MemPol: Polling-Based Microsecond-Scale Per-Core Memory Bandwidth Regulation

Alexander Zuepke[1*], Andrea Bastoni[1], Weifan Chen[2], Marco Caccamo[1], Renato Mancuso[2]

[1*]Chair of Cyber-Physical Systems in Production Engineering, Technical University of Munich, Boltzmannstraße 15, 85748 Garching, Germany.
[2]Cyber-Physical Systems Lab, Boston University, 665 Commonwealth Avenue, Boston, MA 02215, USA.

*Corresponding author(s). E-mail(s): alex.zuepke@tum.de;
Contributing authors: andrea.bastoni@tum.de; wfchen@bu.edu;
mcaccamo@tum.de; 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. Memory regulation via throttling is one of the most practical techniques to mitigate interference. 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 conduct an in-depth evaluation on the Xilinx Zynq UltraScale+ ZCU102, NXP i.MX8M and NXP S32G2 platforms using the San Diego Vision Benchmark Suite.

Keywords: real-time system, multi-core, memory bandwidth regulation, feedback control

1 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 (Lugo et al., 2022). 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 (Xilinx, 2024b,a) or NVIDIA's Jetson AGX Xavier and Orin (NVIDIA, 2024b,a) 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 (Yun et al., 2013; Yun et al., 2016) 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 (ARM, 2016a; Xilinx, 2024b), interconnect (ARM, 2016b), and memory controller (Sohal et al., 2020; Saeed et al., 2022). Third, it forces to integrate additional system-level software components at the OS (Yun et al., 2013) or hypervisor level (Modica et al., 2018; Sohal et al., 2020), 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.
- A detailed evaluation that includes the assessments of key memory parameters for three Arm platforms and a comparison of *MemPol* with the state-of-the-art.

This paper is an extended version of a previously published work at the RTAS 2023 conference (Zuepke et al., 2023).

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 range of Arm platforms, namely on the Xilinx Zynq UltraScale+ ZCU102 (Xilinx, 2024b), the NXP i.MX8M (NXP, 2024a) and the NXP S32G2 (NXP, 2024c) platforms. All platforms feature four Arm Cortex-A53 application cores, but also a number of smaller Arm-based real-time cores. *MemPol* is deployed on one of the real-time cores and regulates the application cores with 6.25 to 10 μs granularity.

For each platform, we precisely characterized the *sustainable bandwidth* using a practical, empirical methodology to measure it. We further correlate the sustainable bandwidth to the *MemPol* regulation parameters and the associated cost model.

Although questionably suitable for certified environments, we have validated the practical feasibility of our debug-based methodology (see Sec. 4.2) on multiple Arm-based boards such as Raspberry Pi 4 (Raspberry Pi Ltd, 2024) and NXP LX2160A (NXP, 2024b) which feature Arm Cortex-A72 application cores, but lack small real-time cores. 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. 2 discusses limitations of *MemGuard* designs and proposes alternatives. Sec. 3 presents the new regulator design, and Sec. 4 its implementation. Sec. 5 assesses the sustainable bandwidth on our platforms and derives parameters for *MemPol* regulation. Sec. 6 evaluates *MemPol* and compares to the state-of-the-art. Sec. 7 discusses related work, and Sec. 8 concludes.

2 Background and Motivation

This section summarizes the key aspects of PMC-based regulation—with focus on its most common variant, *MemGuard* (Yun et al., 2013; Yun et al., 2016)—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 pre-defined 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 (Yun et al., 2016; Schwaericke et al., 2021) or from the DRAM memory controller (Sohal et al., 2020; Saeed et al., 2022). Since overutilization of memory controllers is detrimental to predictability (Sohal et al., 2020), 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 (see Sec. 5). When DRAM controller performance counters are not available, determining this value requires know-how of the target platform and non-trivial experimental setups (Serrano-Cases et al., 2021; Schwaericke et al., 2021).

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 (Yun et al., 2016; Saeed et al., 2022), or by stalling the core at the hypervisor level (Sohal et al., 2020; Schwaericke et al., 2021). 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 integration 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.

2.1 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.

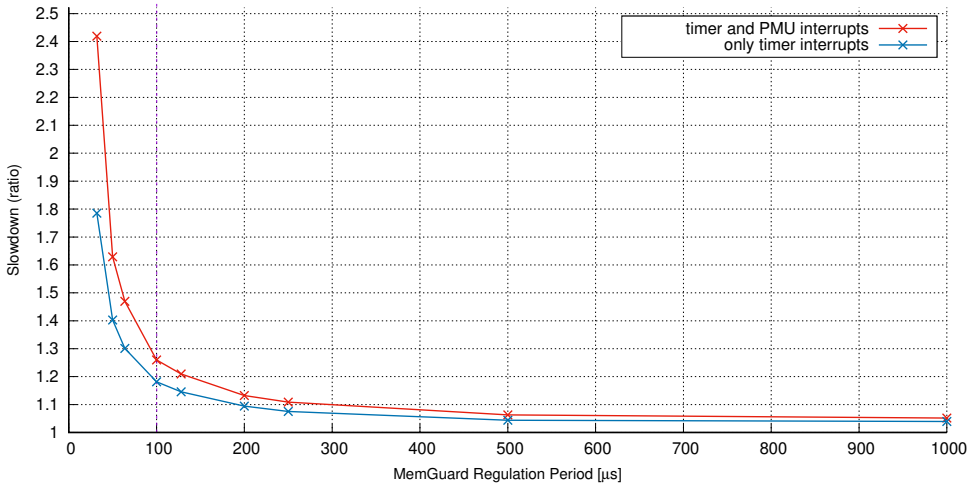

Fig. 1 Impact (slowdown) of *MemGuard*'s timer and regulation overheads on a memory-intensive application as a function of the replenishment period. Implementation on Linux on the Xilinx Zynq UltraScale+ ZCU102 (Xilinx, 2024b). Results are in line with other work (Yun et al., 2016; Saeed et al., 2022) and extended beyond 100 µs.

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. 6). 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 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 (Yun et al., 2016; Saeed et al., 2022) 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.

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

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 (Sohal et al., 2020). 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. 6.2).

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 (Sohal et al., 2020) and in an uneven distribution of memory bandwidth within the period. This complicates the adoption of, *e.g.*, automotive techniques (Moon et al., 2021) that use offsetting to distribute the peak load of read-execute-write (Hamann et al., 2017; Pellizzoni et al., 2011) workloads over successive periods. Moreover, as mentioned in Sec. 1, it can cause accelerators to receive less bandwidth than their assigned quota.

2.2 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 (Sohal et al., 2020; Saeed et al., 2022) or regulation (Yun et al., 2016; Bechtel and Yun, 2019)—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. 4.2), 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. 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. 3 presents *MemPol*, a poll-based regulation design that operates from outside the cores and regulates multiple monitoring dimensions with low overhead.

[2] In Bechtel and Yun (2019), 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.

3 MemPol – Regulation from Outside the Cores

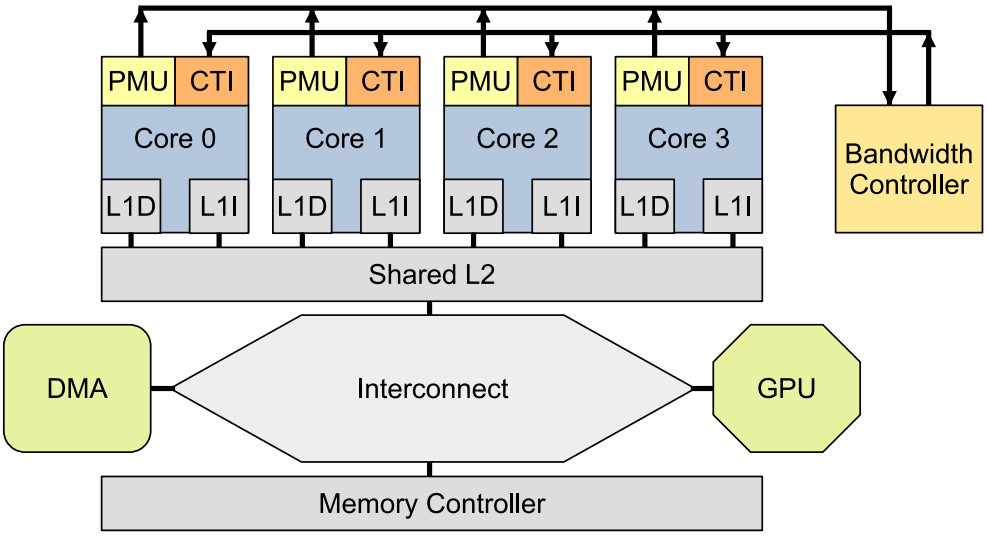

Fig. 2 *MemPol* architecture. Applications cores c_0 to c_3 are regulated by an external controller logic that accesses the application cores' PMU counters as memory-mapped devices and that halts the cores via their debug interfaces.

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. 4.2) 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. 3.1). 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. 3.2).

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. 3.3) 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. 3.4) and a combined strategy to

account for non-memory-intensive phases (Sec. 3.5). 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. 3.6).

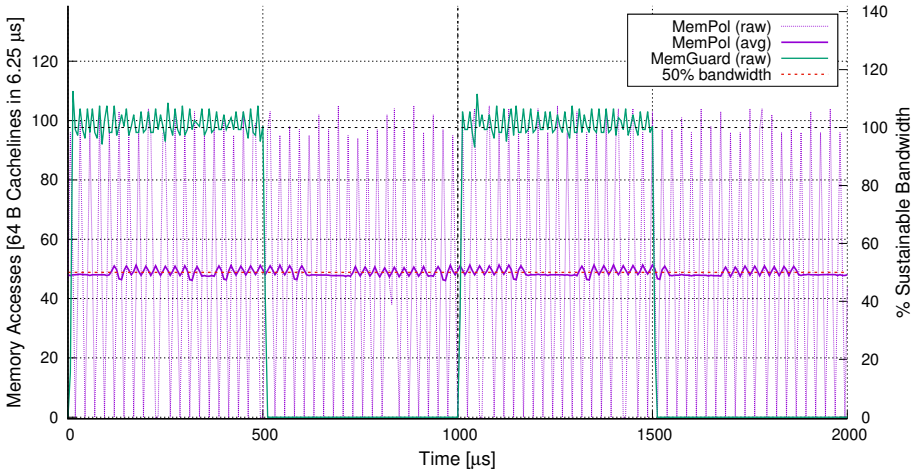

Fig. 3 Comparison of the regulation behavior of *MemPol* (polling at 6.25 µs, sliding window size 50 µs) and *MemGuard* (regulation period 1 ms) on ZCU102 regulating a worst-case memory reader at 50% sustainable memory bandwidth. In both cases, PMU counters are sampled every 6.25 µs. For *MemPol*, the 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.

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. 3.7). 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. 6.1 for details.)

3.1 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 (see Sec. 5). 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 (Sohal et al., 2020). 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 $A_i^{budget} = B_i * P$.

