Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs

Ahsan Saeed
Robert Bosch GmbH, Stuttgart, Germany

Denis Hoornaert
Technische Universität München, Germany

Dakshina Dasari
Robert Bosch GmbH, Stuttgart, Germany

Dirk Ziegenbein
Robert Bosch GmbH, Stuttgart, Germany

Daniel Mueller-Gritschneder
Technische Universität München, Germany

Ulf Schlichtmann
Technische Universität München, Germany

Andreas Gerstlauer
The University of Texas at Austin, TX, USA

Renato Mancuso
Boston University, MA, USA

Abstract

Temporal isolation is one of the most significant challenges that must be addressed before Multi-Processor Systems-on-Chip (MPSoCs) can be widely adopted in mixed-criticality systems with both time-sensitive real-time (RT) applications and performance-oriented non-real-time (NRT) applications. Specifically, the main memory subsystem is one of the most prevalent causes of interference, performance degradation and loss of isolation. Existing memory bandwidth regulation mechanisms use static, dynamic, or predictive DRAM bandwidth management techniques to restore the execution time of an application under contention as close as possible to the execution time in isolation.

In this paper, we propose a novel distribution-driven regulation whose goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain target execution time for the RT applications. Using existing interconnect-level Performance Monitoring Units (PMU), we can observe the Cumulative Distribution Function (CDF) of the per-request memory latency. Regulation is then triggered to enforce first-order stochastical dominance with respect to a desired reference. Consequently, it is possible to enforce that the overall observed execution time random variable is dominated by the reference execution time. The mechanism requires no prior information of the contending application and treats the DRAM subsystem as a black box. We provide a full-stack implementation of our mechanism on a Commercial Off-The-Shelf (COTS) platform (Xilinx Ultrascale+ MPSoC), evaluate it using real and synthetic benchmarks, experimentally validate that the *timeliness objectives* are met for the RT applications, and demonstrate that it is able to provide 2.2x more overall throughput for NRT applications compared to DRAM bandwidth management-based regulation approaches.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases temporal isolation, memory latency, real-time system, multi-core

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.4

Funding *Ahsan Saeed*: This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669.
Denis Hoornaert: Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.
Renato Mancuso: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grants number CCF-2008799 and CNS-2238476.

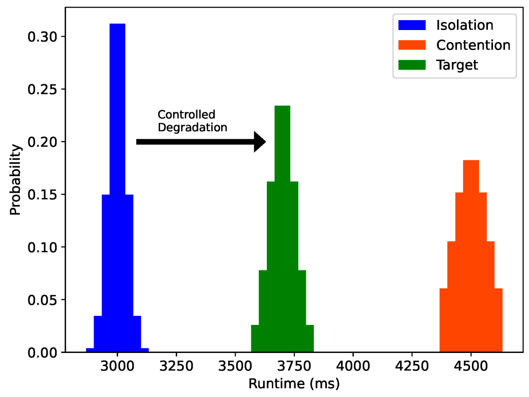


Figure 1 Execution time distributions in isolation (blue) and contention (red). The controlled degradation target can be expressed by reasoning in terms of controlled distribution shift (green).

1 Introduction

An important trend across industrial, automotive and avionics domains is the adoption of MPSoCs. However, a key barrier in designing mixed-criticality systems is the presence of shared resources like the main memory, the cache and the interconnect, which makes it non-trivial to bound the execution time of RT applications running on these MPSoCs. This is because when two or more applications are executed in parallel on different cores, which we refer to as the contention scenario, the interaction between them on shared hardware resources can lead to unforeseen and unpredictable delays [8, 34, 36]. It is well known that memory contention is a key source for performance degradation [7], and practitioners across the industry and academia are looking for solutions that facilitate temporal isolation between applications while using COTS platforms.

Existing hardware-oriented mechanisms for memory interference control require dedicated hardware [2, 11, 13] that is not feasible in COTS multi-core platforms. In contrast, software-oriented memory bandwidth management-based regulation mechanisms are promising grassroots techniques to approach the problem of controlling memory interference by periodically monitoring the memory bandwidth originating from each core and stalling cores when the egress memory bandwidth exceeds a pre-defined threshold. This threshold can be (1) fixed and computed offline for a given combination of applications [5, 41], (2) predicted on the fly [5, 41, 42] or (3) computed dynamically by instrumenting the current memory utilization at the memory controller [23]. A common denominator across the above approaches is that (1) the system parameters for regulation are based on experimental evaluation and not on a formal analysis (2) they focus on restoring the execution time of an application under contention as close as possible to the execution time in isolation.

Ideally, however, the aggressiveness of regulation should directly depend on the target execution time. Indeed, if the RT applications have sufficient slack, less aggressive regulation is desirable as it enables better progress for the NRT applications. Consider the qualitative situation depicted in Figure 1. On the left (resp., right) side of the figure, we depict the distribution of execution time of an application executing in isolation, blue area (resp., contention, red area). Controlled degradation (green area) is achieved if a *bounded shift* is allowed from the solo case and in the direction of the contention case. With this intuition, a *timeliness objective* can be non-ambiguously expressed as a (1) target execution time and (2) a condition on the mass of the execution time distribution that can cross said target.

In this paper, we propose a distribution-driven regulation approach, whose goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain target execution time. This definition allows us to unite WCET-like constraints and high-percentile latency constraints typical of real-time cloud systems (tail latency). The basic premise of our approach stems from the observation that the latency distribution of memory transactions of an application under contention gets skewed compared to the execution in isolation. Therefore, it is possible to precisely influence the overall application execution time so long as we can (1) characterize this distribution and (2) affect its shape via regulation. With this basic principle, we first theoretically compute the reference CDF from the distribution of the per-request memory latency for a given target execution time. Then, we enforce first-order stochastical dominance by periodically checking that the CDF of the observed memory latency distribution of the RT application (obtained by sampling at the PMU) stays above the reference CDF of the per-request memory latency. In case this condition is violated, the NRT cores are suspended till the condition of first-order stochastical dominance holds again. If the reference per-request memory latency first-order stochastically dominates the observed latency, then it follows that the overall execution time random variable is dominated by the reference execution time random variable. Consequently, the observed execution time achieves the *timeliness objective*.

The proposed distribution-driven regulation truly considers the impact of memory contention on the latency and execution time of an application, as opposed to memory bandwidth-based [5, 41, 42] or memory utilization-based approaches [23]. Furthermore, we can also control the level of degradation while guaranteeing timeliness by varying the reference CDF of the per-request memory latency.

With this work, we make the following contributions:

1. To the best of our knowledge, our work is the first that demonstrates the use of an interconnect-level PMU to capture the latency distribution of memory transactions and to leverage it for precise control over an application's execution time under contention.
2. We mathematically characterize the distribution of memory latency for an application and demonstrate its effect when the application is executed in isolation and contention.
3. We provide a formal mathematical proof supporting how our proposed approach meets the imposed *timeliness objective* for the RT applications, ultimately enabling controlled degradation.
4. Finally, we perform an evaluation on a COTS platform (Xilinx Ultrascale+ MPSoC) using an extensive set of realistic and synthetic benchmarks from the San Diego Vision Benchmarks [35], DAPHNE [30], and IsolBench [33] suites. We demonstrate its effectiveness in (1) allowing controlled degradation, (2) providing probabilistic guarantees for RT application, and (3) reducing the execution time of NRT applications by up to 2.2x compared to DRAM bandwidth management-based regulation approaches.

The rest of the paper is organized as follows: Section 2 provides the survey of related work. Section 3 describes the system model and the main assumptions of our approach. After presenting the main theory behind our approach and its mathematical formalization in Section 4, Section 5 describes the overall architecture and the main algorithm of our approach. Section 6 describes the implementation, and Section 7 discusses the experimental setup and presents the results. Finally, Section 8 concludes with a summary and outlook on future work.

2 Related Work

There has been a significant amount of work [18] to tackle the issue of memory interference.

The first category includes techniques that essentially employ *memory bandwidth management-based regulation*. In this category of approaches, the effects of memory contention are statically regulated by controlling the outgoing memory bandwidth from each core as in MemGuard [5,41,42], or by directly measuring the utilization at the memory controller [23] and then based on the observed utilization, dynamically regulating the outgoing memory bandwidth from each of the cores. In these approaches, the designer has to experimentally derive the correct system parameters, and furthermore, there are no formal techniques to guarantee the impact of such a regulation on the execution time of the application.

