MemPol: Policing Core Memory Bandwidth from Outside of the Cores

Alexander Zuepke[§] Andrea Bastoni[§] Weifan Chen[†] Marco Caccamo[§] Renato Mancuso[†]
[§] Technical University of Munich, {alex.zuepke|andrea.bastoni|mcaccamo}@tum.de
[†] Boston University, {wfchen|rmancuso}@bu.edu

***Abstract*—In today's multiprocessor systems-on-a-chip (MPSoC), the shared memory subsystem is a known source of temporal interference. The problem causes logically independent cores to affect each other's performance, leading to pessimistic worst-case execution time (WCET) analysis. One of the most practical techniques to mitigate interference is memory regulation via throttling. Traditional regulation schemes rely on a combination of timer and performance counter interrupts to be delivered and processed on the same cores running real-time workload. Unfortunately, to prevent excessive overhead, regulation can only be enforced at a millisecond-scale granularity.**

In this work, we present a novel regulation mechanism from *outside the cores* that monitors performance counters for the application core's activity in main memory at a microsecond scale. The approach is fully transparent to the applications on the cores, and can be implemented using widely available on-chip debug facilities. The presented mechanism also allows more complex composition of metrics to enact load-aware regulation. For instance, it allows redistributing unused bandwidth between cores while keeping the overall memory bandwidth of all cores below a given threshold. We implement our approach on a host of embedded platforms and carry out an in-depth evaluation on the Xilinx Zynq UltraScale+ ZCU102 platform using the SD-VBS.

I. Introduction

Homogeneous multi-core systems became mainstream in the real-time embedded community about a decade ago. From a predictability standpoint, these platforms came with formidable challenges that have been the focus of a host of research works [1]. But in many ways, such systems are already obsolete. Modern embedded multiprocessor systems-on-a-chip (MPSoC) embrace heterogeneity. This is necessary due to the increasing adoption of data-intensive artificial intelligence (AI) algorithms in embedded and safety-critical domains. CPUs, GPUs, TPUs, on-chip programmable logic (FPGA), and smart network interfaces (NICs) are some examples of top-tier processing elements in current-generation MPSoCs. Xilinx's UltraScale+ and Versal [2], [3] or NVIDIA's Jetson AGX Xavier and Orin [4], [5] are among the most recent examples of this trend.

Unfortunately, the explosion in heterogeneity has exacerbated the existing challenges related to the management of shared memory hierarchy resources. One such challenge is quality of service (QoS) driven regulation of main memory bandwidth consumption from heterogeneous processing elements (PE). Software regulation of the memory bandwidth based on monitoring of performance counters (PMC) has received significant attention [6], [7] thanks to its wide applicability to a broad range of MPSoC that are normally equipped with performance counter units (PMU).

PMC-based regulation, however, comes with important compromises. Most prominently, it is inherently CPU-centric, because it relies on the ability to install and process PMC-generated interrupts. Secondly, by design, it does not allow to implement complex regulation policies accounting for both per-PE activity and global system behavior. Worse yet, it is challenging to define complex software regulation policies that account for more than a single performance metric. This contrasts with the wide range of performance metrics exported by modern platforms at multiple levels of their complex memory hierarchy—*e.g.*, at the level of PE [2], [8], interconnect [9], and memory controller [10], [11]. Third, it forces to integrate additional system-level software components at the OS [6] or hypervisor level [10], [12], with the corresponding engineering and performance overheads.

This paper stems from the question: *Can memory bandwidth regulation be enforced following a drastically different approach?* And, ideally, one that can achieve fine-grained regulation, acceptably low overheads, and customizable regulation policies capable of capturing multiple nuances in the performance of complex memory hierarchies.

In light of this goal, we propose *MemPol*: a novel approach for memory bandwidth regulation that targets the aforementioned objectives. By exploiting the heterogeneous computing elements of MPSoCs, *MemPol* adopts a low-overhead, polling-based design that enables *microsecond-scale* memory bandwidth regulation and monitoring. *MemPol* moves away from interrupt-based regulation and relies on debug primitives to control bandwidth consumption with minimum intrusiveness. Furthermore, *MemPol* allows defining complex regulation functions that combine contributions of *multiple* performance counters. Thus, we make the following key contributions:

- A microsecond-scale memory bandwidth *monitor* based on periodic polling of performance counters from "outside" of the cores. *MemPol* does not cause performance degradation of the applications executing on the cores.
- A low-overhead memory bandwidth *regulator* that throttles monitored cores using built-in on-chip debug facilities without causing memory perturbations.
- Per-core memory bandwidth regulation using an *on-off controller* design.
- The possibility to define software regulation profiles with functions based on multiple PMC metrics.

- A combination of per-core (local) regulation and global regulation of all cores to redistribute unused bandwidth between cores, while keeping the overall memory bandwidth below a given threshold.

MemPol's regulation logic can be fully implemented outside of the core-complex. Our regulator enables the unconstrained use of the most powerful cores of a platform for application-related workloads by dedicating *e.g.*, energy-efficient, real-time oriented cores to the management of the regulation logic. Because *MemPol* leverages debug primitives, it can be extended to pause/resume the activity of PEs other than CPUs—albeit our initial prototype is focused on CPU regulation.

As a proof of concept, we implemented *MemPol* on a Xilinx Zynq UltraScale+ ZCU102 [2] platform that features four Arm Cortex-A53 application cores and two Arm Cortex-R5 real-time cores. *MemPol* is deployed on one of the Cortex-R5 cores and regulates the application cores with 6.25 μs granularity. Although questionably suitable for certified environments, we have validated the practical feasibility of our debug-based methodology (see Sec. IV-B) on multiple Arm-based boards such as NXP S32G274 [13], Raspberry Pi 4 [14], and NXP LX2160A [15]. Our evaluation showcases the ability of *MemPol* to enforce complex regulation policies, such as proportional bandwidth redistribution, by monitoring a combination of local and global bandwidth consumption. By instantiating *MemPol* with legacy policies, we also compare its performance overhead with state-of-the-art PMC-based regulation.

The rest of this paper is structured as follows. Sec. II discusses limitations of *MemGuard* designs and proposes alternatives. Sec. III presents the new regulator design, Sec. IV its implementation, and Sec. V its evaluation. Sec. VI discusses related work, and Sec. VII concludes.

II. Background and Motivation

This section summarizes the key aspects of PMC-based regulation—with focus on its most common variant, *MemGuard*—and details the most important limitations of the approach that constitute the motivation for our search for a different approach to memory bandwidth regulation.

MemGuard regulates the maximum number of memory transactions that cores are allowed to perform over a predefined period of time (*i.e.*, their memory bandwidth). Cores are assigned a memory budget that is consumed when cores perform memory transactions and that is periodically replenished. Cores are idled when the budget is depleted. Its implementation relies on three main features: (1) a memory bandwidth monitor; (2) a mechanism to deliver regulation and replenishment interrupts; and (3) a mechanism to idle cores.

Memory bandwidth is monitored using performance counters. Depending on platforms capabilities, implementations of *MemGuard* have used PMCs from cores' PMUs [7], [16] or from the DRAM memory controller [10], [11]. Since overutilization of memory controllers is detrimental to predictability [10], hard real-time systems dimension the memory budget allowed for regulated cores using the principle of *maximum sustainable bandwidth*. That is the maximum bandwidth that a memory controller can sustain under worst-case memory workload, *e.g.*, row misses in the same bank, without experiencing overutilization. When DRAM controller performance counters are not available, determining this value requires know-how of the target platform and non-trivial experimental setups [16], [17].

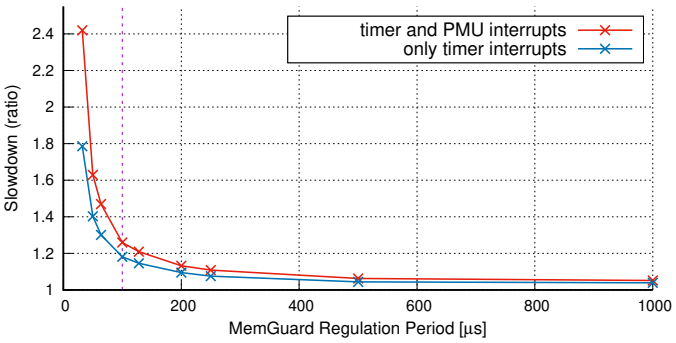


Fig. 1. Impact (slowdown) of *MemGuard*'s timer and regulation overheads on a memory-intensive application as a function of the replenishment period. Results are in line with other work [7], [11] and extended beyond 100 µs.

MemGuard relies on the capabilities of the PMU to deliver a regulation interrupt to a core upon budget depletion. When such an interrupt is received, the core idles by either scheduling a CPU-intensive high-priority task [7], [11], or by stalling the core at the hypervisor level [10], [16]. One timer interrupt periodically replenishes the budget and possibly unblocks the regulated core.

Note that regulation at hypervisor level can only provide a coarse regulation at core level, while regulation at OS level can enable more fine-grained regulation at task level. However, the latter also requires changes to the operating system. Although *MemPol* could be extended to achieve tighter intergration with the operating system and enable per-task regulation, in this work, we focus on the lower-level mechanisms to implement bandwidth regulation, and assume per-core regulation. We defer further integration with the OS to future work.

A. MemGuard Limitations

Interrupt overheads. *MemGuard* delivers interrupts to a core to signal both regulation and replenishment. Such an interrupt-based approach generates an overhead that increases with the frequency of the interruptions, *i.e.*, with shorter replenishment periods, or with smaller budget assignments. Interrupt overheads pose severe constraints on the enforcement of both small memory budgets and short regulation periods.