3.2 Overshooting

In *MemGuard*, the 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 (*e.g.*, see Sec. 5). 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, sending a *halt* request to the core, and the point where a core actually *stops* issuing further memory requests. We denote this delay as D and assume that the core stops in reasonable time $D \leq P$ within the polling period P (see Sec. 4.3). The product $2\beta_i$ then describes the worst-case overshooting when a core c_i accesses memory at peak bandwidth and exceeds its budget at the beginning of P, but takes to the beginning of the next period to halt.

3.3 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}.

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

3.4 Sliding Window Technique to Control Burstiness

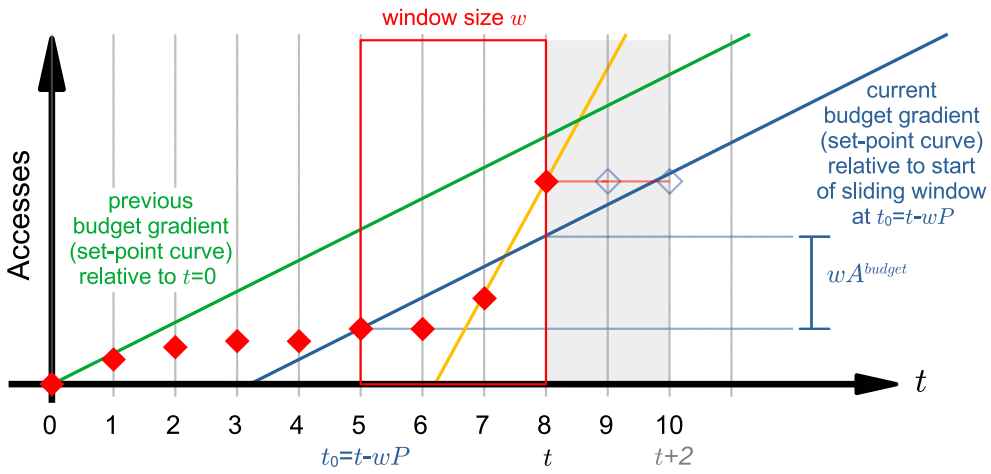

Fig. 4 Sliding window technique. At time t=8, the burst (yellow gradient) is within a previous budget gradient from time t=0 (green gradient), but not within the current budget gradient at the start of the sliding window at time t=5 (blue gradient). Based on its recent history in $(t-wP, t)$ (red box), the core will be rate-limited for at least two periods in $(t, t+2)$. See Sec. 3.4.

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. 3.3 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 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).

3.5 Resulting Combined Control Strategy

MemPol's controller combines the strategies from Sec. 3.3 and Sec. 3.4 depending on the behavior in the previous w polling periods.

Not rate-limited. A sliding window (Sec. 3.4) 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

Algorithm 1: Controller implementation (Sec. 3.5)

```
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 (0..w−1) or (initially) not (w)
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 rate-limiting
14 |  else                                        ◁ non rate-limited mode
15 |  |  spv = hist[i] + w * A_i^budget           ◁ spv from start of sliding window
16 |  end
17 |  val = α_r * pmc_r + α_w * pmc_w             ◁ read PMCs and apply weighting
18 |  delta = val - spv                           ◁ signed delta value, integer overflow
19 |  if delta > 0 then                           ◁ PMC above set-point value, throttle
20 |  |  t_lrt = 0                                ◁ (re-)start aging of rate-limiting
21 |  |  spv_lrt = spv                            ◁ further budgeting based on spv
22 |  |  hist[i] = spv_lrt                        ◁ update history with rate-limited value
23 |  |  throttle()                               ◁ halt core if running
24 |  else          ◁ PMC below set-point value, resume or keep running
25 |  |  hist[i] = val                            ◁ update history with current value
26 |  |  resume()                                 ◁ resume core if halted
27 |  end
28 |  i = (i + 1) MOD w                           ◁ select next position in history data
```

(Sec. 3.3) 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 (Lines 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 (Lines 11–13) from the variable spv_{lrt} set at the start of rate-limiting (Line 21). Otherwise, spv is set to the history value at the start of the sliding window (Line 15). Afterwards, the controller samples the current

PMC value (Line 17). If the PMC value is above spv, the controller enters rate-limiting mode (Lines 20–23): 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 (Lines 24–26).

3.6 Setting Regulator's Budgets

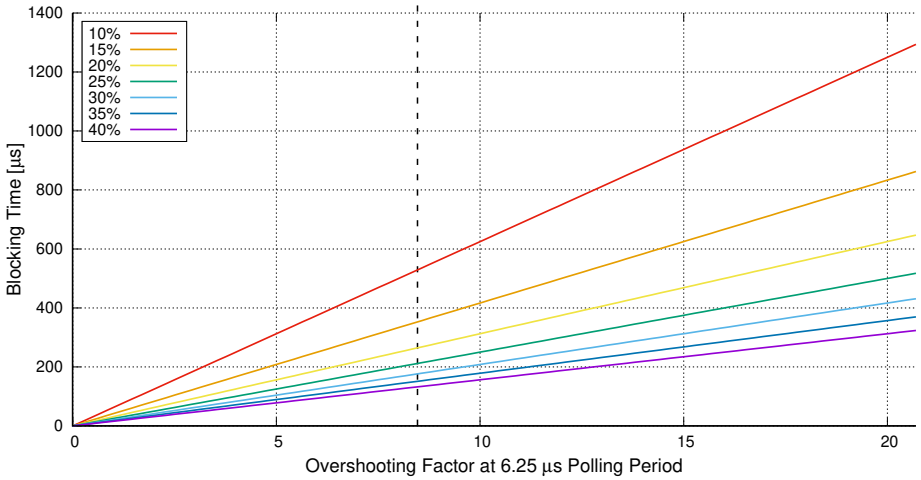

Fig. 5 Overshooting in relation to $B_{sustainable}$ by a certain factor (x axis) and the resulting blocking time (y axis) for different bandwidth levels (%) in a regulation at 6.25 µs. Lower bandwidth levels observe higher blocking times. The maximum observed overshooting in relation to $B_{sustainable}$ on the ZCU102 is factor 8.46 (dotted vertical line), see Sec. 5.2.

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 $2\beta_i P$. 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, as used in our regulation on the ZCU102 (see Sec. 5.2), this corresponds to 625 µs blocking time. The maximum overshooting factor normalized to $B_{sustainable}$ observed in our experiments was $\beta =$ 8.46 on the ZCU102 (see Sec. 5.2), 11.08 on the i.MX8M (see Sec. 5.3), and 6.51 on the S32G2 (see Sec. 5.4).

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.

3.7 Combined Local Per-Core and Global Regulation

The logic presented in Sec. 3.1–3.5 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.

3.8 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. 3.4 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.

4 Implementation

Before explaining the main components of *MemPol*, we briefly summarize the relevant features of the Arm architecture and the commonalities of the platforms that have been used for our implementations on the Xilinx Zynq UltraScale+ ZCU102 (Xilinx, 2024b), the NXP i.MX8M (NXP, 2024a), and the NXP S32G2 (NXP, 2024c).

4.1 SoC Architecture and CoreSight Debugging Capabilities

Our target platforms include four Arm Cortex-A53 *application processor (AP)* cores and additionally one or more Arm Cortex-M or Cortex-R *real-time processor (RP)* cores. The AP cores feature private L1 caches and a shared L2 cache (LLC) and reside in the full-power domain (1 GHz speed or faster) of the SoC. The RP cores are connected to the low-power domain (200–500 MHz speed) of the SoC and have access to private tightly-coupled memories (TCM). A central cache-coherent interconnect (*e.g.*, Arm CCI-400) connects the low- and full-power domains and the main memory controller(s).

Arm defines a common infrastructure (*CoreSight*) for hardware debugging of its cores (ARM, 2017). CoreSight specifies registers of memory-mapped debug devices on a low-bandwidth APB bus that can be accessed through a *debug access port (DAP)*. Additionally, the CoreSight infrastructure is accessible for on-chip debugging via the low-power domains on most Arm SoCs. To debug devices connected to CoreSight, the typical setup comprises per-core debug interfaces, performance counters (PMU), trace interfaces, cross trigger interfaces (CTI), and a shared cross trigger matrix (CTM) (ARM, 2018a, 2016a). The CTI exposes core-specific input signals to halt and resume a core, and an output signal to indicate that the core triggered a halting condition. The CTM connects the input and output signals from the CTIs of the cores and allows halting multiple cores on a debug event in a synchronized manner.

The memory-mapped debug interface configures debug trigger conditions, such as breakpoints and watchpoints. It also provides access to a bi-directional debug communication channel register and allows the injection of instructions into the pipeline once the core is halted. A debugger obtains indirect access to the core's registers by injecting instructions to load or store the core's current registers from or to the debug communication channel register. Being at the highest privilege level, the debugger has access to all of the core's registers. Similarly, information provided by performance counters can also be controlled by the memory-mapped PMU interface. Arm mentions the workflows for debugging by an external hardware debugger or by a *self-hosted* software debugger running on other cores (ARM, 2016a).

4.2 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 or resume request requires write transactions to the CTI's *trigger pulse* register, and to the CTI's *trigger acknowledge* register for the acknowledgment of a previous debug request. We discovered experimentally that polling for previous requests is not required if there is a sufficient delay between the writes to the acknowledge register and the

trigger register to resume the core. This reduces the number of required memory transactions for a halt-resume cycle to three writes to CTI registers: trigger halt, acknowledge, and trigger resume. In any case, access to the core's debug interface is not needed, as the core's state is not to be modified.

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. 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. Likewise, the Arm documentation mentions that cache- and memory-related PMCs do not impact a core's execution behavior (ARM, 2016a). This allows for *interference-free remote monitoring*.

4.3 MemPol Regulator

We implemented the regulator on one of the real-time cores on the specific SoCs. The regulator exposes a memory-mapped interface in the TCM of its core. Following the design of hardware registers, this 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. This allows inspecting and debugging the regulator's state from the AP cores. For tracing purposes, we used the remaining TCM 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.

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 (trigger nothing + trigger nothing). We further interleave the CTI accesses of all cores, *i.e.*, perform the first CTI transactions for $c_0..c_3$, then followed by the second CTI transactions for $c_0..c_3$. This pattern and the dummy writes ensure a similar execution time in each regulation loop and 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 on the ZCU102 where a Cortex-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 (ARM, 2018a). Since all four cores can have outstanding transactions, we assume the worst-case halt delay to be at most 1.5 µs on the ZCU102. In our experiments, we observed a delay of around 1 µs.

The regulator is implemented in a bare-metal C application and compiled to Arm Thumb (Cortex-M) or Arm code (Cortex-R). The implementation requires between 4 and 8 KB code (the larger version includes formatted console output and tracing), 3 KB of data (controller state), and 1 KB stack. Code and data of the regulator is kept in the TCM of the RP, so instruction fetches and data accesses of the regulator do not cause memory interference to the APs. The regulator uses standard 32-bit integer arithmetic and multiplication; no division is needed.

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 specific regulator implementation on our platforms (see Sec. 5). Mapping the CoreSight registers as *shared device*, rather than using a *uncached strongly-ordered* mapping, significantly speeds up write operations, as regulator core does not need to wait for transactions to complete. This allows the writes to the CTI registers to be queued and serialized by the interconnect next to the APB bus rather than the core. We place a `DSB` memory barrier instruction at the end of the control loop to reduce jitter in the control loop. This ensures that any outstanding writes to CTI registers have finished before starting a new round and reading from the PMU.

