

Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs


Ahsan Saeed ✉ 
Robert Bosch GmbH, Germany

Denis Hoornaert ✉ 
Technical University of Munich, Germany

Dakshina Dasari ✉
Robert Bosch GmbH, Germany

Dirk Ziegenbein ✉
Robert Bosch GmbH, Germany

Daniel Mueller-Gritschneider ✉ 
Technical University of Munich, Germany

Ulf Schlichtmann ✉ 
Technical University of Munich, Germany

Andreas Gerstlauer ✉ 
The University of Texas at Austin, U.S.A.

Renato Mancuso ✉ 
Boston University, U.S.A.

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 stochastic 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.

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



© Ahsan Saeed and Denis Hoornaert and Dakshina Dasari and Dirk Ziegenbein and Daniel Mueller-Gritschneider and Ulf Schlichtmann and Andreas Gerstlauer and Renato Mancuso; licensed under Creative Commons License CC-BY 4.0

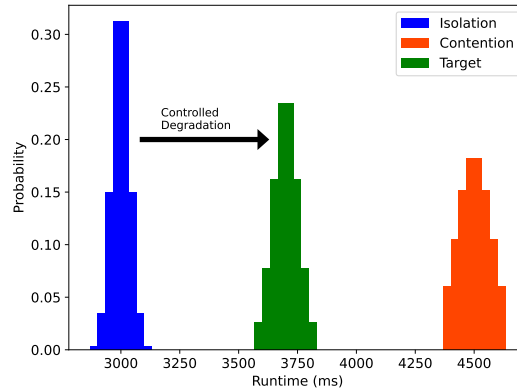
35th Euromicro Conference on Real-Time Systems (ECRTS 2023).

Editor: Alessandro V. Papadopoulos; Article No. 4; pp. 4:1–4:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **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).

42 shared resources like the main memory, the cache and the interconnect, which makes it
 43 non-trivial to bound the execution time of RT applications running on these MPSoCs. This
 44 is because when two or more applications are executed in parallel on different cores, which
 45 we refer to as the contention scenario, the interaction between them on shared hardware
 46 resources can lead to unforeseen and unpredictable delays [8, 34, 36]. It is well known that
 47 memory contention is a key source for performance degradation [7], and practitioners across
 48 the industry and academia are looking for solutions that facilitate temporal isolation between
 49 applications while using COTS platforms.

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

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

71 In this paper, we propose a distribution-driven regulation approach, whose goal is to
 72 achieve a *timeliness objective* formulated as a constraint on the probability of meeting a
 73 certain target execution time. This definition allows us to unite WCET-like constraints and
 74 high-percentile latency constraints typical of real-time cloud systems (tail latency). The

75 basic premise of our approach stems from the observation that the latency distribution
76 of memory transactions of an application under contention gets skewed compared to the
77 execution in isolation. Therefore, it is possible to precisely influence the overall application
78 execution time so long as we can (1) characterize this distribution and (2) affect its shape
79 via regulation. With this basic principle, we first theoretically compute the reference CDF
80 from the distribution of the per-request memory latency for a given target execution time.
81 Then, we enforce first-order stochastic dominance by periodically checking that the CDF
82 of the observed memory latency distribution of the RT application (obtained by sampling at
83 the PMU) stays above the reference CDF of the per-request memory latency. In case this
84 condition is violated, the NRT cores are suspended till the condition of first-order stochastic
85 dominance holds again. If the reference per-request memory latency first-order stochastically
86 dominates the observed latency, then it follows that the overall execution time random
87 variable is dominated by the reference execution time random variable. Consequently, the
88 observed execution time achieves the *timeliness objective*.

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

94 With this work, we make the following contributions:

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

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

116 **2 Related Work**

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

118 The first category includes techniques that essentially employ *memory bandwidth manage-*
119 *ment-based regulation*. In this category of approaches, the effects of memory contention are
120 statically regulated by controlling the outgoing memory bandwidth from each core as in

121 MemGuard [5, 41, 42], or by directly measuring the utilization at the memory controller [23]
 122 and then based on the observed utilization, dynamically regulating the outgoing memory
 123 bandwidth from each of the cores. In these approaches, the designer has to experimentally
 124 derive the correct system parameters, and furthermore, there are no formal techniques to
 125 guarantee the impact of such a regulation on the execution time of the application.

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

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

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

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

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

161 **3 System Model and Assumptions**

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

165 **A1: Multicore Platform Topology.** We assume a system comprised of m application
 166 CPUs Π_1, \dots, Π_m . For simplicity, we assume that the high-criticality workload is only

167 deployed on CPU Π_1 , which can be considered the *real-time core*. The memory hierarchy
168 comprises zero or more levels of cache. Cache misses caused by `load` or `store` instructions
169 at the last-level cache (LLC) cause read/write memory requests to be initiated towards a
170 single shared main memory subsystem via a single shared bus. Note that we distinguish
171 between memory instructions (`load/store`) and the resulting traffic that they might cause
172 in terms of read (and possibly write) requests to the underlying main memory subsystem.

