Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores

Ahsan Saeed[1,3], Dakshina Dasari[1], Dirk Ziegenbein[1], Varun Rajasekaran[2], Falk Rehm[1], Michael Pressler[1], Arne Hamann[1], Daniel Mueller-Gritschneder[3], Andreas Gerstlauer[4] and Ulf Schlichtmann[3]
[1]Robert Bosch GmbH, [2]Bosch Rexroth AG, [3]Technical University of Munich, [4]The University of Texas at Austin
{ahsan.saeed,dakshina.dasari,dirk.ziegenbein,falk.rehm,michael.pressler,arne.hamann}@de.bosch.com, varun.rajasekaran@boschrexroth.de, {daniel.mueller,ulf.schlichtmann}@tum.de, gerstl@ece.utexas.edu

Abstract—**Temporal isolation is one of the key challenges for co-running mixed-criticality applications on Commercial Off-The-Shelf (COTS) multi-core platforms. In particular, the main memory subsystem is one of the most prominent causes of interference and loss of isolation. Existing mechanisms for memory bandwidth regulation are limited to conservative bandwidth reservation, use pessimistic worst-case execution time (WCET) estimations or require dedicated hardware that is not feasible in COTS multi-core platforms.**

In this paper, we propose a novel mechanism for memory interference control that uses feedback-based control to dynamically regulate memory accesses of individual cores in a multi-core platform. Our mechanism directly regulates the source of interference by leveraging information about memory utilization, acquired from existing hardware performance counters provided by modern COTS-based memory controllers. The proposed solution is implemented on Linux as a loadable kernel module. The results of evaluating our approach with real and synthetic benchmarks on a COTS multi-core (NXP S32V234) platform demonstrate that it is able to provide temporal isolation with up to 4x and 2x more overall throughput for non-real-time applications compared to static and dynamic memory bandwidth-based regulation approaches, respectively, while maintaining guarantees for applications running on the real-time core.

Index Terms—**temporal isolation, memory bandwidth regulation, real-time system, feedback control, multi-core**

I. Introduction

The increased demand for computational power in emerging applications across different embedded domains (automotive, avionics and industrial automation) has driven the adoption of multi-core systems. A key challenge on such platforms is the ability to consolidate applications with varying Quality of Service (QoS) requirements, ensuring freedom from interference and meeting timing guarantees while also utilizing the system effectively. However, the presence of shared resources (like the cache and main memory) leads to undesired interference between applications [1]–[3] and, as a result, non-negligible, context-dependent variability of the execution time. This problem is further exacerbated considering scenarios in which the set of executing applications is not fixed and applications dynamically enter or leave the system (e.g., by over-the-air or modular upgrades). In such a scenario, it is either infeasible or prohibitively expensive to enumerate all the possible resource usage scenarios and resulting interference effects at design time, thereby rendering any offline resource allocation methods inadequate.

In particular, interference through the shared main memory system among applications on different cores accounts for significant degradation in application performance and response time [4]. A key problem with modern COTS-based memory controllers is that memory request scheduling is governed by proprietary performance-oriented mechanisms [5]. These mechanisms are agnostic to the priority/criticality of the requestor, which can lead to the unfavorable consequence that requests originating from time-critical tasks may be delayed by requests of low-priority tasks. These complex memory scheduling policies combined with innumerable memory request patterns from co-running applications renders any worst-case design-time estimation of request service times difficult.

Existing hardware-oriented mechanisms for memory interference control require dedicated hardware [6]–[8] that is not feasible in COTS multi-core platforms. On the other hand, software-oriented memory bandwidth regulations mechanisms, like MemGuard [9], [10] approach the problem of controlling memory interference by periodically monitoring the memory bandwidth originating from each core and stalling cores when egress memory bandwidth exceeded a pre-defined threshold. These methods may result in a very conservative usage of the total memory bandwidth and may prove inadequate in controlling interference, since they base their regulation on an indirect measure of utilization (derived from the core-egress bandwidth) and not on the actual memory utilization. In fact, many modern memory controllers expose interfaces that directly reflect the utilization of the DRAM subsystem for a given observation interval [11].

In order to understand the relationship between the egress bandwidth at the cores and the resulting utilization at the DRAM subsystem, we monitored these values and observed that memory traffic demand with a certain bandwidth can have different impact on the utilization of the DRAM subsystem (as seen in Figure 1). This is attributed to the fact that COTS-based memory controllers have complex performance oriented optimizations, due to which not only the number of requests, but also the target addresses, request sequences, and many other factors affect the number of requests served by the controller in a given observation interval. This explains the *non-linear relationship* between the memory bandwidth (observed at the core egress) and the memory utilization reported by the memory controller. This is highly relevant as *memory interference only comes into play when the instantaneous memory utilization saturates and reaches 100%, causing applications to stall* [11], leading to the premise that the regulation mechanism must be cognizant of the memory utilization to truly isolate memory interference effects.

In this paper, we propose a novel mechanism that is aimed at directly controlling the overall memory utilization to provide temporal isolation. It aims to optimally regulate memory bandwidth of each core using feedback-based control in such a way that the memory utilization is maintained around a pre-defined threshold below saturation. Such an online memory regulation mechanism is capable of handling diverse workloads that arrive dynamically and have input-data dependent memory requirements, for which an exhaustive memory usage profiling is infeasible at design time. To address mixed-criticality deployments, we consider a setup in which a dedicated core hosts a real-time (RT) application while the remaining other cores host best-effort, non-real-time (NRT) applications. The proposed approach increases overall memory utilization and provides better responsiveness to NRT applications than existing approaches that require conservatively overthrottling the respective cores in order to provide guarantees to real-time applications.

The key contributions of this work are:

1) We highlight that memory bandwidth measured at the source cores by itself is not a key indicator of underlying memory saturation and hence interference. Based on this observation, we propose a mechanism for regulating memory bandwidth based on memory utilization measured at the DRAM subsystem.
2) We propose a feedback-based control mechanism to dynamically regulate the memory accesses from each core and ensure temporal isolation based on memory utilization.
3) We implement our approach as a loadable Linux kernel module and deploy our solution on a COTS multi-core (NXP S32V234) platform. We evaluate our system on an extensive set of realistic benchmarks from the San Diego [12], DAPHNE [13] and synthetic IsolBench [14] suite, and demonstrate its effectiveness compared against the static and dynamic bandwidth-based regulation approaches. Results show that our approach reduces execution time of non-real-time applications by up to 4x compared to bandwidth-based regulation approaches while maintaining guarantees for real-time applications.

We do not construct a formal model of the DRAM subsystem, nor formulate provable guarantees. The correctness of our approach is corroborated by a full system evaluation, which provides evidence that the work presented is practical for industrial applications.

The rest of the paper is organized as follows: after a motivation of the memory bandwidth regulation problem in Section II, Section III describes the overall architecture and Section IV explains the main algorithm of our approach. Section V then discusses the experimental setup and presents results followed by survey of related work in Section VI. Finally, Section VII concludes with a summary and outlook on future work.

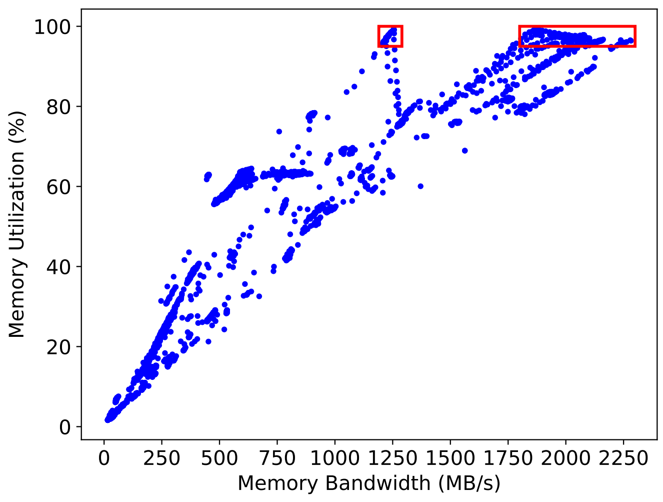


Fig. 1. Impact of memory bandwidth on utilization for SB-VDS benchmark profiled in isolation.