4.4 Side Effects

We have observed the following side effects when using *MemPol*.

Deeper CPU idle modes. Access to the CoreSight registers require that the Cortex-A53 cores are online. This interferes with the power management subsystem of the Linux kernel which turns cores off in deeper power saving modes. Unfortunately, this takes the cores' CoreSight registers offline as well. This causes any access to the core's CoreSight registers to either fail with a data abort exception or get stuck. We therefore have to disable any deeper power saving modes beyond the `WFI` instruction to idle the cores.[5] We do not consider this to be a problem for real-time systems that need memory bandwidth regulation, as waking up from deeper power saving modes increases interrupt latencies and is therefore typically disabled.

Freezing system timer in debug mode. Cores entering debug halt state might also halt the global system timer that drive the cores' private virtual and physical timer interrupts. Halting the time and related timer interrupts is a handy feature for system software development when using an external hardware debugger, however this feature interferes with time keeping of the cores when *MemPol* is used. Likewise, other peripherals can change their behavior in debug mode as well. This behavior depends on the SoC and needs to be disabled in the specific peripherals. We also do not consider

[5] E.g. `echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/state1/disable`.

this to be a problem when using *MemPol*, as any problems with non-working timer interrupts and I/O show early during testing.

External Hardware Debugging. The setup of CTI and PMU requires taking ownership of the debug interface by disabling software lock registers and then configuring the devices. This interferes with any external hardware debugger that also claims these devices. We have not fully tested hardware debugging together with *MemPol*, but using an external hardware debugger will likely interfere with the regulation. For example, the integrated logic analyzer (ILA) for FPGA development on the ZCU102 takes priority when using the SoC's debugging features and disables *MemPol*'s capabilities to halt or resume cores.

SoC Debugging and TrustZone. *TrustZone* is a feature of Arm processors that introduces secure and non-secure execution modes of the cores and related access bits for all components in an SoC ARM (2016a). This allows to fully isolate security-sensitive software in the SoC, while Linux or an RTOS run in non-secure mode. To separate debugging of secure from non-secure components down to the hardware level, the Arm architecture defines an authentication interface of four signals for invasive / non-invasive debugging in secure / non-secure execution state. Access to the CTI and PMU registers requires at least the invasive resp. non-invasive debugging of non-secure execution state (`DBGEN`, `NIDEN`) to be enabled. Monitoring and debugging in secure execution state (*TrustZone* mode) is instead enabled by `SPIDEN` and `SPNIDEN` signals. We have not tested *MemPol* with TrustZone, and we do not consider regulating secure applications to be relevant for real-time use cases, as TrustZone introduces additional jitter and interference in the caches. Note that *MemGuard* faces similar challenges in setting up PMU counters to monitor secure applications from a non-secure hypervisor or operating system. See Ning et al. (2021) for further details on the security impact of on-chip monitoring and debugging facilities.

5 Platform Assessment and Sustainable Bandwidth

We now evaluate our platforms *w.r.t.* their sustainable bandwidth and their CoreSight register access timing to derive platform-specific settings for the *MemPol* regulation.

5.1 Determining the Sustainable Bandwidth

We use a dedicated benchmark to evaluate the sustainable memory bandwidth of the platforms.[6] Similar to the *USTRESS* benchmark (Sohal et al., 2020), the benchmark probes the memory bandwidth of the DRAM memory controller with different memory access patterns and increasing step sizes over a large memory buffer.

As the memory controller reads and writes memory in units of full cachelines, the benchmark issues various *read*, *write* and *modify* operations on cachelines. The difference between *write* and *modify* operations is that *write* operations always write to full cachelines, while *modify* operations only update a part of a cacheline, *e.g.*, by overwriting just a single byte. Arm CPUs detect full writes to cachelines and in this case suppress fetching cachelines from the memory controller (ARM, 2018a).

[6] The benchmark is available at https://gitlab.com/azuepke/bench.

Therefore, *read* and *write* operations stress the read and write performance of the memory controller independently, while a large number of *modify* operations eventually leads to an interleaved *read/write* pattern once all cachelines in the caches become dirty, as for each modification a new cachelines is read and an older one is written back. The interleaved *read/write* pattern additionally stresses the internal scheduling capabilities of the DRAM controller, which prioritizes reads over writes, leading to worst-case scenarios. Lastly, by increasing the step size of memory accesses in power-of-two steps, the benchmark probes specific bits of the physical addresses to trigger the worst-case behavior of DRAM, *i.e.*, row misses in the same DRAM bank. The recent work of Fernandez-De-Lecea et al. (2023) provides a comprehensive overview on the multicore interference effects in DRAM controllers.

We obtain the sustainable bandwidth results by running the benchmark on Linux. Except for the default processes by the specific distributions, the Linux system is mostly idle. No graphical user environment is running. We disabled power-saving[7] and configured each system to support 128 MiB of huge pages.[8] The benchmark is pinned to the first CPU. We let the benchmark test different memory access patterns for 10 seconds each on a 32 MiB sized memory buffer that is mapped using 2 MiB huge pages.[9]

Figures 6, 7, and 8 show the results of the benchmark runs on our platforms. Straight lines show the observed memory bandwidth on the CPU core, while dotted lines show the sum of the two PMCs relevant for bandwidth regulation (see Sec. 4.3).

The benchmark performs three types of *read* operations, namely *load* using normal load instructions, *ldnp* using non-temporal loads, and *prfm L1* using prefetches to the L1 cache (`PRFM PLDL1KEEP` instruction). Prefetches to the L2 cache (not shown) yield similar results. Prefetches achieve much read higher performance in general, as they don't block the pipeline and get handled by the memory subsystem in the background.

Likewise, the benchmark performs three types of *write* operations (to full cache-lines), *store* using normal store instructions, *stnp* using non-temporal stores, and *dc zva* using the data cache zero instruction. The different types of stores show similar performance characteristics. However, the figures show that the selected PMCs `0x17` for L2 data cache refills and `0x18` for L2 data cache write-backs slightly undercount (dotted lines) the bandwidth observed at the core.

Lastly, the benchmark performs two types of *modify* operations by using normal store (*mod*) and non-temporal store (*mod stnp*) instructions. As expected, figures show that the PMC bandwidth is twice as high as the one at the core, since *modify* comprises both *read* and *write* operations.

The results on all three platforms show that the achievable memory bandwidth drops when the step size increments, until it plateaus at a specific minimum bandwidth (the empirically obtained *sustainable bandwidth*). The results then slightly increase again at step sizes 131072 and 262144. This is most likely a side effect of the benchmark as the number of accessed cachelines shrink in each increment and cache hits become more likely.

[7] We set the `scaling_governor` setting of all CPUs to `performance`.
[8] `sysctl -w vm.nr_hugepages=64`
[9] `bench --delay 10000 --size 32 --huge --perf --cpu 0 --auto --all --csv x.csv`

Running multiple instances of the benchmark on each CPU in parallel confirms that the memory controller is the bottleneck, rather than the CPU cores, the interconnect, or the caches.

Our selection of α_r and α_w parameters for the regulation is guided by the differences in achievable sustainable bandwidth shown by *read* and *write* operations. For example, if *writes* show a significantly lower bandwidth behavior than *reads*, we want the regulator to penalize *write*-heavy applications over *read*-heavy ones, and adjust the two factors inversely proportional to their bandwidth. In practice, we keep $\alpha_r = 1$ and increase $\alpha_w > 1$ accordingly to compensate for the heavier impact of the *writes*. This results in a simple linear model of bandwidth usage for both *reads* and *writes*. Note that the factors can be set differently, *e.g.*, to account for possible denial-of-service attacks on the writeback buffers in the shared cache (Bechtel and Yun, 2019), although we haven't conducted further evaluations on this aspect.

We discuss the individual results in the following sections.

5.2 Xilinx Zynq UltraScale+ ZCU102

The Xilinx Zynq UltraScale+ ZCU102 (Xilinx, 2024b) is a revision 1.0 board equipped with a *zu9eg* SoC and 4 GiB DDR4 RAM. Each Cortex-A53 core has separate 32 KiB L1 caches for instruction and data. The four APs are configured in a single cluster configuration and share 1 MiB of L2 cache. Next to the APs running at 1.2 GHz, the SoC provides two Cortex-R5 RPs running at 500 MHz. Each Cortex-R5 core is equipped with 128 KiB of local memory (TCM). The SoC additionally includes a programmable logic (PL) part (an FPGA) that is not used by our experiments. We include the regulator in the `BOOT.BIN` file of the system and load the regulator on the first Cortex-R5 core at boot time. We further use the PetaLinux 2021.1 distribution provided by Xilinx with Linux kernel 5.4.

5.2.1 ZCU102 Bandwidth Assessment

The bandwidth assessment in Fig. 6 shows a peak read bandwidth of $B_{peak-core,r} =$ 4393 MB/s (*prfm L1*) and a peak write bandwidth $B_{peak-core,w} =$ of 8460 MB/s (*store*). We also observe an undercounting of write operations in PMCs of about 3% (dotted lines). However, with an increment of 128 KiB, we observe a sustainable bandwidth of 1027 MB/s for reading, 985 MB/s for writing and 483 MB/s for *modify*. Because read and write bandwidths are within 5% difference, we assume a single sustainable memory bandwidth value of $B_{sustainable} \approx 1000$ MB/s (954 MiB/s) for the ZCU as simplification and to improve readability. Then fractions of the bandwidth then compute nicely to bandwidth values, *e.g.*, 20% is 200 MB/s.

These results are in line with previously reported performance metrics of the same platform (Schwaericke et al., 2021). We observe a slightly lower bandwidth on a second ZCU102 board in our lab that is equipped with different DRAM (*read* 1015 MB/s, *write* 935 MB/s, *modify* 478 MB/s, slow-down already at 64 KiB step size).

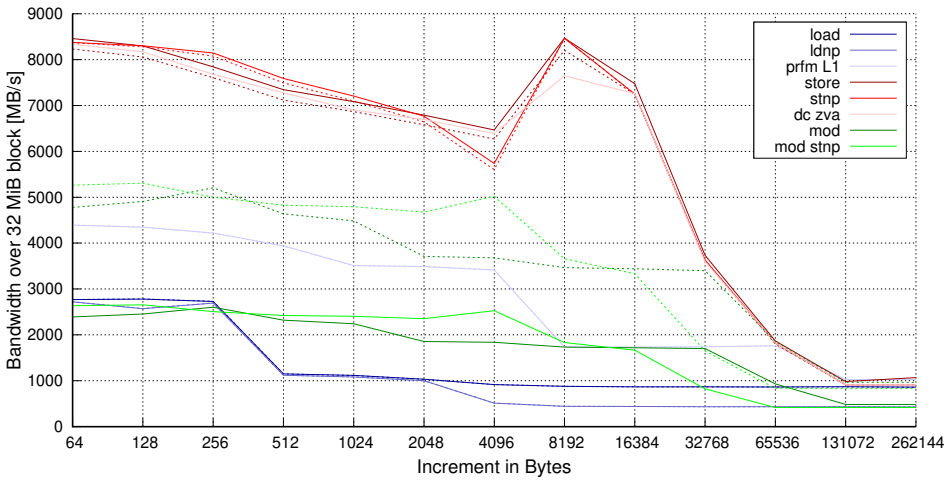

Fig. 6 Sustainable bandwidth on Xilinx ZCU102: Assessment of memory bandwidth over 16 MiB block with different step sizes to trigger worst-case performance behaviour in the memory subsystem. Lines represent the bandwidth observed by the core. Dotted lines track the PMCs relevant for bandwidth regulation. Sec. 5.1 explains details.