As an example, Fig. 1 reports the overheads of timer and regulation interrupts in our setup for the version of *MemGuard* that we have used in our experimental comparison (see Sec. V). The figure shows the slowdown of a memory-intensive application[1] as function of the replenishment period. The budget is measured as the number of L2 cache refills. Fig. 1 separately shows the impact of timer *and* regulation (PMU) interrupt, and timer interrupts only. As shown, for

[1] `bandwidth` from the *IsolBench* testsuite (https://github.com/CSL-KU/IsolBench).

short regulation periods (32 μs), *MemGuard* is affected by extremely high overhead—up to 2.4 slowdown ratio. These effects are in-line with previous studies [7], [11] that have shown around 10% overheads for periods of around 100 μs.

Inherent pessimism. Although interrupt handlers normally have minimum memory footprint, they generate memory transactions that are reflected *in the very same* metrics monitored by *MemGuard*. Precisely accounting for this interference is complex, resulting in pessimistic worst-case bandwidth thresholds.

Single monitoring dimension. To reduce implementation complexity and the number of interrupts, *MemGuard* monitors only one memory consumption metric—*e.g.*, cache write-backs, cache refills, or memory controller utilization—at a time.[2] Store instructions on the cores result in higher memory controller utilization than load instructions, because they cause write-backs. Therefore, if only cache refills are monitored, the worst-case scenario consists of a 1:1 ratio between refills and write-backs [10]. But assuming so leads to overall memory under-utilization. At the same time, regulation only based on cache refills might not correctly take into account write-heavy phases that do not generate linefills (see Sec. V-B).

Coarse regulation. Access to memory often results in bursts of cache refills and transactions. To avoid excessive idling of regulated cores and to smooth out the impact of such bursts, *MemGuard*'s budgets and periods must be set to relatively large values. Although beneficial to reduce the impact of interrupt overheads, regulating over large periods results in prolonged memory bursts [10] and in an uneven distribution of memory bandwidth within the period. This complicates the adoption of, *e.g.*, automotive techniques [19] that use offsetting to distribute the peak load of read-execute-write [20], [21] workloads over successive periods. Moreover, as mentioned in Sec. I, it can cause accelerators to receive less bandwidth than their assigned quota.

B. An Alternative Regulation Design

Interrupt overheads and a non-flexible single-dimension monitoring lead to severe compromises for *MemGuard*-based systems. In particular, regulating using core-managed interrupts (either for polling [10], [11] or regulation [7], [18]) cannot eliminate the overheads reported in Fig. 1.

An alternative to avoid interrupting useful computation on the regulated cores is to exploit the heterogeneity of MPSoCs and monitor the PMU counters from *outside* the core cluster, *e.g.*, using one of the many real-time cores available on such platforms. However, while, *e.g.*, on Arm platforms, per-core performance counters are also accessible from outside of a core (see Sec. IV-A), per-core PMU interrupts can only be delivered to other cores on the same complex.[3] Currently, therefore, the only suitable design to perform PMC-based regulation from the outside is to combine *polling* of PMU counters with a control action to throttle (*i.e.*, idle) the cores.

[2]In [18], cache refills *and* write-backs are considered in separated regulations, but their memory contributions cannot be *combined* together.

[3]For GICv3-based systems, Arm recommends using local PPI interrupt 23.

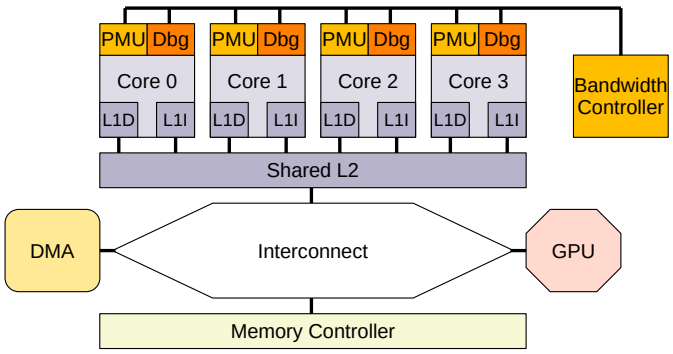


Fig. 2. *MemPol* architecture. Applications cores c_0 to c_3 are regulated by an external controller logic that accesses the application cores' PMU counters as memory-mapped devices and that halts the cores via their debug interfaces.

To fully prevent interrupt overheads, the control action should also be done from the outside and must not involve any type of notification to the to-be-regulated cores. Furthermore, a poll-based design enacts the *simultaneous* use of multiple performance counters to perform regulation, while keeping overheads constant.

Sec. III presents *MemPol*, a poll-based regulation design that operates from outside the cores and regulates multiple monitoring dimensions with low overhead.

III. MemPol – Regulation from Outside the Cores

The first objective of *MemPol* is to remove *any* overheads from the cores to be regulated. This is achieved with a design that operates *from the outside* of the target cores and specifically (1) monitors the last-level cache (LLC) activity by polling the cores' PMU counters, and (2) uses a core-independent interface (*e.g.*, the *CoreSight* debugging interface, see Sec. IV-A) to halt cores when they exceed their given memory budget. The controlling logic of *MemPol* can be implemented on one of the application cores, on a smaller companion core, *e.g.*, Cortex-M and Cortex-R cores, or even in an FPGA. Fig. 2 presents the architecture of *MemPol*.

The second objective of *MemPol* is to enable a multi-dimensional regulation based on the combined contribution of multiple PMU counters, without impacting overheads. In particular, we consider the *accumulated read and write activity* of a core, *i.e.*, the sum of last-level cache misses and write-backs (Sec. III-A). Since the controller *polls* PMU counter values, within a polling period, cores can generate a high number of transactions—thus potentially *overshooting* their assigned budget—that can be only accounted for in the next polling instant. To contrast overshooting effects, *MemPol* has a short polling period P in the *microsecond* range (Sec. III-B).

Compared to *MemGuard*, *MemPol* realizes a different regulation logic that *does not* periodically replenish cores' budgets. Instead, regulation is enacted every polling period P via an *on-off controller* logic (Sec. III-C) that can idle cores for time intervals as short as P. As programs show different behavior during their execution, *i.e.*, memory-intensive phases vs. computation-intensive phases, we limit the *burstiness* of memory accesses using both a *sliding window method* (Sec. III-D)

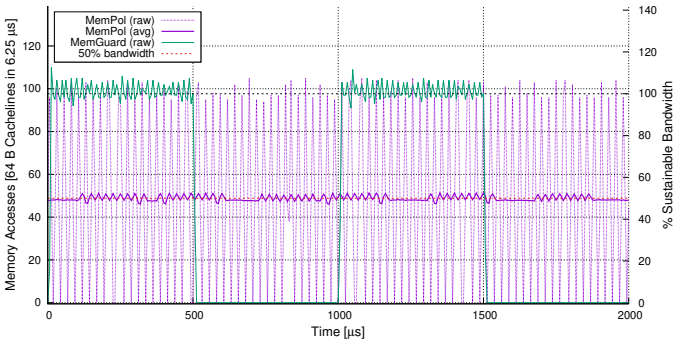


Fig. 3. Comparison of the regulation behavior of *MemPol* (polling at 6.25 µs, sliding window size 50 µs) and *MemGuard* (regulation period 1 ms) on ZCU102 regulating a worst-case memory reader at 50% sustainable memory bandwidth. In both cases, PMU counters are sampled every 6.25 µs. For *MemPol*, the average over 200 µs is also shown for better visualization of its resulting regulation. In the given example, both mechanisms achieve the same regulation results over longer time spans. *MemPol* just regulates faster.

and a combined strategy to account for non-memory-intensive phases (Sec. III-E). Overall, cores can experience multiple on/off transitions during the length R of the sliding window, but can also idle for periods longer than R due to overshooting under small bandwidth-regulation (Sec. III-F).

As an example of the low-overhead, high-resolution capabilities enabled by the *MemPol* design, we implement *two regulation strategies* that operate at microsecond scale: (i) a *local per-core controller* that regulates a core's memory bandwidth *w.r.t.* a given *local per-core budget* independently for each core, and (ii) a *global controller* that redistributes unused bandwidth to demanding cores, but keeps the overall bandwidth of all cores below a given *global budget* (Sec. III-G). Contrary to the complex interactions among cores that would be needed to realize a global controller under *MemGuard*, our global controller relies on the poll-based regulation and only requires minimal additions compared to the local one. Fig. 3 gives an overview of the fine-grained actions performed by *MemPol* in comparison to the coarse-grained ones used by *MemGuard*. (See Sec. V-A for details.)

A. Regulation Cost Model

Assuming a system comprising a set of cores C, we model a core c_i's performance counters for read and write accesses as functions over time $PMU_i^r(t)$ resp. $PMU_i^w(t)$, which return non-decreasing integer values that relate to memory accesses. We introduce the coefficients α_r and α_w to account for different impacts that reads and writes have on the saturation level of the memory subsystem.[4] We then sample the PMC values every P time units and aggregate the memory activity as a monotonic function $A_i(t) = \alpha_r PMU_i^r(t) + \alpha_w PMU_i^w(t)$.

The memory bandwidth that can be extracted from the memory controller highly depends on the memory access patterns and can deviate between best-case and worst-case scenarios by an order of magnitude or more. Previous experiments have shown that in best-case conditions like linear memory accesses the cores are the limiting factor, while in worst-case conditions like continuous row-misses the memory controller becomes a bottleneck [10]. Given our real-time focus, the cost model for regulation is based on the sustainable memory bandwidth $B_{sustainable}$, *i.e.*, the minimum bandwidth that can be extracted by all cores in parallel in worst-case scenarios. We can therefore assign a fraction of the sustainable bandwidth to each core c_i as B_i, $\sum_{j \in C} B_j \leq B_{sustainable}$. The maximum allowed number of aggregated accesses to stay within the budget limits during time P is $A_i^{budget} = B_i * P$.