II. Memory Utilization

In order to demonstrate the benefit of using memory utilization as a metric on which memory bandwidth regulation can be based, we need to further understand the latent relationship between the memory requests originating from the cores and the resulting memory utilization. Applications suffer from memory interference and incur stalls when the overall memory requests exceed the rate at which the memory controller can serve them. An important criterion to avoid such interference-related stalls is to ensure that the egress memory request at the cores can be kept below the memory controller service rate i.e. 100% utilization.

The memory utilization in an observation interval, in the context of this work, refers to the memory service rate in that interval. This is measured by leveraging the profiling mechanism present in modern memory controllers that count *busy cycles*, which reflect the total number of cycles where any memory requests are pending in the memory request queue (FIFO) during the observation interval. These memory requests are pending as long as different phases of memory transfer including arbitration, control and actual data transfers have not finished. Memory requests made towards DRAM are first received in the memory request queue (FIFO), which is shared by different memory banks and command control unit. Hence if the memory request queue is filled, no more memory requests are served causing applications to stall. By monitoring the *busy cycles* B in an observation interval of L cycles length, the percentage memory utilization U can be derived by $U = (B/L) * 100$.

Fig. 1 plots the ingress memory bandwidth (relative to the DRAM) and the subsequent memory utilization at the DRAM subsystem. We profiled the SB-VDS benchmark [12] applications on our evaluation platform (NXP S32V234 [15]) at a period of 1ms in isolation for a duration of 5 seconds. The memory bandwidth is measured using the memory controller profiling mechanism, which measures the total number of bytes transferred in read and in write memory requests. We observe that the DRAM subsystem can achieve saturation, that is a near 100% utilization, for a wide range of memory bandwidth values ranging from 1800 to 2200 MB/s and as low as 1250 MB/s. In other words, only observing the memory bandwidth corresponding to outgoing last-level cache (LLC) misses and write-backs at the core egress is not sufficient to predict interference. This non-predictability can be explained by the fact that the rate at which the memory controller serves requests is not fixed but highly variable and non-trivial to compute. It depends on several factors including the access location of each memory request and the current state of DRAM subsystem, the memory access pattern, the memory controller scheduling algorithm, the page policy and the power-management policy [16], [17], all of which are hidden from the user in COTS multi-core platforms.

Therefore in the example of Fig. 1, temporal isolation strategies to conservatively bound the memory interference by only measuring memory bandwidth have to assume 1250 MB/s as the maximum guaranteed bandwidth to be distributed among applications. At all higher memory bandwidths, the memory utilization could reach 100%, which implies that no more memory requests can be fulfilled and applications are stalled and experience execution time increases caused by memory interference. This conservative approach enables guaranteed application execution times but wastes a significant amount of memory bandwidth. In order to mitigate this, we propose the use of memory utilization as a metric to measure the saturation level of the DRAM subsystem along with a feedback-based control approach that maximizes the available memory bandwidth by ensuring that the DRAM subsystem operates below its saturation threshold (100% utilization). The effectiveness of our approach is demonstrated by a full system evaluation, which is presented in detail in Section V.

III. System Overview

An overview of our system architecture is depicted in Fig. 2. Our setup considers a multi-core platform in which a core designated as RT core is dedicated to host real-time applications, while the other cores are designated as NRT cores that host best-effort applications. The RT core must be allocated sufficient memory bandwidth such that real-time applications meet their timing requirements. The NRT cores may occasionally suffer performance degradation due to a reduced memory bandwidth allocation.

We introduce a Dynamic Regulator (DR) whose purpose is to maximize the overall memory bandwidth utilization while maintaining guarantees for applications running on the RT core. To this end, the DR ensures that the DRAM subsystem is never saturated as well as that no NRT core exceeds its allocated memory bandwidth usage. The DR thus primarily focuses on controlling interference at the DRAM subsystem

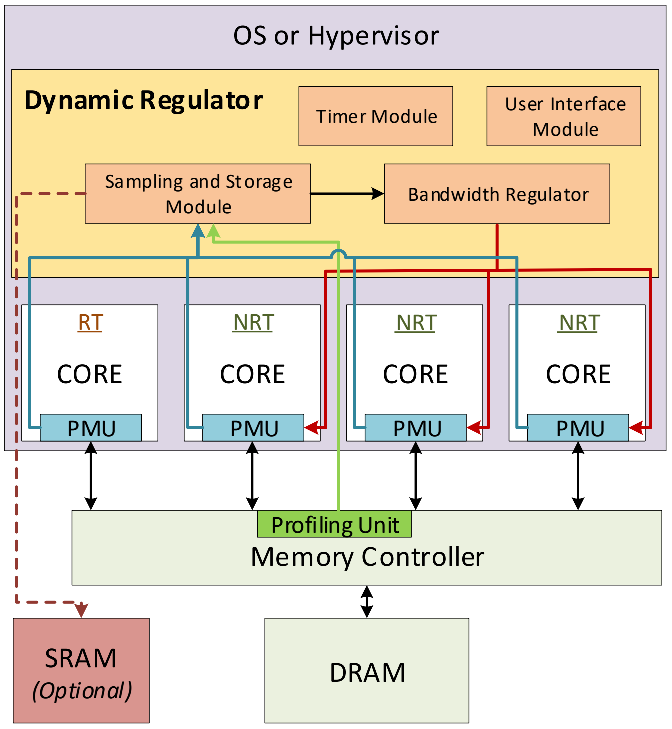


Fig. 2. System architecture.

by observing the memory utilization, and limiting the NRT core bandwidth.

The DR in principle can reside either in the Operating System (OS), or in the hypervisor. The DR keeps track of the memory accesses from individual cores by collecting hardware performance counter values from the Performance Monitoring Unit (PMU) of each core, as highlighted by blue lines in Fig. 2. In addition, the DR computes the memory utilization using information from the Profiling Unit of the memory controller, which is highlighted by green lines in Fig. 2. These values are then used to dynamically configure the memory bandwidth budgets assigned to the cores, which are then also enforced by the DR (highlighted by red lines in Fig. 2). For analysis and evaluation of the mechanism, the DR optionally stores counter samples and key characteristics in SRAM memory to avoid additional traffic to the main memory.

Although shared-cache contention is a major issue, in this work, we focus on the issue of contention induced by the shared DRAM. As such, we assume that the LLC is either partitioned on a per-core basis or is private. Different techniques have been proposed to achieve the cache-partitioning (See [2], [18]–[21]) and can be combined with our approach.

The proposed method can be implemented on any platform on which we are able to measure i) memory utilization and ii) per core memory accesses (reads and writes) towards the DRAM. Many modern COTS platforms (e.g., NXP iMX6-based and S32 automotive platforms) include a profiling mechanism inside the memory controller that provides the ability to calculate the memory utilization together with read and write access statistics towards the DRAM subsystem for a given observation interval. Furthermore, modern processor-microarchitectures (e.g., ARMv8-A, Intel-Xeon (5600/E7), AMD-17h-Zeppelin/Zen) provide per-core-specific hardware performance events within a PMU that can measure LLC misses and write-backs. When a load or store instruction causes a cache miss, a read transaction is initiated to load the cache block from main memory. If the line being evicted from the cache was dirty, then it is written back to memory with a write transaction referred to as cache write-back. Hence the combination of LLC misses and write-backs can essentially translate into the total number of memory accesses for a core.