5.2.2 ZCU102 MemPol Regulation

We measured the access time from both the APs and RPs to CoreSight registers. On the ZCU102, we measured a mean overhead for reading resp. writing of 303 resp. 213 ns from the Cortex-A53 cores and of 274/216 ns from the R5 cores. 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 Cortex-A53 cores. This hints to bottlenecks at the interconnect level between the A53 cores and the low-power domain. 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. We stressed the routers in the low-power domain by accessing I/O devices in the low-power domain from the A53 cores in parallel, but this did not increase the latencies for accesses from the R5 cores much.

Profiling of the *MemPol* regulation running on the first R5 core showed that the execution of the control loop takes between 4.8 to 5.2 µs. Overall, we add a safety margin to the observed values and set the period of the control loop to 6.25 µs to get a nice factor for human readable timings.

5.2.3 ZCU102 Cost Model

In the cost model of the *MemPol* controller, the sustainable memory bandwidth of $B_{sustainable} \approx 1000$ MB/s 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, as read and write performance are quite similar.[10]

[10] 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.

Based on the peak bandwidth, we assume an overshooting factor $\beta = max(B_{peak-core,*})/B_{sustainable} = 8.46$, or peaks of up to 826 cachelines in 6.25 µs. Experiments with the benchmark from Sec. 5.1 show peak PMC values of 456 refills, 831 write-backs, and 831 for the sum of both counter values.

5.3 NXP i.MX8M

The NXP i.MX8M Quad (NXP, 2024a) is evaluated on the Coral Dev Board (*Phanbell*) by Google. It supports a single cluster of four Cortex-A53 cores running at 1.5 GHz, 32 KiB L1 instruction and data caches each, a shared 1 MiB L2 cache, and 1 GiB LPDDR4 memory. The real-time companion core is a Cortex-M4 with 256 KiB TCM which is clocked at 200 MHz on the Coral Dev Board. We load the regulator binary with the `bootaux` command of the U-Boot bootloader. We use the Mendel Eagle distribution with Linux kernel 4.14.98.

To prevent side effects, we have to clear the `HDBG` bit in the `SYS_CTR_CONTROL_CNTCR` register to prevent the core timers to be halted when a core is halted (see Sec. 4.4). Also, the UART reacts to the debug signals and must be properly configured (NXP, 2021).

5.3.1 i.MX8M Bandwidth Assessment

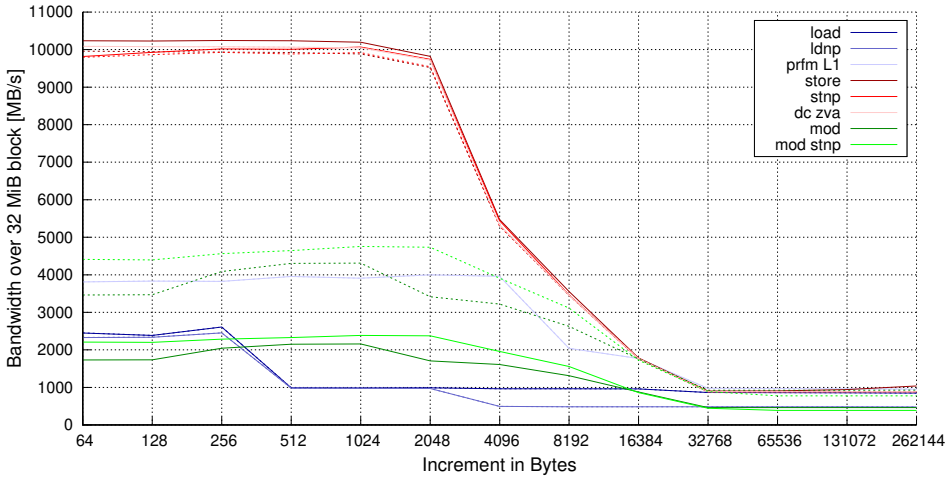

Fig. 7 Sustainable bandwidth on NXP i.MX8M: Assessment of memory bandwidth over 16 MiB block with different step sizes to trigger worst-case performance behaviour in the memory subsystem. Lines represent the bandwidth observed by the core. Dotted lines track the PMCs relevant for bandwidth regulation. Sec. 5.1 explains details.

Fig. 7 shows the bandwidth measurements on the i.MX8M. We observe a peak read bandwidth of $B_{peak-core,r} = 3813$ MB/s (*prfm L1*) and a peak write bandwidth $B_{peak-core,w} =$ of 10235 MB/s (*store*). We already see the bandwidth dropping at an increment of 32 KiB, with 976 MB/s for reading, 911 MB/s for writing and 462 MB/s

when modifying cachelines. We again use a unified sustainable memory bandwidth value of $B_{sustainable} \approx 924$ MB/s (882 MiB/s) for the i.MX8M, even if the difference between reading is about 7%. Like on the ZCU102, we observe an undercounting of writes in PMCs of about 3%.

5.3.2 i.MX8M MemPol Regulation

We measured the access time to the CoreSight registers from the Cortex-M4 core in a tight loop while the Cortex-A53 were active. Reading a CoreSight registers takes between 47 and 57 cycles (235 ns to 285 ns), while writing takes 51 to 60 cycles (255 ns to 300 ns) on the M4. Activity on the A53 cores did not further increase the latencies. We measured a worst-case of 1371 cycles (6.855 µs) for the regulation loop of the *MemPol* regulator. We add a safety margin and use a 10 µs period for the control loop.

5.3.3 i.MX8M Cost Model

On the the i.MX8M, the sustainable memory bandwidth of $B_{sustainable} \approx 924$ MB/s relates to 144.375 cachelines per 10 µs period, and we set the weight-factors $\alpha_r = \alpha_w = 1$ for both reading and writing.

The overshooting factor of $\beta = max(B_{peak-core,*})/B_{sustainable} = 11.08$ is higher than on the ZCU102 due to the higher peak performance. We can expect peaks of up to 1600 cachelines in 10 µs. Our experiments show peak PMC values of 709 refills, 1599 write-backs, and 1599 for the sum of both PMCs in practice.

5.4 NXP S32G2

The NXP S32G274 is designed for automotive purposes (NXP, 2024c). We evaluate the SoC in revision 2.0 on a MicroSys S32G274AR2SBC2 evaluation board with 4 GiB LPDDR4 RAM. The S32G2 provides two clusters of two Cortex-A53. This allows the two cores of each cluster to run in a lock-step configuration. Each core has the usual 32 KiB L1 data and instruction caches. The two cluster have 512 KiB shared L2 cache each. On the RP side, the S32G2 has six Cortex-M7 cores in dual lock-step, so the software side sees three cores. The M7 cores have 64 KiB of TCM and also 32 KiB L1 data and instruction caches. A Network-on-a-Chip (NoC) interconnect connects all components on the SoC. The A53 cores run at 1 GHz, while the M7 cores use 400 MHz. The manual mentions that the debug APB is clocked at 50 MHz (NXP, 2023).

We run the *MemPol* regulator on the first Cortex-M7 core. The regulator code is kept in the internal SRAM at address `0x34100000`, as the M7 core lacks a dedicated TCM for instructions. We configure the instruction cache to speed up execution. The regulator's data is kept in the data TCM of the M7 core. We start the Cortex-M7 using the `startm7` command from U-Boot. We further evaluate the S32G with Linux kernel version 5.15.73 by the CPU vendor.

5.4.1 S32G2 Bandwidth Assessment

The S32G2 shows a different behavior for its memory bandwidth in Fig. 8 than the ZCU102 or the i.MX8M. From a peak read bandwidth of $B_{peak-core,r} = 2000$ MB/s

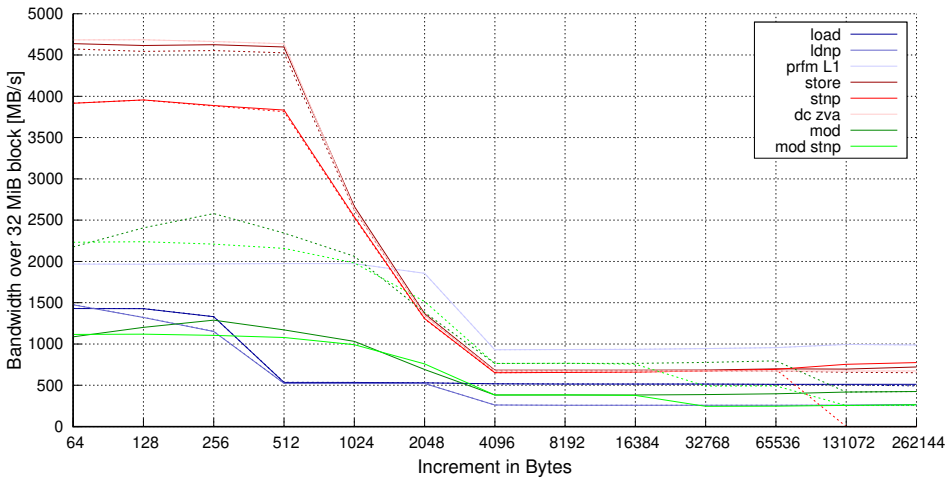

Fig. 8 Sustainable bandwidth on NXP S32G2: Assessment of memory bandwidth over 16 MiB block with different step sizes to trigger worst-case performance behaviour in the memory subsystem. Lines represent the bandwidth observed by the core. Dotted lines track the PMCs relevant for bandwidth regulation. Sec. 5.1 explains details.

(*prfm L1*) and a peak write bandwidth $B_{peak-core,w}$ = of 4420 MB/s (*store*), we quickly drop off to the low bandwidth plateau at a step size of 4 KiB. The then observe $B_{sustainable,r}$ = 956 MiB/s for reading, $B_{sustainable,w}$ = 679 MiB/s for writing, and 394 MiB/s when changing cachelines. This makes it hard to assign a single sustainable bandwidth value. Instead, we assign the *two* values for reading and writing as sustainable bandwidth (see Sec. 5.4.3).

5.4.2 S32G2 MemPol Regulation

Accessing the CoreSight registers on the S32G2 from the first main Cortex-A53 core takes 450 resp. 257 ns for reading resp. writing. The Cortex-M7 core can read registers faster at 420 ns, but writing takes the same time. The timing on the Cortex-M7 core is 420 resp. 257 ns for reading resp. writing. For the regulation loop of the *MemPol* regulator, we observed a worst-case execution time of 2987 cycles (7.468 µs) during our tests. Like on the i.MX8M, we again use a 10 µs period for *MemPol*'s control loop.