The second category includes *profile-driven approaches* like E-WarP [27,29] and the work in [1], where an application's behavior is profiled to sufficiently characterize it. Then, together with insights into the underlying regulation mechanism – E-WarP uses Memguard under the hood – it is possible to accurately predict the worst-case execution time. In contrast, the proposed approach in this paper is not about predicting the WCET but rather about setting a target execution-time distribution and adjusting the regulation scheme accordingly.

The third category of approaches falls broadly into the category of *WCET estimation approaches* [14, 18, 20]. These approaches perform WCET estimation by leveraging detailed models of the memory subsystem and do not assume any specific regulation approach. They only consider worst-case memory access latencies considering a certain arbitrary memory placement (bank arrangement) and the underlying workload.

Next, there are the hardware-based regulation mechanisms, which include using a dedicated memory controller [2] or additional hardware like FPGAs [11,13], which is orthogonal to our approach. In addition, embedded high-performance platforms are increasingly offering QoS modules [25,31,45] on the interconnect between masters (CPUs, GPUs, DMAs) and main memory to regulate and prioritize memory requests. However, the existing QoS modules account for the traffic generated by the core cluster connected to the interconnect as a single master, which does not alleviate cross-core contention [21]. Secondly, a static QoS configuration may lead to inefficiencies in the utilization of the underlying DRAM subsystem for dynamic workloads.

Other hardware-based techniques for COTS platforms, such as RDT [9,28] and MPAM [44], essentially enforce a desired memory bandwidth limit at the hardware-level. This reduces the regulation overhead and significantly improves the granularity of bandwidth regulation. The recently proposed MemPol [46] loosely belongs to this category because it leverages debug interfaces to halt/resume CPUs with the goal of enforcing a target bandwidth. Despite said benefits, the aforementioned shortcomings of memory bandwidth management-based regulation are still present. Nonetheless, a promising direction for future work entails combining the techniques proposed in this paper with hardware-based bandwidth enforcement.

We approach the problem from a different perspective by not relying on the notion of DRAM bandwidth. Instead, we directly reason on the properties of the observed distribution of latencies for the memory transactions performed by the application under analysis. Our approach starts by considering design-time timeliness constraints and uses one such specification to construct a target cumulative distribution (CDF). The latter is then used to enact regulation. The proposed approach also makes no assumptions on the memory transactions generated by the contending applications.

3 System Model and Assumptions

We hereby review the key assumptions and the system model required for the results presented in Section 4 to hold. These assumptions are also experimentally validated in Section 7.2 and Section 7.3.

A1: Multicore Platform Topology. We assume a system comprised of m application CPUs $\Pi_1, \ldots, \Pi_m$. For simplicity, we assume that the high-criticality workload is only deployed on CPU Π_1, which can be considered the *real-time core*. The memory hierarchy comprises zero or more levels of cache. Cache misses caused by `load` or `store` instructions at the last-level cache (LLC) cause read/write memory requests to be initiated towards a single shared main memory subsystem via a single shared bus. Note that we distinguish between memory instructions (`load`/`store`) and the resulting traffic that they might cause in terms of read (and possibly write) requests to the underlying main memory subsystem.

A2: Cache Model. We assume that (1) either all the cache levels are private per-core caches, or (2) if shared cache levels exist, they can be partitioned among the cores to prevent inter-core cache interference. All the cache levels adopt a write-back, write-allocate policy. By write-allocate, `store` instructions that cause a cache miss to trigger a read memory request downstream to fill the cacheline to be modified. A cacheline that has been modified is marked as *dirty*. By write-back, cache refills might trigger a write memory request downstream if the cache replacement policy has selected a dirty cacheline for eviction. We make no assumption about the specific cache replacement policy adopted by the cache controllers at the different levels. We make no assumption about the inclusiveness of adjacent cache levels.

A3: In-order CPUs. We assume that the considered CPUs are unable to reorder instructions. Thus, the latency incurred by pending `load` instructions is additive with respect to the time spent executing instructions that do not perform memory operations. The same is true for `store` instructions. This assumption is pessimistic yet safe if out-of-order CPUs are considered instead.

Timing anomalies arising due to microarchitectural effects can violate this assumption. In this work, we followed a measurement-based evaluation approach. Therefore, timing anomalies are accounted for in the measured runtime. If these anomalies are to be estimated using static analysis, the work in [12] demonstrates that timing anomalies can be statically bounded and accounted for at design time without introducing an intractable amount of pessimism.

A4: Blocking reads, non-blocking writes. As per A2, both `load` and `store` instructions cause an LLC cache miss to trigger a read request to the main memory. As per A3, the latency incurred by such read requests is additive with respect to the time spent by the rest of the instructions that do not generate main memory requests. Conversely, if a memory instruction triggers a write-back to the main memory, the resulting write memory transaction is carried out non-blockingly with respect to the instruction stream under analysis. Therefore, the latency of read requests in main memory is *on the critical path* from the standpoint of total execution time, while the latency of write requests is not. This is not to say that the contention generated by write requests is not considered, but rather that what matters is their impact on the latency of read transactions.

Note that, in typical DRAM subsystems, batched write requests could be prioritized over reads, causing read requests to temporarily stall. However, by controlling the latency distribution of read requests, one can control how this reflects into the total execution time, essentially factoring in the overall impact of write requests.

A5: Measurable Read Latency Distribution. We assume that the platform provides a performance monitoring unit (PMU) capable of collecting measurements on the latency of read memory requests. The PMU shall be located at the interface of the shared bus and main memory subsystem. The latency is measured as the difference between the timestamp at which a read request is forwarded to the main memory and the timestamp at which the response for the said request is returned (request turnaround time). We assume that, when queried, the PMU can return (an approximation of) the distribution of the observed latencies of read requests issued by a core Π_k under analysis. We will discuss the ability to do so in commercial platforms in Section 6.

A6: Computation and Read-latency Additivity. By A4 and A5, we can decompose the worst-case execution time E as a sum of two contributions $E = C + L$, where L is the total latency of read memory transactions. Let N denote the worst-case number of read requests and let us indicate the per-request latency as l_i, then $L = \sum_{i=1}^{N} l_i$. C denotes the time spent for anything other than waiting for read responses, and is a constant, regardless of whether the workload executes in isolation vs. contention. Conversely, l_i and thus L and E are random variables that are affected by the level of congestion of the main memory subsystem. In practice, we observe a small deviation (less than 1.8%) in the value of C when measured in isolation vs. under contention, as evaluated in detail in Section 7.3. One such deviation might arise from contention over Miss Status Holding Registers (MSHR) [33] or LLC tag/data banks [6]. For the sake of simplicity, C is assumed to be constant in our theoretical formulation. In practical instantiations of our framework, this value should be experimentally derived and a safe upper-bound on the compute-only time shall be used.

A7: Profiled Critical Workload. We assume that the high-criticality workload deployed on Π_1 can be profiled offline to derive the worst-case execution time E_{isol} and total read latency L_{isol} in isolation. This can be done using traditional measurement-based approaches and allows us to upper-bound the value of $C = E_{isol} - L_{isol}$, which is the time spent by the CPU to carry out any other operation except waiting for read requests to be fulfilled. As per A2, C is computed with statically partitioned shared caches (if any). As per A5, L_{isol} measurement is enabled by the PMU.

A8: I.I.D. Read Transaction Latencies. We assume that l_i are independent samples from the same (unknown) distribution. Intuitively, the independence arises from the fact that between any two subsequent read transactions, a random amount of time can elapse, and a random amount of congestion can be caused by interfering CPUs. Thus, l_i's are independent and identically-distributed (i.i.d.) random variables.

4 Distribution-Driven Regulation

In this section, we introduce the theoretical results that represent the foundation of the proposed distribution-driven regulation. We introduce the notations in Table 1.

Table 1 Summary of notation used.

