Fine-Grained QoS Control via Tightly-Coupled Bandwidth Monitoring and Regulation for FPGA-based Heterogeneous SoCs

Giacomo Valente, Gianluca Brilli, *Member, IEEE,* Tania Di Mascio, *Member, IEEE,* Alessandro Capotondi, *Member, IEEE,* Paolo Burgio, *Member, IEEE,* Paolo Valente, *Member, IEEE,* Andrea Marongiu, *Member, IEEE*

Abstract—Commercial embedded systems increasingly rely on heterogeneous architectures that integrate general-purpose, multi-core processors, and various hardware accelerators on the same chip. This provides the high performance required by modern applications at a low cost and low power consumption, but at the same time poses new challenges. Hardware resource sharing at various levels, and in particular at the main memory controller level, results in slower execution time for the application tasks, ultimately making the system unpredictable from the point of view of timing. To enable the adoption of heterogeneous systems-on-chip (SoCs) in the domain of timing-critical applications several hardware and software approaches have been proposed, bandwidth regulation based on monitoring and throttling being one of the most widely adopted. Existing solutions, however, are either too coarse-grained, limiting the control over computing engines activities, or strongly platform-dependent, addressing the problem only for specific SoCs. This paper proposes an innovative approach that can accurately control main memory bandwidth usage in FPGA-based heterogeneous SoCs. In particular, it controls system bandwidth by connecting a runtime bandwidth regulation component to FPGA-based accelerators. Our solution offers dynamically configurable, fine-grained bandwidth regulation – to adapt to the varying requirements of the application over time – at a very low overhead. Furthermore, it is entirely platform-independent, capable of integration with any FPGA-based accelerator. Developed at the register-transfer level using a reference SoC platform, it is designed for easy compatibility with any FPGA-based SoC. Experimental results conducted on the Xilinx Zynq UltraScale+ platform demonstrate that our approach (i) is more than $100\times$ faster than loosely-coupled, software controlled regulators; (ii) is capable of exploiting the system bandwidth 28.7% more efficiently than tightly-coupled hardware regulators (e.g., ARM CoreLink QoS-400, where available); (iii) enables task co-scheduling solutions not feasible with state-of-the-art bandwidth regulation methods.

Index Terms—Embedded Systems, Memory Interference, Bandwidth Monitoring, Bandwidth Regulation

1 Introduction

The current generation of embedded systems widely relies on Heterogeneous System on Chips (HeSoCs) where general-purpose multi-cores are coupled to HW accelerators and application-specific processors [1]–[3]. The adoption of such systems provides abundant computing power to satisfy the needs of modern applications, with plenty of SW and HW tasks executing in parallel, but – at the same time – poses novel challenges. In particular, as the number of on-chip *compute engines* (CE) grows – including higher CPU and GPU core counts, as well as more application-specific *accelerators* – the interference due to main memory sharing significantly impacts the application tasks' execution time [4], [5]; in turn, this makes the system unpredictable from the point of view of timing. This constitutes a major problem for the adoption of Commercial Off-the-shelf (COTS) HeSoCs in application domains where timing predictability is required [6]. Several solutions have been proposed to tackle this problem, ranging from static memory partitioning techniques [7] to task execution models that guarantee predictable memory access [8] and memory bandwidth regulation strategies [9]. The latter, in particular, are increasingly being made available also in commercial products [10]–[12], and rely on bandwidth *monitoring* and *throttling* mechanisms. *Throttling* is an effective way of limiting the bandwidth allowed for a particular CE by interspersing the required memory transaction with idle periods at the system bus/interconnection level, typically at the granularity of small bursts of few hundred bytes [13]. To satisfy the Quality of Service (QoS) requirement of one (or more) *critical* CE, the remaining CEs should have their bandwidth usage limited to a degree that does not slow down the execution time of the *critical* task beyond what is tolerated. Throttling should only be applied to the minimum extent necessary to satisfy the predictability requirements. Any bandwidth that remains unused should be made available for reuse by other system components. This unused bandwidth is referred to as *residual* bandwidth.

Statically configuring the QoS level of every CE in the system is ineffective in contexts where several HW and SW tasks come and go in a very dynamic manner, for several

- *G. Valente and T. Di Mascio are with DISIM Department, University of L'Aquila, 67100 L'Aquila, Italy. E-mail: {giacomo.valente, tania.dimascio}@univaq.it*
- *G. Brilli, A. Capotondi, P. Burgio, P. Valente, and A. Marongiu are with Department of Physics, Informatics, and Mathematics, University of Modena and Reggio Emilia, 41125 Modena, Italy. E-mails: {gianluca.brilli, alessandro.capotondi, paolo.burgio, paolo.valente, andrea.marongiu}@unimore.it*

Manuscript received Jul 17, 2024; revised Nov 08, 2024.

reasons: (i) the same CE can host different tasks over time, with different predictability requirements; (ii) different tasks might exhibit different use of the memory bandwidth, thus contributing to the interference phenomenon to different degrees; (iii) the QoS regulations for the CEs hosting such tasks change depending of the overall system load in that particular moment (i.e., the number of CEs concurrently accessing memory and mutually interfering on execution time).

QoS regulation and *throttling* should thus rely on continuous *monitoring* of the actual bandwidth usage by the various CEs, resulting in a **coupled interaction** between the *monitoring* and *throttling* phases operating in a closed-loop manner. In the literature, the concept of a *System Controller* has been proposed [14], which, upon the admission of a new task to the system or the completion of a previously admitted task, inspects the bandwidth usage of all running tasks and resets the QoS regulations for all the involved CEs. Intuitively, the finer the **granularity** at which this operation can be achieved, the wider the scope of application of the technique. For example, in application domains such as advanced system automation (e.g., robotics, autonomous cars, unmanned aerial vehicles), numerous HW and SW tasks with durations and periods below the millisecond boundary are involved [15], [16]. The granularity of the technique depends mainly on two aspects: (i) the size of the unit data transfer that is monitored and upon the duration of which the idle period is determined; (ii) the coupling between *monitoring* and *throttling*. Concerning the first point, intuitively the smaller the data transfer, the shorter the time to complete a full regulation cycle but also the more impactful the overhead for the *monitoring* operation itself. Concerning the second point, the closed-loop interaction between *monitoring* and *throttling* also implies an overhead, which can only be reduced by tightly coupling the operation of these two phases. Needless to say, the looser the coupling, the less precise the QoS guarantee provided on *critical* tasks.

Focusing on commercial HeSoCs based on Field-Programmable Gate-Arrays (FPGA), previous research has mostly explored software-based, loosely-coupled approaches for bandwidth regulation on the CPU cores. These methods have been explored both in industry [17], [18] and academic [14], [16], [19], [20], with only few approaches targeting the regulation of accelerators deployed on the FPGA logic and rarely considering interference at the whole SoC level [21]. Hardware-based approaches, both from the research community [22]–[24] and from commercial products [13], [25], [26], although more fine-grained and tightly-coupled, tend to address the problem only for specific SoCs or interconnect protocols, limiting their application to those platforms where such hardware is available.

In this paper, which is a significantly extended version of previously published work [27], we propose *an innovative, fine-grained and platform-independent Runtime Bandwidth Regulator (RBR) for disciplined main memory bandwidth usage in COTS FPGA-based HeSoCs*. The RBR is meant as a non-invasive extension of a standard Direct-Memory Access (DMA) interface for FPGA-based accelerators, aimed at delivering tightly-coupled *monitoring* and *throttling* of main memory bandwidth, effectively delivering the desired QoS levels with high precision while efficiently exploiting the *residual* bandwidth. The proposed RBR quickly adapts to time-varying QoS levels and is completely platform-independent, as it is developed targeting a general reference architecture of an FPGA-based HeSoC and implemented as an HDL design that can be synthesized to every FPGA-based HeSoC with minimal area and time overhead.

Our experimental results show that the proposed tightly-coupled bandwidth regulation scheme can precisely track and very quickly adapt to dynamic QoS variations as small as 1% with a timing resolution – defined as the minimum time interval required to adjust the bandwidth to a specific value – of just $32\mu s$ for an entire worst-case regulation cycle (99% of the cycle is idle time). If coarser regulation steps resolutions are sufficient for the application at hand, the RBR can be reconfigured at runtime to operate at a finer timing resolution of up to $0.33\mu s$ for the entire worst-case regulation cycle.

This approach makes bandwidth regulation highly effective for applications with timing resolution one to two orders of magnitude finer than what is achievable with state-of-the-art solutions [19], [20] based on loosely-coupled, SW-controlled monitoring + throttling. Although fully HW-based regulation is not always available on COTS platforms, we also compare our proposal to the ARM CoreLink QoS400 regulators. Experimental results show that we achieve comparable regulation speed to QoS-400, with a finer regulation step and 28.7% better exploitation of the *residual* bandwidth. This allows a higher number of SW and HW tasks to safely co-execute compared to other approaches, where a less efficient system-wide exploitation of the system bandwidth can only meet the QoS requirements by conservatively reducing the number of CEs accessing main memory in parallel.