IV. Dynamic Regulator

In this section, we describe the design and implementation of the Dynamic Regulator (DR) as introduced in the previous section and depicted in Fig. 2. The DR dynamically regulates and enforces the bandwidths of the NRT cores, while the RT core is left unregulated such that its hosted RT applications are non-throttled.

The main concept of the DR mechanism is to maintain the memory utilization below a pre-defined threshold. Because the DR samples the memory utilization periodically, the memory utilization threshold must be set below 100% in order to avoid overshoots in memory utilization caused by the reactive nature of DR as a function of the regulation interval.

The DR uses a *Sampling and Storage Module* to collect hardware performance counter values periodically according to the time period defined by the *Timer Module*. These values are then used by *Bandwidth Regulator* to form a feedback-based control and dynamically regulate the bandwidth budget of each NRT core. Finally, a *User Interface Module* is used to configure different DR parameters. The various components of the mechanism are depicted in Fig. 2 and described in the following subsections.

A. Timer Module

The timer module is responsible for triggering periodic events. The time period configured in the timer module sets the value of i) the regulation interval at which the memory bandwidth budgets of the NRT cores are regulated and enforced, as well as ii) the observation interval for measuring memory utilization. For creating a timer, the implementation relies on the high-resolution timer (in the nanosecond range) infrastructure provided by the Linux kernel in order to provide precise timing.

The choice of the regulation interval is a trade-off between bandwidth regulation granularity and overhead (details in Section V-B) due to the generation of more frequent timer interrupts. The overhead of the periodic tick could be reduced by directly using the scheduler tick. A regulation interval of 1 ms has shown to yield good results and is set throughout the evaluation setup.

B. Sampling and Storage Module

The sampling unit is mainly used for collecting required hardware performance counter values at every regulation interval:

- Per-core hardware performance events like LLC misses and LLC write-backs to measure memory accesses.
- DRAM cycles and busy cycles from the memory controller that provide the average memory utilization U of the DRAM subsystem, which forms the basis of our algorithm.

Our sampling-based approach does not measure instantaneous but average utilization over the observation interval. As such, there is a tradeoff between capturing transient saturations and practical limits due to performance overheads explained in the Section V-B. To compute the total memory bandwidth usage for each core, we multiply the measured LLC miss count and LLC write-back count with the cacheline size, which is 64 bytes for our evaluation platform (NXP S32V234 [15]).

The storage of this collected information is optional and only used for offline analysis and evaluation of the mechanism. The DR has no inherent requirements for data storage. If the storage option is used, then the collected samples are stored in SRAM to avoid additional traffic to the main memory.

C. Bandwidth Regulator

The bandwidth regulator module is the core component responsible for regulating the memory bandwidth of the NRT cores in every regulation interval.

The DR first computes the previous memory utilization at the start of each regulation interval. The memory budget (combined count of LLC misses and LLC write-backs) for each NRT core for the next regulation interval is then computed by comparing this utilization to the pre-defined memory utilization threshold. Afterwards, the computed memory bandwidth budget is used to set overflow interrupts for each NRT core. When the memory requests from a core exceed the set budget, the preset overflow interrupt for that core is triggered, and the interrupt handler then suspends the corresponding core from requesting any further memory requests until the regulation interval ends. We use the perf event infrastructure to configure these overflow interrupts, which are present inside the PMU.

Algorithm 1 sketches the working of the Dynamic Regulator. Let the total number of NRT cores be denoted by q. Let $b_{i,r}$ denote the memory budget allocated to each NRT core π_i in a given regulation interval r. Furthermore we denote by $b_{i,1}$, the initial budget assigned to the core π_i. The initial budgets can be user-specified or estimated by profiling the memory request patterns of the NRT applications and arriving at a conservative estimate. Let G_r denote the global budget, available to all the NRT cores in regulation interval r. An initial global budget G_1 is calculated as the sum of the individual budgets of the NRT cores in the first regulation interval and is given by $G_1 = \sum_{i=1}^{q} b_{i,1}$.

At the beginning of each regulation interval $r > 1$, the bandwidth regulator first computes the previous memory utilization U_{r-1} (Line 4). It then compares this utilization U_{r-1}

Algorithm 1: Dynamic Regulator

```
input: number of cores q, step size $\delta$, threshold $U_T$,
       initial NRT core budget $b_{i,1} \forall\, i \in 1 \ldots q$,
       initial util $U_{init}$
1  $U_1 = U_{init}$, $r = 2$, ovrFlag = true
2  $G_1 = \sum_{i=1}^{q} b_{i,1}$
3  foreach regulation interval $r$ do
4      Compute the previous utilization $U_{r-1}$
5      $\delta = \frac{|U_T - U_{r-1}|}{2}$
6      if $U_{r-1} < U_T \wedge ovrFlag = true$ then
7          $G_r = G_{r-1} * (1 + \delta)$
8      else
9          $G_r = G_{r-1} * (1 - \delta)$
10     end
11     ovrFlag = false
12     foreach non-real-time core $\pi_i$, $\{i \in 1 \ldots q\}$ do
13         $b_{i,r} = G_r * \frac{m_{i,r-1}}{\sum_{k=1}^{q} m_{k,r-1}}$
14         Set overflow interrupt with budget $b_{i,r}$
15     end
16     foreach non-real-time core $\pi_i$, $i \in \{1 \ldots q\}$ do
17         Monitor cache misses and write-backs $m_{k,r}$
           // Whenever $m_{k,r}$ exceeds $b_{i,r}$ raise
              interrupt
18         if Event: overflow interrupt triggered then
19             suspend core $\pi_i$
20             ovrFlag = true
21         end
22     end
23     foreach non-real-time core $\pi_i$, $i \in \{1 \ldots q\}$ do
24         Reset overflow interrupt
25     end
26     $r = r + 1$
27 end
```

against the pre-defined threshold U_T (Line 6) and accordingly determines the memory budget for NRT cores for the next regulation interval. In general, in order to regulate the memory utilization, a global budget in regulation interval r denoted by G_r is computed as

$$G_r = G_{r-1} * (1 \pm \delta),$$

where δ represents the step size by which the budget is increased or decreased. In the Algorithm 1, the value of δ is set in an adaptive way as shown in Line 5, but the algorithm can also run with a static step size ranging between (0,1), which is further evaluated in Section V-A2. The difference between U_{r-1} and U_T, divided by a constant factor, is used as the adaptive step size. We arbitrarily chose a factor of 2 to represent a half of the difference between U_{r-1} and U_T.

The global budget G_r for the interval r is distributed among the q NRT cores $\pi_i, i \in 1 \ldots q$ proportional to the number of memory accesses $m_{i,r-1}$ issued by each NRT core π_i in the previous regulation interval $r-1$ (Line 13), such that the budget $b_{i,r}$ for core π_i is given by

$$b_{i,r} = G_r * \frac{m_{i,r-1}}{\sum_{i=1}^{q} m_{i,r-1}}$$

With this, the higher the memory request ratio ($\frac{m_{i,r-1}}{\sum_{i=1}^{q} m_{i,r-1}}$) of the core in the current regulation interval is, the higher its share of the budget for the next interval will become.

Finally, the bandwidth regulator then configures an overflow interrupt in the PMU with the newly computed budget $b_{i,r}$ (Line 14). It then monitors the relevant hardware performance events (e.g., LLC misses and write-backs) to keep track of the memory usage of the cores (Line 17). If the core π_i exceeds the configured budget $b_{i,r}$ at any point in the regulation interval, an overflow interrupt is triggered by the PMU and the Dynamic Regulator suspends core π_i until the end of the interval (Line 19).