Symbols	Descriptions	Symbols	Descriptions
E_{isol}	Total execution time in isolation	$\bar{l}_{\sigma^2}$	Variance of read memory transactions reference
E_{reg}	Total execution time under regulation	L_{isol}	Total latency of read memory trans. in isolation
$\bar{E}$	Total execution time target	l_{min}	Min read latency
C	Non-memory compute time	l_{max}	Max read latency
L	Total latency of read memory transactions	l_i	Latency of an individual read memory transaction i
l_μ	Mean latency of read memory transactions	N	Worst-case number of read requests
l_{σ^2}	Variance of read memory transactions	α	Acceptable tolerance for execution time to exceed $\bar{E}$
$\bar{l}_\mu$	Mean latency of read memory trans. reference		

Regulation Goal. Unlike the related literature surveyed in Section 2, our goal is to achieve a *timeliness objective* formulated as a constraint on the probability of meeting a certain execution time target $\bar{E}$. Formally, given an execution time target $\bar{E}$ and an acceptable error $\alpha \in [0, 1]$, the goal of regulation can we written as

$$P(E_{reg} \leq \bar{E}) \geq 1 - \alpha, \tag{1}$$

where E_{reg} is the actual execution time observed under regulation and (possibly) in the presence of main memory contention for the application under analysis. When α is such that $\alpha \to 0$, then $\bar{E}$ represents a worst-case execution time (WCET) constraint. Note however that the timeliness constraint formulation in Eq. 1 is more generic. For instance, setting $\alpha = 0.01$ expresses a 99^{th}-percentile tail latency requirement on E_{reg}.

Goal-driven Regulation Strategy. We hereby describe how the regulation strategy can be built from the goal formulated in Eq. 1 given a value of $\bar{E}$ and α. Following the notation and assumptions in A6 (Section 3), we can rewrite Eq. 1 as follows:

$$P(C + L \leq \bar{E}) = P\left(\sum_{i=1}^{N} l_i \leq \bar{E} - C\right) \geq 1 - \alpha. \tag{2}$$

The key insight into our approach is that, by controlling the distribution of per-request latency l_i via regulation, we can directly control the distribution of the total memory latency L and thus impact the distribution of E_{reg} to satisfy Eq. 1.

As we previously mentioned, l_i's are independent and identically-distributed random variables (as per A8) following an unknown distribution. Call l_μ and l_{σ^2}, respectively, the (unknown) mean and variance of the l_i random variables. From the Central Limit Theorem (CLT) [10], it holds that the random variable Z constructed as

$$Z = \frac{\sum_{i=1}^{N} l_i - N l_\mu}{\sqrt{N l_{\sigma^2}}} = \frac{L - N l_\mu}{\sqrt{N l_{\sigma^2}}} \sim \mathcal{N}(0, 1), \tag{3}$$

follows a standard normal distribution, i.e. a normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$. The latter property is captured by the notation $Z \sim \mathcal{N}(0, 1)$. Note that Eq. 3 only holds for large values of N. Since our goal is to analyze and regulate memory-intensive applications, this condition holds. In fact, our experiments described in Section 7 highlight that for the considered applications, the order of magnitude of N is somewhere between 10^6 and 10^7.

From Eq. 3 we can derive that $L \sim \mathcal{N}(N l_\mu, N l_{\sigma^2})$. Let us indicate with $\Phi(x)$ the Cumulative Distribution Function (CDF) of the standard normal distribution. We can then rewrite Eq. 2 as follows:

$$P(L \leq \bar{E} - C) = \Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \geq 1 - \alpha. \tag{4}$$

So far, we have treated l_μ and l_{σ^2} as unknown values. The insight at this point is that, when regulation is performed (by pausing/resuming the activity of interfering cores), we can exert direct control over the underlying distribution of $L = \sum_{i=1}^{N} l_i$ and thus over the value of Nl_μ and Nl_{σ^2}. In fact, our goal is not to enforce a specific value of l_μ and l_{σ^2}. Instead, it is enough to identify two values $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that the following inequality holds for every value of $\bar{E} \in \mathbb{R}^+$:

$$\Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \geq \Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \geq 1 - \alpha. \tag{5}$$

Regulation Condition. Recall from A5 in Section 3 that we are able to periodically *snapshot* the distribution of read latencies. By enacting start/stop control over the interfering cores, we can impact such distribution. We are now ready to derive the condition according to which, given a snapshot, we should pause or resume the activity of the interfering cores.

More specifically, we can *observe* the CDF of the random variable l_i while the application under analysis is running. Call this observed CDF function $F_l(t) = P(l_i \leq t)$. If regulation is applied such that

$$\forall t \in \mathbb{R}^+, F_l(t) \geq \Phi\left(\frac{(\bar{E} - C) - \bar{l}_\mu}{\sqrt{\bar{l}_{\sigma^2}}}\right) = \bar{F}_l(t), \tag{6}$$

then we have two properties. The first, is that $\bar{F}_l(t)$ is the CDF of a random variable $l_i^{norm} \sim \mathcal{N}(\bar{l}_\mu, \bar{l}_{\sigma^2})$. The second is that l_i^{norm} is said to first-order stochastically dominate l_i [26]. Indeed, Eq. 6 is one possible definition of first-order stochastic dominance, also indicated with the notation $l_i^{norm} \geq_1 l_i$.

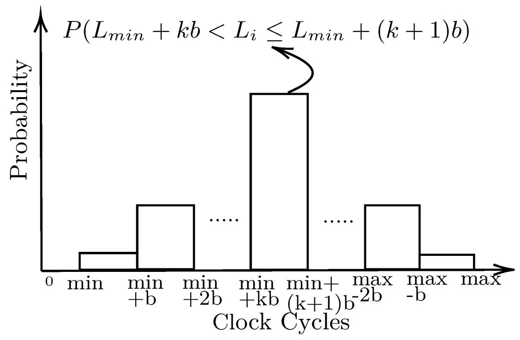
It is a known result [26, Theorem 1.A.3] [19, Lemma 6] that stochastical dominance between random variables implies stochastical dominance in the aggregate. Formally, given two random variables X and Y and a positive integer k, if Y is k-th order stochastically dominated by X (i.e., $X \geq_k Y$), then $\forall n \in \mathbb{N}^+$ and i.i.d. replicas $X_1, \ldots, X_n$ of X and $Y_1, \ldots, Y_n$ of Y it holds that

$$\sum_{i=1}^{n} X_i \geq_k \sum_{i=1}^{n} Y_i \implies \sum_{i=1}^{n} X_i \geq_1 \sum_{i=1}^{n} Y_i. \tag{7}$$

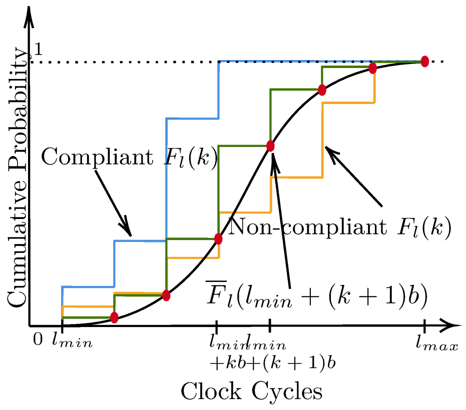
Next, we note that from Eq. 7 and 6 it immediately follows that $\sum_{i=1}^{N} l_i^{norm} \geq_1 \sum_{i=1}^{N} l_i$. Moreover, by leveraging the properties of the normal distribution [17], we know that $\sum_{i=1}^{N} l_i^{norm} \sim \mathcal{N}(N\bar{l}_\mu, N\bar{l}_{\sigma^2})$. This brings us to the final step. That is, the random variable L under regulation is first-order stochastically dominated by a normal distribution of mean $N\bar{l}_\mu$ and variance $N\bar{l}_{\sigma^2}$. This means that, as long as Eq. 6 is ensured via regulation, Eq. 5 holds.

Final Formulation. Putting everything together, we have the following workflow. First, given the target $\bar{E}$ and α, numerically compute $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that

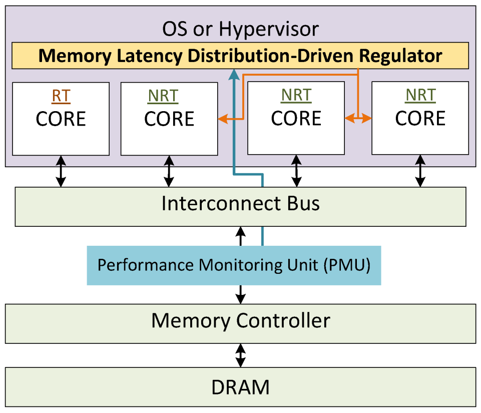
$$\Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \geq 1 - \alpha \tag{8}$$



■ **Figure 2** Visual representation of the read-latency PMF.



■ **Figure 3** Discretized, compliant and non-compliant latency CDF.



■ **Figure 4** Overview of our system architecture consisting of MPSoC.

holds. Second, use the same values of $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ to construct the target per-request latency CDF $\bar{F}_l$ as described in Eq. 6. Next, at runtime, observe the CDF of l_i, namely F_l, and pause/resume (regulate) the activity of the non-real-time CPUs to ensure that $\forall t \in \mathbb{R}^+, F_l(t) \geq \bar{F}_l(t)$. So long as this inequality holds, it also holds that $P(C + L \leq \bar{E}) = P(L \leq \bar{E} - C) \geq 1 - \alpha$ because Eq. 5 holds.