The rest of the paper is organized as follows: Section 2 positions our contribution with respect to related work. Section 3 introduces background information on top of which we build our proposal. Section 4 presents the proposed tightly-coupled regulation mechanism. Section 5 provides the evaluation of the proposed mechanisms on the Zynq UltraScale+ platform. Section 6 concludes the paper.

2 Related Work

Memory interference can be very impactful on the performance of modern HeSoCs. This has motivated a lot of characterization work in the recent past, focusing on the effects on the main CPU [4], [28], GPGPU accelerators [29], [30], FPGA-based accelerators [21], [31], and external I/O components [10], [32], [33]. All the previous work showed that unmanaged concurrent accesses to main memory on HeSoCs can lead to dramatic slowdowns, making the system unpredictable from the point of view of timing behavior.

A simple, widespread approach to mitigating these effects is that of enforcing mutually exclusive memory access to the main memory (DRAM) [28]. Several works rely on this principle [8], [34]–[36] across a wide range of target architectures, granularity settings and scheduling approaches. Although functional, these approaches are often too conservative and pessimistic, as their *one-at-a-time* execution model induces a severe under-utilization of the available

DRAM bandwidth in modern HeSoCs, limiting the overall throughput.

Other works try to overcome such underutilization by allowing a controlled number of tasks to access DRAM at the same time [37], [38]. Task-based scheduling for memory accesses, however, does not allow for fine-grained control. Other approaches improve DRAM bandwidth usage by relying on offline profiling to devise efficient task scheduling and bandwidth allocation [39], [40]. The main limitation of such approaches resides in their static nature, which is not always practical or altogether feasible in the context of modern time-critical applications. Controlled Memory Request Injection (CMRI) [41] also allows more than one task at a time to access DRAM, interspersing memory requests from the tasks with a controlled amount of idle cycles. This is achieved by wrapping task execution within fine-grained, controllable duty cycles. Both compiler-level code instrumentation and dynamic task throttling, at SW-level (e.g., the *Memguard* solution [19]), hypervisor-level [42], or DMA-level [21], can be used to implement the duty cycling. However, CMRI techniques are limited to millisecond-scale timing resolution. As noted in the works [14], [16], attempting to regulate bandwidth at a sub-millisecond timing resolution with CMRI techniques results in substantial overhead (i.e., slowdown on the regulated task) – up to 10% for a timing resolution of 100 us – making them unsuitable for applications demanding finer timing control.

New techniques are being introduced to protect real-time applications running on CPU cores from memory interference, achieving sub-millisecond timing resolution [14], [16]. This shift to finer timing resolution is made possible by dedicated hardware components that handle bandwidth monitoring and throttling. By offloading these operations from the CPU, the components significantly reduce the processing load on the system, allowing for more precise control of memory access and ensuring minimal interference with real-time operations.

Saeed et al. [14] developed a mechanism to control memory interference by considering DRAM utilization through HW performance monitors and regulating interference sources at the task execution level on a per-core basis. Their solution employs core-managed interrupts from HW performance monitors, leading to a high regulation overhead that prevents a finer timing resolution than 500 us. To achieve finer timing resolution regulation, Zuepke et al. [16] introduced *MemPol*, a BR mechanism that uses HW performance monitors for monitoring and debugging mechanisms for bandwidth control. The approach relies on a dedicated core to control the monitoring and throttling loop with tighter coupling compared to other SW approaches, thereby significantly reducing the associated overhead. Although its experimental evaluation has been conducted on FPGA-based HeSoCs, MemPol – like all the other approaches cited up to this point – focuses on BR across CPU cores only, with currently no support for the control of HW accelerators. Also Farshchi et al. [9], who rely on the FPGA logic on a HeSoC to implement a HW throttler, uses the programmable logic only as a medium to control the traffic between the CPU and the shared buses, not to host accelerators. The solution offers loosely-coupled *monitoring + throttling* and a fairly coarse granularity due to the loose physical coupling between the CPU and the FPGA on the SoC, and to the slow speed of the FPGA logic compared to that of the CPU.

HW support for managing QoS and fairness at various levels of the interconnect hierarchy is increasingly being offered on commercial HeSoCs (e.g., ARM QoS400 [13], QVN400 [25], MPAM [26]). The availability of such support on current systems is however still very limited, sometimes only partial and typically very poorly documented (as highlighted in [10], [12]). Schwaricke et al. [20] used the ARM QoS400 for BR to ensure predictable data transfers between virtual machines. Their loosely-coupled approach is limited to AXI-based architectures [43], reducing its portability across different architectures. Other HW mechanisms have been proposed for BR on FPGA-based HeSoCs [22]–[24], again specifically targeted at AXI interconnect protocols. Here the bus activity is monitored to regulate the tasks executed on HW accelerators. Although these approaches, like ours, focus on BR for the HW accelerators in FPGA, the technique focuses on bus-level regulation, whereas our proposal tightly couples the monitoring and throttling logic with the DMA unit of the accelerator itself. Compared to these papers, from the point of view of the evaluation our focus is on interference at the whole SoC level, not the FPGA only. Moreover, our proposal (and the associated evaluation section) also focuses on efficient exploitation of the *residual* bandwidth. To the best of our knowledge, MemPol [16] is the only other work that evaluates this aspect, while compared to commercial solutions [13] our RBR achieved up to 28.7% better usage of the *residual* bandwidth.

In a nutshell, the proposed RBR design tackles the challenge of achieving fine-grained regulation of the FPGA-based accelerators' bandwidth on commercial HeSoCs by integrating a lightweight HW monitoring system and a bandwidth throttler with the most typical interface to the outer memory of an accelerator, the DMA. The RBR can be seamlessly incorporated into any generic FPGA-based accelerator design with minimal overhead. The RBR dynamically and precisely controls the memory bandwidth generated by the accelerator, enhancing precise QoS control of *critical* tasks in the system and efficient overall system bandwidth utilization.

3 Background

Fig. 1 shows a simplified block diagram of the reference FPGA-based HeSoC. In this template, the main *host* multi-core CPU shares the main DRAM memory with the FPGA logic. Here, one or more application-specific *accelerators* can be deployed. Internally, every *accelerator* includes a *datapath*, namely the core logic that performs the computation, and an efficient *DMA engine*, used to facilitate the staging of data from the DRAM into faster local memories. In modern HeSoCs, the DMAs inside FPGA-based accelerators generate much higher DRAM bandwidth requests than what happens on the CPU cores [12], [31] (e.g., in [44], [45] HeSoCs). This is because CPU cores typically generate a limited number of read and write transactions to memory, constrained by the number of in-flight memory operations they can handle. In contrast, FPGA designs allow multiple accelerators to be connected to the same memory port,

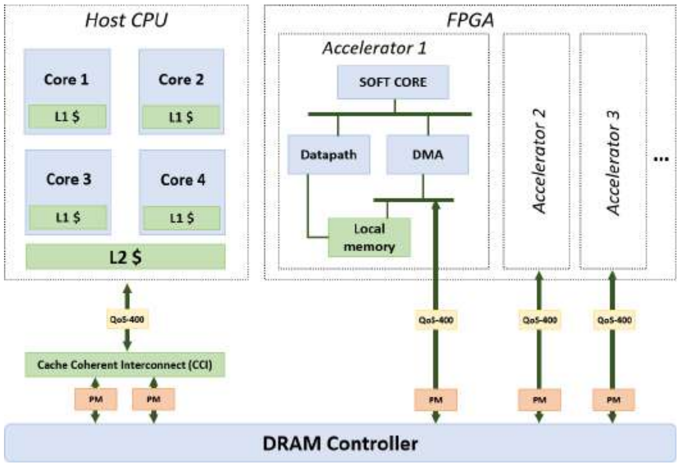
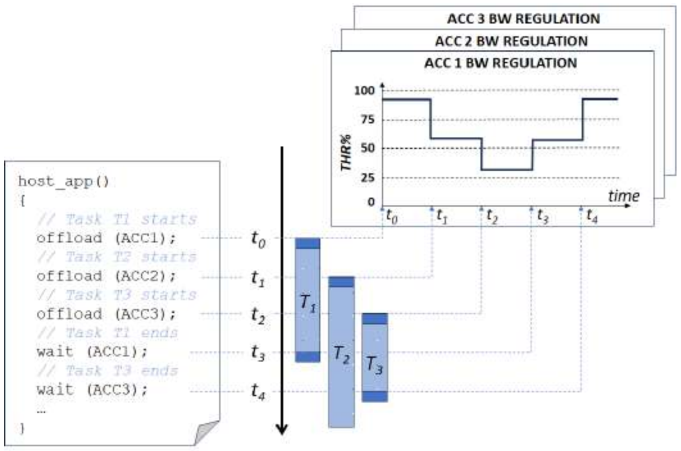


Fig. 1: Architectural template of the target HeSoC.



```
host_app()
{
  // Task T1 starts
  offload (ACC1);
  // Task T2 starts
  offload (ACC2);
  // Task T3 starts
  offload (ACC3);
  // Task T1 ends
  wait (ACC1);
  // Task T3 ends
  wait (ACC3);
  …
}
```

Fig. 2: Tasks scheduling and memory bandwidth regulation.

enabling numerous independent memory requests. If CPU cores and FPGA-based accelerators run in parallel without DRAM access control, the execution time of the CPU tasks can slow down by over $10\times$ [31]. On the other hand, enforcing mutually exclusive DRAM accesses by CPU and FPGA causes severe under-utilization of the available memory bandwidth.