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

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

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

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

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

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

■ **Table 1** Summary of notation used

| Symbols | Descriptions | Symbols | Descriptions |
|-----------------|--|----------------------|---|
| E_{isol} | Total execution time in isolation | \bar{l}_{σ^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}_{μ} | Mean latency of read memory trans. reference | | |

215 commercial platforms in Section 6.

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

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

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

241 4 Distribution-Driven Regulation

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

244 **Regulation Goal.** Unlike the related literature surveyed in Section 2, our goal is to achieve
 245 a *timeliness objective* formulated as a constraint on the probability of meeting a certain
 246 execution time target \bar{E} . Formally, given an execution time target \bar{E} and an acceptable error
 247 $\alpha \in [0, 1]$, the goal of regulation can be written as

$$248 \quad P(E_{reg} \leq \bar{E}) \geq 1 - \alpha, \quad (1)$$

249 where E_{reg} is the actual execution time observed under regulation and (possibly) in the

250 presence of main memory contention for the application under analysis. When α is such that
 251 $\alpha \rightarrow 0$, then \bar{E} represents a worst-case execution time (WCET) constraint. Note however
 252 that the timeliness constraint formulation in Eq. 1 is more generic. For instance, setting
 253 $\alpha = 0.01$ expresses a 99th-percentile tail latency requirement on E_{reg} .

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

$$257 \quad P(C + L \leq \bar{E}) = P\left(\sum_{i=1}^N l_i \leq \bar{E} - C\right) \geq 1 - \alpha. \quad (2)$$

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

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

$$265 \quad Z = \frac{\sum_{i=1}^N l_i - Nl_\mu}{\sqrt{Nl_{\sigma^2}}} = \frac{L - Nl_\mu}{\sqrt{Nl_{\sigma^2}}} \sim \mathcal{N}(0, 1), \quad (3)$$

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

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

$$275 \quad P(L \leq \bar{E} - C) = \Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \geq 1 - \alpha. \quad (4)$$

276 So far, we have treated l_μ and l_{σ^2} as unknown values. The insight at this point is that,
 277 when regulation is performed (by pausing/resuming the activity of interfering cores), we can
 278 exert direct control over the underlying distribution of $L = \sum_{i=1}^N l_i$ and thus over the value
 279 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
 280 is enough to identify two values \bar{l}_μ and \bar{l}_{σ^2} such that the following inequality holds for every
 281 value of $\bar{E} \in \mathbb{R}^+$:

$$282 \quad \Phi\left(\frac{(\bar{E} - C) - Nl_\mu}{\sqrt{Nl_{\sigma^2}}}\right) \geq \Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \geq 1 - \alpha. \quad (5)$$

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

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

289 is applied such that

$$290 \quad \forall t \in \mathbb{R}^+, F_l(t) \geq \Phi\left(\frac{(\bar{E} - C) - \bar{l}_\mu}{\sqrt{\bar{l}_{\sigma^2}}}\right) = \bar{F}_l(t), \quad (6)$$

291 then we have two properties. The first, is that $\bar{F}_l(t)$ is the CDF of a random variable
 292 $l_i^{norm} \sim \mathcal{N}(\bar{l}_\mu, \bar{l}_{\sigma^2})$. The second is that l_i^{norm} is said to first-order stochastically dominate
 293 l_i [26]. Indeed, Eq. 6 is one possible definition of first-order stochastic dominance, also
 294 indicated with the notation $l_i^{norm} \geq_1 l_i$.

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

$$300 \quad \sum_{i=1}^n X_i \geq_k \sum_{i=1}^n Y_i \implies \sum_{i=1}^n X_i \geq_1 \sum_{i=1}^n Y_i. \quad (7)$$

301 Next, we note that from Eq. 7 and 6 it immediately follows that $\sum_{i=1}^N l_i^{norm} \geq_1 \sum_{i=1}^N l_i$.
 302 Moreover, by leveraging the properties of the normal distribution [17], we know that
 303 $\sum_{i=1}^N l_i^{norm} \sim \mathcal{N}(N\bar{l}_\mu, N\bar{l}_{\sigma^2})$. This brings us to the final step. That is, the random variable
 304 L under regulation is first-order stochastically dominated by a normal distribution of mean
 305 $N\bar{l}_\mu$ and variance $N\bar{l}_{\sigma^2}$. This means that, as long as Eq. 6 is ensured via regulation, Eq. 5
 306 holds.