4.1 Discrete-domain Formulation

The results derived so far in Section 4 assume that we are able to snapshot online a *continuous* distribution of read latency accesses. This is practically impossible with realistic hardware. In this subsection, we relax precisely this requirement.

Let l_{min} and l_{max} be the minimum and maximum possible read latency. Consider a realistic PMU that defines K latency observation bins with configurable size b. If a transaction i was counted in the first bin, then its latency l_i was somewhere in the range $[l_{min}, l_{min} + b)$; more in general, if it was counted in the k^{th} bin with $k \in \{0, \ldots, K-1\}$, then its latency was somewhere in the range $[l_{min} + kb, l_{min} + (k+1)b)$.

When queried, the PMU reports the number of read transactions completed by Π_1 whose latency fell in each of the K bins. Assume that this number is cumulative since the time at which the application was launched – if it is reset after a snapshot, e.g. to prevent overflows

Algorithm 1 Memory Latency Distribution-Driven Regulator.

```
input : number of latency bins $K$, reference CDF $\bar{F}_k \; \forall \; k \in \{0 \dots K-1\}$
foreach regulation interval $r$ do
    foreach latency bin $k \in \{0 \dots K-1\}$ do
        Sample the height of latency bin $l_{k,r}$
        $\gamma_{k,r} = \gamma_{k,r-1} + l_{k,r}$
    end
    foreach latency bin $k \in \{0 \dots K-1\}$ do
        $f_{k,r} = \frac{\gamma_{k,r}}{\sum_{k=0}^{K-1} \gamma_{k,r}}$          ▷ Normalize bins to obtain PMF
        $F_{k,r} = \sum_{m=0}^{k} f_{m,r}$          ▷ Construct observed CDF
    end
    if $F_{0,r} < \bar{F}_0 \vee \dots \vee F_{K-1,r} < \bar{F}_{K-1}$ then
        suspend all NRT cores
    else
        resume all NRT cores
    end
    $r = r + 1$
end
```

in the counters, then it can be accumulated in software at each snapshot. In software, divide the number of transactions in each bin (i.e. the height of the bin) by the total number of transactions in the entire snapshot. The result is a valid observed Probability Mass Function (PMF) $f_l(k)$ for the read request latency l_i for the generic request i. Figure 2 provides a visual representation of the PMF. In other words, the height of each bin provides the value of $f_l(k) = P(l_{min} + kb \leq l_i < l_{min} + (k+1)b)$. From the acquired PMF, it is easy to compute the corresponding observed CDF as

$$F_l(k) = \sum_{j=0}^{k} f_l(j) = P(l_i < l_{min} + (k+1)b). \tag{9}$$

Recall that (Eq. 6) we can construct a normal distribution $\bar{F}_l(t)$ of reference with appropriate values of $\bar{l}_\mu$ and $\bar{l}_{\sigma^2}$ such that Eq. 8 is satisfied. At runtime, whenever a new read latency distribution snapshot is acquired, it is enough to check the following condition:

$$\forall k \in \{0, \dots, K-1\}, F_l(k) \geq \bar{F}_l(l_{min} + (k+1)b). \tag{10}$$

This condition is visually depicted in Figure 3. Indeed, if the condition expressed in Eq. 10 holds, then our reference $l_i^{norm} \sim \mathcal{N}(\bar{l}_\mu, \bar{l}_{\sigma^2})$ first-order stochastically dominates l_i. This is the case for the blue curve in Figure 3. Conversely, if for some k Eq. 10 does not hold, the non-real-time CPUs must be paused – regulation must be triggered. This is the case for the orange line in Figure 3. The implicit assumption, which we validate in Section 7.3, is that pausing the interfering CPUs allows to *shift* the observed $F_l(k)$ in subsequent snapshots.

Finally, note that numerically computing the value of $\bar{F}_l(t)$ online can lead to excessive overhead in the regulator. Instead, the K values of $\bar{F}_l(k)$ necessary to check the validity of Eq. 10 can be pre-computed offline and stored in a lookup table for efficient online retrieval. These values are depicted as red dots in Figure 3.

5 System Overview

An overview of our system architecture is depicted in Figure 4. We consider an MPSoC in which a core designated as RT core is dedicated to host time-sensitive RT applications, while the others are designated as NRT cores that host performance-oriented NRT applications.

The purpose of the memory latency distribution-driven regulator introduced in Section 4 is to achieve the *timeliness objective* (Equation (1)) on the execution time of applications running on the RT core. The regulator is activated periodically on each NRT core using a timer interrupt. The timer interrupt triggers the sampling of memory latency distribution using the Performance Monitoring Unit (PMU) (shown in blue in Figure 4) for the memory transactions originating from RT core. This memory latency distribution is normalized to obtain the probability mass function (PMF), as described in Section 4.1 and then is used to derive the cumulative distribution function (CDF). From the CDF, we enforce the rule of first-order stochastic dominance (Equation (6)), which states that if any bin violates the reference CDF for the target distribution of execution time, the regulation is triggered, and all the NRT cores are suspended, as highlighted with red lines in Figure 4.

In principle, the regulator could reside either in software, such as the Operating System (OS) or hypervisor, or in hardware, such as a Field Programmable Gate Array (FPGA). For analysis and evaluation of the mechanism, the regulator optionally stores the PMF and key characteristics in the DRAM memory.

The proposed mechanism can be implemented on any platform on which we are able to measure (1) memory latency distribution and (2) filter the memory transaction on a per core basis.

5.1 Memory Latency Distribution-Driven Regulator Algorithm

Algorithm 1 sketches our proposed distribution-driven regulation. Let the total number of bins in the memory latency distribution be denoted by K. Furthermore, we denote by $\bar{F}_k$ the reference CDF assigned to each bin.

At the beginning of each regulation interval $r > 1$, the regulator first samples the number of transactions (since the last interval) with latency that falls in bin $l_{k,r}$. This is repeated for each bin (Line 3). The samples are accumulated into the variable $\gamma_{k,r}$ (Line 4). We then apply height normalization to derive the PMF f_k (Line 7). The PMF is converted into a CDF F_k by summing up the probabilities associated with the variable up to each bin (Line 8). This CDF F_k is then compared against the reference CDF $\bar{F}_k$ for each bin (Line 10). If the condition in Eq. 10 does not hold, all the NRT cores are suspended (Line 11). They will be resumed only when Eq. 10 holds again (Line 15).

The theoretical formulation provided in Section 4 assumes that the PMF (or CDF) of the per-request latency can be observed infinitely fast. Clearly, this is not possible in realistic hardware, hence a non-zero regulation interval T_r must be picked. Because of that, what could happen is that during T_r, the distribution of memory latencies shifts so drastically that it cannot be recovered. Although this can happen, its effect can be easily bounded. In the worst-case, right after a snapshot that satisfied Eq. 10 (otherwise, the NRT cores would be stopped) with exact equalities between left- and right-hand sides, a back-to-back sequence of memory transactions with latency l_{max} occurs. These can be at most $\lceil T_r/l_{max} \rceil$ because Π_1 is an in-order CPU (A3 in Section 3). Thus, the extra time cost $H = (l_{max} - l_{min})\lceil T_r/l_{max} \rceil$ can be accounted for by computing a new, more restrictive $\bar{E}' = \bar{E} - H$. Interestingly, since we can observe the typical latency distribution under unrestricted contention, it is also possible to compute the probability that such a case can occur.

6 Implementation

We have performed a full-system implementation that includes a partitioning hypervisor augmented to support the proposed memory latency distribution-driven regulator. The implementation is carried out on the Xilinx Ultrascale+ Multi-Processor System-on-Chip (MPSoC) ZCU102 [40]. The SoC features 4 ARM Cortex A53 [4] cores clocked at 1.2 GHz. Each core has its own private L1 data and instruction cache, whereas the 4 cores share a unified L2 cache. The SoC also features a tightly-coupled FPGA, which is not needed to implement the proposed approach. We only use the FPGA for the validation experiments on the nature of DRAM read transaction latencies conducted in Section 7.2.

We use the Jailhouse-RT partitioning hypervisor [15, 27] to partition resources in our system, which is an ideal choice for this type of implementation because it is lightweight, easy to port/modify, includes support for cache coloring [16, 43] and bandwidth regulation, and is open-source.