Monitoring bandwidth usage from the FPGA-based accelerators on COTS HeSoCs can be done by querying in SW the *performance monitors* (PM). PMs are widely available in commercial platforms at various points in the system interconnect, and it is also possible to instantiate similar IPs in the FPGA logic. Intuitively, if said SW executes on the main *host* CPU there will be significant overhead involved for the *monitoring* phase, due to the very loose coupling between the *host* CPU and the FPGA-based accelerator. However, it is more and more common to enrich accelerator templates with a *soft core* for local control of the *datapath* and *DMA* operation, without the need for the costly intervention of the main CPU [46]–[49]. This tightens the coupling and reduces the overhead.

Throttling FPGA *accelerators* can be done by splitting long DMA bursts into several smaller chunks, each of which can be followed by a number of idle cycles (indicated as $idle_{cc}$), determined to reduce the used bandwidth to the percent value specified by the *Throttling Factor* ($\mathrm{THR}_{\%}$). The number of $idle_{cc}$ can be computed as a function of the cycles taken to complete the transfer of the chunk (referred to as copy cycles, indicated as $copy_{cc}$) and the $\mathrm{THR}_{\%}$, as shown in Eq. (1):

$$idle_{cc} = \frac{100 - THR_{\%}}{THR_{\%}} \cdot copy_{cc} \tag{1}$$

Note that $\mathrm{THR}_{\%} = 100$ means 100% bandwidth granted; $\mathrm{THR}_{\%} = 1$ means 1% bandwidth granted. Previous work has explored the use of the *soft cores* for programming the DMA in a duty-cycled loop according to Eq. (1) [21]. The main drawback of throttling *accelerators* via SW is the high programming overhead, which prevents its usage when fine-grained operation is needed. In the following we consider the fully SW-based monitoring + throttling approach as our reference example of loosely-coupled, coarse-grained bandwidth regulation scheme.

Some commercial HeSoCs include support for fine-grained bandwidth regulation at the level of individual master ports (e.g., QoS400 [13], QVN400 [25]). Although these solutions are not universally supported across vendors and SoCs, in the following we consider QoS400 as our reference, state-of-the-art example of fine-grained bandwidth regulation scheme.

Considering the dynamic operation of a real-time system, where tasks come and go continuously – and thus a different number of actors is simultaneously accessing DRAM at different time instants – we need to re-evaluate often the throttling factors to be applied to each accelerator to make sure that two requirements are fulfilled: (i) the timing guarantees are respected for every task; (ii) overall DRAM bandwidth usage (i.e., *residual* bandwidth) is maximized. This situation is illustrated in Fig. 2. Here the application that is running on the *Host CPU* spawns new SW and HW tasks over time. The figure shows, in particular, the offloading of computation on the FPGA-based accelerators – i.e., the creation of new HW tasks T_1, T_2, T_3 – at time instants t_0, t_1, t_2 and the termination of tasks T_1 and T_3 at time instants t_3 and t_4. Upon the admission of every new task in the system or the termination of an old task, some sort of *System Level Bandwidth Controller* (SLBC) could re-evaluate the $\mathrm{THR}_{\%}$ settings for every FPGA-based accelerator. The period of the SLBC policy is designed according to the requirements of the application domain: in autonomous systems applications new tasks can be admitted into the system with μs-scale frequency [15], [16]. With such a short time period in which $\mathrm{THR}_{\%}$ settings can change, it is fundamental to design the low-level *bandwidth regulation* (BR) mechanism to be as responsive and tight as possible, ensuring a fine timing resolution.

The BR timing resolution includes three components: (i) the time to transfer the chunk of data (referred to as monitoring process), which depends on the chunk size and the data size of the transfer and is typically provided by a monitoring system; (ii) the time to compute the $idle_{cc}$, which is influenced by the implementation of the BR control logic; and (iii) the time to perform the throttling, which varies according to the $\mathrm{THR}_{\%}$ requirements. Without loss

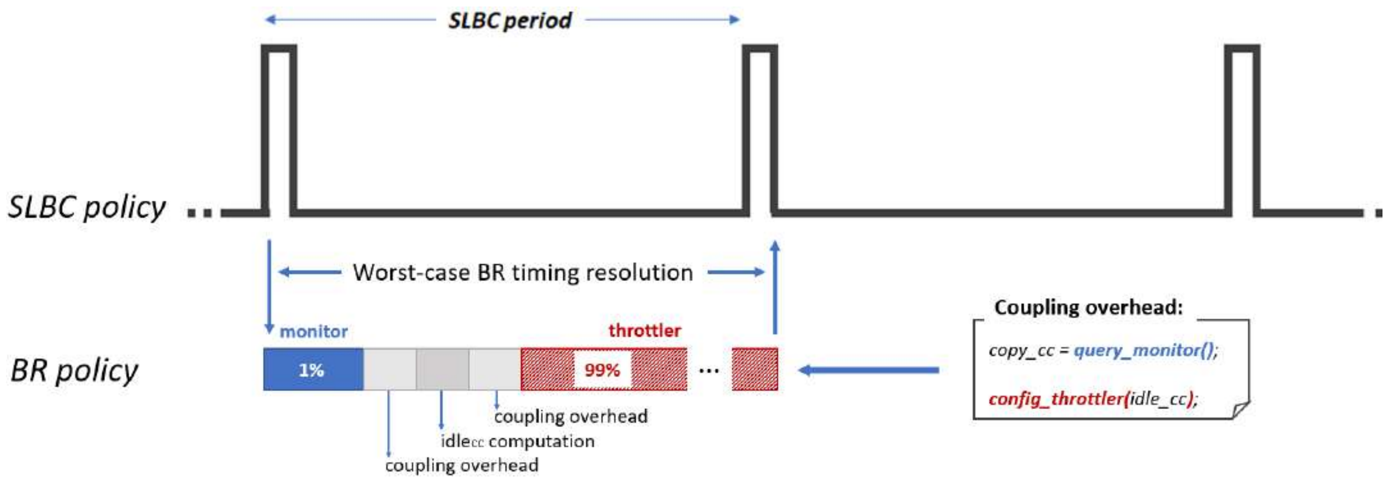


Fig. 3: Worst-case timing resolution ($\mathrm{THR}_{\%} = 1\%$). The worst-case timing resolution should not exceed the SLBC period to avoid missing new $\mathrm{THR}_{\%}$ settings.

of generality, in our scenario, we assume that the smallest bandwidth regulation value is $\mathrm{THR}_{\%}$ = 1%. In case of $\mathrm{THR}_{\%}$ = 1%, if the *monitoring* process takes $copy_{cc} = n$ clock cycles, by applying Eq. (1) the throttling would need to introduce $idle_{cc} = 99n$, giving a total timing resolution of $99n + n = 100n$. This represents the worst-case timing resolution in our scenario.

In addition to components (i) (ii), and (iii), the coupling between the *monitoring* and the *throttling* elements introduces some overhead, which further impacts the timing resolution. Specifically, this coupling overhead arises from the necessity for the BR to query the $copy_{cc}$ before the computation of $idle_{cc}$, and the need to configure the throttling mechanism using the calculated $idle_{cc}$. Intuitively, the tighter the coupling of the two elements, the shorter the time to complete the *monitoring* + *throttling* control loop, and thus the more amenable the timing resolution to the described autonomous systems scenarios. For example, if a particular SoC platform provides HW mechanisms for BW *monitoring* and *throttling*, these will individually be very fast, but their coupling is however controlled via SW, which is very loose in terms of responsiveness.

Fig. 3 illustrates the scenario of worst-case timing resolution. Allowing the timing resolution of BR to exceed the SLBC period would mean that the BR mechanism is not capable of adapting quickly enough to the application requirements, and thus it is not capable of providing the required timing guarantees. Section 5 provides a detailed analysis of the overhead implied by, and thus the speed of, different BR techniques.

4 Runtime Bandwidth Regulator

In the following, we present the proposed Runtime Bandwidth Regulator (RBR), which enables accurate BR of FPGA *accelerators* with minimal overheads.