[4]For example, in flash memory, reading is much faster than writing.

B. Overshooting

In *MemGuard*, a PMU triggers an interrupt whenever a core exceeds its budget. Instead, a polling controller samples PMCs periodically and can only detect budget overruns for the previous period P. This might results in overshooting the target budget. Under real-time constraints, overshooting is even exacerbated. In fact, the regulation is based on the sustainable worst-case bandwidth and not on the real memory utilization at the memory controller, which can handle peak best-case bandwidths much higher than the ones used for regulation (see Sec. IV-D for the ZCU102). We characterize the peak bandwidth that can be accessed by a single core as $B_{peak-core}$ and use the factor $\beta = B_{peak-core}/B_{sustainable}$ to express overshooting in relation to $B_{sustainable}$. We further use the factor $\beta_i = B_{peak-core}/B_i$ to describe the overshooting of a core c_i in relation its configured bandwidth target B_i.

A second contributing factor to overshooting is delays in the control path between *observing* that a core has exceeded its bandwidth budget and the point where a core *stops* issuing further memory requests. We denote this delay as D and express the overshooting by a factor $\delta = D/P$ to the length of the polling period P. The product $\beta_i(1+\delta)$ describes the worst-case overshooting when a core c_i accesses memory at peak bandwidth and exceeds its budget at the beginning of P.

C. On-Off Controller as Bandwidth Limiter

To regulate a core c_i at time $t > t_0$, *MemPol* derives a set-point $sp_i(t, t_0) = A_i(t_0) + \lfloor \frac{t-t_0}{P} \rfloor A_i^{budget}$ based on the core's memory accesses A_i at time t_0 and its configured budget. Using an on-off controller, *MemPol* halts a core if $A_i(t) > sp_i(t, t_0)$, and let the core run (again) if $A_i(t) \leq sp_i(t, t_0)$. At each P, the core's set-value budget is increased by A_i^{budget}.

D. Sliding Window Technique to Control Burstiness

Real-time programs tend to access memory in burst. For example, after long idle or computation phases with few memory accesses, a program might access data again to prepare for the next iteration. The yellow gradient line in Fig. 4 depicts such a burst. Since the on-off controller from Sec. III-C uses as point of reference $t_0 = 0$, it includes the non memory-intensive phase (green gradient line in Fig. 4) of the core. This would allow the core to run and access memory even during the burst at time $t = 8$, which is instead potentially detrimental for the real-time guarantees of other cores.

We therefore cap the budget of a core by "forgetting" the core's unused bandwidth and limit the core's burstiness with a

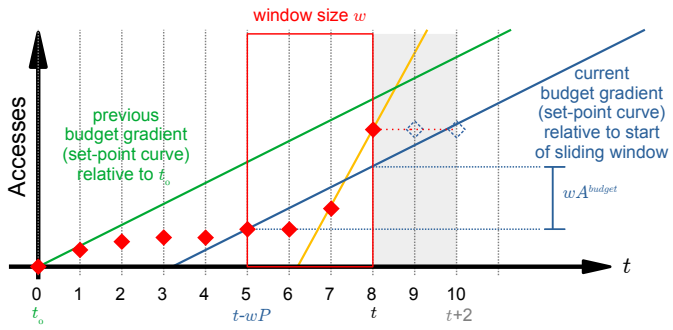


Fig. 4. Sliding window technique. At time t=8, the burst (yellow gradient) is within a previous budget gradient from time 0 (green gradient), but not within the current budget gradient at the start of the sliding window at time 5 (blue gradient). Based on its recent history in $(t-wP, t)$ (red box), the core will be rate-limited for at least two periods in $(t, t{+}2)$. See Sect. III-D.

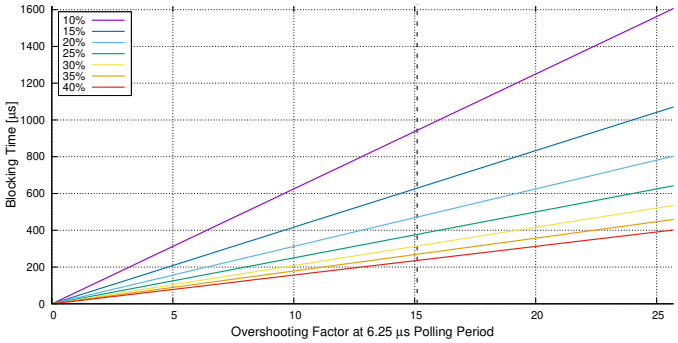


Fig. 5. Overshooting in relation to $B_{sustainable}$ by a certain factor (x axis) and the resulting blocking time (y axis) for different bandwidth levels (%) in a regulation at 6.25 µs. Lower bandwidth levels observe higher blocking times. The maximum observed overshooting in relation to $B_{sustainable}$ on the ZCU102 is factor 15.1 (dotted vertical line), see Sec IV-D.

Algorithm 1: Controller implementation (Sec. III-E)

```
1  input:
2      A_i^budget                          ◁ budget, number of memory accesses
3      w                                   ◁ history size, equal to size of sliding window
4      α_r, α_w                            ◁ weight-factors for reads and writes
5  init:
6      hist[0..w−1] = α_r * pmc_r + α_w * pmc_w     ◁ history data
7      i = 0                               ◁ position in history data, 0..w−1
8      t_lrt = w                           ◁ time since last rate-limited, initially not
9      spv_lrt = undef                     ◁ set-point value at start of rate-limiting
10 loop:
11     if t_lrt < w then                   ◁ rate-limited mode
12         t_lrt = t_lrt + 1               ◁ age rate-limiting
13         spv = spv_lrt + t_lrt * A_i^budget   ◁ spv from start of r.-lim.
14     else                                ◁ non rate-limited mode
15         spv = hist[i] + w * A_i^budget   ◁ spv from start of sl. win.
16     val = α_r * pmc_r + α_w * pmc_w     ◁ read PMCs
17     delta = val - spv                   ◁ signed delta value, integer overflow
18     if delta > 0 then                   ◁ PMC above set-point value, throttle
19         t_lrt = 0                       ◁ (re-)start aging of rate-limiting
20         spv_lrt = spv                   ◁ further budgeting based on spv
21         hist[i] = spv_lrt               ◁ update history, rate limited
22         throttle()                      ◁ halt core if running
23     else                                ◁ PMC below set-point value, resume
24         hist[i] = val                   ◁ update history with current value
25         resume()                        ◁ resume core if halted
26     i = (i + 1) MOD w                   ◁ next position in history data
```

sliding window of w polling periods. At time t, we use $t-wP$ as start of the window, and derive a new budget gradient (the blue gradient line in Fig. 4). We then move the window to the right each polling period (the red box in Fig. 4).

E. Resulting Combined Control Strategy

MemPol's controller combines strategies III-C and III-D depending on the behavior in the previous w polling periods.

Not rate-limited. A sliding window (Sec. III-D) tracks the behavior of a core c_i if at time t is has not exceeded its budget wA_i^{budget} for at least the last w polling periods. In each period P, the reference point t_0 of the budget gradient is *moved* to the current start of the sliding window.

Rate-limited. The first time core c_i exceeds its given budget wA_i^{budget} at time t, the reference t_0 of the sliding window is *frozen* at $t_0 = t - wP$, and the on-off controller (Sec. III-C) regulates c_i until its budget returns below the budget gradient rooted in t_0 for at least w polling periods.

Alg. 1 presents the resulting controller implementation, which stores in $hist[]$ the last w values of $A_i(t)$ and tracks in t_{lrt} (aging counter) the last time that the budget was exceeded. t_{lrt} also defines the current control mode ($0..w{-}1$ rate-limited, w not rate-limited). While in rate-limited mode, the variable spv_{lrt} tracks the set-point value of the budget gradient.

The controller starts in not rate-limited mode and initializes the history data with current PMC values (L. 6–9). In each iteration of the control loop, a current set-point value spv is calculated depending on the current controller mode. In rate-limited mode, the controller ages t_{lrt} and derives spv (L. 11–13) from the variable spv_{lrt} set at the start of rate-limiting (L. 20). Otherwise, spv is set to the history value at the start of the sliding window (L. 15). Afterwards, the controller samples the current PMC value (L. 16). If the PMC value is above spv, the controller enters rate-limiting mode (L. 19–22): it sets $t_{lrt} = 0$ to keep the controller in rate-limited mode for at least the next w loops and it throttles the core. The current spv is copied into spv_{lrt} and defines the base for further budgeting. spv is also stored in the history data to keep the burst bounded. Once active, if rate-limited mode is entered multiple times, the budget gradient established by spv_{lrt} remains constant. When PMC values drop below spv, the controller resumes the core and updates the history data (L. 23–25).

F. Setting Regulator's Budgets

Under *MemPol*'s regulation strategy, the amount of time that a core c_i is throttled depends on "how-much" it overshoots its budget B_i, which is accounted for in β_i. The resulting worst-case blocking time of c_i is therefore $\beta_i(1+\delta)P$. Fig. 5 visualizes such blocking times as function of the overshooting factor normalized to $B_{sustainable}$. For example, if core c_i overshoots $B_{sustainable}$ by factor 10 ($B_{peak-core} = 10 \times B_{sustainable}$) and has an assigned budget B_i of 10% of $B_{sustainable}$, it will be halted for at least 100 polling periods. With a polling period of 6.25 µs (see Sec. IV-D), this corresponds to 625 µs blocking time. The maximum overshooting factor normalized

to $B_{sustainable}$ observed in our experiments was $\beta = 15.1$ on the ZCU102 (see Sec. IV-D).