6.1 AXI Performance Monitor (APM)

We sample the memory latency in the Xilinx Ultrascale+ MPSoC [40] using the AXI Performance Monitor (APM) hardware module. The APM measures the key performance metrics like the amount of read/write memory transactions, min/max/total latency, and other performance metrics for the AMBA AXI [3] in a system. The APMs implemented on Xilinx Ultrascale+ MPSoC [40] are based on the Xilinx AXI Performance Monitor available as a LogiCORE IP [37].

The APM has 10 hardware counters that can be configured to simultaneously monitor up to 10 performance metrics for any interface points called slots on the AXI interconnect. There is also a global-clock counter in addition to these 10 hardware counters that run at the APM clock frequency of 533.5 MHz.

The APM can be configured to monitor the performance metrics for a particular slot using the *Metric Selector* register. Furthermore, the APM contains a Range Incrementer module that compares the performance metric count with the low and high ranges from the *Range* register and increments the count of the given performance metric by one if the value falls within the limits. The Range Incrementer is useful in obtaining the read/write latency ranges that we leverage in this work to sample the memory latency distribution.

We configured 8 *Metric Selector* registers in conjunction with 8 *Range* registers to monitor read memory latency (as defined in Section 3 *A6: Measurable Read Latency Distribution*) with respectively low and high ranges of 0-40, 41-80, 81-120, 121-160, 161-200, 201-240, 241-280, and 281-2000 clock cycles. The rationale behind the selection of these ranges is discussed in Section 7.4. These 8 performance metrics provide the number of read memory transactions that fall within the given read memory latency limits, referred to as bins. Furthermore, 2 *Metric Selector* registers are configured to report the total number of read transactions and total read latency. The total number of read transactions is N, as used throughout the mathematical formalization in Section 3. Additionally, we verify that the total number of read transactions and the sum of all bins are always the same. This ensures that no memory transaction *escapes* the bins. The global-clock counter is used as the reference for all the calculations in this paper. The included hardware counters can be set and read via a memory-mapped interface.

The APM slot is configured to monitor the AXI communication between the cores and the memory controller. In addition, we employ the AXI ID filtering to monitor the transactions emanating from a core with a certain AXI ID. The AXI IDs for the cores are evaluated experimentally. Once the AXI IDs for each core have been determined, we utilize the *Filter* and *Mask* registers to set up AXI ID filtering.

Currently, the APMs are adopted in Xilinx Ultrascale boards. However, since these APM IPs are part of the AXI bus, they are deployable on other SoCs. They can also be deployed in programmable logic (FPGA) to gather statistics on the traffic observed over AXI bus segments generated, for instance, by in-FPGA accelerators.

7 Validation and Evaluation

In this section, we first experimentally validate the key assumptions presented in Section 3. Then we discuss the key design parameters of our system. Finally, we present a full system evaluation where we validate the effectiveness of our approach to ensure the timeliness of different sets of applications.

7.1 Experimental Setup

We evaluate our approach on the Xilinx Ultrascale+ Multi-Processor System-on-Chip (MPSoC) ZCU102 [40] as introduced in Section 6. A combination of real-world [35], [30], and synthetic [33] benchmarks are used to evaluate the proposed approach. For our real-world benchmarks, we use a subset of the benchmarks in the San Diego Vision Benchmark Suite (SD-VBS) [35]. The input dataset for the benchmark applications comes in 9 different sizes. Since we are interested in DRAM-bounded applications, we use the ones with the largest input data size (named *FullHD*). The other benchmark suite is the Darmstadt Automotive Parallel Heterogeneous Benchmark Suite (DAPHNE) [30], which represents parallelizable workloads from the automotive domain. For our evaluation, we used the applications that run exclusively on the CPU. We also use a synthetic 'Bandwidth' benchmark from the IsolBench suite [33] that is engineered to continuously perform memory write operations. In the rest of the paper, we refer to this benchmark as the *MemBomb* application.

Unless otherwise stated, all experiments refer to the *isolation scenario* or simply *isolation* in which the disparity application is running on the designated RT core with no other applications running in parallel. In contrast, a *contention scenario* or simply *contention* happens when the same *disparity* application is running on the designated RT core while synthetic *MemBomb* applications are running on the three NRT cores. The *disparity* application is selected as it has the lowest average IPC and the highest average memory utilization [23] in the benchmark suite, making it an ideal candidate for demonstrating memory interference-related effects.

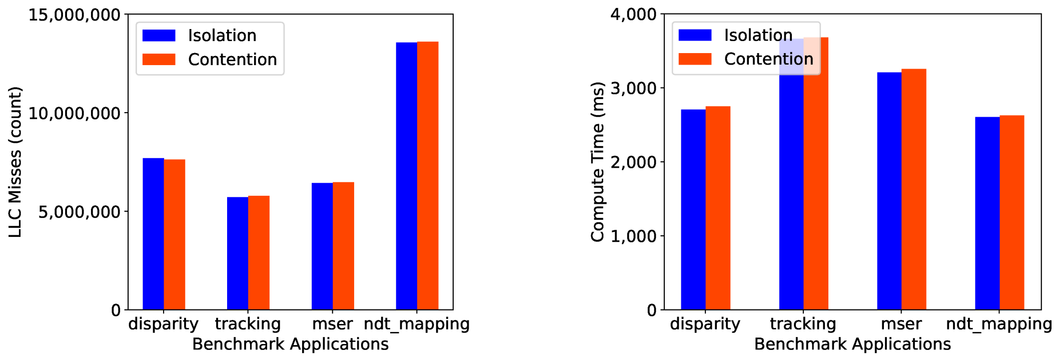
For consistency, we always activate the hypervisor. The regulator is activated on each NRT core to facilitate comparison with a memory bandwidth management-based regulation (MemGuard [5]). However, the current implementation can be extended to sample the PMU values from only one NRT core responsible for suspending the other NRT cores. All the obtained results are calculated on 100 runs for each configuration to remain statistically significant.

7.2 Validation of I.I.D. Assumption A8

In order to validate hypothesis A8 in Section 3, i.e., that the latencies of read memory transactions emitted by the cores are i.i.d., we perform 10 different statistical tests called Permutation Tests [32]. These tests are designed to find evidence that empirical samples are i.i.d.. The rationale is that if i.i.d. holds in all cases, the regulation system is guaranteed to be operated correctly. Conversely, if the i.i.d. property is validated only in some cases, a full-system implementation and evaluation are necessary to assess the correct end-to-end behavior of a system that employs the proposed distribution-driven regulation.

Table 2 Summary of permutation testing results for Synthetic (table upper half) and Real-world (table lower half) memory traffic. Test pass noted with ✓and fail with ×.

Test no.	1	2	3	4	5	6	7	8	9	10	Pass (%)
Synthetic Benchmarks: AXI Traffic Generator											
Rand. Pattern + Rand. ITG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100
Rand. Pattern + Fix ITG	✓	✓	✓	×	✓	✓	×	✓	✓	×	70
Seq. Pattern + Rand. ITG	✓	✓	×	×	✓	✓	✓	✓	✓	✓	80
Seq. Pattern + Fix ITG	✓	✓	×	×	✓	✓	×	✓	✓	×	60
Real-world Benchmarks: SD-VBS											
Best-case	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100
Worst-case	✓	×	✓	×	✓	✓	×	✓	×	×	50
Mode-case	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	90



(a) Similar LLC misses for applications on RT core in isolation and contention.

(b) Similar Compute Time for applications on RT core in isolation and contention.

Figure 5 Experimental results to validate the key assumptions, as stated in detail in Section 3, hold for our system.

Performing permutation testing requires measuring the memory latency of individual memory transactions at the finest granularity. The aforementioned APMs can only measure aggregated latency values and are thus not suitable for the purpose. Instead, and only for these experiments, we leverage the tightly-coupled FPGA of the evaluation platform.

The experiment is divided into four successive steps: (1) generate memory traffic, (2) capture the activity at the AXI level, (3) measure and compile each transaction's response time, and (4) perform a set of permutation tests.

To evaluate the memory latency of both synthetic and real-world benchmarks, we implement two distinct FPGA designs. The first FPGA design is composed of an *AXI Traffic Generator (ATG)* [38], which generates heavy synthetic memory traffic toward the memory controller. We configure the ATG to generate four types of access patterns that combine *random* and *sequential* accesses with *random* and *fixed* inter-transaction gaps (ITG). The traffic activity created by the ATG is captured and stored for post-processing by an *Integrated Logic Analyzer* [39] (ILA), which is also instantiated in the FPGA.