4.1 Runtime Bandwidth Regulator Architecture

Since the main interface of an *accelerator* to the DRAM is the *DMA*, we suggest that bandwidth *monitoring* and *throttling* for its operation should happen at this level. The RBR is thus introduced as a non-intrusive component of the accelerator template, as shown in Fig. 4. It contains two main blocks: (i) a *monitor* that probes the outgoing channel to the DRAM to unobtrusively measure the time ($copy_{cc}$ from Eq. (1)) to transfer an amount of bytes configured via a parameter called the *threshold* (which defines the granularity of the technique); (ii) a *throttler* that computes the $idle_{cc}$ as a function of the $copy_{cc}$ and throttling factor $\mathrm{THR}_{\%}$, and stops DMA operations for that amount of time. It is worth noting that the RBR can work with bytes transferred through either read or write transactions. To avoid complicating the presentation, in the following, we consider just one type of transactions (the same considerations apply to the other). The value of the *threshold*, and the throttling factor $\mathrm{THR}_{\%}$ are provided during RBR configuration via the *soft core*[1]. This is expected to happen every time the underlying middleware (e.g., the RTOS or the hypervisor) modifies the QoS requirements for the tasks currently scheduled, for example because a new task has been just admitted.

The $copy_{cc}$ required to transfer *threshold* bytes can significantly vary based on the load of the system. While in absence of contention this transfer would usually take nom_{cc} nominal cycles to complete, under heavy contention the actual $copy_{cc}$ can grow significantly. Applying Eq. (1) in such scenario would further penalize the core/accelerator that has suffered this slowdown during memory transfer by further imposing an amount of $idle_{cc}$ that is computed based on this slowed-down transfer time. Any BR mechanism is thus typically capable of detecting contention-induced slowdown and to accordingly computing the appropriate amount of $idle_{cc}$. Our RBR mechanism does that as follows:

$$idle_{cc} = \max\{\delta \cdot nom_{cc} - (copy_{cc} - nom_{cc}), 0\} \quad (2)$$

1. Note that the configuration happens via memory-mapped registers, so the main or a secondary CPU could also configure *threshold* and $\mathrm{THR}_{\%}$, albeit with a higher latency compared to the *soft core*.

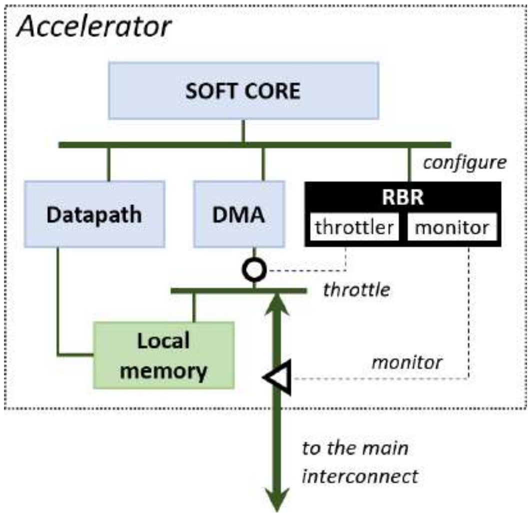


Fig. 4: The FPGA *accelerator* with the RBR.

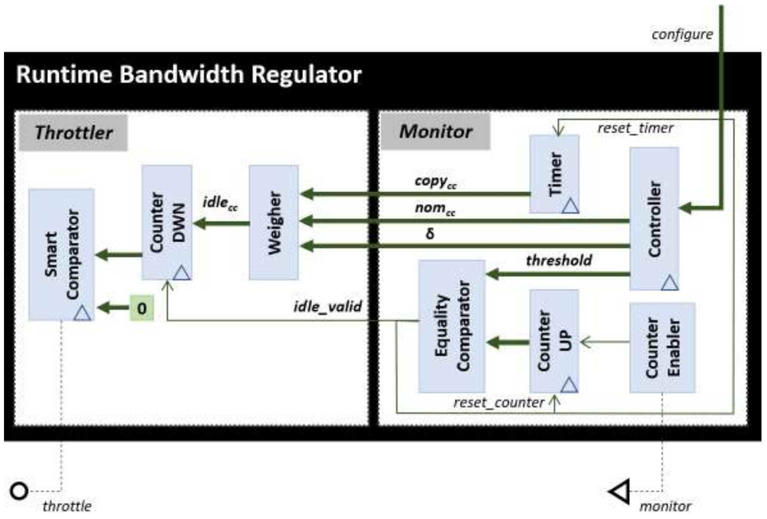


Fig. 5: RBR internal design.

where:

$$\delta = \frac{100 - THR_{\%}}{THR_{\%}} \tag{3}$$

The first term of Eq. (2) first computes the idle cycles in nominal conditions as $\delta \cdot nom_{cc}$. Then, those extra cycles that the transaction already took to complete because of the interference, are subtracted. The *max* function is introduced to prevent $idle_{cc}$ assuming negative values. With reference to Eq. (2), during the RBR configuration stage, THR$_{\%}$, *threshold*, and nom_{cc} are provided. Specifically, when the THR$\%$ value is supplied, the soft-core processor computes the corresponding δ value using Eq. (3). Instead of providing the raw THR$\%$ value to the RBR, the pre-computed δ value is sent directly. This prevents the RBR from performing floating point divisions, ensuring it does not impact the target frequency of the whole design. On the other hand, the nom_{cc} value (i.e., the number of clock cycles that a transfer of *threshold* size would take without contention) is recalculated by the soft processor whenever the *threshold* value changes, based on the specific characteristics of the throttled bus.

A close-up of the internals of the RBR is shown in Fig. 5. Upon RBR configuration, the *threshold*, nom_{cc}, and THR$_{\%}$ values are stored into a *Controller block* in the *monitor*. It is worth noting that all the sequential blocks work in the same clock domain and they are indicated with a triangle inside (highlighting the fact that they work at rising edge of the clock). The clock signal is not reported.

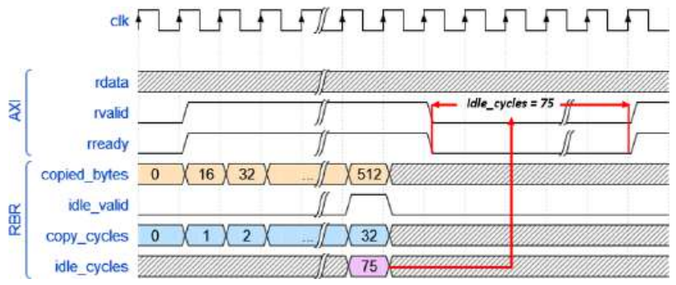
The *monitor* is developed by using the framework proposed in [50]; the *monitor* relies on a *Timer* block to measure the $copy_{cc}$. For every byte read through the outgoing channel to the DRAM, a *Counter Enabler* activates the increment of a *Counter Up*. An *Equality Comparator* detects when the *threshold* has been reached; when that happens, the *Equality Comparator* asserts an *idle_valid* signal, to trigger the operation of the *throttler*, and to reset the *Timer* and the *Counter Up* blocks. The *monitor* operates in a single clock cycle: upon observing the transmission of the last byte of the sequence (i.e., upon reaching the *threshold*), the *idle_valid* signal is asserted within the same clock cycle.

The *throttler* receives three inputs: $copy_{cc}$, nom_{cc}, and THR$_{\%}$ (expressed as δ of Eq. (3)), which are provided by the *monitor*. To compute the $idle_{cc}$, it relies on a *Weigher* block. The *Weigher* takes these inputs, where δ is represented in fixed-point format, and calculates Eq. (2) in a fully combinatorial manner, ensuring an efficient computation of the throttling cycle idle time based on the provided values. For every rising-edge of *idle_valid* signal, the Weigher output $idle_{cc}$ is written inside a down counter (*Counter DWN*) block, which also triggers the operation of a second equality comparator, referred to as *Smart Comparator* block. This block acts as a state machine that transitions between a *PASS-THROUGH* and a *WAIT* state. As long as the *Counter DWN* contains a number higher than zero, the *Equality Comparator* remains in the *WAIT* state. In this state, DRAM accesses from the accelerator are blocked. The *throttler* also operates in one clock cycle: when the Throttler receives the rising edge of *idle_valid* signal, it takes one clock cycle to stop the DRAM access from the accelerator.

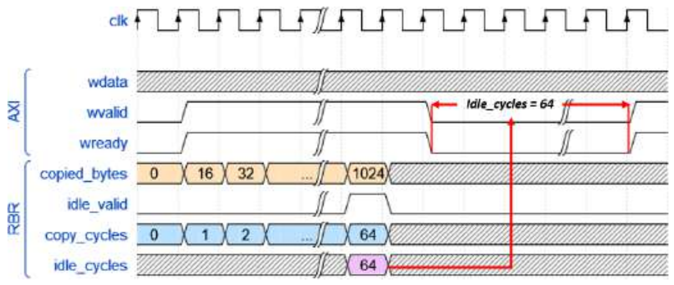
4.2 Runtime Bandwidth Regulator SoC Integration

The proposed RBR fully offloads *accelerator* control from the main CPU and is fully platform-independent as it was designed with a generic and representative reference architecture for an FPGA-based HeSoC. Porting the solution to a specific architecture only requires adapting the outbound monitoring and throttling signals to match the bus protocol.