Under *MemGuard* regulation instead, the blocking time is constant and upper-bounded by the length of a replenishment period. In practice, though, the blocking time of *MemGuard* can be even higher than *MemPol*'s, since the typical regulation period of *MemGuard* is 1 ms.

G. Combined Local Per-Core and Global Regulation

The logic presented in Sec. III-A–III-E implements *local per-core controllers* that are independent of each other. However, the polling-based regulator can be easily extended to implement a *global controller* that uses the same regulation logic, but observes the *sum* of the memory accesses of all cores and the *sum* of all budgets. We note that, contrary to *MemGuard*-based regulation, the global controller can be implemented alongside the local one and does not require complicated interaction among cores.

The control decision of the global controller to halt or run cores impacts the local per-core controllers as follows:

- per-core controller=run ▷ **run**
- per-core controller=halt ∧ global controller=run ▷ **run**
- per-core controller=halt ∧ global controller=halt ▷ **halt**

The global controller overrides a per-core controller decision only if the previous bandwidth demand of all cores was below the configured budget. Additionally, the global controller updates per-core controller's t_0 to t, thus forcing cores to acknowledge the actual used bandwidth and preventing penalties due to the overriding forced by the global controller. The redistribution scheme stops as soon as the bandwidth demand increases.

H. Regulator Sliding Window Size Settings

The regulation model allows for different sliding window sizes w and bandwidth settings B for the per-core and the global controller. An assignment is valid as long as $w_{global} \leq \max_{j \in C}(w_j)$ and $\sum_{j \in C} B_j \leq B_{global}$.

Setting per-core w_i value is particularly sensitive to the burstiness of applications executing on core c_i. Although an actual value should be derived from the temporal behavior of the regulated applications, Sec. III-D hints to the possible compromise of limiting the budget during a burst to $w_i A_i^{budget}$, and the time the regulator "forgets" previous bursts to $w_i P$.

On the global-controller side, one would intuitively try to set the w_{global} to a very small value. But as the global controller has no influence on the distribution of memory bursts on the cores and the decisions of the per-core controllers, a small w_{global} value would not result in a better regulation than setting w_{global} to similar values as for the per-cores controllers.

In this paper, we opted to use the same w value for all per-core and the global controller and leave an evaluation of different w trade-offs for future work.

IV. Implementation

Before explaining the main components of *MemPol*, we briefly summarize the relevant features of the Xilinx Zynq UltraScale+ ZCU102 SoC [2] that has been used for our prototype. The ZCU102 includes four Arm Cortex-A53 *application processor (AP)* cores and two Arm Cortex-R5 *real-time processor (RP)* cores. The AP cores are connected to the full-power domain and feature private L1 caches and a shared L2 cache (LLC). The RP cores are connected to the low-power domain and have access to private tightly-coupled memories (TCM). An Arm CCI-400 acts as a central cache-coherent interconnect between low- and full-power domains and the main DDR4 memory controller.

Arm defines a common infrastructure (*CoreSight*) for hardware debugging of its cores [22]. CoreSight defines registers of memory-mapped debug devices on a low-bandwidth APB bus that can be accessed through a *debug access port (DAP)*. The CoreSight infrastructure is available on the ZCU102 and it is also present on most boards featuring A57 and A72 cores. To debug devices connected to CoreSight, the typical setup comprises per-core debug interfaces, PMCs, trace interfaces, cross trigger interfaces (CTI), and a shared cross trigger matrix (CTM) [8], [23]. The CTI is normally used by an external hardware debuggers to halt and resume cores, while the debug interface provides access to the core's internal state. The CTM connects instead input and output signals from the CTI and allows halting multiple cores on a debug event in a synchronized manner. Similarly, information provided by PMCs can also be controlled.

A. Exploiting Memory-Mapped Debug and PMU Registers

In the standard workflow to halt a core via the memory-mapped CTI registers, a debugger triggers the *debug request* input of the core. The core eventually enters *debug halt* state. Before a new request can be sent, the debugger acknowledges the previous debug request, then polls the CTI to ensure that the previous request has been properly de-asserted. To resume a core, a debugger must trigger a *debug restart* signal via the CTI. The core automatically acknowledges this request.

MemPol mimics the behavior of a debugger and appropriately manipulates the CTI debug registers to stall and restart cores. After initial programming, each halt and resume request requires a single write transaction to the CTI's registers and the acknowledgment of previous debug requests. We discovered experimentally that polling for previous requests is not required if there is a sufficient delay between writes to acknowledge and resume registers.

To monitor the PMCs, the PMU register interface provides full access to all six performance counters of a core. After initialization, reading a PMC requires a single read transaction.

The setup of CTI and PMU requires taking ownership of the debug interface by disabling software lock registers and then configuring the devices. The Arm architecture defines an authentication interface of four signals for invasive / non-invasive debugging in secure / non-secure execution state. Both CTI and PMU require invasive resp. non-invasive debugging of non-secure execution state (`DBGEN`, `NIDEN`) to be enabled. Regulating secure applications is out of scope for this work.[5]

B. Accessing Debug and PMU Registers

Internal overhead: In our experiments, accesses to a core's memory-mapped PMU registers in a tight loop from a second core show no measurable impact on the performance on the first core. This allows for *interference-free remote monitoring*.

External overhead and applicability: The memory-mapped CTI and PMU registers are connected through a low-bandwidth APB bus that allows only one outstanding transaction [25]. We measured the access time from the main cores to CoreSight registers on multiple platforms and asserted the possibility to stop/resume cores using the debug interface. When accessing CoreSight registers on the ZCU102, we measured a mean overhead for reading/writing of 303/213 ns from the A53 cores and of 274/216 ns from the R5 cores. Likewise, we measured 450/257 ns on an NXP S32G274 [13] (Cortex-A53), 135/122 ns on a Raspberry Pi 4 [14] (Cortex-A72) and 374/366 ns on an NXP LX2160A [15] (Cortex-A72). Note that the full regulation has only been implemented and evaluated on the ZCU102. Compared to using an *uncached strongly-ordered mapping*, mapping registers as *shared device* (so cores do not need to wait for transactions to complete), significantly speeds up write operations. The reported measures have been performed on systems that were otherwise idle during testing. Instead, while stressing the memory subsystem in parallel to the tests, we observed that latencies on our ZCU102 increase up to 1146/643 ns for access from the A53 cores.[6]

C. MemPol Regulator

We implemented the regulator on the first Cortex-R5 core. The regulator exposes a memory-mapped interface in the TCM of its core. Following the design of hardware registers, the interface comprises status and control registers. After booting, a main loop polls the control registers and updates status registers periodically. The interface also exposes the full internal state of the four per-core controllers and the global controller with history buffers of up to 128 entries. For tracing purposes, we used the TCM of the second R5 core as a trace buffer to record PMC values.

When enabled, the regulator first programs the last two PMCs of each core (events `0x17` L2 data cache refill, `0x18` L2 data cache write-back), initializes the regulator, and starts the control loop. In each iteration of the control loop, the regulator (1) reads the two PMU counters of each of the four AP cores; (2) takes control decisions for each core based on the per-core and the global controller settings; (3) halts, resumes, or leaves the core's state unchanged; and (4) waits for the start of the next control loop period.

[5]Monitoring and throttling in secure execution state (*TrustZone*) is enabled by `SPIDEN` and `SPNIDEN` signals. This allows to separate secure from non-secure workloads down to the hardware level, but also requires a security concept for *all* components in an SoC. See [24] for further details on the security impact of on-chip monitoring and debugging facilities. Note that *MemGuard* faces similar challenges in setting up PMU counters to monitor secure applications from a non-secure hypervisor or operating system.

[6]This hints to bottlenecks at the interconnect level between the A53 cores and the CoreSight registers that we would like to investigate in future work. Accessing the CoreSight registers from the R5 cores shows lower latencies, as the transactions take a different path and stay in the SoC's low-power domain.

To give cores sufficient time to acknowledge a previous halt request before resuming, we spread the sequence of halting/resuming a core (three memory transactions with delays) as either two CTI transactions in the halting case (trigger halt + trigger nothing) and two CTI transactions in the resume case (acknowledge + trigger resume). If a core's state is unchanged, we perform two dummy writes to the CTI trigger register. We further interleave each CTI access with accesses to all other cores' states. Such patterns ensure that cores can fully halt (resp. resume) their activities in parallel to the remaining execution of the control loop and the reading of the PMU registers (in the next loop iteration). In fact, our experiments showed that, after sending the halt signal, cores do not immediately stop, but remain active for some time in the presence of outstanding memory transactions. In an experiment where an A53 core sends a halt signal to itself and then monitors a timer to detect when it eventually halts, we observed a maximum delay of 320 ns by adding read-modify-write operations (store byte) to cold cachelines before and after the halt request. The core was able to emit up to 8 further read-modify-write operations after sending the halt. This number matches the 8 outstanding linefills per core documented for the L2 memory subsystem of the Cortex-A53 core complex [23]. Since all four cores can have outstanding transactions, we assume the worst-case halt delay to be at most 1.5 µs. In our experiments, we observed a delay of around 1 µs.

The implementation—standard 32-bit integer arithmetic and multiplication—requires 7 KB Arm code (kept in TCM, with 2.6 KB code for formatted console output), 3 KB of data (controller state in TCM) and a 1 KB stack (also in TCM).