The second FPGA design is implemented to evaluate the real-world memory traffic by observing the activity originating from the main CPUs running SD-VBS benchmarks in isolation. The design is a simplified version of the approach introduced in [22] and consists of only a loopback IP linking the core cluster with the memory controller through the FPGA (i.e., no transformations are performed on the transactions' address). Similarly, the Jailhouse-RT hypervisor [15] is instrumented to target the FPGA memory range instead of the memory controller, making the hypervisor and benchmark memory traffic observable via an ILA. We run different SD-VBS benchmarks with different inputs in a sequence and randomly acquire fragments of memory traces. Thus, while we know that the captured activity belongs to *some* SD-VBS benchmark, we cannot determine which trace corresponds to which specific benchmark.

Table 2 shows the results of the first 10 permutation tests performed on the two FPGA designs, on the top and bottom, respectively. For synthetic benchmarks, the number of passed tests increases as randomness in the pattern, and ITG is introduced. Therefore, for ATG with random memory access pattern and random ITG has the highest tests pass of 100%, whereas sequential memory access pattern with fixed ITG has the lowest test pass of 60%. Hence, the percentage of tests pass increases as access pattern and ITG randomness grow.

For real-world benchmarks, 30 snapshots of memory traffic are captured. Since applications have different phases, the ILA buffer is small, and memory transactions are captured asynchronously, we observed variation in the results of permutation tests. In the best-case scenario, all tests are passed, although pass percentages as low as 50% have been seen on rare occasions. The mode (value that appears most often) indicates a 90% pass.

In summary, the permutation testing indicates that not all tests are passed under all scenarios, albeit an indication that A8 holds in most of the cases has emerged. Nonetheless, we conduct a full-stack implementation to verify that the *timeliness objective* (Equation (1)) we impose is, in fact, met with real-world applications.

7.3 Validation of Other Key System Assumptions

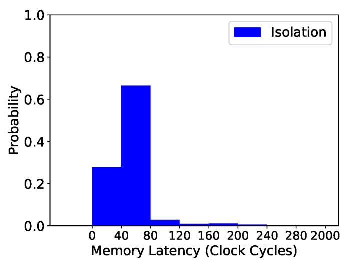
In this subsection, we experimentally validate that the key assumptions, as stated in detail in Section 3, hold for our system.

Validation of A2: Cache Model. First, we show that the total numbers of LLC misses for an application executed in isolation and contention scenarios are comparable. Figure 5a illustrates the average total number of LLC misses that occur during 100 runs for *disparity*, *tracking*, *mser* and *ndt_mapping* in isolation and contention, respectively. It can be observed that the total number of LLC misses is comparable in both scenarios, with an average difference of less than 1% in their counts. This demonstrates that there is no inter-core cache interference, which is consistent with assumption *A2*.

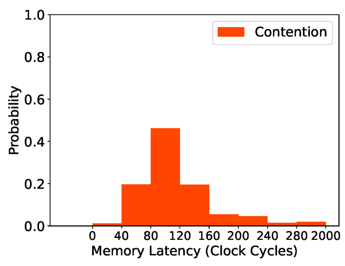
Validation of A6: Computation and Read-latency Additivity and A7: Profiled Critical Workload. Next, we show that the compute time C of an application remains the same in isolation and contention. We measure the worst-case execution time E and the total latency of read memory transactions L and determine the compute time C by: $C = E - L$

In Figure 5b, it is shown that the compute time of the application under consideration (*disparity*, *tracking*, *mser* and *ndt_mapping*) is similar in both the scenarios, with an average difference of less than 1.8%. Thus, assumptions *A6* and *A7* hold.

Validation of A5: Measurable Read Latency Distribution. Finally, we demonstrate the capability of measuring (an approximation of) the latency distribution of read memory transactions in a COTS platform – without redirecting memory transactions through the FPGA – as stated in *A5*.



(a) Isolation.



(b) Contention.

Figure 6 Impact of memory interference on the shape of normalized memory latency distribution for *disparity* on RT core.

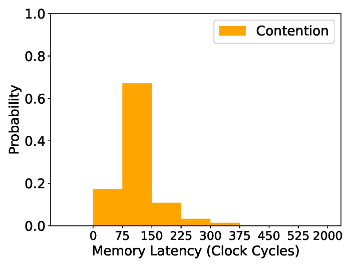


Figure 7 Impact of bin size on the shape of memory latency distribution.

Figure 6 shows the normalized read memory latency distribution obtained from the APM present in the evaluation platform (Xilinx Ultrascale+ MPSoC [40]) in isolation and contention. According to Figure 6a, the majority of individual memory read transactions for *disparity* have a latency of less than 80 clock cycles in isolation.

When multiple contending *MemBomb* applications are running in parallel, the *disparity* benchmark experiences a significant increase in memory latency, resulting in a shift of the memory latency distribution to the right (higher memory latency bins), as seen in Figure 6b. Under contention, the majority of individual memory read transactions have latency in the range of 41 to 160 clock cycles.

7.4 Configuration Parameters

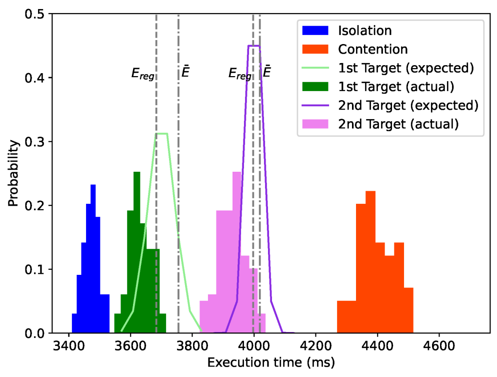
Configuring the proper system parameters is one of the primary challenges system designers face when implementing any regulating mechanism. In this subsection, we explain the key design parameters of our approach and the rationale behind their selection.

7.4.1 Regulation Interval

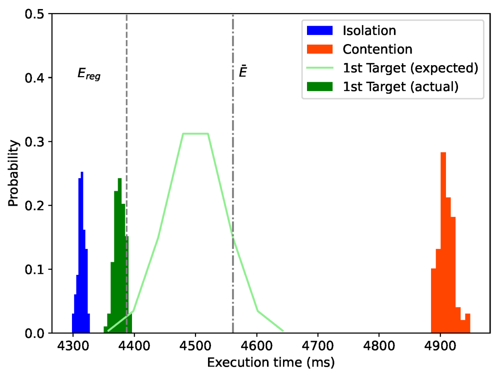
The choice of the regulation interval T_r is a trade-off between regulation granularity and overhead due to the generation of more frequent timer interrupts. The smaller regulation granularity is beneficial for finer grain control over the enforcement of our regulator. A regulation interval $T_r = 1$ ms has shown to yield good results and is set throughout the evaluation setup.

7.4.2 Total Bins

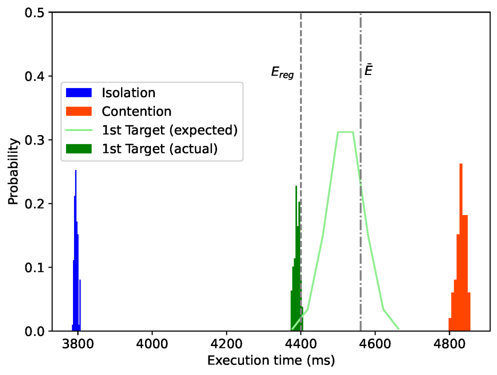
The number of bins defines the quantization that can be used to approximate the memory latency distribution. The PMU present in our evaluation platform offers 10 hardware counters as described in Section 6.1, which can be accordingly used to set 10 latency bins. However, we only dedicate 8 hardware counters for measuring memory latency distribution, resulting in 8 bins. The other two hardware counters are reserved for the purposes of (1) measuring the total number of read transactions as well as (2) the total read latency. This is done to validate the key system assumptions that are specified in Section 3.



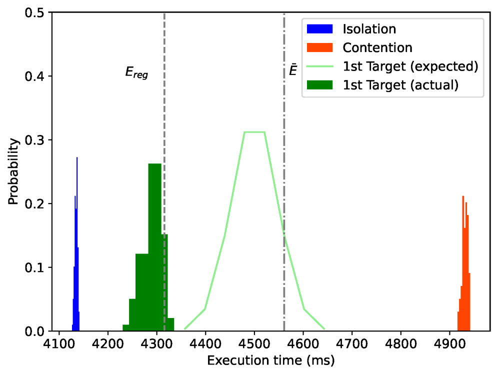
(a) *disparity* on RT core and *MemBomb* on NRT cores.