For example, consider an accelerator with a DMA connected to the main memory through an AMBA AXI4 bus with a data bus size of 128-bit [43], operating at a frequency of 300 MHz. In this setup, the accelerator can achieve a maximum bandwidth of 4.8 GB/s for both read and write transactions. Suppose we want to regulate the bandwidth so that the accelerator reads at 30% and writes at 50% of the maximum bandwidth, corresponding to 1.44 GB/s for reading and 2.4 GB/s for writing. In this scenario, we connect two instances of RBR, one for each channel. For BR of the *writes*, the monitor block inside the RBR takes the signals *wvalid* (from the master), *wready* (from the slave), and *wstrobe* (from the master) to count the transmitted data,



(a) Bandwidth regulation of the AXI4 read channel using $threshold_{rd} \leftarrow 512$ Bytes and $\mathrm{THR}_{\%_{rd}} \leftarrow 30\%$. The resulting bandwidth is 1.44 GB/s.



(b) Bandwidth regulation of the AXI4 write channel using $threshold_{wr} \leftarrow 1024$ Bytes and $\mathrm{THR}_{\%_{wr}} \leftarrow 50\%$. The resulting bandwidth is 2.40 GB/s.

Fig. 6: Example scenario with a read and write memory bandwidth regulation using RBR.

while the throttler takes as input *wvalid* and *wready* and propagates them as output when in *PASS-THROUGH* state. When in the *WAIT* state, the throttler blocks both *wvalid* and *wready*, pausing the communication. For BR of the *reads*, the monitor block takes only *rvalid* (from the slave) and *rready* (from the master), while the throttler takes the pair *rvalid* and *rready*, propagating or blocking them in case of *PASS-THROUGH* or *WAIT* state, respectively.

To demonstrate the RBR functionality, we set different thresholds for *reads* and *writes*: $threshold_{rd} \leftarrow 512$ Bytes for *reads* and $threshold_{wr} \leftarrow 1024$ Bytes for *writes*. Considering the size of the data bus at 128-bit, this produces $nom_{cc,rd}$=32 and $nom_{cc,wr}$=64. The RBR *throttlers* are configured with $\mathrm{THR}_{\%_{rd}} \leftarrow 30\%$ and $\mathrm{THR}_{\%_{wr}} \leftarrow 50\%$ for *reads* and *writes*, respectively. The timing diagrams of the regulator activity are shown in Fig. 6a and Fig. 6b. Focusing on Fig. 6a, when the first byte of the DMA read transaction flows through the AXI4 bus, the monitor activates its internal timer. After the 512th byte is transmitted, the monitor forwards the $copy_{cc}$ to the throttler (during the rising edge of the *idle_valid* signal). The throttler then computes the $idle_{cc}$ and pauses the communication accordingly. Once the idle period ends, the throttler resumes communication. This monitor-throttle loop repeats every 512 bytes, resulting in 1200 iterations for a 600 kB transfer. The writing process in Fig. 6b works similarly. As shown in Fig. 6a and Fig. 6b, the obtained bandwidth meets the requirements.

The timing resolution of the BR depends on the *threshold* value, which can be configured as needed. However, the choice of the *threshold* has a direct impact on the accuracy of the $idle_{cc}$ computation and insertion performed by the RBR. Here, accuracy is defined as the deviation between the expected value of $idle_{cc}$ needed to achieve a specific bandwidth and the actual value computed and introduced by the RBR. This accuracy in turn influences the resolution of the BR step, which should be further evaluated.

4.3 Choosing the *threshold*

As described is Section 3 and shown in Fig. 3, the worst-case setting for the $\mathrm{THR}_{\%}$ parameter, which is 1%, lengthens the overall BR cycle duration (monitoring + throttling) to 100× the monitoring time. Thus, intuitively, the finer the granularity of the monitoring (i.e., the smaller the *threshold* parameter) the better. With our technique, the *threshold* can be as small as the size of the AXI4 bus, which is at maximum 1 *beat* = 16 Bytes on Zynq Ultrascale+ devices [44]. However, the downside of picking a very small *threshold* value is a loss of accuracy in the $idle_{cc}$ computation and insertion.

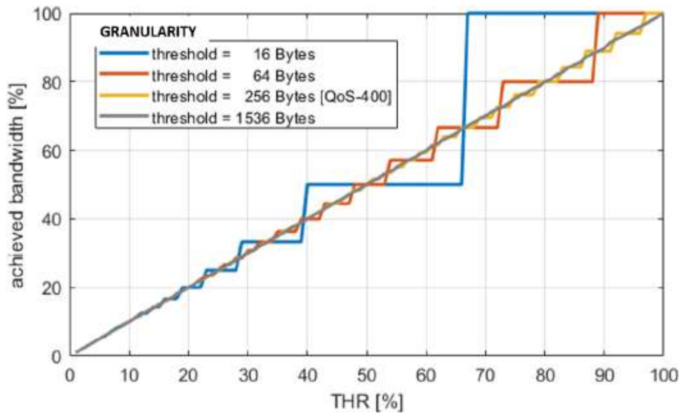


Fig. 7: Accuracy of the bandwidth regulation as a function of $\mathrm{THR}_{\%}$ for various granularities.

TABLE 1: RBR timing resolution and regulation step resolution for THR%=1% for various threshold values. The regulation step resolution is expressed with respect to the THR% value of the previous regulation cycle.

Threshold (bytes)	16	64	256	1536
Timing resolution [μs]	0.33	1.32	5.28	31.68
Regulation step resolution	50%	20%	5.88%	1%

Fig. 7 shows this effect by plotting how accurately different *threshold* settings (different curves) allow the BR in RBR (Y axis) to match the required $\mathrm{THR}_{\%}$ in the full [1%, 100%] range (X-axis). It is clear from the plot that choosing a small monitoring *threshold* negatively affects the accuracy of the BR for high $\mathrm{THR}_{\%}$ values. A *threshold* setting of 1 *beat* implies that the technique can just exploit an on/off decision with respect to the introduction of $idle_{cc}$: the next *beat* transfer can either be immediately issued or skipped for the next bus cycle. This results in either 100% or 50% BR, which affects the regulation step resolution, with no additional feasible settings in between, as illustrated by the blue curve. The grey curve is obtained with a *threshold* of 1536 Bytes, and it represents the best compromise between latency and BR accuracy since it is the smallest window that supports

TABLE 2: FPGA resource usage of the RBR. For comparison, the resource usage of a XILINX DMA IP is also provided.

	Component	FFs	LUTs	BRAM
DMA	AXI Datamover	2129	2246	16
	Interface	263	892	0
	TOT	2392	3138	16
RBR	Monitor	431	119	0
	Throttler	223	488	0
	TOT	654	607	0
	RBR area % wrt DMA	**27.34%**	**19.34%**	**0%**
	RBR area % wrt SoC	**0.119%**	**0.221%**	**0%**

1% $\text{THR}_{\%}$ variations in the high range. By comparison, the granularity of 256 Bytes, represented as yellow curve in the graph and adopted by the ARM QoS400 [13] on the target device, is insensitive to $\text{THR}_{\%}$ variations finer than 4-6% for $\text{THR}_{\%} \in [80\%, 100\%]$ (see Fig. 8 later on). Note that, if the application at hand does not require this regulation step resolution, our technique can be easily and quickly reconfigured for smaller *threshold* values. Table 1 reports the timing resolution of the RBR, for the four considered thresholds. The values refer to the worst-case scenario of $\text{THR}_{\%} = 1\%$.

5 Experimental Results

We implement our proposal on a Xilinx Zynq Ultrascale+, *XCZU9EG* HeSoC [44], an FPGA-based HeSoC integrating an ARM Cortex-A53 quad-core (referred to as APU), an ARM Cortex-R5 dual-core (referred to as RPU), and an FPGA. RPU cores are connected to a single DRAM controller port. APU cores are connected to two DRAM controller ports of 128-bit size. The FPGA can access the DRAM through four AXI4 [43] ports of 128-bit size, multiplexed to three DRAM controller ports of 128-bit size. The base accelerator template was modeled using Xilinx IPs for the DMA[2], soft-core, and interconnects. As we are only interested in measuring the worst-case interference effects, our accelerators: (i) are configured to work as traffic generators [31], i.e., they perform only memory accesses without computation[3]; (ii) perform all memory accesses with a sequential stride (i.e., addresses are sequential between separate burst requests, and memory accesses inside a burst are sequential), as this pattern generates much higher bandwidth compared to a random access pattern [31], and thus produces the highest contention for the memory controller.