Overall, the 16 transactions to CoreSight registers—*i.e.*, eight to read PMU counters and eight to throttle cores—dominate the execution time of the controller. Profiling showed that the execution of the control loop takes between 4.8 to 5.2 µs. We experimentally observed that using a `DSB` instruction at the end of the control loop reduces jitter, as the R5 core flushes any outstanding writes to CTI registers before starting a new round and reading from the PMU. In our benchmarks, the control loop never reached the worst-case overheads observed on the A53 cores in Sec. IV-B: we hypothesize that the smaller amount of accesses and the `DSB` at the end of the control loop prevent changes in the priority of the traffic at interconnect level. Overall, we add a safety margin to the observed values and set the period of the control loop to 6.25 µs.

D. Cost Model on ZCU102

Using the *USTRESS* benchmark [10], we observe a sustainable memory bandwidth of $B_{sustainable} \approx 1000$ MB/s (954 MiB/s) for both reading and writing on the ZCU102. In the cost model of the controller, this translates to 97.656 cachelines of 64 B per 6.25 µs period with weight-factors $\alpha_r = \alpha_w = 1$ for both reading and writing.[7] Assuming instead

[7]The implementation uses a factor of $\alpha = 1000$ and a budget of 97656 cachelines per loop to compensate any loss of precision in the decimal places.

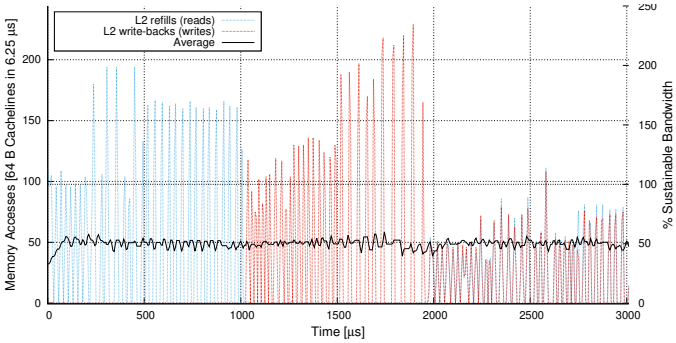


Fig. 6. Polling regulation at 6.25 µs of a core at 50% sustainable memory bandwidth. The core performs three series of four different memory access patterns every 250 µs: four read patterns, four write patterns, then four modify (read-write) patterns. The overall number of memory accesses is the same each time, but peak-behavior increases within a series. 200 µs averages.

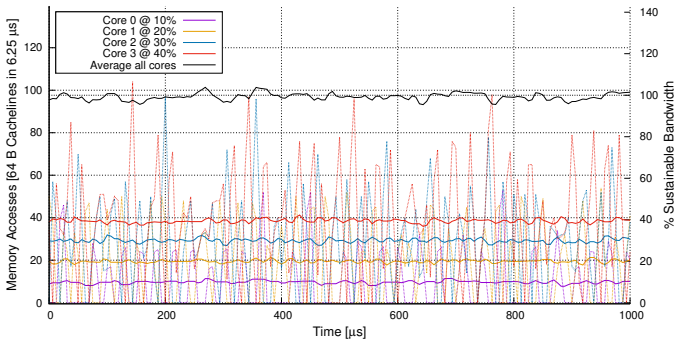


Fig. 7. *MemPol* regulates cores at different bandwidth levels: c_0 worst-case reader at 10%, c_1 worst-case writer at 20%, c_2 peak reader at 30%, c_3 peak writer at 40%. Polling 6.25 µs. 50 µs sliding window size. 200 µs averages.

linear memory accesses, the peak memory bandwidth is $B_{peak-core,r} = 4240$ MB/s for reading and $B_{peak-core,w} = 8162$ MB/s for writing. This results in an overshooting factor $\beta = max(B_{peak-core,r}, B_{peak-core,w})/B_{sustainable} = 8.16$, or peaks of up to 797 cachelines in 6.25 µs.

Lastly, we assume a maximum poll-control delay $D = 4$ µs in the $P = 6.25$ µs control loop, *i.e.*, around 2.5 µs between reading a core's performance counters and signaling the debug halt plus a delay of 1.5 µs until a core halts, resulting in a control delay factor of $\delta = D/P \approx \frac{2}{3}$. The maximum overshooting factor in relation to $B_{sustainable}$ we expect the controller to handle is therefore a factor of $\beta(1+\delta) = 13.6$. Note that, since the exact moment *when* a core stops after signaling *debug halt* is unknown, β is an approximation of the overshooting. Our experiments showed peak outlier PMC values of 695 refills, 893 write-backs, or 1479 for the combined counter values, which results in factor $\beta_{observed} = 15.1$ slightly above the computed β.

V. Evaluation

We evaluate *MemPol* on the ZCU102 platform. The regulator runs bare-metal on the R5 core and is independent of the operating system on the application cores. It is loaded during system startup as part of the boot loader configuration, and it remains inactive until the benchmarks configure its parameters and start it. The regulator polls PMU counters every 6.25 µs and using a default sliding window size w of 8 entries (50 µs).

We evaluate the details of *MemPol*'s regulation with a set of experiments on a lightweight RTOS, which allows full control of cores activities and of the physical memory layout. We have implemented *MemGuard* on the RTOS for low-level comparisons with *MemPol*. Furthermore, we perform a comparison of *MemPol* and *MemGuard* from [18] on Linux (PetaLinux 2021.1, Linux 5.4 kernel) using the San Diego Vision Benchmark Suite (SD-VBS) [26]. In the SD-VBS, we hook into `photonStartTiming()` and `photonEndTiming()` to measure execution times and to precisely coordinate the start of the regulation. The plots in this section show the aggregated core's L2 cache activity over time as memory accesses (number of cachelines) and as the percentage of the sustainable bandwidth. Averages over t-100 µs to t+100 µs are shown as thick lines.[8]

A. Per-Core Regulation

We first present experiments of the per-core regulation based on both read and write access measurements. The test applications generate different memory access patterns. The patterns differ in the access type (loads, stores, or modifications of full cachelines) and in the stress they cause in the memory controller (worst-case accesses or linear accesses).

Fig. 3 shows a worst-case reader regulated by both *MemGuard* and *MemPol*. In both cases, we can observe the number of L2 cache refills matches the worst-case of approx. 97 cachelines per 6.25 µs. The worst-case readers use `PRFM PLDL1KEEP` instructions to prefetch data to the L1 cache instead of using normal loads. This removes any dependencies in the core's pipeline to wait for the loaded data.

Focusing on *MemPol* only, Fig. 6 shows different memory access patterns changing every 250 µs on a core regulated at 50% of the sustainable bandwidth. Starting from the left, the application first performs worst-case *loads* (each load causes a bank switch) for 250 µs. In the subsequent ranges of 250 µs each, the test performs 2, 4, and 8 memory accesses to the same bank before switching bank. In the next four ranges, the application repeats the same patterns, but with stores to *whole cachelines* instead of loads, thus ensuring that cachelines bypass the cache (write-through). Finally, the application does *read-modify-write* accesses to cachelines. The number of memory accesses is the same in each test, but the latencies at the memory controller differ. Fig. 6 shows three main trends. (1) Linear memory accesses are handled faster than worst-case ones. (2) As expected, higher overshooting corresponds to longer idle times. (3) Buffering of write transactions causes more frequent and higher spikes than reads. We also note that a variation of the worst-case load pattern starting at 250 µs generates higher overshooting than peak accesses at 750 µs.

Fig. 7 shows the behavior of *MemPol* in simultaneously enforcing different bandwidth levels. Here, cores c_0 and c_1 at 10% (20%) levels perform worst-case reads (writes—to whole

[8] A moving average of 200 µs proved to be a good trade-off to show the regulation trends even in case of overshooting.

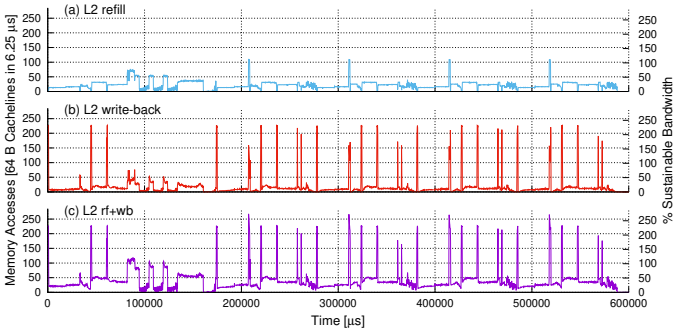


Fig. 8. 200 µs averages of PMCs of a run of `tracking` in VGA resolution. The graphs show (a) L2 refills, (b) L2 write-backs, and (c) combined L2 refills and write-backs. *MemGuard* regulates based on (a), *MemPol* based on (c).

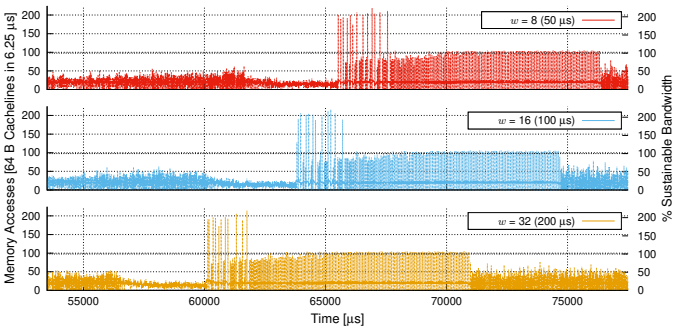


Fig. 9. Three runs of `tracking` in VGA resolution regulated at 20% sustainable memory bandwidth. The graphs detail the first write peak (Fig. 8 at around 45 ms) for different sliding window sizes of 50 µs, 100 µs and 200 µs. Larger sliding window sizes allow the benchmark to reach the peak earlier, *i.e.*, at around 60 ms (200 µs) instead of 63.8 ms or 65.5 ms (50 µs).

cachelines), while cores c_2 and c_3 at 30% (40%) levels perform linear reads (writes). Overall, the cores meet their average bandwidth targets, despite the visible overshooting of cores c_2 and c_3. Note the quite regular distance between spikes for the individual cores, and that the height of the spikes relates to the memory access pattern.