307 **Final Formulation.** Putting everything together, we have the following workflow. First,
 308 given the target \bar{E} and α , numerically compute \bar{l}_μ and \bar{l}_{σ^2} such that

$$309 \quad \Phi\left(\frac{(\bar{E} - C) - N\bar{l}_\mu}{\sqrt{N\bar{l}_{\sigma^2}}}\right) \geq 1 - \alpha \quad (8)$$

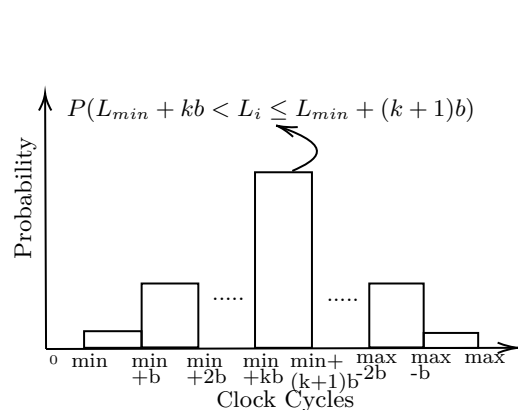
310 holds. Second, use the same values of \bar{l}_μ and \bar{l}_{σ^2} to construct the target per-request
 311 latency CDF \bar{F}_l as described in Eq. 6. Next, at runtime, observe the CDF of l_i , namely
 312 F_l , and pause/resume (regulate) the activity of the non-real-time CPUs to ensure that
 313 $\forall t \in \mathbb{R}^+, F_l(t) \geq \bar{F}_l(t)$. So long as this inequality holds, it also holds that $P(C + L \leq \bar{E}) =$
 314 $P(L \leq \bar{E} - C) \geq 1 - \alpha$ because Eq. 5 holds.

315 4.1 Discrete-domain Formulation

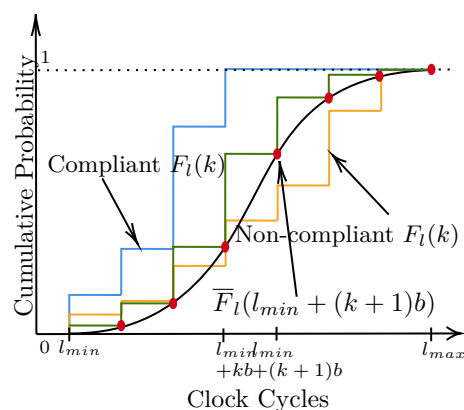
316 The results derived so far in Section 4 assume that we are able to snapshot online a *continuous*
 317 distribution of read latency accesses. This is practically impossible with realistic hardware.
 318 In this subsection, we relax precisely this requirement.

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

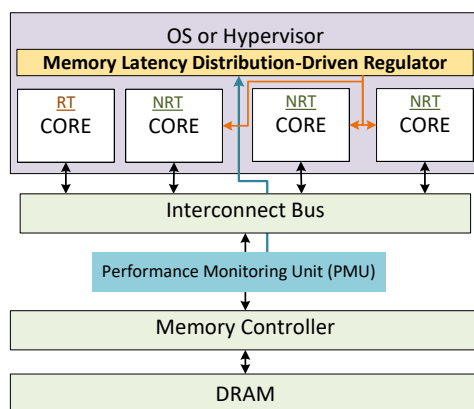
324 When queried, the PMU reports the number of read transactions completed by Π_1 whose
 325 latency fell in each of the K bins. Assume that this number is cumulative since the time at
 326 which the application was launched—if it is reset after a snapshot, e.g. to prevent overflows
 327 in the counters, then it can be accumulated in software at each snapshot. In software, divide



■ **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.

328 the number of transactions in each bin (i.e. the height of the bin) by the total number of
 329 transactions in the entire snapshot. The result is a valid observed Probability Mass Function
 330 (PMF) $f_l(k)$ for the read request latency l_i for the generic request i . Figure 2 provides a
 331 visual representation of the PMF. In other words, the height of each bin provides the value of
 332 $f_l(k) = P(l_{min} + kb \leq l_i < l_{min} + (k + 1)b)$. From the acquired PMF, it is easy to compute
 333 the corresponding observed CDF as

$$334 \quad F_l(k) = \sum_{j=0}^k f_l(j) = P(l_i < l_{min} + (k + 1)b). \quad (9)$$

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

$$338 \quad \forall k \in \{0, \dots, K - 1\}, F_l(k) \geq \bar{F}_l(l_{min} + (k + 1)b). \quad (10)$$

339 This condition is visually depicted in Figure 3. Indeed, if the condition expressed in Eq. 10
 340 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
 341 the case for the blue curve in Figure 3. Conversely, if for some k Eq. 10 does not hold, the