We deploy three accelerators, each connected to a dedicated high performance port towards the main memory controller. Each of these paths has dedicated performance monitors and QoS-400 regulators. The accelerators are extended with our RBR, configured to work with a threshold of 1536 bytes and a $nom_{cc} = 96$ cc (since the bus between the DMA and memory transfers data at a rate of 16 Bytes per clock cycle without contention). The resulting design was synthesized with a target frequency of 300 MHz. Table 2 reports the raw area utilization of the proposed RBR in terms of FFs, LUTs, and BRAM. For reference we also report the area utilization for the simplest DMA engine that can be instantiated with Xilinx IPs (i.e., just an AXI Datamover[2] and the necessary interfaces). As shown in Table 2, compared to the DMA, the RBR uses the 27.34% of the FFs, the 19.34% of the LUTs, and the 0% of the BRAM. Overall, the RBR uses less than 1% of the FPGA resources available on the SoC.

2. https://docs.amd.com/r/en-US/pg022_axi_datamover
3. Note that this is without loss of generality, as well-designed accelerators overlap computation with memory transactions, using double-buffered transfers to avoid stalling DMA

Listing 1: BW regulation via SW-controlled DMA

```
void SW_BR_DMA (void *src, void *dst, int size)
{
  int NTRANS = size / threshold;
  for (int i=0; i < NTRANS; i++) {
    // DMA transfer and monitoring
    int offset = i * threshold;
    start_monitor();
    DMA_prog (src+offset, dst+offset, threshold);
    stop_monitor();
    int copy_cycles = read_monitor();
    // compute idle cycles and throttle
    int idleCycles = computeIdleCycles (THR,
        copy_cycles);
    Wait (idleCycles);
  }
}
```

Our experiments are aimed at comparing the proposed RBR mechanism to other BR approaches, both in terms of (i) speed and adaptation to dynamically varying QoS requirements (Subsection 5.1) and (ii) effective usage of the overall system bandwidth (Subsection 5.2). To this aim, we compare the following four approaches to BR (LCMT stands for Loosely-Coupled Monitoring and Throttling, TCMT stands for Tightly-Coupled and Throttling):

- **LCMT-SW-DMA**: Loosely-coupled regulation implemented by coupling AXI Perfomance Monitor (APM)[4] monitoring and explicit SW-based DMA throttling;
- **LCMT-RBR**: Loosely-coupled regulation implemented by coupling APM[4] monitoring and RBR throttling;
- **TCMT-RBR**: Our tightly-coupled regulation solution, entirely based on the proposed RBR;
- **TCMT-QoS-400**: Tightly-coupled regulation implemented by using the QoS400 regulator [13];

Concerning BR in SW, this can be achieved by relying on the APM for the *monitoring* phase and by explicitly duty cycling the DMA operation in SW [21] for the *throttling* phase. Listing 1 shows the pseudo-code to be executed on the *soft core* of each accelerator in place of regular DMA transfers. The original transfer of *size* bytes is split in `NTRANS` smaller transfers, each the size of the *threshold* parameter. Between one small transfer and the other the `Wait` function is invoked, which stalls the DMA for $idle_{cc}$ cycles, which is computed according to Eq. (2).

Concerning BR via the ARM CoreLink QoS-400 [13], on the Xilinx Zynq Ultrascale+ platform multiple regulators are available, enabling distinct regulations for various components. The QoS-400 performs transaction rate regulation based on a parameter referred to as ax_r (average rate), representing the average number of transactions allowed per

4. https://docs.amd.com/v/u/en-US/pg037_axi_perf_mon

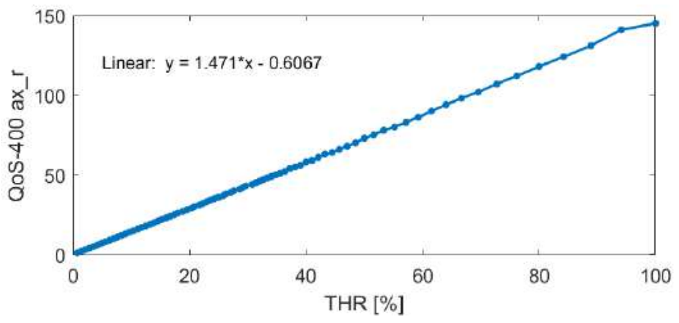


Fig. 8: Model of the QoS-400 correlating $\mathrm{THR}_{\%}$ values to *ax_r*. The equation is the linear interpolation.

clock cycle. Separate regulations are possible for *write* and *read* requests by writing into a memory-mapped register. We experimentally characterized the behavior of the QoS-400, and derived a model to easily correlate our $\mathrm{THR}_{\%}$ parameter to the corresponding ax_r setting, as shown in Fig. 8. Being the QoS-400 regulation granularity 256 Bytes (16 *beats* of 16 Bytes each), it is evident the loss of regulation precision as we move to high $\mathrm{THR}_{\%}$ values (as also shown in Fig. 7).

5.1 Bandwidth regulation mechanisms speed

This experiment is aimed at comparing how effectively the various BR approaches (monitoring+throttling) adapt to a trace of dynamically evolving QoS (i.e., $\mathrm{THR}_{\%}$) settings.

As discussed earlier, we envision a system where tasks start and complete execution dynamically, and a global SLBC (hypothetically a component of the operating system) is responsible for configuring the bandwidth regulators of the various accelerators to meet the evolving required $\mathrm{THR}_{\%}$ levels. The SLBC is modeled in our experimental setup using one of the available ARM Cortex R5 core on the platform, to avoid burdening the main CPU cores. The controller reads the $\mathrm{THR}_{\%}$ settings from the available On-Chip Memory (OCM). This operation requires $\approx 0.34\mu s$ and dictates the maximum speed of the SLBC.

Fig. 9 compares how the various approaches adapt to a QoS requirement trace that evolves with period $T = 32\mu$s. This is the nominal speed at which TCMT-RBR handles a worst-case 1% BR step request, using a monitoring window of 1536B. This magnitude matches the task admission frequency for control-oriented real-time applications [15], [16]. The X axis shows timestamps along a temporal line, while the Y axis shows the percentage of the maximum bandwidth that the accelerator under scrutiny is requesting. The black curve represents the trace of $\mathrm{THR}_{\%}$ setting requests, while the other curves show how the various BR approaches adapt to these requests over time.

As expected, both TCMT approaches precisely follow the $\mathrm{THR}_{\%}$ profile in every operating condition, since their nominal latency is less or equal than 32μs. As the coupling between *monitoring* and *throttling* phases loosens, it is impossible to adjust to a QoS trace evolving this fast. This is of course to be ascribed to the high overhead implied by the frequent DMA programming operations of the SW technique. Fig. 10 shows the performance penalty to split a single DMA transfer of 1536 KBytes in increasingly

TABLE 3: Speed of the different bandwidth regulation techniques (minimum timing resolution for $THR_{\%} = 1\%$). The TCMT-RBR is configured with a threshold of 1536 B, as this provides the highest precision. The numbers are referred to a 300 MHz implementation.

	TCMT-RBR	TCMT-QoS400	LCMT-RBR	LCMT-SW-DMA
Cycles	9600	2364	57600	1093500
T [μs]	32	7.88	192	3645

smaller and more numerous ones, in the absence of throttling ($idle_{cc}$=0). Transferring 1536 KBytes in `NTRANS`=1024 chunks of `threshold`=1536 Bytes each costs ten times a single transfer of 1536 KBytes, requiring around $3ms$. If the worst-case $\mathrm{THR}_{\%} \leftarrow 1\%$, the technique can process a new request every $(3ms/1024) * 100 = 300\mu$s.

Table 3 shows the minimum timing resolution, expressed as microseconds and clock cycles, for the various BR techniques, assuming the worst-case $\mathrm{THR}_{\%} = 1\%$. LCMT-RBR and LCMT-SW-DMA are respectively **6**×, and **114**× slower than TCMT-RBR. Our TCMT approach makes BR effective for applications with timing resolutions one to two orders of magnitude smaller than what is possible for LCMT approaches. If coarser regulation steps are sufficient for the application at hand, the RBR operates at a timing resolution of 0.33μs, namely **24**× faster than the TCMT-QoS400.

5.2 Co-scheduling

In a context where various processing engines coexist within the system, the efficient utilization of memory bandwidth becomes crucial. While tightly-coupled monitoring and throttling primarily serves the purpose of effectively controlling the QoS requirement of one or more tasks/cores in the system, it is also extremely important that the technique allows maximal exploitation of residual bandwidth. The following sections aim to compare the residual bandwidth utilization for the various BR techniques.