The S32G2 provides an alternative mechanism to obtain the relevant performance counters. The Cortex-A53 core exports some of its internal signals that feed the PMCs also on the *PMUEVENT bus*, including the ones related to L2 cache activity. This allows external hardware to monitor the core from the outside without using the CoreSight registers (ARM, 2018a). The S32G2 implements one *PMUEVENT bus observer* unit for each A53 core with dedicated 8-bit wide counters for each signal on the PMUEVENT bus (NXP, 2023). We measured that these counters can be read in 293 ns from the Cortex-M7 cores. However, we cannot reliably use these counters for regulation, as the peak memory in a regulation period would overflow the counters.[11]

[11] The PMUEVENT bus observer unit is primarily intended to count cache, TLB and bus error events.

5.4.3 S32G2 Cost Model

For the cost model on the S32G2, we cannot use a single metric for the sustainable memory bandwidth. From the measured metrics of $B_{sustainable,r} = 956$ MiB/s for reading and $B_{sustainable,w} = 679$ MiB/s for writing, we can derive different weight-factors of $\alpha_r = 1$ for reading and $\alpha_w = 1.408$ for writing to account for the differences. This means that the value of the PMC monitoring the L2 data cache write-back (`0x18`) gets multiplied by 1.408 by the regulator, and $B_{sustainable} = 956$ MiB/s is reduced to a single value.[12]

However, this now inflates the overshooting of the peak write bandwidth by the factor α_w as well. Our overshooting factor becomes $\beta = \alpha_w B_{peak-core,w}/B_{sustainable} = 6.51$, or peaks of up to 972 *weighted* or 690 *unweighted* cachelines in 10 µs. In our experiments, we observed peak raw (unweighted) PMC values of 367 for L2 cache refills and 834 for write-backs and the sum of both counter values.

5.5 Further Platforms

We additionally evaluated the feasibility of *MemPol* on further platforms.

Raspberry Pi 4. On the Raspberry Pi 4 (Raspberry Pi Ltd, 2024), we benchmarked that the reading resp. writing of CoreSight registers from its Cortex-A72 cores takes 135 resp. 122 ns. We are also able to halting and resuming of cores through the debug interface. A *MemPol* regulation would be possible on the Raspberry Pi 4 (probably even with a fast regulation cycle of 2.5 µs as the numbers suggest), but we skipped further evaluation of this platform as the regulation would have to run on one of the system's four Cortex-A72 cores.

NXP LX2160A. We run the same experiment on the NXP LX2160A (NXP, 2024b) and observe 374 resp. 366 ns for CoreSight accesses from the Cortex-A72 cores. Also, halting and resuming of cores through the CTI works as expected. We also did not further consider this platform for evaluation for the same reason as for the Raspberry Pi 4.

NVIDIA Jetson AGX Orin. The same experiment to access the other cores' CoreSight registers failed on the NVIDIA Jetson AGX Orin development kit with its twelve Cortex-A78 cores (NVIDIA, 2024a). The platform additionally includes a Cortex-R5 that could be used to host the regulation. Here, the firmware did not enable the platform's debug authentication signals (`DBGEN`, `NIDEN`), thus making an evaluation impossible (see Sec. 4.4).

6 Evaluation

We perform most of the evaluation of *MemPol* on the ZCU102 platform. Here, 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

[12] In the implementation, we set $\alpha_r = 1000$ and $\alpha_w = 1408$ to prevent the need for floating-point arithmetic.

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) (see Sec. 5.2).

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 Bechtel and Yun (2019) on Linux using the San Diego Vision Benchmark Suite (SD-VBS) (Venkata et al., 2009). 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.[13]

6.1 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. 9 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. 9 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. 10 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 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

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

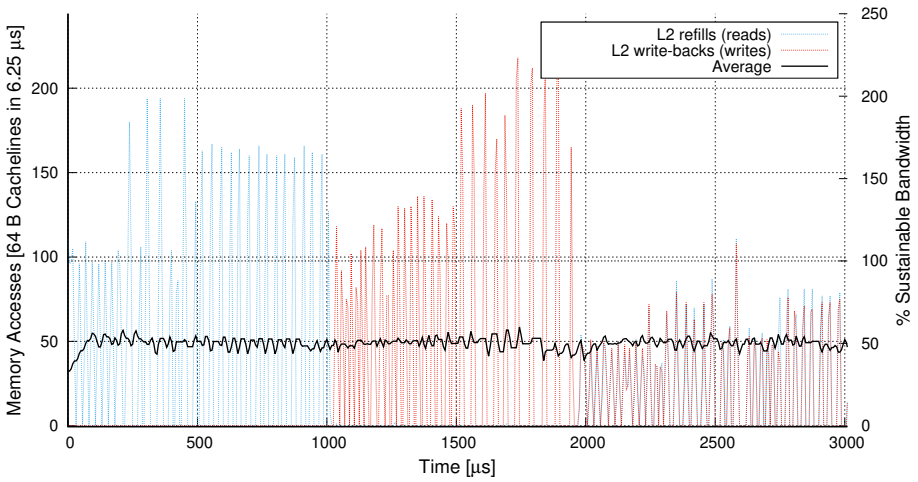

Fig. 9 Polling regulation at 6.25 µs of a core at 50% sustainable memory bandwidth. The core performs three series of four different memory access patterns every 250 µs: four read patterns, four write patterns, then four modify (read-write) patterns. The overall number of memory accesses is the same each time, but peak-behavior increases within a series. 200 µs averages.

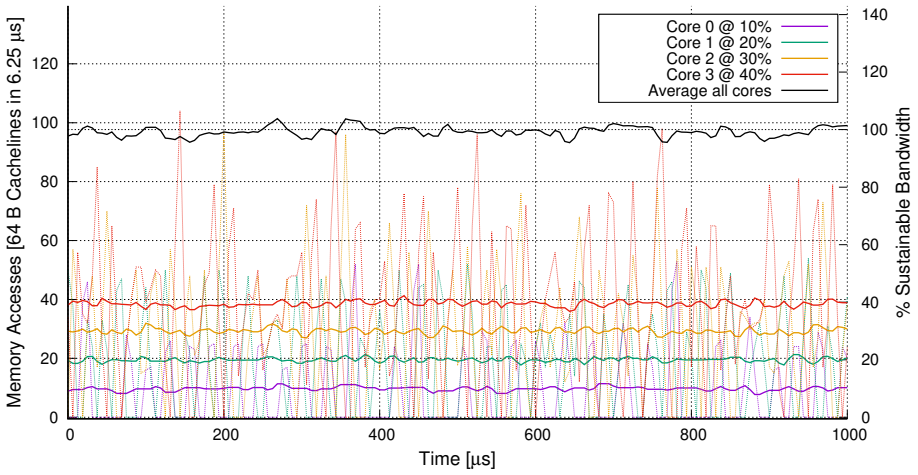

Fig. 10 *MemPol* regulates cores at different bandwidth levels: c_0 worst-case reader at 10%, c_1 worst-case writer at 20%, c_2 peak reader at 30%, c_3 peak writer at 40%. Polling 6.25 µs. 50 µs sliding window size. 200 µs averages.

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.

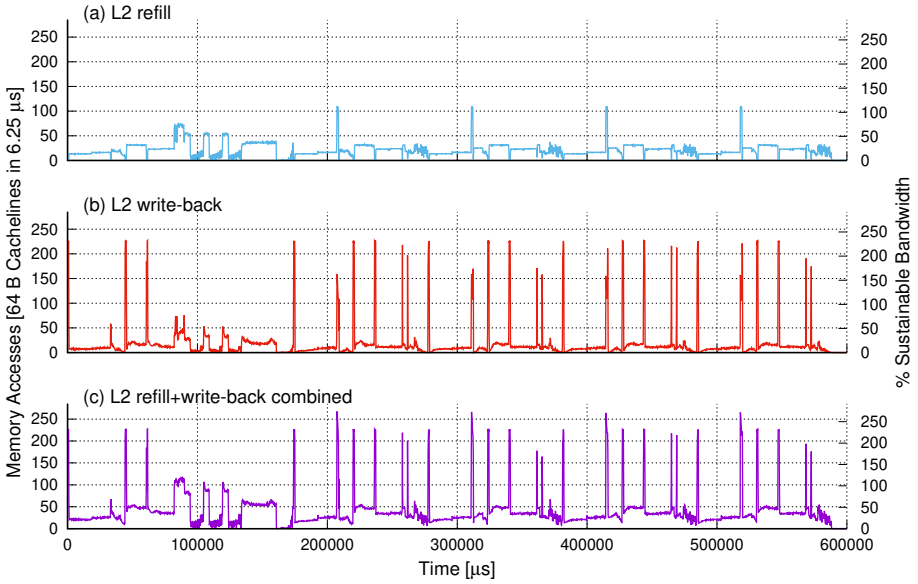

Fig. 11 200 µs averages of PMCs of a run of `tracking` in VGA resolution. The graphs show (a) L2 refills, (b) L2 write-backs, and (c) combined L2 refills and write-backs. *MemGuard* regulates based on (a), *MemPol* based on (c).

6.2 Regulation based on L2 Data Cache Refill and Write-Back

As mentioned in Sec. 2, 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. 3.1). For this, we record the PMU counters for a full *unregulated* run of the `tracking` SD-VBS benchmark. Fig. 11 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. 11 (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. 11 (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. 11 (c) shows the combined L2 cache counters that are used by *MemPol*-regulation following the cost model in Sec. 3.1. 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 (Sohal et al., 2020).

6.3 Impact of Sliding Window Size

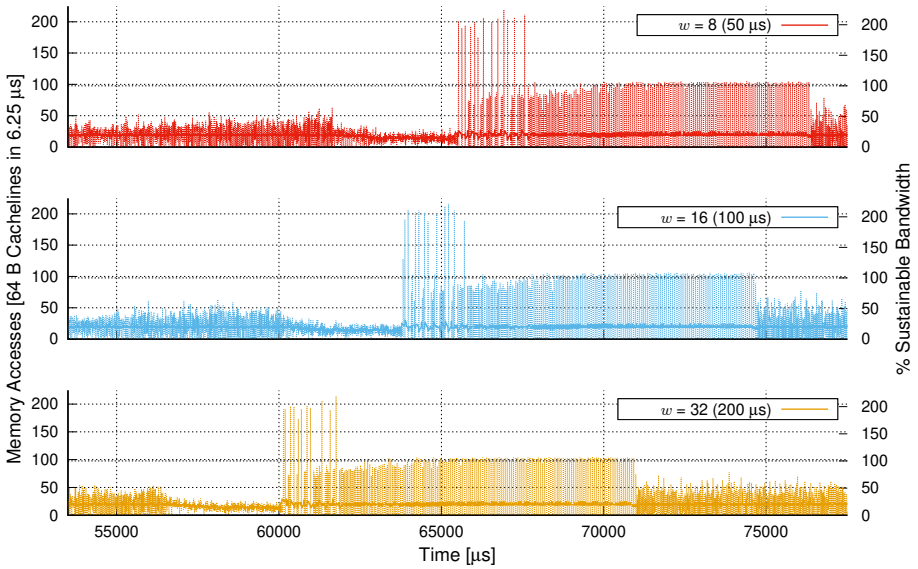

Fig. 12 Three runs of `tracking` in VGA resolution regulated at 20% sustainable memory bandwidth. The graphs detail the first write peak (Fig. 11 at around 45 ms) for different sliding window sizes of 50 µs, 100 µs and 200 µs. Larger sliding window sizes allow the benchmark to reach the peak earlier, *i.e.*, at around 60 ms (200 µs) instead of 63.8 ms or 65.5 ms (50 µs).