■ **Algorithm 1** Memory Latency Distribution-Driven Regulator

```

input : number of latency bins  $K$ , reference CDF  $\bar{F}_k \forall k \in \{0 \dots K - 1\}$ 
1 foreach regulation interval  $r$  do
2   foreach latency bin  $k \in \{0 \dots K - 1\}$  do
3     | Sample the height of latency bin  $l_{k,r}$ 
4     |  $\gamma_{k,r} = \gamma_{k,r-1} + l_{k,r}$ 
5   end
6   foreach latency bin  $k \in \{0 \dots K - 1\}$  do
7     |  $f_{k,r} = \frac{\gamma_{k,r}}{\sum_{k=0}^{K-1} \gamma_{k,r}}$  ▷ Normalize bins to obtain PMF
8     |  $F_{k,r} = \sum_{m=0}^k f_{m,r}$  ▷ Construct observed CDF
9   end
10  if  $F_{0,r} < \bar{F}_0 \vee \dots \vee F_{K-1,r} < \bar{F}_{K-1}$  then
11    | suspend all NRT cores
12  else
13    | resume all NRT cores
14  end
15   $r = r + 1$ 
16 end

```

342 non-real-time CPUs must be paused—regulation must be triggered. This is the case for the
 343 orange line in Figure 3. The implicit assumption, which we validate in Section 7.3, is that
 344 pausing the interfering CPUs allows to *shift* the observed $F_l(k)$ in subsequent snapshots.

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

349 5 System Overview

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

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

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

368 The proposed mechanism can be implemented on any platform on which we are able to

369 measure (1) memory latency distribution and (2) filter the memory transaction on a per core
370 basis.

371 5.1 Memory Latency Distribution-Driven Regulator Algorithm

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

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

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

395 6 Implementation

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

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

408 6.1 AXI Performance Monitor (APM)

409 We sample the memory latency in the Xilinx Ultrascale+ MPSoC [40] using the AXI
410 Performance Monitor (APM) hardware module. The APM measures the key performance
411 metrics like the amount of read/write memory transactions, min/max/total latency, and

412 other performance metrics for the AMBA AXI [3] in a system. The APMs implemented on
 413 Xilinx Ultrascale+ MPSoC [40] are based on the Xilinx AXI Performance Monitor available
 414 as a LogiCORE IP [37].

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

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

425 We configured 8 *Metric Selector* registers in conjunction with 8 *Range* registers to monitor
 426 read memory latency (as defined in Section 3 *A6: Measurable Read Latency Distribution*) with
 427 respectively low and high ranges of 0-40, 41-80, 81-120, 121-160, 161-200, 201-240, 241-280,
 428 and 281-2000 clock cycles. The rationale behind the selection of these ranges is discussed in
 429 Section 7.4. These 8 performance metrics provide the number of read memory transactions
 430 that fall within the given read memory latency limits, referred to as bins. Furthermore,
 431 2 *Metric Selector* registers are configured to report the total number of read transactions
 432 and total read latency. The total number of read transactions is N , as used throughout
 433 the mathematical formalization in Section 3. Additionally, we verify that the total number
 434 of read transactions and the sum of all bins are always the same. This ensures that no
 435 memory transaction *escapes* the bins. The global-clock counter is used as the reference for
 436 all the calculations in this paper. The included hardware counters can be set and read via a
 437 memory-mapped interface.

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

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

447 **7 Validation and Evaluation**

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

452 **7.1 Experimental Setup**

453 We evaluate our approach on the Xilinx Ultrascale+ Multi-Processor System-on-Chip
 454 (MPSoC) ZCU102 [40] as introduced in Section 6. A combination of real-world [35], [30], and
 455 synthetic [33] benchmarks are used to evaluate the proposed approach. For our real-world
 456 benchmarks, we use a subset of the benchmarks in the San Diego Vision Benchmark Suite

■ **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 |

(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.

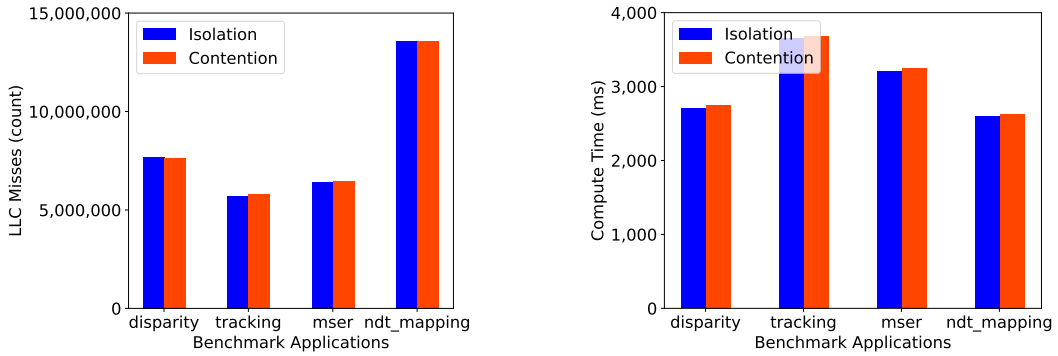
Unless otherwise stated, all experiments refer to the *isolation scenario* or simply *isolation* in which the disparity application is running on the designated RT core with no other applications running in parallel. In contrast, a *contention scenario* or simply *contention* happens when the same *disparity* 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.

Performing permutation testing requires measuring the memory latency of individual



(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.