The bandwidth regulator also keeps track of the NRT cores that exceed their budget and had to be suspended in the previous regulation interval to adjust the global budget. If $U_{r-1} < U_T$ and there exists at least one NRT core that was suspended in the previous interval $r-1$, then it implies that some core is underprovisioned and more budget must be allocated. Therefore, it computes a new global budget G_r for the interval r by increasing the previous budget G_{r-1} by δ. Similarly, if $U_{r-1} > U_T$ then the bandwidth regulator computes G_r by decreasing G_{r-1} budget of the interval r by δ.

D. User Interface Module

The Dynamic Regulator framework provides an interface for letting the user configure the key parameters, namely the timer period of the regulation interval, threshold U_T, number of samples to be stored and step size δ. Furthermore, the storage module can also be enabled or disabled. The interface is implemented using the *debugfs* file system of the Linux kernel.

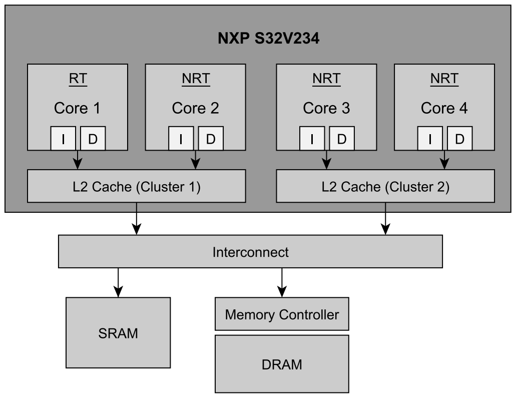


Fig. 3. Hardware architecture of our evaluation platform.

V. Evaluation Results and Analysis

We evaluate our approach on the NXP S32V234 [15] embedded platform. As shown in Fig. 3, the SoC features

TABLE I
BENCHMARK CHARACTERISTICS IN ISOLATION.

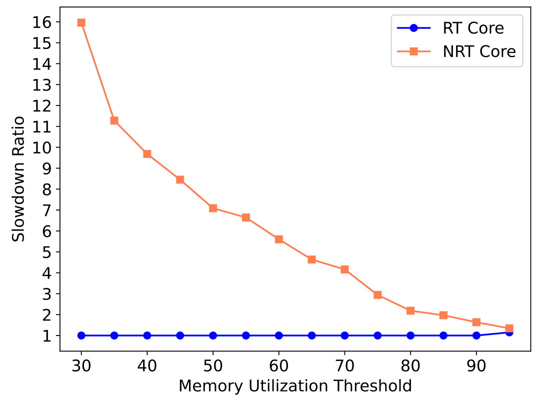
Benchmarks	Applications	Avg. IPC	Avg. Bandwidth
SD-VBS	multi_ncut	0.88	450 MB/s
	disparity	0.50	441 MB/s
	tracking	0.59	406 MB/s
	mser	0.67	328 MB/s
	sift	0.69	126 MB/s
	stitch	0.90	124 MB/s
	texture_synthesis	0.72	013 MB/s
DAPHNE	ndt_mapping	0.54	582 MB/s
	euclidean_cluster	0.87	301 MB/s
	points2image	0.85	189 MB/s
Isol-Bench	MemBomb	0.15	2321 MB/s

4 ARM Cortex A53 [22] CPUs, organized into 2 clusters each having 2 cores and clocked at 1GHz. Each core has its own private L1 data and instruction cache whereas the 2 cores within a cluster share a unified L2 cache. As our focus is not the shared cache, we use cores that do not share the same LLC for the majority of experiments. We include an analysis on four cores using applications that are not cache-sensitive by nature (e.g., where the working set size of each application is bigger than the LLC size). While this work is a proof of concept, cache partitioning is orthogonal and can be implemented within a hypervisor [23] and used in combination with the dynamic regulator.

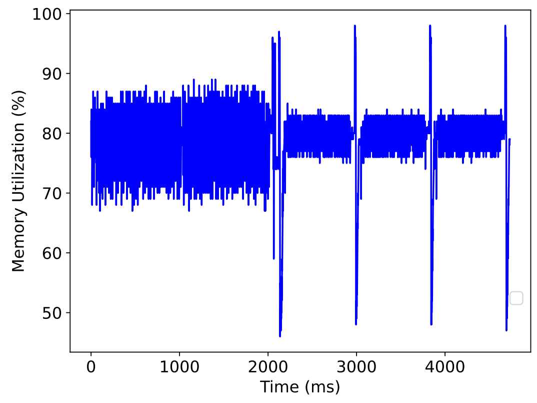
The Profiling Unit of the memory controller in our platform exposes a set of memory-mapped performance counters that report: (1) the number of DDR cycles elapsed, (2) the number of busy DDR cycles, (3) the total number of bytes transferred in read and (4) in write memory requests. The ratio between (2) and (1) provides the memory utilization, which forms the basis of our approach. We implement the Dynamic Regulator in Linux version 4.19 as a loadable kernel module and pin it to run on core 1.

A combination of real-world [12], [13] and synthetic [14] benchmarks are used to gain insight into the platform and evaluate the proposed approach. For our real-world benchmarks, we use a subset of the benchmarks in the San Diego Vision Benchmark Suite (SD-VBS) [12]. The input dataset for the benchmark applications comes in 9 different sizes. Since we are interested in applications that are DRAM-bound, we use the ones with the largest input data size (named *FullHD*). The other benchmark suite is the Darmstadt Automotive Parallel Heterogeneous Benchmark Suite (DAPHNE) [13], which represents parallelizable workloads from the automotive domain. For our evaluation, we used the applications that run exclusively on the CPU. We also use a synthetic 'Bandwidth' benchmark from the IsolBench suite [14] that is engineered to continuously perform memory write operations. In the rest of the paper, we refer to this benchmark as the *MemBomb* application.

Table I summarizes the characteristics of each benchmark considered in our evaluations. Benchmarks are listed in increasing order of average memory bandwidth usage measured when each benchmark runs in isolation on the evaluation platform. Notice that the benchmarks cover a wide range of memory bandwidth usage, ranging from 13MB/s (*texture_synthesis*) up to 2.3GB/s (*MemBomb*). We have excluded the *sift*, *texture_synthesis*, *stitch* and *point2image* benchmarks from our evaluation given their low bandwidth requirements and, therefore, insignificant impact of memory interference on their execution time.



(a) Improved performance (reduced slowdown) of the NRT application with higher utilization threshold



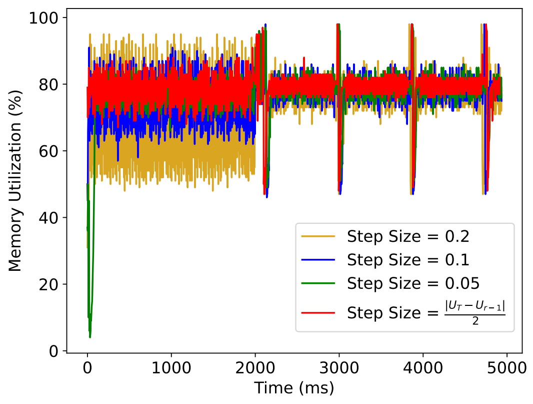
(b) Scenario showing overshooting of total memory utilization above pre-defined threshold of 80%

Fig. 4. Impact of the memory utilization threshold U_T on Dynamic Regulator behavior for *disparity* on RT core and *MemBomb* on NRT core.