B. Regulation based on L2 Data Cache Refill and Write-Back

As mentioned in Sec. II, the single monitoring dimension used by *MemGuard* may lead to memory under-utilization and may not correctly account for *e.g.*, write-heavy behaviors. By monitoring multiple dimensions *at once*, *MemPol* can instead overcome these limitations as shown in this experiment that measures the impact of L2 cache write-backs on the regulation model (Sec. III-A). For this, we record the PMU counters for a full *unregulated* run of the `tracking` SD-VBS benchmark. Fig. 8 shows the sampled L2 cache refill and write-back counters. After initial preparation (up to approx. 180 ms), the benchmark starts to track objects in four consecutive images for about about 100 ms each.

The bandwidth reported by the L2 cache refill counter (Fig. 8 (a)) shows that the bandwidth stays mostly below the 25% mark during the execution, with one larger and four minor spikes beyond the 50% mark. This is the data that *MemGuard* uses for regulation. In contrast, when also monitoring the L2 cache write-back counter, Fig. 8 (b) shows that the benchmark typically consumes between 10 to 15% of the bandwidth, but causes many frequent write-peaks beyond the 200% mark. Fig. 8 (c) shows the combined L2 cache counters that are used by *MemPol*-regulation following the cost model in Sec. III-A. We see that the overall bandwidth demand accumulates and sometimes exceeds the 250% mark.

Compared to *MemGuard*, *MemPol* can precisely track the write behavior and correctly account for the previous state of the L2 cache. Instead, to correctly regulate, *MemGuard* must make pessimistic assumptions on the write behavior, or must use statistical information obtained by prior profiling [10].

C. Impact of Sliding Window Size

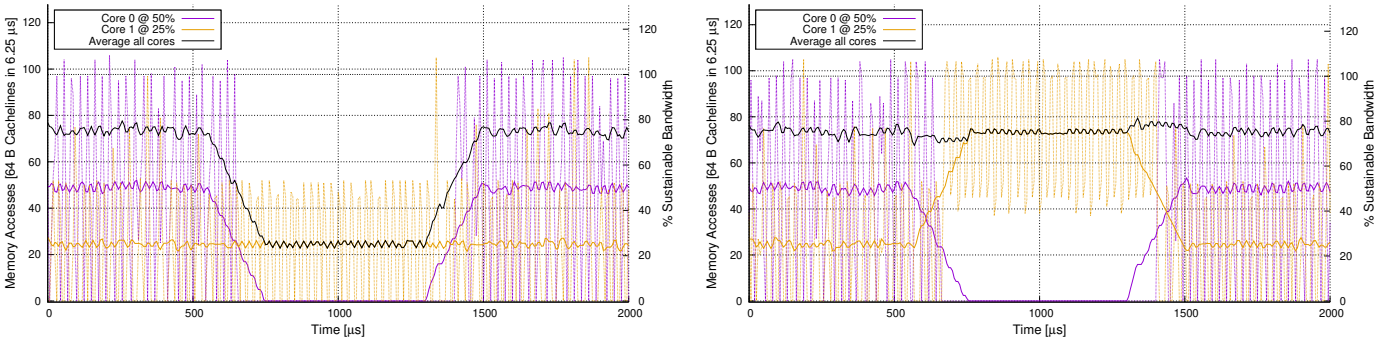
Fig. 9 compares three *regulated* runs of the `tracking` SD-VBS benchmark at 20% sustainable bandwidth with different settings for w focusing on the first write peak at around 45 ms in the unregulated run Fig. 8. In the experiment, smaller w causes larger slowdown (*i.e.*, the spikes appear later) than bigger w values. For example, at w = 8 (50 µs), the execution is slowed down for up to 5.5 ms. This shows that certain workloads are *sensitive* to the sliding window size and require profiling to find acceptable settings. Obviously, for small sliding windows the regulation is less tolerant to periodically repeating spikes, as the margins to compensate for the spikes in non-memory-intensive phases reduce.

D. Redistribution of Memory Bandwidth by Global Regulator

Fig. 10 shows the redistribution of unused memory bandwidth of *MemPol*'s global regulator. Here, core c_0 (regulated at 50%) alternates between memory access and idle phases, while core c_1 (regulated at 25%) always performs memory accesses. When the global regulation is disabled (Fig. 10 (a)), the overall bandwidth drops to 25% when c_0 is idle. Instead, when the global regulation is enabled (Fig. 10 (b)), c_1 is allowed to use any remaining bandwidth up to the global configured limit of 75%. In both cases, we can observe a slight overshooting of the average global bandwidth when c_0 returns from being idle, as the local regulator for c_0 lets the core consume the bandwidth up to its budget. The global regulator cannot prevent this, as it can only override the *halt* decision of the local per-core regulator as described in Sec. III-G.

E. Comparing Regulation of MemPol and MemGuard

We compare the regulation of *MemPol* and *MemGuard* using SD-VBS. For comparable results between *MemPol* and *MemGuard*, we constraint *MemPol* to use only the L2 cache refill counter instead of the more precise combined model (Sec. V-B). We measure the execution time of the benchmarks under regulation and co-scheduled with other benchmarks, and we compare the results to unregulated executions in isolation. After several initial runs, we observed that `disparity`, `mser`, `sift`, `stitch`, and `tracking` provide the most noteworthy result for this experiment. We use sliding window sizes of 50 µs, 100 µs, and 200 µs for *MemPol*, and compare them to replenishment periods of 50 µs, 200 µs, and 1 ms for *MemGuard*.



(a) Global regulation disabled

(b) Global regulation enabled

Fig. 10. *MemPol* bandwidth redistribution: Core c_0 is regulated at 50% bandwidth and alternates memory access and idle phases every 750 µs. Core c_1 is regulated at 25% bandwidth and accesses memory all the time. Both cores perform worst-case reading. (a) The global regulator is disabled. (b) The global regulator is enabled and redistributes unused bandwidth from c_0 to c_1 while c_0 is idle, but keeps the overall bandwidth at 75%, which the sum of both cores' configured bandwidth. Polling at 6.25 µs. 50 µs sliding window size. 200 µs averages.

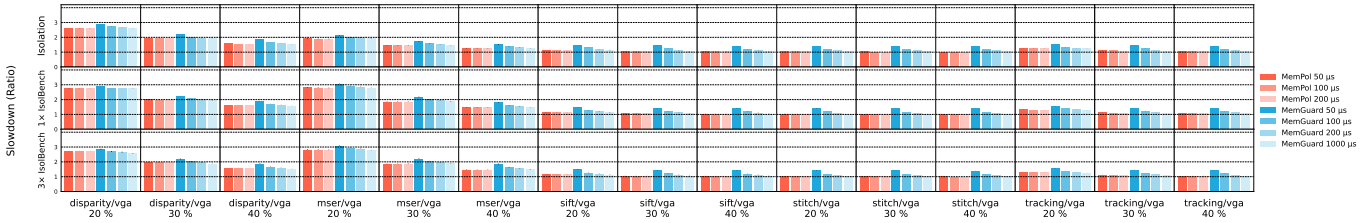


Fig. 11. Slowdown ratio in execution time of SD-VBS regulated at 20%, 30% or 40% sustainable bandwidth compared to unregulated execution as baseline (first horizontal group). The slowdown is caused by memory bandwidth regulation (*MemPol*, *MemGuard*) and by implementation overheads (interrupt handling in *MemGuard*, see Sec. II-A). The colored bars represent the relative mean overhead of 10 runs. The small vertical black lines show min/max. The benchmarks run alone or in parallel with *IsolBench* on one or three other cores. We evaluate *MemPol* and *MemGuard* at different sliding window sizes / regulation periods. *MemPol* regulates on L2 cache refill counters only, like *MemGuard*. *MemPol*'s global regulation is turned off.

In our first set of experiments (Fig. 11), global regulation is disabled, and we evaluate the regulated benchmarks at 20%, 30%, and 40% of the sustainable bandwidth, which are typical settings for one core in a four core setup. We run the benchmarks in isolation (first horizontal group in Fig. 11) and together with *IsolBench* on another core (60% bandwidth) or on three other cores (3 x 20% bandwidth), and we measure the slowdown ratio. As expected, overheads in execution time compared to the unregulated baseline increase for smaller regulation periods and lower bandwidths. In both *MemPol* and *MemGuard* setups, `mser` is the most affected one by the parallel execution with *IsolBench*, while, in general, the number of co-runners has no significant impact on the regulation. Overall, even when using only the L2 cache refill counter, *MemPol* regulates comparably to *MemGuard*, with *MemGuard* showing higher overheads at smaller regulation periods due to the increased interrupt load.

In our second set of experiments (Fig. 12), we evaluate the benchmarks executing in parallel on two cores with an equal regulation of 20% and 30% (Fig. 11 shows that 20% and 30% are the most interesting bandwidth settings). Here we also enable *MemPol*'s global regulation[9] and use 40% resp. 60% for the global bandwidth. From the benchmarks, we select `disparity`, `sift`, and `tracking` as co-runners, as they run for a longer time. Similarly to Fig. 11, the regulations of *MemPol* and *MemGuard* are in general comparable. The global regulation never causes higher overheads, but its benefits are strongly dependent on the benchmark combinations (`disparity` and `mser` benefit the most). Interestingly, *MemPol*'s global regulation helps `disparity` when run in parallel to `tracking`, but not vice versa (bottom left vs. top right), because `tracking` is compute-bound (see Fig. 9 (a)), but `disparity` is memory-bound.