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

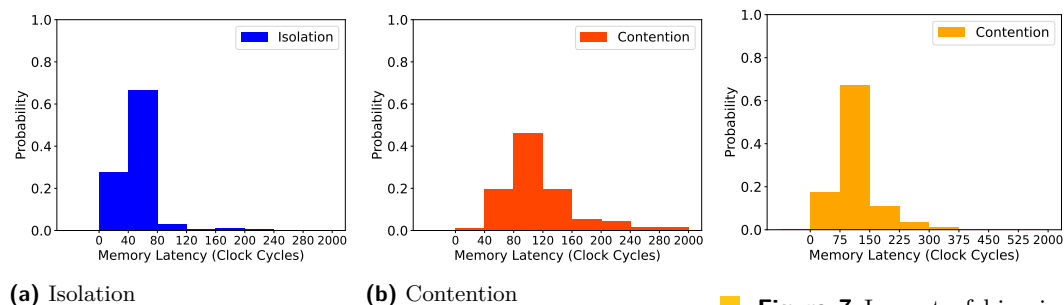
491 The experiment is divided into four successive steps: (1) generate memory traffic, (2)
 492 capture the activity at the AXI level, (3) measure and compile each transaction’s response
 493 time, and (4) perform a set of permutation tests.

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

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

512 Table 2 shows the results of the first 10 permutation tests performed on the two FPGA
 513 designs, on the top and bottom, respectively. For synthetic benchmarks, the number of
 514 passed tests increases as randomness in the pattern, and ITG is introduced. Therefore, for
 515 ATG with random memory access pattern and random ITG has the highest tests pass of
 516 100%, whereas sequential memory access pattern with fixed ITG has the lowest test pass of
 517 60%. Hence, the percentage of tests pass increases as access pattern and ITG randomness
 518 grow.

519 For real-world benchmarks, 30 snapshots of memory traffic are captured. Since applications



■ **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.

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

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

528 7.3 Validation of Other Key System Assumptions

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

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

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

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

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

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

553 When multiple contending *MemBomb* applications are running in parallel, the *disparity*

554 benchmark experiences a significant increase in memory latency, resulting in a shift of the
 555 memory latency distribution to the right (higher memory latency bins), as seen in Figure 6b.
 556 Under contention, the majority of individual memory read transactions have latency in the
 557 range of 41 to 160 clock cycles.

558 7.4 Configuration Parameters

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

562 7.4.1 Regulation Interval

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

568 7.4.2 Total Bins

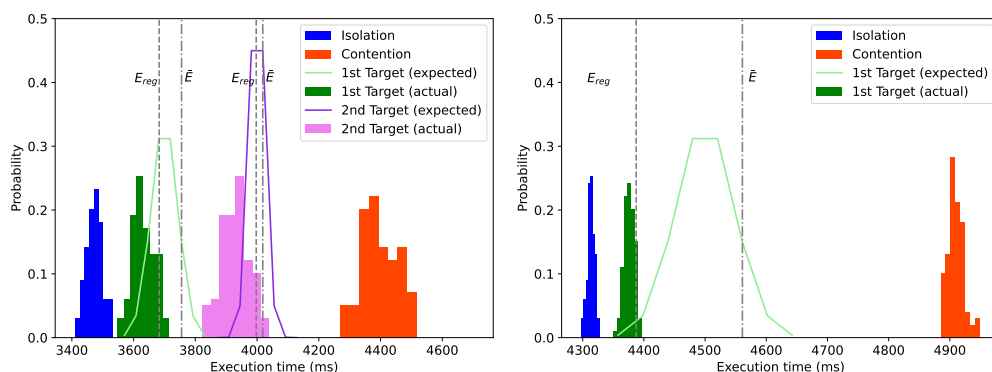
569 The number of bins defines the quantization that can be used to approximate the memory
 570 latency distribution. The PMU present in our evaluation platform offers 10 hardware counters
 571 as described in Section 6.1, which can be accordingly used to set 10 latency bins. However,
 572 we only dedicate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