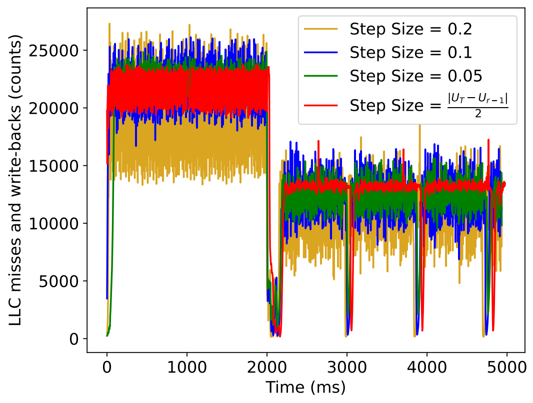
A. Sensitivity Analysis

One of the key issues for system designers in using any regulation mechanism is to configure the right system parameters. In this subsection, we first discuss the key design parameters of our approach, in particular the utilization threshold (U_T) and the step size (δ). Since our approach is self-regulatory due to dynamic feedback-based control, the impact of initial budgets is insignificant to the overall performance of the system. Therefore, we arbitrarily choose an initial budget of 50 MB/s for all our experiments.

1) Memory Utilization Threshold: We evaluate the impact of the pre-defined threshold U_T on the slowdown ratio of co-running applications. We define the slowdown ratio of an application as *the ratio of execution time in contention to the*



(a) System-wide Memory Utilization.



(b) Memory Access pattern of application running on NRT Core.

Fig. 5. Impact of the step size on Dynamic Regulator behavior for *disparity* on RT core and *MemBomb* on NRT core.

execution time in isolation. In the experiment shown in Fig. 4, the *disparity* application is run on the designated RT core, and an instance of the synthetic *MemBomb* application is run on the NRT core. This application is selected as it has the lowest average IPC and the highest average memory utilization in the benchmark suite, making it an ideal candidate for demonstrating memory interference-related effects.

A high memory utilization threshold causes bandwidth to be more aggressively allocated to the NRT core, which in turn results in a reduction of its slowdown ratio. This behavior can be clearly observed in Fig. 4(a), where the slowdown ratio of the application running on the NRT core decreases with an increase in the threshold, whereas the slowdown ratio of the application running on the RT core remains constant at about 1 until the threshold is set to a value of 95%, for which the RT core starts to see a slowdown due to overshoots in memory utilization caused by the reactive nature of DR as a function of the regulation interval.

These overshoots can be seen in Fig. 4(b) for a setup in which the threshold is set at 80%. The Dynamic Regulator targets to keep the memory utilization of the DRAM subsystem around this threshold. However, the memory utilization can exceed the pre-defined threshold for two reasons. Firstly, it is possible that the memory utilization requirement of the RT application, which is kept unregulated, exceeds this threshold. For example in Fig. 4(b), the RT application (*dispartiy*) memory utilization pattern shows four instances (around 2000 ms, 3000 ms, 3800 ms and 4600 ms) where the memory utilization is above 80%. These are the section of application, where the RT application in isolation has the memory utilization requirement of more than 80%. Hence, when the RT application is co-run with an NRT application (*MemBomb*), the memory utilization will exceed the pre-defined threshold of 80% to meet the RT application requirements.

Secondly, the Dynamic Regulator decreases the memory accesses of the NRT cores only when the memory utilization exceeds the pre-defined threshold (Line 9 in Algorithm 1). Therefore, a small memory utilization overshoot is expected due to the reactive nature of the mechanism, which has no significant impact on the slowdown ratio of the RT core (Fig. 4) as long as the utilization remains below 100%. This is why, at the 95% threshold, the RT application slowdown ratio rises to 1.1 (a 10% increase) due to overshoot that results in the saturation of DRAM subsystem. Note that with enough insight into DRAM operation, an adversarial NRT application could exploit the reactive DR nature by injecting overshoots as a function of the regulation interval and threshold. For these reasons, we set a conservative utilization threshold of 80% to account for overshooting margins and to ensure that we introduce no additional saturation at the DRAM subsystem.

2) Step Size: In this experiment, we evaluate the impact of step size δ on the performance of the Dynamic Regulator. The step size determines the delta value by which the memory budgets allocated to the NRT cores are increased or decreased in every regulation interval. We consider two scenarios for step sizes: (1) static and (2) adaptive. In the first scenario, the step size is assigned a fixed value for every regulation interval r (Line 5 in Algorithm 1), whereas in the case of an adaptive step size, the value for every regulation interval is given by $\delta = \frac{|U_T - U_{r-1}|}{2}$.

We co-run a real application from the SB-VDS suite (*disparity*) on the RT core with a synthetic application (*MemBomb*) on the NRT core and observe the memory utilization and memory bandwidth pattern of the workloads. As highlighted by the results in Fig. 5(a), setting a static step size leads to higher oscillations around the pre-defined threshold of 80% in comparison with an adaptive step size. This, in turn, causes higher oscillations in the allocated bandwidth for NRT cores, as seen in Fig. 5(b). Higher oscillations can lead to memory utilization overshoots beyond the set threshold, which can result in missing real-time guarantees in case the memory utilization reaches 100%.

B. Overhead Analysis

We further conducted experiments to evaluate the performance overhead incurred by our proposed mechanism. The key sources of the overhead arise from 1) the periodic interrupt that each core receives (either timer or IPI interrupt) at the beginning of every regulation interval; 2) the hardware performance counter overflow interrupt that is triggered whenever the actual

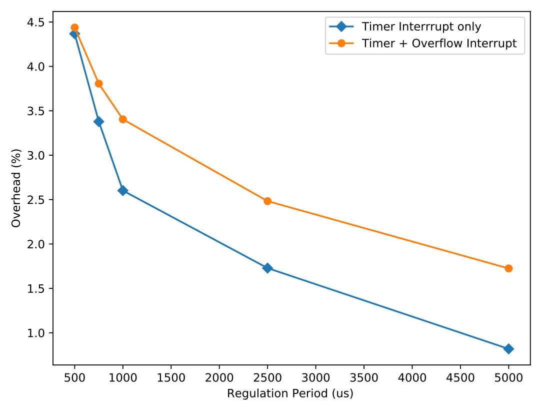


Fig. 6. Execution time overhead versus regulation interval.

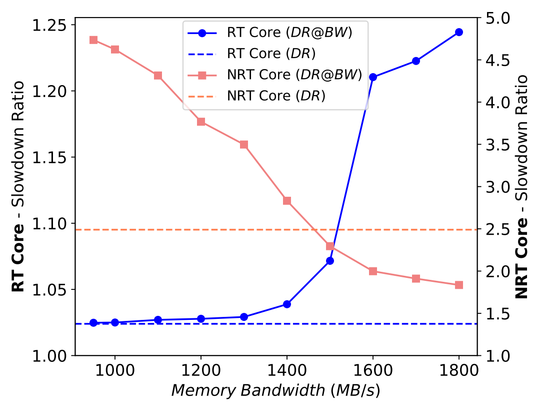


Fig. 7. Impact of the Memory Bandwidth threshold B_T on slowdown ratio for *disparity* on RT core and *MemBomb* on NRT core.

memory bandwidth of an NRT core exceeds its currently configured value, and 3) the increased memory accesses due to cache evictions by the execution of interrupt handlers of the DR, which indirectly increases the application execution time.

In order to quantify the overhead caused by the Dynamic Regulator, we conducted an experiment by running an instance of *MemBomb* in isolation on one of the NRT cores. In the first case, we measure the execution time of *MemBomb* in an unregulated and standalone scenario to obtain its baseline execution time. We then repeat the experiment using the Dynamic Regulator under two different scenarios: 1) timer-only and 2) timer with overflow. In the timer-only scenario, the utilization threshold is set above 100%, so that the core receives only the periodic timer interrupts at every regulation interval. In the timer with overflow scenario, NRT cores are set to a fixed low budget of 10 MB/s (Line 11 in Algorithm 1), so as to trigger an overflow interrupt at every regulation interval in addition to the periodic timer interrupt. Since setting a low budget can itself cause an increase in execution time due to core suspension, we disable core suspension, i.e. comment out Line 19 in Algorithm 1.