5.2.1 Effective bandwidth exploitation

The objective of the experiment presented in this Section is that of measuring the overall memory bandwidth usage of the system, while guaranteeing a certain QoS requirement for *critical* tasks. To that end, we assume that an ARM Cortex-A53 core executes the *critical* task on top of a Petalinux kernel[5], and we express its QoS requirement in terms of maximum tolerated slowdown. Following a widely adopted convention in literature [12], [16], [39], we consider two QoS thresholds: 10% and 20% maximum slowdown. We consider that the three FPGA-based accelerators execute best-effort tasks, and thus we regulate their bandwidth usage to satisfy the QoS requirement of the *critical* task. To model and evaluate the system dynamism in terms of workload variations over time, we rely on a pre-computed trace that instructs our SLBC – running on an ARM Cortex R5 core – on which FPGA-based accelerators to start/stop every 32μs. For each of the two QoS requirements, we provide a plot where we show the use of the residual system bandwidth by the FPGA-based accelerators, under the guarantee that the various regulation techniques deliver the required QoS

5. https://docs.amd.com/v/u/en-US/dh0016-petalinux-tools-hub

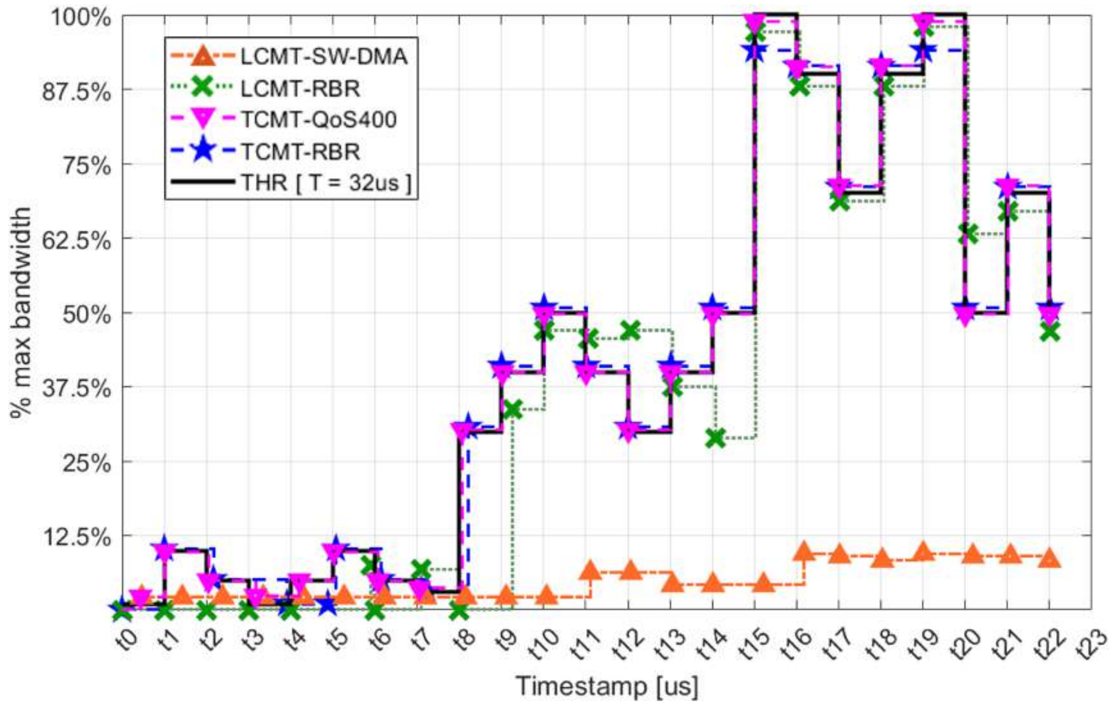


Fig. 9: Comparison of LCMT- and TCMT- regulation in adapting to a trace of THR% settings. The trace evolves dynamically with a period of T = 32μs.

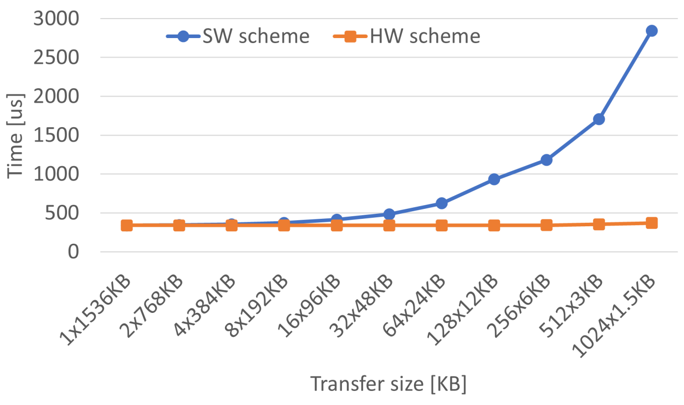


Fig. 10: SW versus HW *throttling* cost when splitting a DMA transfer of $1536kB$ in smaller transfers.

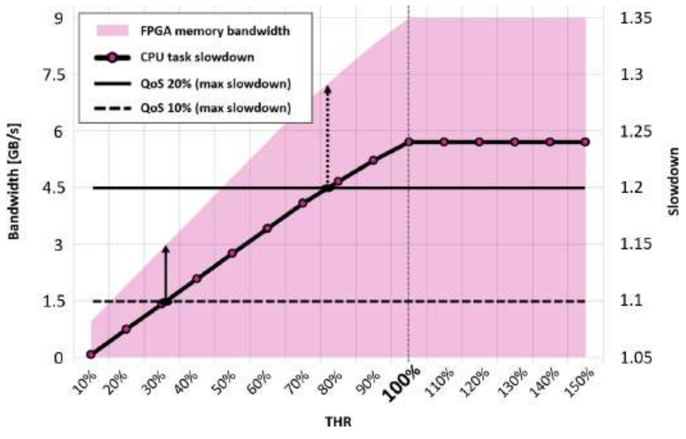
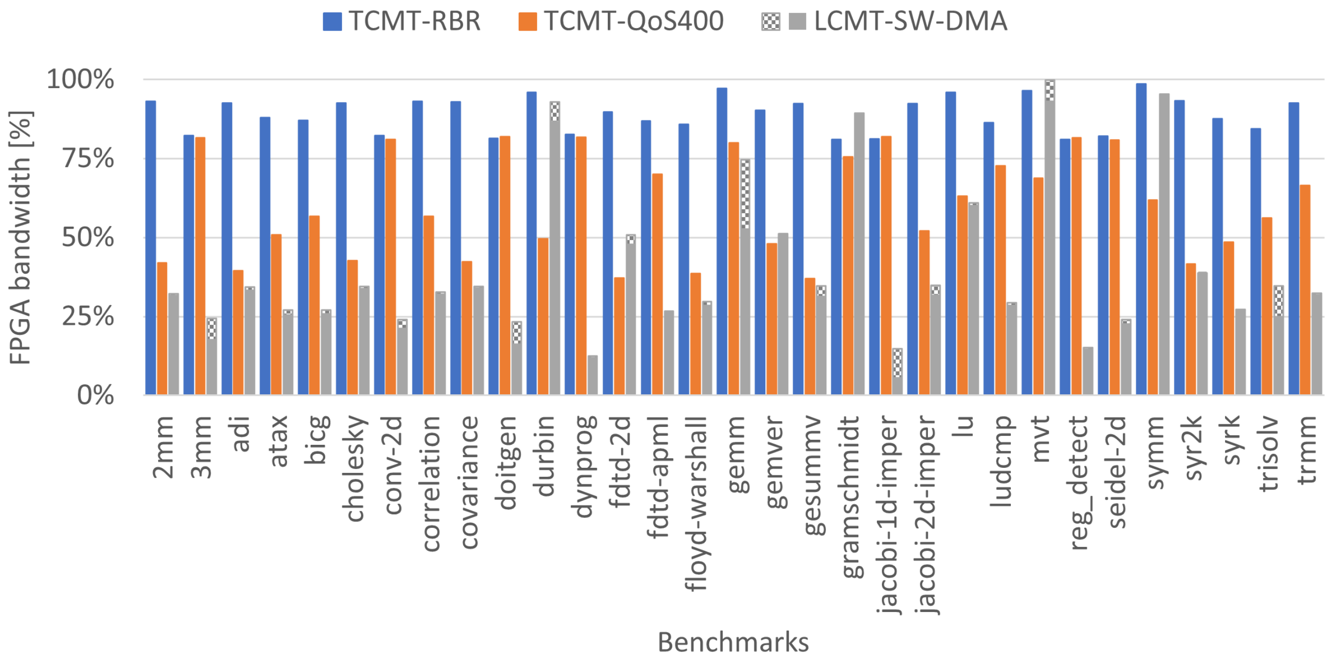


Fig. 11: Experimental setup. For each benchmark we identify the highest THR$_{\%}$ value that doesn't exceed the QoS requirement (10% or 20% max slowdown). The resulting FPGA bandwidth is plotted in Fig. 12.