(b) *tracking* on RT core and *MemBomb* on NRT cores.



(c) *mser* on RT core and *MemBomb* on NRT cores.



(d) *ndt_mapping* on RT core and *MemBomb* on NRT cores.

Figure 8 Execution Time Distribution (for 100 runs each).

7.4.3 Bin Size

The bin size of the memory latency distribution needs to be chosen in such a way that all possible individual read memory latencies can be covered while ensuring that distribution shifts can be effectively captured. Simultaneously, one must ensure that the bins are equally spaced and without discontinuities to provide a well-formed distribution snapshot when sampled. To determine the appropriate bin size, the APM is initially configured to measure the minimum and maximum memory latency values during a set of application runs. We observed a minimum read latency of 38 clock cycles and a maximum of approximately 600 clock cycles. Based on these values, we fix 40 clock cycles as the bin size. We also experimented with a larger bin size of 75 clock cycles with the same setup as shown in Figure 6b, which resulted in nearly empty bins with memory latency values greater than 375 clock cycles, as seen in Figure 7. We set the upper limit of the last bin to 2000 clock cycles in order to capture all conceivable memory latencies that a memory transaction may encounter.

7.5 Effectiveness of the Approach

The objective of this experiment is to show that, given $\bar{E}$ and α, Eq. 1 holds. Figure 8 summarizes the execution time distribution of applications during 100 runs and compares the target execution time $\bar{E}$ against the actual execution time E_{reg}. As a point of reference, the

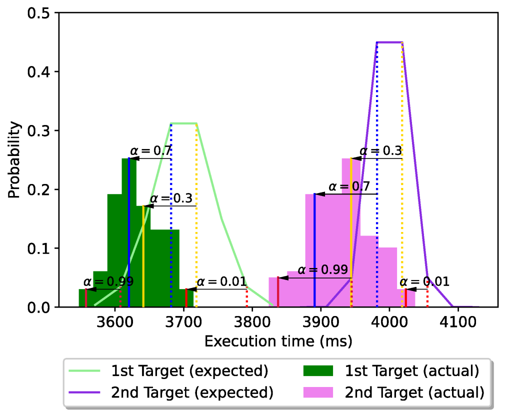


Figure 9 Validation of timeliness objective for various values of acceptable error α.

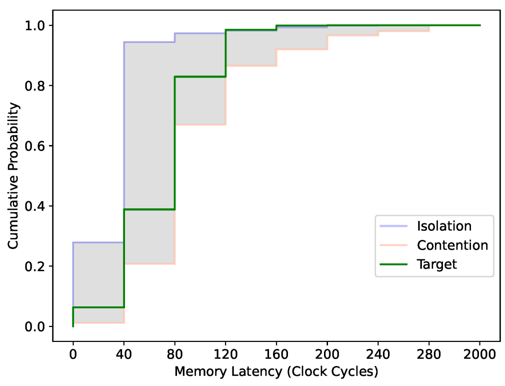


Figure 10 CDF for *disparity* on RT core.

execution time distribution in isolation (blue) and contention (orange) are also provided. The expected discretized execution time distribution of the target execution time is theoretically computed and is depicted as a discretized bell curve, whereas the actual execution time distribution is experimentally evaluated and depicted by a bar plot.

Figure 8a presents the target execution time $\bar{E}$ of 3755 ms and 4018 ms with an acceptable error $\alpha = 0.10\%$ for 1st and 2nd target execution times, respectively. Notably, as there are multiple possible normal distributions for a given $\bar{E}$, we fix $\sigma = \sqrt{\frac{\mu}{2}}$ and $\sigma = \sqrt{\frac{\mu}{6}}$ for the 1st and 2nd $\bar{E}$, respectively and then find the corresponding mean μ that is evaluated to 3700 ms and 4000 ms, respectively. Lowering/rising the standard deviation σ only narrows/widens the normal distribution curve and thus controls the tightness of the timeliness objective. The actual execution time E_{reg} for the given α is 3683 ms and 3997 ms , respectively and less than the target execution time. Hence the timeliness goal defined in Equation (1) is satisfied.

In order to validate the applicability of the approach for diverse workloads, we applied the same methodology to a number of different applications. We considered *tracking*, *mser* and *ndt_mapping* to be RT applications hosted on the RT core, while *MemBomb* running on the three NRT cores, as shown in Figure 8b, Figure 8c and Figure 8d, respectively. We use the same target execution time $\bar{E}$ of 4560 ms with an acceptable error $\alpha = 0.10\%$ for all three sets of experiments. Also, we use the same $\sigma = \sqrt{\frac{\mu}{2}}$. The actual execution time E_{reg} for *tracking*, *mser* and *ndt_mapping* was measured as 4387 ms, 4399 ms and 4315 ms, respectively, which is less than the target execution time and hence satisfies the timeliness goal defined in Equation (1).

Figure 9 shows the validation of the timeliness objective for various values of α for the same set of applications and experimental setup used in Figure 8a. We consider four values of α: 0.01, 0.3, 0.7 and 0.99. These are applied to both the expected and achieved target execution time distribution and highlighted by dashed and solid lines, respectively, in Figure 9. We found out that, for any value of α, the criteria $E_{reg} < \bar{E}$ holds. This provides empirical evidence to corroborate our expectation that the timeliness constraint formula presented in Equation (1) indeed holds for arbitrary values of α.

Finally, we illustrate the CDF of read memory latency observed by the *dispartiy* application in isolation and under contention, as well as the enforced reference CDF. The reference CDF $\bar{F}$ used in Figure 8 for the 1st target execution time $\bar{E}$ of 3755 ms is highlighted with green lines in Figure 10. The CDF in isolation (blue lines) and contention (orange lines) are

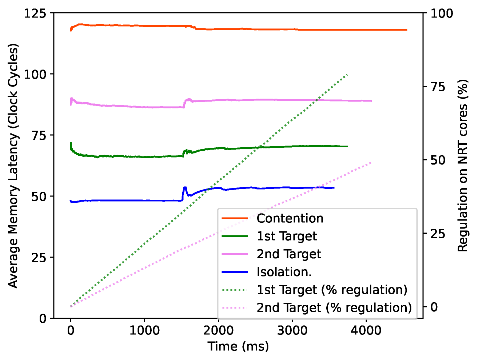


Figure 11 Impact of regulation on the average memory latency for *disparity* on RT core.

computed using the same PMFs previously shown in Figure 6. It can be noted that the $\bar{F}(k)$ computed for each bin k lies between the envelope defined by the CDFs measured in isolation (upper bound) and under contention (lower bound). These $\bar{F}(k)$ values are subsequently used by the memory latency distribution-driven regulator (Algorithm 1) to achieve an execution time E_{reg} that meets the timeliness objective.

7.6 Impact of Regulation on the Average Memory Latency

The selection of the target execution time $\bar{E}$ impacts the aggressiveness of the regulation, which in turn affects the average memory latency of an application. The average memory latency is defined as the total read memory latency divided by total number of read memory transactions over the $T_r = 1$ ms regulation interval.

The average memory latency of *disparity* under the same experimental setup as in Figure 8a is shown in Figure 11. However, instead of presenting the average memory latency over 100 runs, we present the WCET case: where the observed execution time is the highest.

It can be observed that the average memory latency for the 1st target execution time, with an observed WCET of 3714 ms, is around 70 clock cycles. For the 2nd target with an observed WCET of 4037 ms, the average memory latency is around 90 clock cycles. Consequently, the average memory latency is proportional to the target execution time $\bar{E}$.

As the target execution time for the 2nd target is more relaxed relative to the 1st target, the overall percentage of regulation that is enforced on the NRT cores decreases from 75% to 50% as seen in Figure 11. The percentage of regulation is calculated by dividing the total number of regulation intervals in which the NRT cores are suspended by the total number of regulation intervals in the experiment. Hence, the percent regulation that is enforced on the NRT cores is inversely proportional to the target execution time $\bar{E}$.

It is worthwhile to note that traditional DRAM bandwidth management-based regulation mechanisms [5, 23, 41] tend to bring the actual execution time as close as possible to the isolation scenario. However, our approach allows for the actual execution time to be anywhere between the execution time in contention to isolation.