Fig. 6 shows the performance overhead, i.e., the increased execution time with respect to the unregulated baseline scenario as a function of the regulation interval. It can be noted that the overhead for a regulation interval of 1ms is only 2.6%. Overhead increases with a reduction in the regulation interval, reaching up to 4.4% for a 500μs regulation interval. Based on the results, we pick 1ms as the default regulation interval and use it throughout our evaluation.

C. Comparing Dynamic Regulation based on Utilization and Bandwidth Thresholds

To demonstrate that memory utilization as a metric is more beneficial than memory bandwidth, we implemented another version of Dynamic Regulator that is based on a memory bandwidth threshold instead. In order to realize this, we make the following changes in Algorithm 1 to have regulation based on a memory bandwidth threshold:

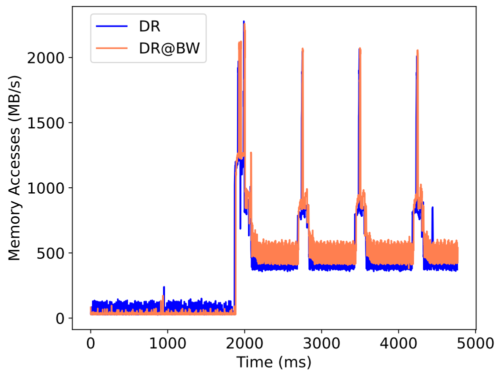
1) The previous memory utilization U_{r-1} is set to measure the previous total memory requests toward the DRAM subsystem from all the cores (Line 4 in Algorithm 1).
2) The step size δ is set to a small static value of 0.05 (Line 5 in Algorithm 1) for a gradual change in memory budgets for NRT cores.
3) The threshold U_T is replaced by B_T, representing a predefined memory bandwidth threshold.

In the remainder of the paper, we refer to this modified algorithm as *DR@BW*. The total number of memory requests are measured using the Profiling Unit of the memory controller. We use the guaranteed (worst-case) bandwidth [9] of the DRAM subsystem as the memory bandwidth threshold. The guaranteed bandwidth of the DDR3 memory used in our evaluation is approximately 950 MB/s based on the work in [11].

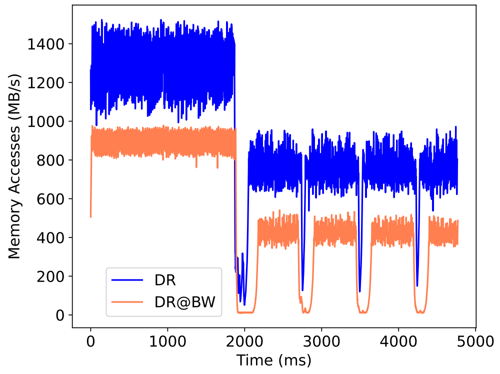
We further highlight the rationale of using guaranteed bandwidth as the threshold in *DR@BW* by experimentally observing the impact of various memory bandwidth thresholds on the slowdown ratio of the applications in RT and NRT cores. For this experiment, we run a real application (*disparity*) on the RT core alongside a synthetic application (*MemBomb*) on the NRT core.

A high memory bandwidth threshold implies that bandwidth is more aggressively allocated to the NRT core, which in turn results in a reduction of its slowdown ratio. This behavior can be clearly observed in Fig. 7, where the slowdown ratio of application run on the NRT core decreases with the increase in the threshold. However, when the memory bandwidth threshold is raised over the guaranteed bandwidth (950 MB/s), the RT application suffers from memory interference, resulting in an increase in the slowdown ratio. Hence, in order to make sure that the guarantees for the RT applications are always met, the memory bandwidth threshold is set to the guaranteed bandwidth of the DRAM subsystem.

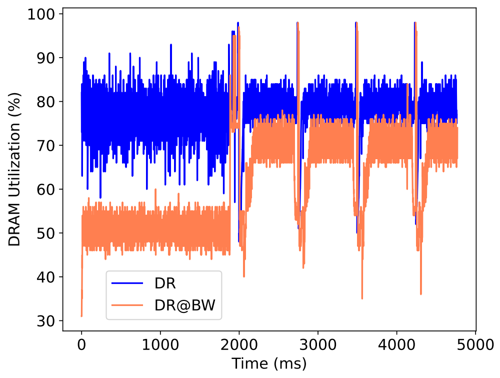
We evaluate the overall performance benefit of using memory utilization rather than memory bandwidth as the basis for memory bandwidth regulation by comparing DR effectiveness against *DR@BW*. We co-run a real application (*disparity*) on



(a) Similar memory behavior for RT application.



(b) The DR provides 2x more NRT memory bandwidth than *DR@BW*.



(c) Higher memory utilization achieved by the DR in comparison to the *DR@BW*.

Fig. 8. Comparison of the performance of Dynamic Regulator with Memory Bandwidth and Memory Utilization as threshold for *disparity* on RT core and *MemBomb* on NRT core.

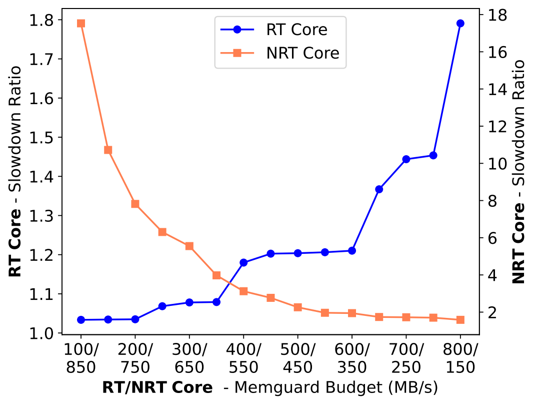


Fig. 9. MemGuard budget selection for *disparity* on RT core and *MemBomb* on NRT core.

the RT core and a synthetic application (*MemBomb*) on the NRT core. To ensure fair comparison, we used the same step size of 0.05 when comparing the two mechanisms.

Fig. 8(a) and Fig. 8(b) show the memory bandwidth usage, over time, of the RT core and NRT core respectively for both these approaches. While Fig. 8(c) shows the overall system-wide memory utilization over time. The RT core shows similar memory patterns in Fig. 8(a) under both approaches as they both aim to maintain native execution time without stalls.

Fig. 8(b) shows that *DR@BW* limits the combined memory bandwidth of RT and NRT cores based on the pre-defined memory bandwidth threshold of 950 MB/s (guaranteed bandwidth). In order to do so, in each regulation interval, the *DR@BW* observes the bandwidth consumed by the RT core and allocates the residual bandwidth (guaranteed bandwidth - RT core bandwidth) to the NRT cores. As seen in Fig. 8(a), during the initial 2000 ms of execution, the RT application has a relatively low bandwidth requirement and therefore the regulation approach allocates a major portion of the available memory bandwidth to the NRT core during this phase of execution. After the initial 2000 ms, the RT application has a varying bandwidth requirement with an average bandwidth of 500 MB/s and peak bandwidth of approximately 2000 MB/s, and it can be seen how the *DR@BW* correspondingly varies the allocation to the NRT cores, to limit the combined bandwidth usage of the RT and NRT cores, to the guaranteed bandwidth.

The clear advantage of using the utilization as a metric comes across in Fig. 8(c). Here we see that the overall memory utilization achieved by DR is higher than that achieved by *DR@BW*. Secondly, this also validates that DR limits the memory utilization of the entire system according to the pre-defined memory utilization threshold of 80%. Overall, DR using memory utilization is able to improve the overall memory utilization by 1.5x and provide 2x more NRT memory bandwidth than *DR@BW* for this experimental setup, all while maintaining guarantees for applications running in the RT core.