576 7.4.3 Bin Size

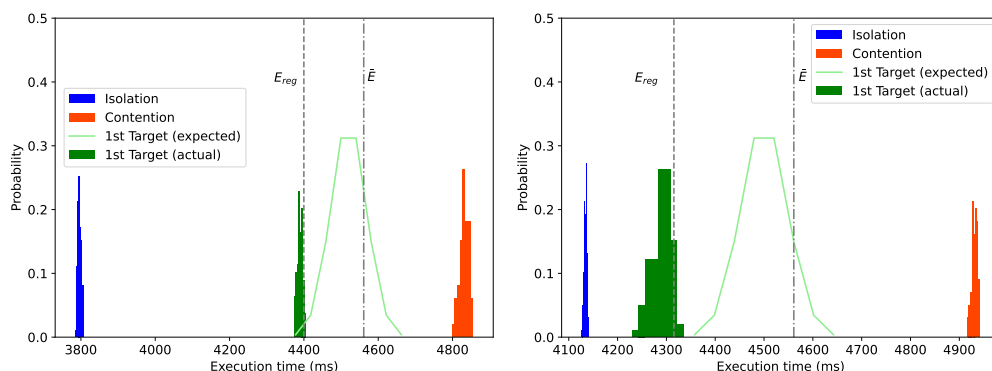
577 The bin size of the memory latency distribution needs to be chosen in such a way that all
 578 possible individual read memory latencies can be covered while ensuring that distribution
 579 shifts can be effectively captured. Simultaneously, one must ensure that the bins are equally
 580 spaced and without discontinuities to provide a well-formed distribution snapshot when
 581 sampled. To determine the appropriate bin size, the APM is initially configured to measure
 582 the minimum and maximum memory latency values during a set of application runs. We
 583 observed a minimum read latency of 38 clock cycles and a maximum of approximately
 584 600 clock cycles. Based on these values, we fix 40 clock cycles as the bin size. We also
 585 experimented with a larger bin size of 75 clock cycles with the same setup as shown in
 586 Figure 6b, which resulted in nearly empty bins with memory latency values greater than 375
 587 clock cycles, as seen in Figure 7. We set the upper limit of the last bin to 2000 clock cycles in
 588 order to capture all conceivable memory latencies that a memory transaction may encounter.

589 7.5 Effectiveness of the Approach

590 The objective of this experiment is to show that, given \bar{E} and α , Eq. 1 holds. Figure 8
 591 summarizes the execution time distribution of applications during 100 runs and compares the
 592 target execution time \bar{E} against the actual execution time E_{reg} . As a point of reference, the
 593 execution time distribution in isolation (blue) and contention (orange) are also provided. The



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



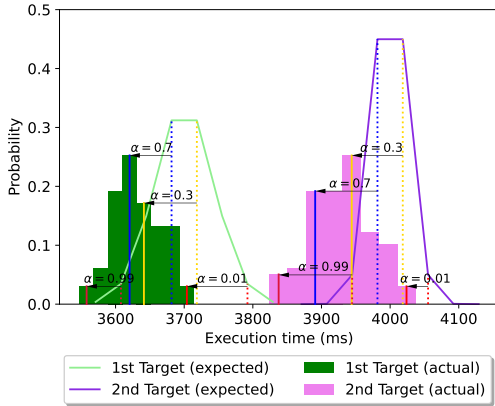
(c) *mser* on RT core and *MemBomb* on NRT cores. (d) *ndt_mapping* on RT core and *MemBomb* on NRT cores.

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

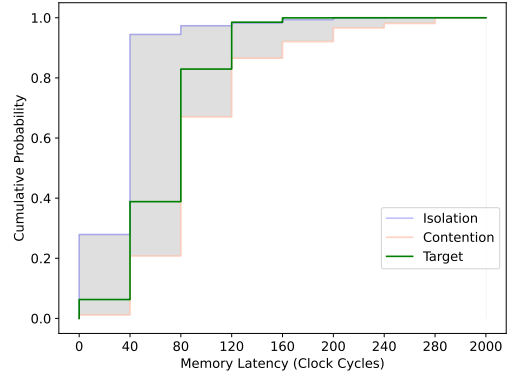
594 expected discretized execution time distribution of the target execution time is theoretically
 595 computed and is depicted as a discretized bell curve, whereas the actual execution time
 596 distribution is experimentally evaluated and depicted by a bar plot.

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

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



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



■ **Figure 10** CDF for *disparity* on RT core.

613 goal defined in Equation (1).

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

621 Finally, we illustrate the CDF of read memory latency observed by the *disparity* application
 622 in isolation and under contention, as well as the enforced reference CDF. The reference
 623 CDF \bar{F} used in Figure 8 for the 1st target execution time \bar{E} of 3755 ms is highlighted with
 624 green lines in Figure 10. The CDF in isolation (blue lines) and contention (orange lines) are
 625 computed using the same PMFs previously shown in Figure 6. It can be noted that the $\bar{F}(k)$
 626 computed for each bin k lies between the envelope defined by the CDFs measured in isolation
 627 (upper bound) and under contention (lower bound). These $\bar{F}(k)$ values are subsequently used
 628 by the memory latency distribution-driven regulator (Algorithm 1) to achieve an execution
 629 time E_{reg} that meets the timeliness objective.

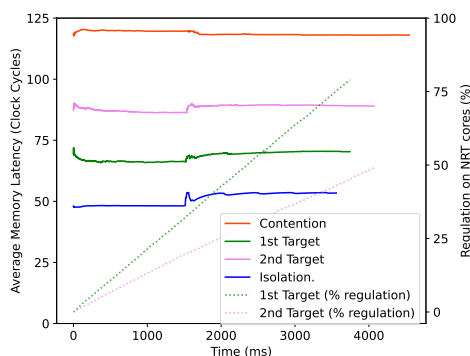
630 7.6 Impact of Regulation on the Average Memory Latency

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

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

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

642 As the target execution time for the 2nd target is more relaxed relative to the 1st target,
 643 the overall percentage of regulation that is enforced on the NRT cores decreases from 75% to



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

644 50% as seen in Figure 11. The percentage of regulation is calculated by dividing the total
 645 number of regulation intervals in which the NRT cores are suspended by the total number of
 646 regulation intervals in the experiment. Hence, the percent regulation that is enforced on the
 647 NRT cores is inversely proportional to the target execution time \bar{E} .