Fig. 12 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. 11. 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.

6.4 Redistribution of Memory Bandwidth by Global Regulator

Fig. 13 and 14 show 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. 13), the overall bandwidth drops to 25% when c_0 is idle. Instead, when the global regulation is enabled (Fig. 14), 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

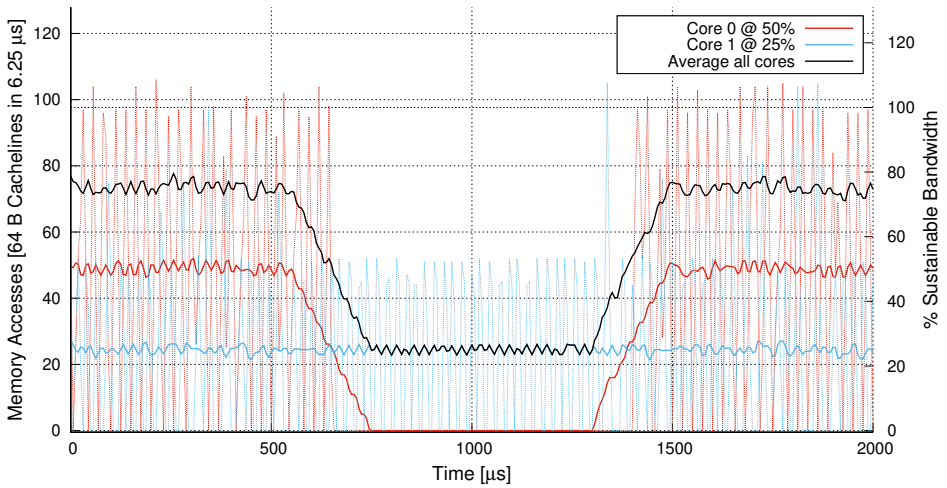

Fig. 13 *MemPol* bandwidth redistribution: global regulation *disabled*. Core c_0 is regulated at 50% bandwidth and alternates memory access and idle phases every 750 µs. Core c_1 is regulated at 25% bandwidth and accesses memory all the time. Both cores perform worst-case reading. The global regulator is disabled and unused bandwidth is not redistributed. Polling at 6.25 µs. 50 µs sliding window size. 200 µs averages.

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. 3.7.

6.5 Comparing Regulation of MemPol and MemGuard

We compare the regulation of *MemPol* and *MemGuard* using SD-VBS. We leverage the framework in Nicolella et al. (2022) to run automated tests to measure the execution time of all 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, 100 µs, 200 µs, and 1 ms for *MemGuard*.

In our first set of experiments (Fig. 15), 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. 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. 6.2). Also, *MemPol*'s global regulation is disabled. We run the benchmarks in isolation (first horizontal group in Fig. 15) 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

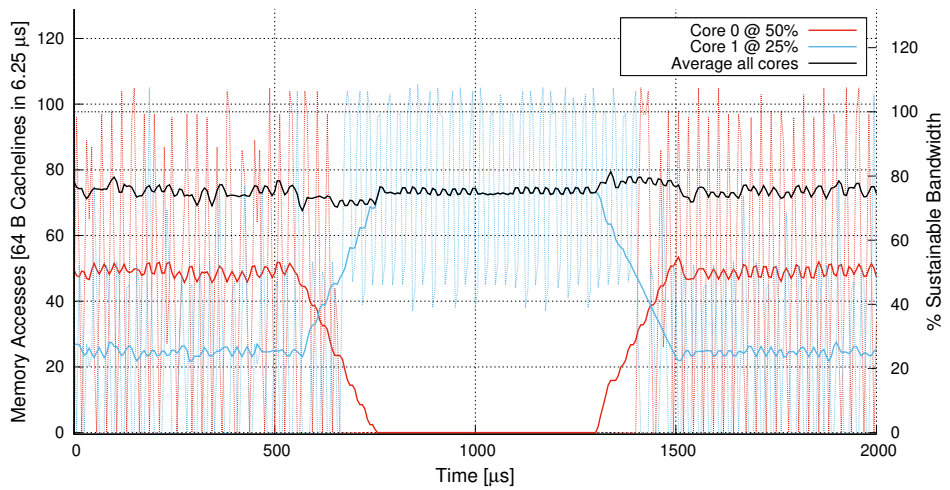

Fig. 14 *MemPol* bandwidth redistribution: global regulation *enabled*. Core c_0 is regulated at 50% bandwidth and alternates memory access and idle phases every 750 µs. Core c_1 is regulated at 25% bandwidth and accesses memory all the time. Both cores perform worst-case reading. The global regulator 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.

refill counter, *MemPol* regulates comparably to *MemGuard*, with *MemGuard* showing higher overheads at smaller regulation periods due to the increased interrupt load.

Table 1 SD-VBS read and write memory bandwidth settings for MemGuard

	average L2 PMCs per run			200 MB/s		300 MB/s		400 MB/s	
Benchmark	refills	write-backs	ratio	read	write	read	write	read	write
`disparity`	11557693	7454823	1.550	121.6	78.4	182.4	117.6	243.2	156.8
`mser`	1697628	528748	3.211	152.5	47.5	228.8	71.2	305.0	95.0
`sift`	4447803	3771100	1.179	108.2	91.8	162.4	137.6	216.5	183.5
`stitch`	870178	871010	0.999	100.0	100.0	149.9	150.1	199.9	200.1
`tracking`	2238342	2318700	0.965	98.2	101.8	147.4	152.6	196.5	203.5

In our second set of experiments we compare *MemPol* to *MemGuard* with write regulation enabled. To setup *MemGuard* bandwidth levels for its write regulation correctly, we first measured the ratio of L2 refills and L2 write-backs for each SD-VBS benchmark in isolation. We ran each benchmark for 50 iterations in VGA resolution and obtained the L2 refill and write-back PMCs before and after the runs. Table 1 shows that the benchmarks fluctuate between 3.2:1 (`mser`) and 1:1.04 (`disparity`) in their read:write ratio. With these insights, we calculate benchmark-specific read and write bandwidth settings for *MemGuard*. Table 1 shows the bandwidth values for a target bandwidth of 20%, 30% and 40% of the sustainable bandwidth. For *MemPol*, we simply configure the combined bandwidth value (Sec. 6.2). *MemPol*'s global regulation

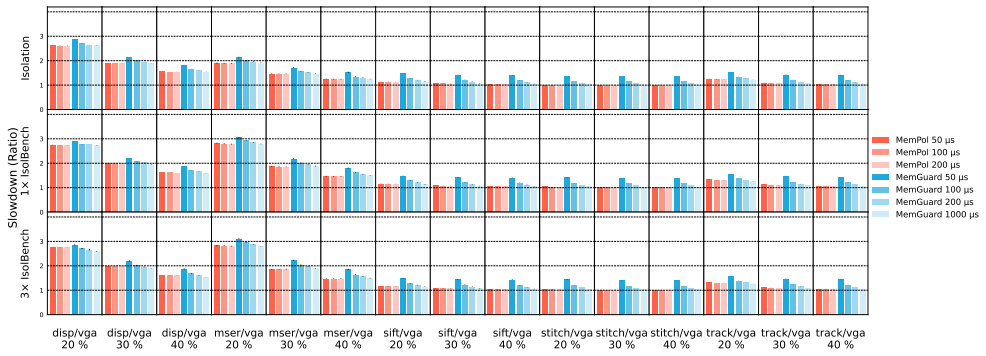

Fig. 15 Slowdown ratio in execution time of SD-VBS regulated at 20%, 30% or 40% sustainable bandwidth with read regulation compared to unregulated execution (slowdown ratio 1.0) as baseline. The slowdown is caused by memory bandwidth regulation (*MemPol*, *MemGuard*) and by implementation overheads (interrupt handling in *MemGuard*, see Sec. 2.1). The colored bars represent the relative mean overhead of 10 runs. The small vertical black lines on top of the bars 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 using L2 cache refill counters only, like *MemGuard*. *MemPol*'s global regulation is turned off.

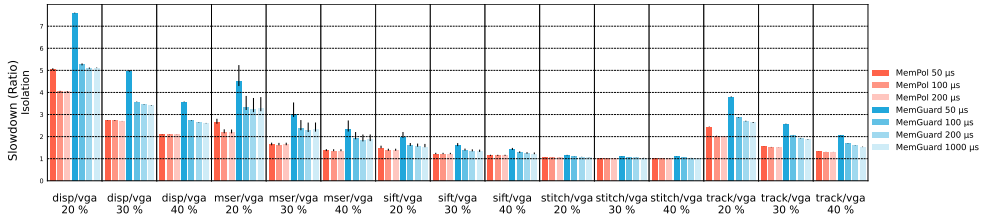

Fig. 16 Slowdown ratio in execution time of SD-VBS regulated at 20%, 30% or 40% sustainable bandwidth with read/write regulation compared to unregulated execution (slowdown ratio 1.0) as baseline. The slowdown is caused by memory bandwidth regulation (*MemPol*, *MemGuard*) and by implementation overheads (interrupt handling in *MemGuard*, see Sec. 2.1). The colored bars represent the relative mean overhead of 10 runs. The small vertical black lines on top of the bars show min/max. The benchmarks run in isolation, like in the first row in Fig. 15. We evaluate *MemPol* and *MemGuard* at different sliding window sizes / regulation periods. *MemPol* regulates using both L2 cache refill and write-back counters, while *MemGuard* uses the bandwidth settings in Table 1. *MemPol*'s global regulation is turned off.

is again disabled. Fig.16 shows the comparison between *MemPol* and *MemGuard* for a run of each benchmark at the given bandwidth levels on the first core of an otherwise idle system. Compared to the similar run using just read-regulation in the top horizontal group in Fig. 15, the read-write-based regulation causes a higher slowdown for `tracking`, as the regulation has now to account for the write-spikes shown in Fig. 11. This affects both *MemPol* and *MemGuard*. `disparity` and `mser` follow this trend, but are less affected. We can also observe that the selected approach of using the ratio between L2 refill and write-back PMCs to derive the regulation parameters for *MemGuard* does not lead to similar outcomes as for *MemPol*. Especially `disparity`, `mser` and `tracking` show higher overheads for *MemGuard* beyond what the read regulation in Fig. 15 shows. This is because the ratio is not homogeneous during the execution of

the benchmark, as especially the higher slowdowns during shorter regulation periods show. Lastly, the measured bandwidth ratios in Table 1 are no longer applicable when the system is under load and the benchmarks cannot exclusively monopolize the L2 cache. This also shows that more sophisticated profiling approaches are needed to find the right regulation parameters for a write-back-based regulation for *MemGuard*.