[9]It does not make sense to evaluate bandwidth redistribution with memory hogs like *IsolBench*.

F. Discussion

The evaluation section has shown the potential of the fine-grained regulation, flexibility, and low-overheads enabled by *MemPol*. Additionally, even when considering only one regulation dimension, *MemPol* achieves comparable or better results than *MemGuard*. While *MemGuard* shows no control delays and halts cores when they reach or exceed their bandwidth limits, *MemPol*'s behavior is driven by both the polling frequency in its control loop and delays in halting via the debug interface. This leads to overshooting, which is amplified by the difference between sustainable bandwidth targets (needed by regulation in real-time systems), and the peak bandwidth the memory controller can deliver in best-case conditions. On the other hand, *MemPol* can consider a wider range of metrics for regulation (compared to just a single PMU counter in *MemGuard*'s case) and enables microsecond-scale regulation that also help to mitigate the side effects of overshooting and to bound blocking times of the cores.

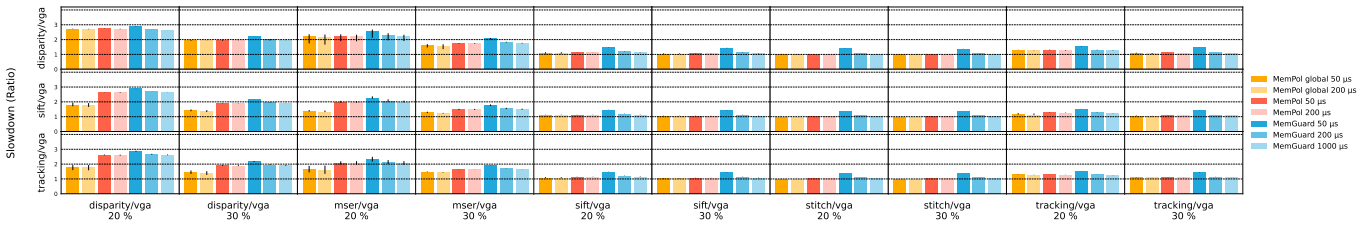


Fig. 12. Slowdown ratio in execution time of SD-VBS regulated at 20% and 30% sustainable bandwidth compared to unregulated execution as baseline. The slowdown is caused by memory bandwidth regulation (*MemPol*, *MemGuard*) and by implementation overheads (interrupt handling in *MemGuard*, see Sec. II-A). The colored bars represent the relative mean overhead of 10 runs. The small vertical black lines show min/max. The benchmarks run in parallel with another instance of a benchmark with the same bandwidth settings on a second core. We evaluate *MemPol* and *MemGuard* at different sliding window sizes / regulation periods. We also include results with *MemPol*'s global regulation enabled at 40% resp. 60% global bandwidth. *MemPol* regulates on L2 cache refill counters only, like *MemGuard*.

Although *MemPol* is a good starting point for novel regulation schemes based on polling, our investigation have shown that non-polling-based regulators (*e.g.*, *MemGuard*) would benefit from a smarter PMU architectures that allow aggregating the sum of multiple PMU counters for regulation. However, such an improved PMU would still be limited, as it does not include data of other IP blocks such as the memory controller. Using polling, [11] shows that the aggregation of data from multiple sources is *necessary* to reduce the heavy pessimism in memory regulation caused by the spread in real bandwidth behavior. In any case, it would be beneficial for all types of regulators if hardware vendors provide PMU counters with fast access for outside agents at any level in the memory hierarchy and disclose information on how to use them.

With *MemPol*, we show a regulation that uses multiple PMU counters (read and write regulation) and even considers combined results of all cores for its global regulation. Furthermore, instead of relying on the pessimistic sustainable bandwidth metric, *MemPol*'s bandwidth redistribution of the global regulation can easily be extended to sample utilization of the memory controller if available on the platform (*e.g.*, [11]). Note that *MemGuard* also supports bandwidth redistribution, but its *bandwidth reclaiming mechanism* redistributes future budgets that it predicts will remain unused based on the history of per-core memory consumption. The approach offers no guarantees that a donating core can reclaim its budget when needed [7]. Compared to *MemGuard* with typical regulation periods of 1 ms, the 50 µs setting for *MemPol* may lead to a pessimistic control behavior for programs with memory-intensive phases that exceed the configured budgets. On the other hand, a low setting for w reduces the window for temporal interference with other bus masters. This is a trade-off that must be considered in the overall design, and requires profiling of the regulated applications.

We currently implement *MemPol* in software on one of the smaller real-time cores. However, the implementation is simple enough to be realized in hardware or in an FPGA. Compared to less flexible regulation approaches, (*e.g.*, Arm CCI-400 [9], which uses counters to bound bursts), *MemPol* requires storage for the execution history in the last w polling periods. In order to implement regulation at OS task level, window sizes and budgets on each core should change dynamically. The current implementation of the regulator supports such dynamism by considering budget updates in the next cycle of the control loop. However, penalties due to overshooting in previous cycles cannot be eliminated. In this work, we have not evaluated the impact of dynamically changing the sliding window size w at run-time.

Currently, *MemPol* throttles cores via debug interfaces. Arm documents the approach as a valid solution for self-hosted debugging in the A53 and A72 manuals [23], [27]. In our experiments, we did not observe any problems with, *e.g.*, atomic synchronization or idle management of the cores. However, debug interfaces seem to be second class citizens *w.r.t.* safety features, *e.g.*, the debug APB interface lacks ECC on the R5 cores [28]. Another limitation is that the debug interfaces provide no simple way for operating systems to disable throttling in critical sections. An alternative to the debug interface to throttle cores would be using regulation interrupts and poll—from a light-weight interrupt handler—the end of the throttling phase in a status register of the regulator. Another possibility is to combine both mechanisms, *e.g.*, use the debug interface to throttle cores for short blocking times and raise interrupts if longer blocking times are expected. This would allow an OS to handle interrupts during longer throttling phases, as incoming interrupts are queued in the interrupt controller when a core is halted in debug state and delivered when the core is released again. On Arm, the often unused FIQ interrupt would be a good candidate for interrupt-based throttling. While the ZCU102 platform provides means to send interrupts to the application cores from the R5 cores, we did not further evaluate this approach, as even a fast interrupt handler requires support from the operating system and causes memory accesses during execution. We leave as future work the evaluation of interrupt-based throttling and the fine-grained regulation at OS task-level. Finally, note that the lack of control mechanisms for an OS to disable throttling during critical sections and the inability to handle OS-level interrupts during throttling are shared by all *MemGuard* implementations at hypervisor level that we are aware of.

VI. Related Work

The problem of regulating memory interference on complex MPSoC platforms has received considerable attention and several software and hardware approaches have been proposed. While software-based approaches to memory regulation bene-

fit from greater flexibility and are widely applicable to existing commercial-off-the-shelf (COTS) platforms, hardware-based approaches are capable of higher control resolution and—given their vantage point view of the system—can precisely monitor and regulate memory traffic.

On the software side, the initial work on PMC-based regulation (*MemGuard*) [6], [7] has been followed by multiple studies [12], [29], [30], including implementations of *MemGuard* also at the hypervisor level to prevent modifications in the host OS, thus allowing for improved applicability. Notably, in [18] the *MemGuard* implementation for Linux[10] was extended to support separate regulation on read (cache-refills) or write (write backs) memory traffic for each core.

Performance counters can only provide an approximation of the load effectively generated on the interconnect and on the DRAM memory controller and the discrepancies between memory traffic generated by the CPUs and the utilization of the memory DRAM controller have been outlined in [10], [11]. In these works, actual memory utilization is determined via performance counters exposed by the memory-controller. Unfortunately, the internals of the memory controllers are rarely made available by hardware vendors [31], and only a limited subset of MPSoCs (mostly from NXP, *e.g.*, [13], [32]) exposes some PMCs for the memory controller.

The work in [11] shares similarities with ours as the memory utilization is periodically *sampled*. Nonetheless, standard *MemGuard*'s interrupts—and associated overheads—are used to regulate cores and to trigger the sampling.

In addition to PMCs, modern MPSoCs provide other QoS or monitoring features (*e.g.*, [33]). The works of [10], [17], [34], [35] have exploited such primitives to implement bandwidth regulation. Although effective, integrated platform monitors and regulators, *e.g.*, [9], only offer a pre-defined set of regulation possibilities, and—since they monitor at the platform interconnect level—make it complex to attribute monitored traffic to specific cores [10]. In parallel to PMC-based regulation, other approaches [36], [37] base their regulation strategy on worst-case memory budget *estimations* derived with offline analysis of statically known workloads.

On the hardware side, to enable higher monitoring resolution, the works of [38], [39] develop custom hardware components to implement bandwidth regulation directly at hardware level, while [40] implements an FPGA module to monitor and regulate different types of requests simultaneously. This proposal was also deployed on a prototype RISC-V design [41]. Adaptations for the memory controller have been proposed in [42]–[45] to reduce the worst-case latency of memory requests under multicore contention. Time Division Multiplexing hardware implementations have also been proposed in [46]–[49] to improve predictability of the memory interconnect level. On MPSoCs (*e.g.*, [2]) that feature an on-chip programmable logic, [50] proposed an architecture to schedule individual memory transactions by redirecting CPU memory traffic through the FPGA, while an FPGA-based closed-loop controller is proposed in [51].

Architecture-level features such as Arm's MPAM [52] or Intel's RDT [53] aim to deliver improved (QoS) control over the memory subsystem, but their availability on current systems is still very limited. Furthermore, in the case of Arm MPAM, all its control interfaces are defined as optional and it is therefore unclear, which controls will be available in actual implementations.