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

652 7.7 Comparison with DRAM bandwidth-based regulation

653 To demonstrate that distribution-driven regulation is more beneficial than traditional DRAM
 654 bandwidth-based regulation mechanisms, we compare the slowdown ratio experienced by
 655 the applications running under the following scenarios (1) unregulated execution, in which
 656 the applications are running in parallel on their respective cores without any regulation
 657 mechanism, (2) a memory bandwidth management-based regulation (MemGuard [5])¹, and
 658 (3) distribution-driven regulation. We define the slowdown ratio of an application as *the*
 659 *ratio of execution time under contention to the execution time in isolation*.

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

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

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.

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

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

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

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

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

687 8 Conclusion and Future Work

688 In this work, we presented a novel distribution-based regulation mechanism that enforces a
 689 *timeliness objective* formulated as a constraint on the probability of meeting any execution
 690 time target, which can be anywhere between the execution time in isolation and contention
 691 scenario. The *timeliness objective* is met by directly controlling the distribution of total
 692 memory latency via regulation, which eventually impacts the distribution of the observed
 693 execution time.

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

704 — References —

- 705 1 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic
 706 Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *IEEE Real-Time Systems
 707 Symposium (RTSS)*, 2018.

- 708 2 Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable SDRAM
709 memory controller. In *IEEE/ACM/IFIP International Conference on Hardware/Software*
710 *Codesign and System Synthesis (CODES+ISSS)*, 2007.
- 711 3 ARM. An introduction to AMBA AXI. <https://developer.arm.com/documentation/102202>.
- 712 4 ARM. ARM® Cortex®-A53 MPCore Processor - Technical Reference Manual. <https://static.docs.arm.com/ddi0500/f/DDI0500.pdf>.
- 713 5 Michael Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore:
714 Analysis and Prevention. In *IEEE Real-Time and Embedded Technology and Applications*
715 *Symposium (RTAS)*, 2019.
- 716 6 Michael Bechtel and Heechul Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore
717 ARM Processors. In *29th IEEE Real-Time and Embedded Technology and Applications*
718 *Symposium (RTAS 2023)*, San Antonio, Texas, USA, May 2023.
- 719 7 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference charac-
720 terization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE*
721 *International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.
- 722 8 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters.
723 Identifying the sources of unpredictability in COTS-based multicore systems. In *IEEE*
724 *International Symposium on Industrial Embedded Systems (SIES)*, 2013.
- 725 9 Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing Intel’s Memory
726 Bandwidth Allocation for resource limitation in real-time systems. In *IEEE International*
727 *Symposium On Real-Time Distributed Computing (ISORC)*, 2022.
- 728 10 H. Fischer. *A History of the Central Limit Theorem: From Classical to Modern Probability*
729 *Theory*. Sources and Studies in the History of Mathematics and Physical Sciences. Springer
730 New York, 2010. URL: <https://books.google.com/books?id=v7kTwafIiPsC>.
- 731 11 Johannes Freitag and Sascha Uhrig. Closed Loop Controller for Multicore Real-Time Systems.
732 In *Architecture of Computing Systems (ARCS)*, 2018.
- 733 12 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling Compositionality for Multicore
734 Timing Analysis. In *International Conference on Real-Time Networks and Systems (RTNS)*,
735 RTNS ’16, 2016.
- 736 13 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infra-
737 structure for Multi-Core Systems with Re-Programmable Logic. In *Euromicro Conference on*
738 *Real-Time Systems (ECRTS)*, 2021.
- 739 14 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Rangunathan
740 Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE*
741 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
- 742 15 J. Kiszka, V. Sinitsin, H. Schild, and contributors. Jailhouse Hypervisor. URL: <https://github.com/siemens/jailhouse>.
- 743 16 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko
744 Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded
745 systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium*
746 *(RTAS)*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.
- 747 17 D.S. Lemons, P. Langevin, and A. Gythiel. *An Introduction to Stochastic Processes in*
748 *Physics*. Johns Hopkins Paperback. Johns Hopkins University Press, 2002. URL: https://books.google.com/books?id=Uw6YDkd_CXcC.
- 749 18 Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I.
750 Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM*
751 *Computing Surveys (CSUR)*, 52(3):1–38, 2019.
- 752 19 Xiaosheng Mu, Luciano Pomatto, Philipp Strack, and Omer Tamuz. From blackwell dominance
753 in large samples to renyi divergences and back again, 2019. URL: [https://arxiv.org/abs/](https://arxiv.org/abs/1906.02838v3)
754 [1906.02838v3](https://arxiv.org/abs/1906.02838v3), doi:10.48550/ARXIV.1906.02838.
- 755 20 Rodolfo Pellizzoni and Heechul Yun. Memory Servers for Multicore Systems. In *IEEE*
756 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.
- 757
758
759