7.7 Comparison with DRAM bandwidth-based regulation

To demonstrate that distribution-driven regulation is more beneficial than traditional DRAM bandwidth-based regulation mechanisms, we compare the slowdown ratio experienced by the applications running under the following scenarios (1) unregulated execution, in which the applications are running in parallel on their respective cores without any regulation

Table 3 Slowdown Ratio of benchmarks in contention without regulation and with different regulation mechanisms.

RT Core			**NRT Cores**		
disparity			*MemBomb on each NRT Core*		
Unregulated	MemGuard	Distribution-Driven	Unregulated	MemGuard	Distribution-Driven
1.28	1.03	1.03	3.79	16.67	7.05
disparity			*MemBomb (HB) on each NRT Core*		
1.25	1.03	1.03	1.41	8.07	3.49

mechanism, (2) a memory bandwidth management-based regulation (MemGuard [5])[1], and (3) distribution-driven regulation. We define the slowdown ratio of an application as *the ratio of execution time under contention to the execution time in isolation.*

We use the latest implementation of MemGuard [5] that regulates LLC write-backs in addition to LLC misses, ported to the partitioning hypervisor and configured for static bandwidth reservation. The key parameter used by MemGuard is the guaranteed (worst-case) bandwidth, which is approximately 960 MB/s for our evaluation platform based on the work in [24]. We allocated half of the said bandwidth for the application running in the RT core, and the remaining is distributed equally among the three applications running in the NRT cores.

Once the configurations for MemGuard have been selected, the parameters of the distribution-driven regulator (target execution time $\bar{E}$ and acceptable error α) are selected in such a way that the actual execution time E_{reg} for the application running on the RT core is the same under MemGuard and distribution-driven regulation. This allows for a fair comparison of slowdown ratios for applications running on NRT cores while keeping the same slowdown ratios for the application running on the RT core.

We conducted the evaluation with two different sets of applications. In the first set of applications, *disparity* is running on the designated RT core while synthetic *MemBomb* applications are running on the three NRT cores. In the second set of applications, only the *MemBomb* is modified to perform memory write operations for half of its duration periodically. We refer to this modified *MemBomb* application as *MemBomb Half Blast (HB).*

Table 3 shows the slowdown ratios for different run settings compared to the execution times in isolation. We compare (1) unregulated runs in which the applications are executed concurrently in the respective cores with no regulation mechanism in place to (2) the proposed distribution-driven regulator and to (3) regulation done using MemGuard.

As expected, both regulation approaches achieve the same slowdown ratios of 1.03 for *disparity*. However, with MemGuard, both sets of applications running on the NRT cores suffer the highest slowdowns of 16.67 and 8.07, respectively. By contrast, the distribution-driven regulator is able to improve the slowdown ratio of the NRT applications on average by 2.2× compared to MemGuard.

8 Conclusion and Future Work

In this work, we presented a novel distribution-based regulation mechanism that enforces a *timeliness objective* formulated as a constraint on the probability of meeting any execution time target, which can be anywhere between the execution time in isolation and contention

[1] Comparison against a more recent work [23] is not possible due to the unavailability of memory utilization metric in our evaluation platform, which is necessary for the latter work.

scenario. The *timeliness objective* is met by directly controlling the distribution of total memory latency via regulation, which eventually impacts the distribution of the observed execution time.

We implemented our solution inside the Jailhouse-RT hypervisor [15] and deployed it on a COTS platform (Xilinx Ultrascale+ MPSoC) to demonstrate its effectiveness in meeting the *timeliness objective* for time-sensitive RT applications. Our approach can also be extended to handle multiple RT cores by assigning ranks to the RT cores based on their criticality level. The level of criticality then determines the order of suspension of the cores. If the observed CDF is below the reference CDF, the NRT cores are suspended first, followed by the RT core with the lowest criticality level, and so on, until the observed CDF no longer remains below the reference CDF. This is not immediately feasible with the same PMU due to the limited number of AXI ID filtering blocks. However, APM blocks can be instantiated on the on-chip FPGA, and memory traffic can be observed through-FPGA instead.

References

1 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

2 Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable SDRAM memory controller. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

3 ARM. An introduction to AMBA AXI. `https://developer.arm.com/documentation/102202`.

4 ARM. ARM® Cortex®-A53 MPCore Processor – Technical Reference Manual. `https://static.docs.arm.com/ddi0500/f/DDI0500.pdf`.

5 Michael Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

6 Michael Bechtel and Heechul Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore ARM Processors. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2023)*, San Antonio, Texas, USA, May 2023.

7 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.

8 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the sources of unpredictability in COTS-based multicore systems. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

9 Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing Intel's Memory Bandwidth Allocation for resource limitation in real-time systems. In *IEEE International Symposium On Real-Time Distributed Computing (ISORC)*, 2022.

10 H. Fischer. *A History of the Central Limit Theorem: From Classical to Modern Probability Theory*. Sources and Studies in the History of Mathematics and Physical Sciences. Springer New York, 2010. URL: `https://books.google.com/books?id=v7kTwafIiPsC`.

11 Johannes Freitag and Sascha Uhrig. Closed Loop Controller for Multicore Real-Time Systems. In *Architecture of Computing Systems (ARCS)*, 2018.

12 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling Compositionality for Multicore Timing Analysis. In *International Conference on Real-Time Networks and Systems (RTNS)*, RTNS '16, 2016.

13 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

14 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

15 J. Kiszka, V. Sinitsin, H. Schild, and contributors. Jailhouse Hypervisor. URL: `https://github.com/siemens/jailhouse`.

16 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. `doi:10.1109/RTAS.2019.00009`.

17 D.S. Lemons, P. Langevin, and A. Gythiel. *An Introduction to Stochastic Processes in Physics*. Johns Hopkins Paperback. Johns Hopkins University Press, 2002. URL: `https://books.google.com/books?id=Uw6YDkd_CXcC`.

18 Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CSUR*, 52(3):1–38, 2019.

19 Xiaosheng Mu, Luciano Pomatto, Philipp Strack, and Omer Tamuz. From blackwell dominance in large samples to renyi divergences and back again, 2019. `doi:10.48550/arXiv.1906.02838`.

20 Rodolfo Pellizzoni and Heechul Yun. Memory Servers for Multicore Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

21 Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high - performance platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021.

22 Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle: Cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309, 2020. `doi:10.1109/RTAS48715.2020.00006`.

23 Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

24 Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A Real-Time Virtio-Based Framework for Predictable Inter-VM Communication. In *IEEE Real-Time Systems Symposium (RTSS)*, 2021.

25 Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

26 Moshe Shaked and J. George Shanthikumar, editors. *Stochastic Orders*. Springer New York, 2007. `doi:10.1007/978-0-387-34675-5`.

27 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

28 Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A Closer Look at Intel Resource Director Technology (RDT). In *International Conference on Real-Time Networks and Systems (RTNS)*, 2022.

29 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and cpus in qos-enabled platforms. *Real-Time Syst.*, 58(3):235–274, September 2022. `doi:10.1007/s11241-022-09382-x`.

30 Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. DAPHNE – An automotive benchmark suite for parallel programming models on embedded heterogeneous platforms: work-in-progress. In *International Conference on Embedded Software Companion (EMSOFT)*, 2019.

31 Ashley Stevens. Quality of Service (QoS) in ARM Systems: An Overview. In *ARM White paper*, 2014.

32 Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, and Michael Boyle. Recommendation for the Entropy Sources Used for Random Bit Generation, 2018. URL: `https://csrc.nist.gov/publications/detail/sp/800-90b/final`.

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

34 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing Isolation Challenges of Non-Blocking Caches for Multicore Real-Time Systems. *ACM Real-Time Systems*, 53(5):673–708, 2017.

35 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 *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

36 Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

37 Xilinx. AXI Performance Monitor LogiCORE IP Product Guide (PG037). `https://docs.xilinx.com/v/u/en-US/pg172-ila`.

38 Xilinx. AXI Traffic Generator v3.0 LogiCORE IP Product Guide (PG125). `https://docs.xilinx.com/v/u/en-US/pg125-axi-traffic-gen`.

39 Xilinx. Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide (PG172). `https://docs.xilinx.com/v/u/en-US/pg037_axi_perf_mon`.

40 Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`.

41 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 (TC)*, 65(2):562–576, 2016.

42 Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Transactions on Computers (TC)*, 66(7):1247–1252, 2017.

43 Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *ACM European Conference on Computer Systems*, EuroSys '09, 2009.

44 Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Transactions on Computers (TC)*, 72(1):168–182, 2023.

45 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.

46 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2023)*, San Antonio, Texas, USA, May 2023.