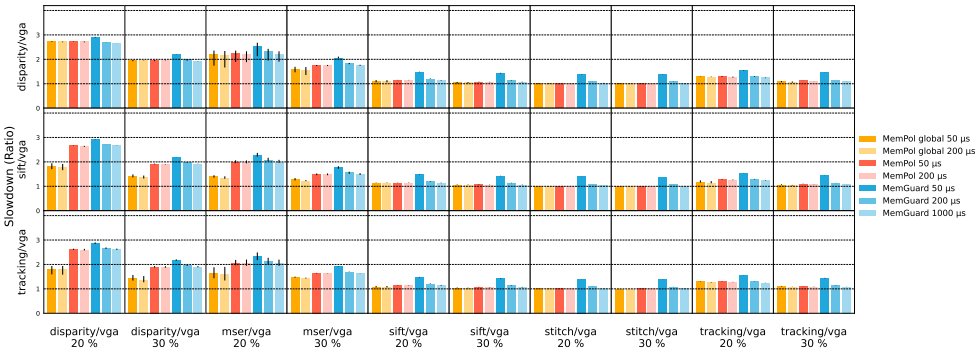

Fig. 17 Slowdown ratio in execution time of SD-VBS regulated at 20% and 30% sustainable bandwidth compared to unregulated execution (slowdown ratio 1.0) as baseline. The slowdown is caused by memory bandwidth regulation (*MemPol*, *MemGuard*) and by implementation overheads (interrupt handling in *MemGuard*, see Sec. 2.1). The colored bars represent the relative mean overhead of 10 runs. The small vertical black lines on top of the bars 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 using L2 cache refill counters only, like *MemGuard*.

In our third set of experiments (Fig. 17), we evaluate the benchmarks executing in parallel on two cores with an equal regulation of 20% and 30% (Fig. 15 shows that 20% and 30% are the most interesting bandwidth settings). Here we also enable *MemPol*'s global regulation[14] and use 40% resp. 60% for the global bandwidth. Similarly to Fig. 15, for a fair comparison, we restrict *MemPol* to only use the L2 cache refill counter for regulation. From the benchmarks, we select `disparity`, `sift`, and `tracking` as co-runners, as they run for a longer time. Similarly to Fig. 15, 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. 12 (a)), but `disparity` is memory-bound.

6.6 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

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

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.

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, Saeed et al. (2022) 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.*, Saeed et al. (2022)). 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 (Yun et al., 2016). 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 (ARM, 2016b), 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 (ARM, 2018a, 2016c). In our experiments, we did not observe any problems with, *e.g.*, atomic synchronization or idle management of the cores. However, it is worth noting that debug interfaces and performance counters, in general, seem to be second class citizens *w.r.t.* safety features. For instance, the debug APB interface to CoreSight registers lacks ECC on the R5 cores (ARM, 2011), and PMCs are underspecified and may exhibit inaccuracies (Mezzetti et al., 2018), as evidenced by the slight undercounting in Sec. 5.1, or even presents bugs (ARM, 2019). Two related questions are whether the right combination of PMU counters will be available on newer Arm core generations, considering that Arm introduces an L3 cache as LLC from the Cortex-A75 onwards (ARM, 2018b), and if the access to the PMCs via the relatively slow CoreSight interface scales beyond a handful of cores. We defer the evaluation of both questions to future work.

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.

7 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 benefit 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*) (Yun et al., 2013; Yun et al., 2016) has been followed by multiple studies (Modica et al., 2018; Dagieu et al., 2016; Martins et al., 2020), including implementations of *MemGuard* also at the hypervisor level to prevent modifications in the host OS, thus allowing for improved applicability. Notably, Bechtel and Yun (2019) extended the *MemGuard* implementation for Linux[15] to support separate regulation on read (cache-refills) or write (write backs) memory traffic for each core. The work of Bechtel and Yun (2023) also extends *MemGuard* to regulate LLC bandwidth offering protection against Cache Bank-Aware Denial-of-Service Attacks.

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 by Sohal et al. (2020) and Saeed et al. (2022). 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 (Rehm et al., 2021), and only a limited subset of MPSoCs (mostly from NXP, *e.g.*, (NXP, 2024c)) exposes some PMCs for the memory controller.

The work by Saeed et al. (2022) 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. The approach proposed by Saeed et al. (2023) also periodically samples PMCs to build distribution-driven memory regulation.

In addition to PMCs, modern MPSoCs provide other QoS or monitoring features (*e.g.*, (ARM, 2014)). The work by Garcia-Esteban et al. (2023) have provided an in-depth analysis of ZCU102 QoS features and the works of Sohal et al. (2020); Serrano-Cases et al. (2021); Houdek et al. (2017); Zini et al. (2022) and Garcia-Esteban et al. (2023) have exploited such primitives to implement bandwidth regulation. Although effective, integrated platform monitors and regulators, *e.g.*, ARM (2016b), 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 (Sohal et al., 2020). In parallel to PMC-based regulation, other approaches (Agrawal et al., 2017; Flodin et al., 2014) 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 Zhou and Wentzlaff (2016) and Farshchi et al. (2020) develop custom hardware components to implement bandwidth regulation directly at hardware level, while Cardona et al. (2019) implements an FPGA module to monitor and regulate different types of requests simultaneously. This proposal was also deployed on a prototype RISC-V design (Wessman et al., 2021). Adaptations for the memory controller have been proposed by Mirosanlou et al. (2020); Hassan et al. (2017); Valsan and Yun (2015); Akesson et al. (2007) and Fernandez-De-Lecea et al. (2023) to reduce the worst-case latency of memory requests under multicore contention. Time Division Multiplexing

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

hardware implementations have also been proposed by Hebbache et al. (2018); Jun et al. (2007); Li et al. (2016) and Kostrzewa et al. (2016) to improve predictability of the memory interconnect level. On MPSoCs (*e.g.*, (Xilinx, 2024b)) that feature an on-chip programmable logic, Hoornaert et al. (2021) 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 by Freitag and Uhrig (2018).

Architecture-level features such as Arm's MPAM (ARM, 2022a) or Intel's RDT (Intel, 2024) aim to deliver improved (QoS) control over the memory subsystem. Real-time characteristics of RDT are analyzed by Sohal et al. (2022) and a theoretical analysis of MPAM characteristics is presented by Zini et al. (2023). Unfortunately, the availability of such architectural-level features 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 (Mancuso et al., 2013; Xilinx, 2020; Kloda et al., 2019) and bank-level partitioning (Yun et al., 2014) 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 NVIDIA's Jetson AGX Orin (NVIDIA, 2024a) as part of Arm's DynamIQ (ARM, 2022b).

An empirical characterization of memory interference for different NVIDIA-based boards is presented by Capodieci et al. (2020) and Cavicchioli et al. (2017), while Brilli et al. (2022) investigates memory interference for FPGA-based heterogeneous MPSoCs.

8 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, the M4 core on the NXP i.MX8M or the M7 core on the NXP S32G2—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 (ARM, 2016b), 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. 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

Agrawal, A., Fohler, G., Freitag, J., Nowotsch, J., Uhrig, S., Paulitsch, M.: Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In: Bertogna, M. (ed.) 29th Euromicro Conference on Real-Time Systems (ECRTS 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, pp. 2–1222. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ECRTS.2017.2

Akesson, B., Goossens, K., Ringhofer, M.: Predator: A predictable SDRAM memory controller. In: 2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 251–256 (2007). https://doi.org/10.1145/1289816.1289877

ARM: Cortex-R5 Technical Reference Manual r1p2. Accessed: 2024-01-01 (2011). https://developer.arm.com/docs/ddi0460/d/

ARM: Quality of Service in ARM Systems: An Overview. Accessed: 2024-01-01 (2014). https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/posts/quality-of-service-in-arm-systems-an-overview

ARM: Arm Architecture Reference Manual for A-profile architecture. Accessed: 2024-01-01 (2016). https://developer.arm.com/docs/ddi0487/ak/

ARM: ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service r1p0. Accessed: 2024-01-01 (2016). https://developer.arm.com/docs/dsu0026/f/

ARM: Arm Cortex-A72 MPCore Processor Technical Reference Manual r0p3. Accessed: 2024-01-01 (2016). https://developer.arm.com/docs/100095/0003/

ARM: Arm CoreSight Architecture Specification. Accessed: 2024-01-01 (2017). https://developer.arm.com/docs/ihi0029/e/

ARM: Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4. Accessed: 2024-01-01 (2018). https://developer.arm.com/docs/ddi0500/j/

ARM: Arm Cortex-A75 Core Processor Technical Reference Manual r3p1. Accessed: 2024-01-01 (2018). https://developer.arm.com/docs/100403/0301/

ARM: Arm Cortex-A53 MPCore Software Developers Errata Notice r0. Accessed: 2024-01-01 (2019). https://developer.arm.com/docs/epm048406/2100/

ARM: Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A. Accessed: 2024-01-01 (2022). https://developer.arm.com/docs/ddi0598/db/

ARM: Arm DynamIQ Shared Unit-AE Technical Reference Manual Revision. Accessed: 2024-01-01 (2022). https://developer.arm.com/docs/101322/0102/

Brilli, G., Capotondi, A., Burgio, P., Marongiu, A.: Understanding and mitigating memory interference in fpga-based hesocs. In: 2022 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1335–1340 (2022). https://doi.org/10.23919/DATE54114.2022.9774768

Bechtel, M., Yun, H.: Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 357–367 (2019). https://doi.org/10.1109/RTAS.2019.00037

Bechtel, M.G., Yun, H.: Cache bank-aware denial-of-service attacks on multicore ARM processors. In: 29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023, pp. 198–208 (2023). https://doi.org/10.1109/RTAS58335.2023.00023

Cavicchioli, R., Capodieci, N., Bertogna, M.: 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), pp. 1–10 (2017). https://doi.org/10.1109/ETFA.2017.8247615

Capodieci, N., Cavicchioli, R., Olmedo, I.S., Solieri, M., Bertogna, M.: 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), pp. 1–10 (2020). https://doi.org/10.