D. Comparison of Dynamic Regulator versus bandwidth-based approaches on 2 cores

In this experiment, we compare the slowdown experienced by the RT and NRT applications co-run on two cores under the following scenarios i) Unregulated execution, in which the applications are co-run in their respective cores with no regulation mechanism in place, 2) Dynamic Regulator, 3) a dynamic bandwidth-based approach (*DR@BW*), and 4) a static bandwidth-based approach (MemGuard [10]).

Comparison against MemGuard: We use the latest implementation of MemGuard [10] that regulates LLC write-backs in addition to LLC misses and configured it to only

TABLE II
SLOWDOWN RATIO OF BENCHMARKS IN CONTENTION WITHOUT REGULATION AND WITH DIFFERENT REGULATION MECHANISMS FOR TWO CORES.

RT Core					**NRT** Core				
Benchmarks	**Unregulated**	**MemGuard**	**DR@BW**	**DR**	**Benchmarks**	**Unregulated**	**MemGuard**	**DR@BW**	**DR**
disparity	1.22	1.03 (750)	1.03	1.02	MemBomb	1.21	7.82 (200)	4.70	2.43
tracking	1.35	1.04 (850)	1.03	1.03	MemBomb	1.16	17.72 (100)	5.06	2.61
mser	1.16	1.04 (700)	1.01	1.01	MemBomb	1.26	8.10 (250)	3.90	2.14
multi_ncut	1.11	1.04 (800)	1.02	1.02	MemBomb	1.13	11.23 (150)	5.84	2.16
ndt_mapping	1.27	1.05 (750)	1.03	1.03	MemBomb	1.43	9.41 (200)	6.84	3.47
euclidean	1.08	1.02 (700)	1.01	1.01	MemBomb	1.29	5.61 (250)	4.93	2.22
Average	***1.20***	***1.04***	***1.02***	***1.02***	***Average***	***1.25***	***9.98***	***5.21***	***2.51***
disparity	1.08	1.02 (750)	1.02	1.02	tracking	1.04	2.30 (200)	1.71	1.37
disparity	1.05	1.01 (750)	1.01	1.02	ndt_mapping	1.03	1.97 (200)	1.68	1.16
tracking	1.09	1.02 (850)	1.01	1.02	ndt_mapping	1.03	3.57 (100)	1.86	1.33
tracking	1.04	1.02 (850)	1.02	1.01	mser	1.05	2.10 (100)	1.43	1.25
tracking	1.05	1.02 (850)	1.01	1.01	multi_ncut	1.04	2.37 (100)	1.61	1.31
Average	***1.06***	***1.02***	***1.02***	***1.02***	***Average***	***1.04***	***2.46***	***1.66***	***1.28***

use static bandwidth reservation. We first determine, for each application, its static memory budget, which is the key parameter used by MemGuard. In order to determine the static bandwidth budget for the RT core, we vary its budget, and observe the corresponding slowdown experienced by the RT application, as seen in Figure 9. We use the lowest budget configuration which results in a similar slowdown as observed in isolation. Then, the static bandwidth budget for the NRT core is computed as the difference of guaranteed bandwidth and static bandwidth budgets for RT core. Fig. 9 illustrates the slowdowns as a function of budgets for two applications *disparity* and *MemBomb* that are co-run on RT and NRT cores, respectively. For this example, the budgets of 750 MB/s and 200 MB/s are selected for the RT and NRT cores, respectively. For our remaining experiments, we use the same methodology to select the memory budgets for MemGuard. Once the configurations for the MemGuard have been selected, we conducted the evaluation with different setups as described in the following.

1) Synthetic benchmarks on NRT core: In the first setup, we studied the detailed system performance by running a real application on the RT core along with a synthetic memory intensive application (*MemBomb*) on the NRT core. This setup is chosen to provide a worst-case scenario where the memory demand from the NRT core (when unregulated) alone can saturate the memory controller by hitting the to 100% memory utilization.

Table II shows the slowdown ratios for different execution settings as compared to the execution times in isolation. We compare unregulated execution in which the applications are run concurrently in the respective cores with no regulation mechanism in place to the proposed Dynamic Regulator, *DR@BW* and MemGuard. The memory utilization threshold of the DR is 80% and the memory bandwidth threshold for *DR@BW* is 950 MB/s. The budget of MemGuard is individually configured for each experiment and mentioned within brackets.

As expected, all regulation approaches successfully ensure enough memory bandwidth for the application on the RT core, denoted by the slowdown ratios close to 1. The application on the NRT core suffers the highest slowdown with MemGuard (on average 9.98). The dynamic regulation improve the slowdown significantly, with the utilization-based DR (on average 2.51) having a clear advantage over the bandwidth-based *DR@BW* (on average 5.21). In summary, the DR is able to improve the slowdown ratio of the NRT application on average by 4x in comparison with MemGuard and by 2x in comparison with *DR@BW*.

2) Real-world benchmarks on NRT core: In this setup, the objective is to conduct a comparison using real-world applications with varying memory loads. Here, real benchmark applications from the SB-VDS and DAPHNE benchmark suite are run on both RT and NRT cores. Again, the RT applications show similar memory behavior under all the regulation mechanisms as they maintain the same execution time, as can be observed in Table II. Due to their lower required memory bandwidth (cf. Table I), the overall slowdown ratios of the real-world NRT applications are lower than of the synthetic *MemBomb* benchmark. Similarly the improvement factors between the different regulation approaches are lower. However, the DR still improves the slowdown ratio of NRT applications on average by 2.2x in comparison with MemGuard and by 1.3x in comparison with *DR@BW*.

E. Comparison of Dynamic Regulator versus bandwidth-based approaches on 4 cores

Table III summarizes and compares the slowdown ratio under different approaches when co-running applications from real and synthetic benchmarks on four cores under four different scenarios. We use the same mechanisms and their configurations for comparison as mentioned earlier for two cores.

The results of the 4 core experiments are in line with the experiments on 2 cores. All regulation approaches effectively isolate the RT core as denoted by the slowdown ratios close to 1. With the synthetic *MemBomb* benchmark running on all 3 NRT cores, the DR shows an average improvement of slowdown ratios compared to MemGuard by 3.1x and compared to *DR@BW* by 1.7x. Similarly, for real-world applications on all 3 NRT cores, the DR improves the slowdown ratios on average

TABLE III
SLOWDOWN RATIO OF BENCHMARKS IN CONTENTION WITHOUT REGULATION AND WITH DIFFERENT REGULATION MECHANISMS FOR FOUR CORES.

Core1: **RT**					Core2: **NRT**				
Benchmarks	**Unregulated**	**MemGuard**	**DR@BW**	**DR**	**Benchmarks**	**Unregulated**	**MemGuard**	**DR@BW**	**DR**
disparity	1.80	1.04 (750)	1.04	1.04	MemBomb	2.47	48.04 (67)	29.74	13.84
tracking	2.02	1.06 (850)	1.04	1.04	MemBomb	2.62	73.47 (33)	29.74	15.71
Average	***1.91***	***1.05***	***1.04***	***1.04***	***Average***	***2.55***	***60.75***	***29.74***	***14.77***
disparity	1.22	1.05 (750)	1.03	1.03	tracking	1.27	3.07 (67)	2.34	1.87
ndt_mapping	1.23	1.05 (750)	1.03	1.03	euclidean	1.08	4.52 (67)	2.07	1.55
Average	***1.23***	***1.05***	***1.03***	***1.03***	***Average***	***1.18***	***3.79***	***2.21***	***1.71***

Core3: **NRT**					Core4: **NRT**				
Benchmarks	**Unregulated**	**MemGuard**	**DR@BW**	**DR**	**Benchmarks**	**Unregulated**	**MemGuard**	**DR@BW**	**DR**
MemBomb	4.30	48.50 (67)	36.20	20.82	MemBomb	4.29	49.47 (67)	35.94	20.82
MemBomb	4.44	73.47 (33)	37.01	23.24	MemBomb	4.47	74.57 (33)	36.47	22.61
Average	***4.37***	***60.99***	***36.61***	***22.03***	***Average***	***4.38***	***62.02***	***36.20***	***21.71***
mser	1.23	2.15 (67)	1.75	1.35	multi_ncut	1.14	2.71 (67)	2.32	1.94
tracking	1.33	2.57 (67)	2.24	1.83	multi_ncut	1.15	2.49 (67)	2.29	1.76
Average	***1.28***	***2.36***	***1.99***	***1.59***	***Average***	***1.15***	***2.60***	***2.31***	***1.85***

by 1.7x compared to MemGuard and by 1.3x compared to *DR@BW*.