In addition to bandwidth regulation, cache partitioning techniques [54]–[56] and bank-level partitioning [57] have been also successfully used to mitigate core-interference at cache and DRAM level respectively. Notably, hardware support for cache partitioning is offered on recent MPSoC such as [5] as part of Arm's DynamicIQ [58].

An empirical characterization of memory interference for different NVIDIA-based boards is presented in [59], [60].

VII. Conclusion

We presented *MemPol*, a novel approach for bandwidth regulation of application cores in today's MPSoCs. *MemPol* enables low-overhead regulation by polling PMU counters from an external processing unit, such as the R5 core on the Xilinx UltraScale+ ZCU102, throttles cores using on-chip debug facilities, and uses an on-off controller design with a sliding window technique to control burstiness. *MemPol* can regulate based on the simultaneous contribution of multiple PMU counters and provides a combination of per-core regulation and global regulation of all cores that allows redistributing unused bandwidth between cores, while keeping the overall memory bandwidth below a given global threshold.

Compared to state-of-the-art PMC-based regulations (*e.g.*, *MemGuard*), *MemPol*: (1) has a more accurate cost model that considers multiple PMU counter for regulation, (2) does not generate timer or PMU interrupt overheads for application cores, and (3) employs a fine-grained microsecond-scale bandwidth regulation allows better cooperation with hardware-based QoS schemes, *e.g.*, in the Arm CCI-400 [9], and prevents starvation of other bus-masters.

The shown implementation focuses on per-core regulation, similar to *MemGuard* implementations found in hypervisors, but can be extended towards regulation at task level as well by including interrupt-based notification to the OS to enforce throttling. We leave an implementation of this for future work.

The presented regulation mechanism is challenging in multiple ways. An on-off-based controller design has to cope with overshooting of memory budgets, delays in the control paths, and unknown behaviors of applications' memory access patterns at a microsecond scale. However, we see this work as a starting point for further research in regulation mechanisms from outside the cores.

Acknowledgments

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799 and CNS-2238476.

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

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

REFERENCES

[1] T. Lugo, S. Lozano, J. Fernandez, and J. Carretero, "A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.

[2] Xilinx, "ZCU 102 MPSoC TRM," https://www.xilinx.com/support/ documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[3] ——, "Xilinx Versal," https://www.xilinx.com/products/silicon-devices/acap/versal.html Accessed: 2022-05-07.

[4] NVIDIA, "NVIDIA Jetson AGX Xavier," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/ Accessed: 2022-05-07.

[5] ——, "NVIDIA Jetson AGX Orin," https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-orin/ Accessed: 2022-05-07.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[7] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, p. 562–576, 2016.

[8] ARM, "Arm Architecture Reference Manual for A-profile architecture," https://developer.arm.com/docs/ddi0487/ Accessed: 2022-05-07.

[9] ——, "ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service," https://developer.arm.com/docs/dsu0026/ Accessed: 2022-05-07.

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

[11] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores ," in *2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, p. 133–145.

[12] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657.

[13] NXP, "NXP S32G," https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32g-vehicle-network-processors/s32g2-processors-for-vehicle-networking:S32G2 Accessed: 2022-05-07.

[14] R. Pi, "Raspberry Pi 4," https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ Accessed: 2022-05-07.

[15] NXP, "NXP LX2160A," https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-lx2120a-lx2080a-processors:LX2160A Accessed: 2022-05-07.

[16] G. Schwaericke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A Real-Time virtio-based Framework for Predictable Inter-VM Communication," in *2021 IEEE International Real-Time Systems Symposium (RTSS)*, 2021.

[17] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 3:1–3:26. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13934

[18] M. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, p. 357–367.

[19] J. Y. Moon, D. Y. Kim, J. H. Kim, and J. W. Jeon, "The Migration of Engine ECU Software From Single-Core to Multi-Core," *IEEE Access*, vol. 9, pp. 55 742–55 753, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3071500

[20] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," in *2017 Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.

[21] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A Predictable Execution Model for COTS-Based Embedded Systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, p. 269–279.

[22] ARM, "Arm CoreSight Architecture Specification," https://developer.arm.com/docs/ihi0029/ Accessed: 2022-05-07.

[23] ——, "Arm Cortex-A53 MPCore Processor Technical Reference Manual," https://developer.arm.com/docs/ddi0500/ Accessed: 2022-05-07.

[24] Z. Ning, C. Wang, Y. Chen, F. Zhang, and J. Cao, "Revisiting ARM Debugging Features: Nailgun and Its Defense," *IEEE Transactions on Dependable and Secure Computing*, no. 01, pp. 1–16, 2021.

[25] ARM, "AMBA APB Protocol Specification," https://developer.arm.com/docs/ihi0024/ Accessed: 2022-05-07.

[26] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 55–64.

[27] ARM, "Arm Cortex-A72 MPCore Processor Technical Reference Manual," https://developer.arm.com/docs/100095/ Accessed: 2022-05-07.

[28] ——, "Cortex-R5 Technical Reference Manual," https://developer.arm.com/docs/ddi0460/ Accessed: 2022-05-07.

[29] N. Dagieu, A. Spyridakis, and D. Raho, "Memguard: A memory bandwith management in mixed criticality virtualized systems memguard KVM scheduling," in *10th Int. Conf. on Mobile Ubiquitous Comput., Syst., Services and Technologies (UBICOMM)*, 2016.

[30] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, ser. OpenAccess Series in Informatics (OASIcs), M. Bertogna and F. Terraneo, Eds., vol. 77. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 3:1–3:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/11779

[31] F. Rehm, J. Seitter, J.-P. Larsson, S. Saidi, G. Stea, R. Zippo, D. Ziegenbein, M. Andreozzi, and A. Hamann, "The Road towards Predictable Automotive High - Performance Platforms," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, p. 1915–1924.

[32] NXP, "NXP S32V234SBC," https://www.nxp.com/design/development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v2-vision-and-sensor-fusion-low-cost-evaluation-board:SBC-S32V234 Accessed: 2022-05-07.

[33] ARM, "Quality of Service in ARM Systems: An Overview," https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/posts/quality-of-service-in-arm-systems-an-overview Accessed: 2022-05-07.

[34] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on ARM-based heterogeneous platforms," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, 2017, pp. 1297–1302.

[35] M. Zini, G. Cicero, D. Casini, and A. Biondi, "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms," *Software: Practice and Experience*, vol. 52, no. 5, pp. 1095–1113, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3053

[36] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, p. 2:1–2:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7174

[37] J. Flodin, K. Lampka, and W. Yi, "Dynamic budgeting for settling DRAM contention of co-running hard and soft real-time tasks," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, 2014, p. 151–159.

[38] Y. Zhou and D. Wentzlaff, "MITTS: Memory Inter-Arrival Time Traffic Shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 532–544. [Online]. Available: https://doi.org/10.1109/ISCA.2016.53

[39] F. Farshchi, Q. Huang, and H. Yun, "BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 364–375.

[40] J. Cardona, C. Hernández, J. Abella, and F. J. Cazorla, "Maximum-contention control unit (MCCU): resource access count and contention time enforcement," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 710–715. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715155

[41] N.-J. Wessman, F. Malatesta, J. Andersson, P. Gomez, M. Masmano, V. Nicolau, J. Le Rhun, G. Cabo, F. Bas, R. Lorenzo, O. Sala, D. Trilla, and J. Abella, "De-RISC: the first RISC-V space-grade platform for safety-critical systems," in *2021 IEEE Space Computing Conference (SCC)*. IEEE, 2021, pp. 17–26.

[42] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "DRAMbulism: Balancing Performance and Predictability through Dynamic Pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 82–94.

[43] M. Hassan, H. Patel, and R. Pellizzoni, "PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, may 2017. [Online]. Available: https://doi.org/10.1145/3019611

[44] P. K. Valsan and H. Yun, "MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems," in *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, 2015, pp. 86–93.

[45] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, p. 251–256.

[46] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the Shackles of Time-Division Multiplexing," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 456–468.

[47] M. Jun, K. Bang, H.-J. Lee, N. Chang, and E.-Y. Chung, "Slack-based Bus Arbitration Scheme for Soft Real-time Constrained Embedded Systems," in *2007 Asia and South Pacific Design Automation Conference*, 2007, pp. 159–164.

[48] Y. Li, K. Akesson, and K. Goossens, "Architecture and analysis of a dynamically-scheduled real-time memory controller," *Real-Time Systems*, vol. 52, no. 5, p. 675–729, Sep. 2016.

[49] A. Kostrzewa, S. Saidi, and R. Ernst, "Slack-based resource arbitration for real-time Networks-on-Chip," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1012–1017.

[50] D. Hoornaert, S. Roozkhosh, and R. Mancuso, "A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:22. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13933

[51] J. Freitag and S. Uhrig, "Closed Loop Controller for Multicore Real-Time Systems," in *Architecture of Computing Systems – ARCS 2018*, M. Berekovic, R. Buchty, H. Hamann, D. Koch, and T. Pionteck, Eds. Cham: Springer International Publishing, 2018, p. 45–56.

[52] ARM, "Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A," https://developer.arm.com/docs/ddi0598/ Accessed: 2022-05-07.

[53] Intel, "Resource Director Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html Accessed: 2022-05-07.

[54] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, p. 45–54.

[55] Xilinx, "Xilinx Xen Support with Cache-Coloring," https://github.com/Xilinx/xen/releases/tag/xilinx-v2020.2 Accessed: 2022-05-07.

[56] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, p. 1–14.

[57] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, p. 155–166.

[58] ARM, "Arm DynamIQ Shared Unit-AE Technical Reference Manual Revision," https://developer.arm.com/docs/101322/ Accessed: 2022-05-07.

[59] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, "Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms," in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2020, p. 1–10.

[60] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, p. 1–10.