1109/RTCSA50079.2020.9203722

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

Dagieu, N., Spyridakis, A., Raho, D.: 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), pp. 21–27 (2016). https://www.thinkmind.org/index.php?view=article&articleid=ubicomm_2016_1_40_10072

Fernandez-De-Lecea, A., Hassan, M., Mezzetti, E., Abella, J., Cazorla, F.J.: Improving Timing-Related Guarantees for Main Memory in Multicore Critical Embedded Systems. In: 2023 IEEE Real-Time Systems Symposium (RTSS), pp. 265–278 (2023). https://doi.org/10.1109/RTSS59052.2023.00031

Farshchi, F., Huang, Q., Yun, H.: BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 364–375 (2020). https://doi.org/10.1109/RTAS48715.2020.00011

Flodin, J., Lampka, K., Yi, W.: 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), pp. 151–159 (2014). https://doi.org/10.1109/SIES.2014.6871199

Freitag, J., Uhrig, S.: Closed Loop Controller for Multicore Real-Time Systems. In: Berekovic, M., Buchty, R., Hamann, H., Koch, D., Pionteck, T. (eds.) Architecture of Computing Systems – ARCS 2018, pp. 45–56. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77610-1_4

Garcia-Esteban, S., Serrano-Cases, A., Abella, J., Mezzetti, E., Cazorla, F.J.: Quasi Isolation QoS Setups to Control MPSoC Contention in Integrated Software Architectures. In: Papadopoulos, A.V. (ed.) 35th Euromicro Conference on Real-Time Systems (ECRTS 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 262, pp. 5–1525. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ECRTS.2023.5

Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F., Ziegenbein, D.: Waters industrial challenge 2017. In: 2017 Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) (2017). https://waters2017.inria.fr/challenge/

Hebbache, F., Jan, M., Brandner, F., Pautet, L.: Shedding the Shackles of Time-Division Multiplexing. In: 2018 IEEE Real-Time Systems Symposium (RTSS), pp. 456–468 (2018). https://doi.org/10.1109/RTSS.2018.00059

Hassan, M., Patel, H., Pellizzoni, R.: PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems. ACM Trans. Embed. Comput. Syst. **16**(4) (2017) https://doi.org/10.1145/3019611

Hoornaert, D., Roozkhosh, S., Mancuso, R.: A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In: Brandenburg, B.B. (ed.) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 196, pp. 2–1222. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECRTS.2021.2

Houdek, P., Sojka, M., Hanzálek, Z.: Towards predictable execution model on ARM-based heterogeneous platforms. In: 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE), pp. 1297–1302 (2017). https://doi.org/10.1109/ISIE.2017.8001432

Intel: Resource Director Technology. Accessed: 2024-01-01 (2024). https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

Jun, M., Bang, K., Lee, H.-J., Chang, N., Chung, E.-Y.: Slack-based Bus Arbitration Scheme for Soft Real-time Constrained Embedded Systems. In: 2007 Asia and South Pacific Design Automation Conference, pp. 159–164 (2007). https://doi.org/10.1109/ASPDAC.2007.357979

Kostrzewa, A., Saidi, S., Ernst, R.: Slack-based resource arbitration for real-time Networks-on-Chip. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1012–1017 (2016). https://doi.org/10.3850/9783981537079_0233

Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., Bertogna, M.: Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–14 (2019). https://doi.org/10.1109/RTAS.2019.00009

Li, Y., Akesson, K.B., Goossens, K.G.W.: Architecture and analysis of a dynamically-scheduled real-time memory controller. Real-Time Systems **52**(5), 675–729 (2016) https://doi.org/10.1007/s11241-015-9235-y

Lugo, T., Lozano, S., Fernandez, J., Carretero, J.: A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. IEEE Access **10**, 21853–21882 (2022) https://doi.org/10.1109/ACCESS.2022.3151891

Modica, P., Biondi, A., Buttazzo, G., Patel, A.: Supporting temporal and spatial

isolation in a hypervisor for ARM multicore platforms. In: 2018 IEEE International Conference on Industrial Technology (ICIT), pp. 1651–1657 (2018). https://doi.org/10.1109/ICIT.2018.8352429

Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., Pellizzoni, R.: Real-time cache management framework for multi-core architectures. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 45–54 (2013). https://doi.org/10.1109/RTAS.2013.6531078

Mirosanlou, R., Hassan, M., Pellizzoni, R.: DRAMbulism: Balancing Performance and Predictability through Dynamic Pipelining. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 82–94 (2020). https://doi.org/10.1109/RTAS48715.2020.00-15

Mezzetti, E., Kosmidis, L., Abella, J., Cazorla, F.J.: High-Integrity Performance Monitoring Units in Automotive Chips for Reliable Timing V&V. IEEE Micro **38**(1), 56–65 (2018) https://doi.org/10.1109/MM.2018.112130235

Moon, J.Y., Kim, D.Y., Kim, J.H., Jeon, J.W.: The Migration of Engine ECU Software From Single-Core to Multi-Core. IEEE Access **9**, 55742–55753 (2021) https://doi.org/10.1109/ACCESS.2021.3071500

Martins, J., Tavares, A., Solieri, M., Bertogna, M., Pinto, S.: Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In: Bertogna, M., Terraneo, F. (eds.) Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020). OpenAccess Series in Informatics (OASIcs), vol. 77, pp. 3–1314. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.NG-RES.2020.3

Nicolella, M., Roozkhosh, S., Hoornaert, D., Bastoni, A., Mancuso, R.: RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications. In: Abdeddaïm, Y., Cucu-Grosjean, L., Nelissen, G., Pautet, L. (eds.) RTNS 2022: The 30th International Conference on Real-Time Networks and Systems, Paris, France, June 7 - 8, 2022, pp. 184–195. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3534879.3534888

NVIDIA: NVIDIA Jetson AGX Orin. Accessed: 2024-01-01 (2024). https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

NVIDIA: NVIDIA Jetson AGX Xavier. Accessed: 2024-01-01 (2024). https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

Ning, Z., Wang, C., Chen, Y., Zhang, F., Cao, J.: Revisiting ARM Debugging Features: Nailgun and Its Defense. IEEE Transactions on Dependable and Secure Computing (01), 1–16 (2021) https://doi.org/10.1109/TDSC.2021.3139840

NXP: NXP S32V234SBC. Accessed: 2024-01-01. https://www.nxp.com/design/

development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v2-vision-and-sensor-fusion-low-cost-evaluation-board:SBC-S32V234

NXP: i.MX 8M Dual/8M QuadLite/8M Quad Applications Processors Reference Manual, Rev. 3.1. NXP document IMX8MDQLQRM (2021)

NXP: S32G2 Reference Manual, Rev. 7. NXP document S32G2RM (2023)

NXP: NXP i.MX8M. Accessed: 2024-01-01. (2024). https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-family-armcortex-a53-cortex-m4-audio-voice-video:i.MX8M

NXP: NXP LX2160A. Accessed: 2024-01-01. (2024). https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-lx2120a-lx2080a-processors:LX2160A

NXP: NXP S32G2. Accessed: 2024-01-01. (2024). https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32g-vehicle-network-processors/s32g2-processors-for-vehicle-networking:S32G2

Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R.: A Predictable Execution Model for COTS-Based Embedded Systems. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 269–279 (2011). https://doi.org/10.1109/RTAS.2011.33

Raspberry Pi Ltd: Raspberry Pi 4. Accessed: 2024-01-01. (2024). https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

Rehm, F., Seitter, J., Larsson, J.-P., Saidi, S., Stea, G., Zippo, R., Ziegenbein, D., Andreozzi, M., Hamann, A.: The Road towards Predictable Automotive High - Performance Platforms. In: 2021 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1915–1924 (2021). https://doi.org/10.23919/DATE51398.2021.9473996

Sohal, P., Bechtel, M., Mancuso, R., Yun, H., Krieger, O.: A closer look at intel resource director technology (rdt). In: Proceedings of the 30th International Conference on Real-Time Networks and Systems. RTNS 2022, pp. 127–139. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3534879.3534882 . https://doi.org/10.1145/3534879.3534882

Serrano-Cases, A., Reina, J.M., Abella, J., Mezzetti, E., Cazorla, F.J.: Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In: Brandenburg, B.B. (ed.) 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 196, pp. 3–1326. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECRTS.2021.3

Saeed, A., Dasari, D., Ziegenbein, D., Rajasekaran, V., Rehm, F., Pressler, M., Hamann, A., Mueller-Gritschneder, D., Gerstlauer, A., Schlichtmann, U.: Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores . In: 2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 133–145 (2022). https://doi.org/10.1109/RTAS54340.2022.00019

Saeed, A., Hoornaert, D., Dasari, D., Ziegenbein, D., Mueller-Gritschneder, D., Schlichtmann, U., Gerstlauer, A., Mancuso, R.: Memory latency distribution-driven regulation for temporal isolation in mpsocs. In: Papadopoulos, A.V. (ed.) 35th Euromicro Conference on Real-Time Systems, ECRTS 2023, July 11-14, 2023, Vienna, Austria. LIPIcs, vol. 262, pp. 4–1423. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPICS.ECRTS.2023.4

Sohal, P., Tabish, R., Drepper, U., Mancuso, R.: E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In: 2020 IEEE Real-Time Systems Symposium (RTSS) (2020). https://doi.org/10.1109/RTSS49844.2020.00039

Schwaericke, G., Tabish, R., Pellizzoni, R., Mancuso, R., Bastoni, A., Zuepke, A., Caccamo, M.: A Real-Time virtio-based Framework for Predictable Inter-VM Communication. In: 2021 IEEE International Real-Time Systems Symposium (RTSS) (2021). https://doi.org/10.1109/RTSS52674.2021.00015

Venkata, S.K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., Taylor, M.B.: SD-VBS: The San Diego vision benchmark suite. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 55–64 (2009). https://doi.org/10.1109/IISWC.2009.5306794

Valsan, P.K., Yun, H.: 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, pp. 86–93 (2015). https://doi.org/10.1109/CPSNA.2015.24

Wessman, N.-J., Malatesta, F., Andersson, J., Gomez, P., Masmano, M., Nicolau, V., Le Rhun, J., Cabo, G., Bas, F., Lorenzo, R., Sala, O., Trilla, D., Abella, J.: De-RISC: the first RISC-V space-grade platform for safety-critical systems. In: 2021 IEEE Space Computing Conference (SCC), pp. 17–26 (2021). https://doi.org/10.1109/SCC49971.2021.00010

Xilinx: Xilinx Xen Support with Cache-Coloring. Accessed: 2024-01-01 (2020). https://github.com/Xilinx/xen/releases/tag/xilinx-v2020.2

Xilinx: Xilinx Versal. Accessed: 2024-01-01 (2024). https://www.xilinx.com/products/silicon-devices/acap/versal.html

Xilinx: Zynq UltraScale+ Device Technical Reference Manual UG1085. Accessed

2024-01-01 (2024). https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

Yun, H., Mancuso, R., Wu, Z.-P., Pellizzoni, R.: 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), pp. 155–166 (2014). https://doi.org/10.1109/RTAS.2014.6925999

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: 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), pp. 55–64 (2013). https://doi.org/10.1109/RTAS.2013.6531079

Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. IEEE Transactions on Computers **65**(2), 562–576 (2016) https://doi.org/10.1109/TC.2015.2425889

Zuepke, A., Bastoni, A., Chen, W., Caccamo, M., Mancuso, R.: Mempol: Policing core memory bandwidth from outside of the cores. In: 29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023, pp. 235–248 (2023). https://doi.org/10.1109/RTAS58335.2023.00026

Zini, M., Casini, D., Biondi, A.: Analyzing arm's MPAM from the perspective of time predictability. IEEE Trans. Computers **72**(1), 168–182 (2023) https://doi.org/10.1109/TC.2022.3202720

Zini, M., Cicero, G., Casini, D., Biondi, A.: Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. Software: Practice and Experience **52**(5), 1095–1113 (2022) https://doi.org/10.1002/spe.3053

Zhou, Y., Wentzlaff, D.: MITTS: Memory Inter-Arrival Time Traffic Shaping. In: Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture. ISCA '16, pp. 532–544 (2016). https://doi.org/10.1109/ISCA.2016.53