- 760 **21** Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk
761 Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive
762 high - performance platforms. In *Design, Automation Test in Europe Conference Exhibition*
763 *(DATE)*, 2021.
- 764 **22** Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle:
765 Cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications*
766 *Symposium (RTAS)*, pages 296–309, 2020. doi:10.1109/RTAS48715.2020.00006.
- 767 **23** Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael
768 Pressler, Arne Hamann, Daniel Mueller-Gritschneider, Andreas Gerstlauer, and Ulf Schlicht-
769 mann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation
770 in Multi-Cores. In *IEEE Real-Time and Embedded Technology and Applications Symposium*
771 *(RTAS)*, 2022.
- 772 **24** Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alex-
773 ander Zuepke, and Marco Caccamo. A Real-Time Virtio-Based Framework for Predictable
774 Inter-VM Communication. In *IEEE Real-Time Systems Symposium (RTSS)*, 2021.
- 775 **25** Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J.
776 Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+
777 MPSoC. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.
- 778 **26** Moshe Shaked and J. George Shanthikumar, editors. *Stochastic Orders*. Springer
779 New York, 2007. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/978-0-387-34675-5)
780 [978-0-387-34675-5](https://doi.org/10.1007/978-0-387-34675-5), doi:10.1007/
978-0-387-34675-5.
- 781 **27** P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: A System-wide Framework for
782 Memory Bandwidth Profiling and Management. In *IEEE Real-Time Systems Symposium*
783 *(RTSS)*, 2020.
- 784 **28** Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A Closer
785 Look at Intel Resource Director Technology (RDT). In *International Conference on Real-Time*
786 *Networks and Systems (RTNS)*, 2022.
- 787 **29** Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory
788 bandwidth management for accelerators and cpus in qos-enabled platforms. *Real-Time Syst.*,
789 58(3):235–274, sep 2022. doi:10.1007/s11241-022-09382-x.
- 790 **30** Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. DAPHNE - An
791 automotive benchmark suite for parallel programming models on embedded heterogeneous
792 platforms: work-in-progress. In *International Conference on Embedded Software Companion*
793 *(EMSOFT)*, 2019.
- 794 **31** Ashley Stevens. Quality of Service (QoS) in ARM Systems: An Overview. In *ARM White*
795 *paper*, 2014.
- 796 **32** Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, and Michael
797 Boyle. Recommendation for the Entropy Sources Used for Random Bit Generation, 2018.
798 URL: <https://csrc.nist.gov/publications/detail/sp/800-90b/final>.
- 799 **33** P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in
800 Multicore Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications*
801 *Symposium (RTAS)*, 2016.
- 802 **34** Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing Isolation Chal-
803 lenges of Non-Blocking Caches for Multicore Real-Time Systems. *ACM Real-Time Systems*,
804 53(5):673–708, 2017.
- 805 **35** S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor.
806 SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on*
807 *Workload Characterization (IISWC)*, 2009.
- 808 **36** Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson.
809 Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms.
810 In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

- 811 37 Xilinx. AXI Performance Monitor LogiCORE IP Product Guide (PG037).
812 <https://docs.xilinx.com/v/u/en-US/pg172-ila>.
- 813 38 Xilinx. AXI Traffic Generator v3.0 LogiCORE IP Product Guide (PG125).
814 <https://docs.xilinx.com/v/u/en-US/pg125-axi-traffic-gen>.
- 815 39 Xilinx. Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide (PG172).
816 https://docs.xilinx.com/v/u/en-US/pg037_axi_perf_mon.
- 817 40 Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit.
818 <https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html>.
- 819 41 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for
820 Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*
821 (*TC*), 65(2):562–576, 2016.
- 822 42 Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. BWLOCK: A Dynamic
823 Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms.
824 *IEEE Transactions on Computers (TC)*, 66(7):1247–1252, 2017.
- 825 43 Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based
826 Multicore Cache Management. In *ACM European Conference on Computer Systems*, EuroSys
827 '09, 2009.
- 828 44 Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm’s MPAM From the
829 Perspective of Time Predictability. *IEEE Transactions on Computers (TC)*, 72(1):168–182,
830 2023.
- 831 45 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and
832 controlling I/O-related memory contention in COTS heterogeneous platforms. *Software:
833 Practice and Experience*, 52(5):1095–1113, 2022.
- 834 46 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso.
835 MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *29th IEEE Real-
836 Time and Embedded Technology and Applications Symposium (RTAS 2023)*, San Antonio,
837 Texas, USA, May 2023.