It is important to remember that COTS-based memory controllers are optimized for high memory throughput [24], but do not provide predictable timing among memory requests from different cores in multi-core platforms. In order to provide predictability for time-critical applications, all regulation approaches need to allocate sufficient memory bandwidth to the RT core such that real-time applications meet their timing requirements, while NRT cores may suffer performance degradation compared to an unregulated scenario due to a reduced memory bandwidth allocation. As a result, a trade-off exists between performance and predictability.

Finally, it is worth noting that even with a conservative memory utilization threshold of 80%, the DR outperforms other mechanisms. With an increased threshold, the DR will show an even higher performance improvement if occasional slowdowns of the RT core are acceptable.

VI. RELATED WORK

Existing studies that addressed memory interference in a multi-core real-time environment can broadly be classified into hardware-oriented and software-oriented mechanisms. On the hardware front, embedded high performance platforms are increasingly offering QoS modules on the interconnect [25], [26] between the masters (CPUs, GPUs, DMAs) and the main memory, which can regulate and prioritize the memory requests. However, even with the existing QoS modules there are two main concerns [5]. Firstly, the QoS module may treat the entire core cluster as a single master, because the core cluster is connected to the interconnect at a single port. With this, the QoS module may offer regulation only at the core-group level and may not differentiate among different cores, which does not solve the problem of cross-core contention. Secondly, a static configuration of the QoS module parameters may not be sufficient to efficiently use the underlying DRAM due to varying workloads; we additionally need a software mechanism that takes into account the changing memory usage patterns of applications and reconfigures these parameters in the QoS modules.

Using a dedicated memory controller [6] or additional hardware like FPGAs [7], [8] is a common approach to reduce memory interference. However, relying on dedicated hardware is not feasible for systems employing COTS components, which come with fixed architectures and memory controller designs.

With the need for easy portability and minimal dependency on platform architectures, OS-level software solutions like MemGuard [9] are preferred. The base version of MemGuard statically regulates memory budgets of each core based on a pessimistic worst-case bandwidth estimation. This can lead to an under-provisioning of memory bandwidth since it does not account for dynamic workload behavior and traffic patterns of applications. We do not compare against the predictive mechanisms proposed by MemGuard, since our approach is orthogonal and is not comparable (the RT core is unregulated and we are not working with global reclamation). However, in order to validate our approach, we make a fairer comparison using the same reactive Dynamic Regulator with either memory utilization or bandwidth as metric for regulating memory budgets for each core.

Other dynamic bandwidth regulation mechanisms [27], [28], [29] perform offline analysis and budget estimation based on worst-case memory access latency metrics, potentially using feedback-based control [30]. However, all these mechanisms suffer from pessimistic budget and worst-case execution time (WCET) estimations. They are agnostic to the memory utilization, and compute the allocated bandwidth values based on an offline analysis, hence are suitable only for a statically known sets of workloads.

By contrast, our work proposes the use of a different metric for bandwidth regulation, namely memory utilization, which provides the actual saturation level of the DRAM. Combining memory utilization with per-core memory bandwidth usage and feedback-based control, Dynamic Regulator is able to provide guaranteed bandwidth to the RT core and efficiently allocate remaining bandwidth to NRT cores. The proposed feedback-based control is closely related to classic PID approaches in control theory.

VII. Summary and Conclusions

In this work, we presented a feedback control-based dynamic memory bandwidth regulation mechanism to support efficient temporal isolation in COTS multi-core platforms. We highlighted with examples that memory bandwidth measured at the source cores by itself is not a key indicator of the underlying memory saturation and hence interference. Therefore, alternative mechanisms that are based on the actual memory utilization are warranted. Based on this insight, we proposed a mechanism for regulating memory bandwidth based on the memory utilization measured at the DRAM controller.

We implemented our solution in Linux as a loadable kernel module and deployed it on a COTS multi-core (NXP S32V234) platform to demonstrate its effectiveness compared against static and dynamic bandwidth-based approaches. Results show that our approach reduces execution time of non-real-time applications by up to 4x while maintaining guarantees for real-time applications. As future work, we plan to extend our approach to a system-level solution by regulating memory requests emerging from other masters like hardware accelerators and GPUs in addition to cores in the systems-on-chip. We also plan to further reduce the implementation overhead in order to use smaller regulation intervals and hence accommodate more fine-grained bandwidth regulation. Our approach can also be extended to handle multiple RT cores by additionally assigning and enforcing static bandwidth limits to each RT core based on offline analysis, in such a way that in any regulation interval the combined memory utilization of all RT cores is always below 100% while also ensuring that sufficient bandwidth limits are set to ensure that the application meets its end-to-end requirements.

Acknowledgement

This work has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 871669.

References

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

[2] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[3] R. Pellizzoni and H. Yun, "Memory Servers for Multicore Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[4] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[5] F. Rehm, J. Seitter, J.-P. Larsson, S. Saidi, G. Stea, R. Zippo, D. Ziegenbein, M. Andreozzi, and A. Hamann, "The road towards predictable automotive high - performance platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021.

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

[7] D. Hoornaert, S. Roozkhosh, and R. Mancuso, "A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

[8] J. Freitag and S. Uhrig, "Closed Loop Controller for Multicore Real-Time Systems," in *Architecture of Computing Systems (ARCS)*, 2018.

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

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

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

[12] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[13] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, "DAPHNE - An automotive benchmark suite for parallel programming models on embedded heterogeneous platforms: work-in-progress," in *International Conference on Embedded Software Companion (EMSOFT)*, 2019.

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

[15] "S32V Vision and Sensor Fusion Evaluation Board." https://www.nxp.com/design/development-boards/automotive-development-platforms/s32v-mpu-platforms/s32v-vision-and-sensor-fusion-evaluation-board:SBC-S32V234.

[16] D. Dasari, B. Akesson, V. Nlis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

[17] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[18] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–36, 2015.

[19] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-Based Multicore Cache Management," in *ACM European Conference on Computer Systems*, EuroSys '09, 2009.

[20] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems," 2009.

[21] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.

[22] "ARM Cortex-A53 MPCore Processor - Technical Reference Manual." https://static.docs.arm.com/ddi0500/f/DDI0500.pdf.

[23] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[24] P. K. Valsan and H. Yun, "MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems," in *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2015.

[25] A. Stevens, "Quality of Service (QoS) in ARM Systems: An Overview," in *ARM White paper*, 2014.

[26] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

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

[28] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study," in *Euromicro Conference on Real-Time Systems (ECRTS)* (M. Bertogna, ed.), Leibniz International Proceedings in Informatics (LIPIcs), 2017.

[29] K. Lampka and A. Lackorzynski, "Resolving contention for networks-on-chips: Combining time-triggered application scheduling with dynamic budgeting of memory bus use," in *International GI/ITG Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, Springer, 2016.

[30] A. Crespo, P. Balbastre, J. Sim, J. Coronel, D. Gracia Prez, and P. Bonnot, "Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems," *IEEE Access*, vol. 6, pp. 50627–50640, 2018.