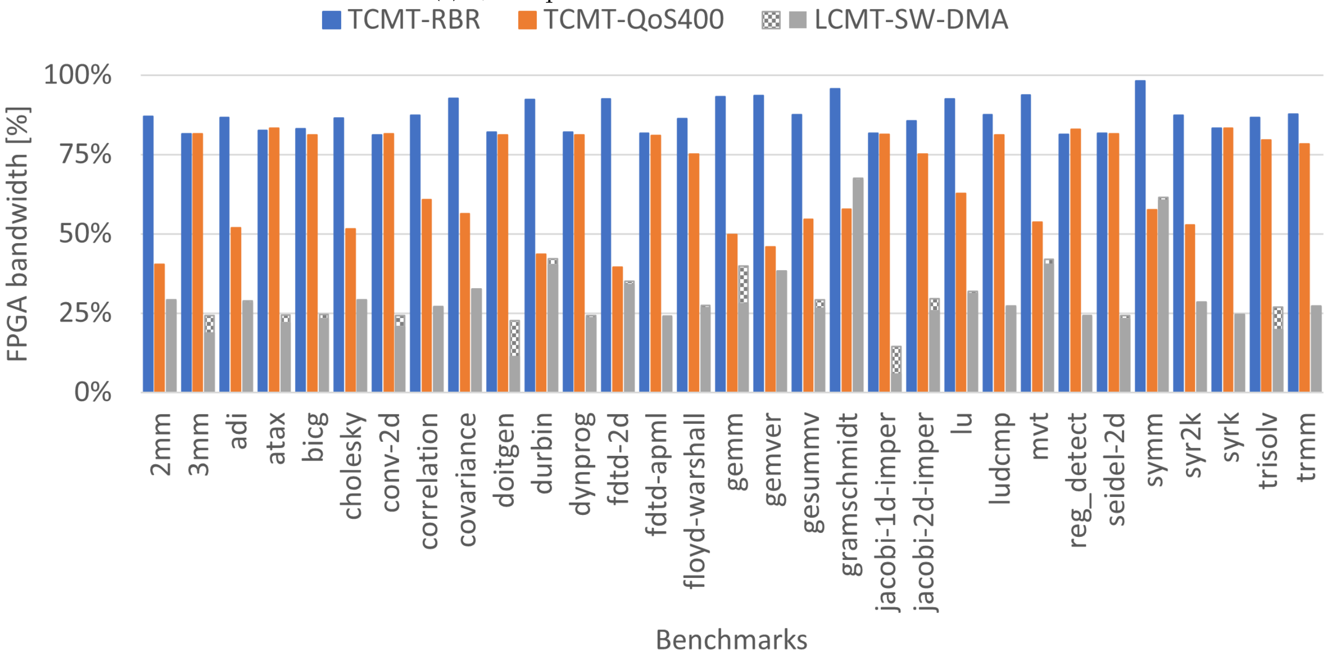
constraint. The results are presented as a set of bar plots for each of the 31 benchmarks from the Polybench suite [51], acting as a *critical* task on the ARM Cortex-A53 core.

Fig. 11 illustrates the experimental setup used to derive the bar plots. The x-axis represents the overall THR$_{\%}$ parameter of the FPGA-based accelerators. Assuming three accelerator templates on the FPGA, a THR$_{\%}$ $\leftarrow$ 100% implies that the i-th accelerator template is configured with one-third of the total THR$_{\%}$ (i.e., $ACT_i = 33.3\%$ THR$_{\%}$). A THR$_{\%}$ $\leftarrow 100\%$ corresponds to utilizing the entire memory bandwidth of the FPGA, as denoted by a vertical dotted line. The black plot with circle markers represents the CPU task execution time slowdown (to be read on the right Y axis), while the red area represents the cumulative bandwidth used by the FPGA-based accelerators (to be read on the left Y axis). The two black horizontal lines represent the two QoS thresholds at 10% and 20% maximum slowdown, respectively. The points where the slowdown curve intersect the horizontal 10% and 20% QoS threshold curves are projected upward to meet the edge of the red area. The bandwidth value identified by these points is used in the bar plots. A plot equivalent to Fig. 11 is derived for every Polybench benchmark, but for a more compact and readable information we only show the bar plots derived as described. The SLBC behavior is based on the maximum THR$_{\%}$ that satisfies the QoS requirement for every benchmark, BR technique and number of active accelerators, based on offline profiling.

The bar plots for the 10% and 20% QoS requirements are shown in Fig. 12a and 12b, respectively. The blue bars rep-



(a) QoS requirement = max 10% slowdown.



(b) QoS requirement = max 20% slowdown.

Fig. 12: Exploitation of the residual bandwidth from the FPGA-based accelerators. CPU tasks are PolyBench benchmarks.

resents the TCMT-RBR, the orange bars the TCMT-QoS400 and the grey bars the LCMT-SW-DMA. For the LCMT-SW-DMA we show two stacked bars, where the solid part is referred to the case where the SLBC trace varies using a slower period of 3645 μs to ensure that the QoS guarantee is satisfied. The hatched part of the bar shows an additional portion of the bandwidth that the SLBC erroneously allows the FPGA-based accelerators to exploit when using the control period of 32 μs. As previously explained, since LCMT-SW-DMA is not fast enough to precisely follow a bandwidth trace that evolves using a 32 μs period, the controller is not capable of satisfying the QoS requirements (see Table 3). Overall, it is evident that the proposed TCMT-RBR surpasses TCMT-QoS400 in terms of residual bandwidth exploration. The advantages are even more pronounced compared to LCMT-SW-DMA, but this was to be expected.

Fig. 13 provides the average memory bandwidth utilization of the 31 benchmark kernels. Here the measured memory bandwidth is reported as a percentage of the ideal residual bandwidth, obtained by applying Eq. (1), given the CPU task, the number of active accelerators, and the actual $THR_{\%}$. From this figure we can derive that the proposed TCMT-RBR approach allows to use additional **28.7**% and **48.2**% memory bandwidth compared to TCMT-QoS400 and

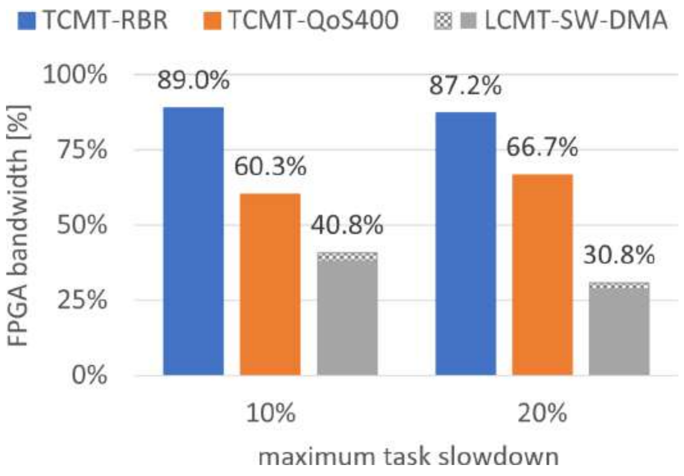


Fig. 13: Exploitation of the residual bandwidth from the FPGA-based accelerators. Average.

TABLE 4: Real-world benchmarks as described by [12].

Scenario		FPGA		APU		RPU	
		ACT1	ACT2	MM	MT	VMA	I2C
VT	(Max 20%)	1.20×	1.20×	1.22×	1.07×	1.70×	1.11×
T	**(Max 40%)**	1.38×	1.38×	1.22×	1.07×	1.35×	1.04×
M	**(Max 60%)**	1.59×	1.59×	1.22×	1.07×	1.31×	1.04×

LCMT-SW-DMA respectively, for the 10% QoS requirement. An increase of about **20.5**% and **56.4**% of memory bandwidth utilization, respect to the other two mechanisms, can be achieved by relaxing the constraint to a 20% of maximum slowdown.

5.2.2 *Unlocking more effective co-scheduling opportunities*

Previous work has explored the use of QoS control in modern HeSoCs to understand how this impacts the performance of co-scheduled SW and HW tasks, targeting the *XCZU9EG* SoC [12]. Here, three FPGA-based accelerators (Xilinx traffic generators) are considered, each attached to a different DRAM controller port. Two *host* cores from the APU, attached to another two DRAM controller ports, execute a matrix multiplication (MM) and a matrix transpose (MT) benchmark, respectively. Two *host* cores from the RPU, sharing a multiplexed channel to the last DRAM controller port, execute a vector add (VMA) and an image to column (I2C) benchmark. Given this co-scheduled workload, five high-level *QoS settings* are considered. In each *setting*, a different X% maximum performance degradation (slowdown) is tolerated: (i) *Very-Tight (VT)*, where X=20%; (ii) *Tight (T)*, where X=40%; (iii) *Moderate (M)*, where X=60%; (iv) *Loose (L)*, where X=80%; (v) *Very-Loose (VL)*, where X=99%. Various hardware QoS knobs available inside the DDR memory controller of the target [44] are then used to try and satisfy the QoS requirements (the most relevant to our discussion of which is the QoS-400). The key finding is that **no available QoS knob could satisfy the M, T, and VT *QoS settings***.

To conduct a direct comparison, we instantiate the exact same setup, with three RBR-enabled accelerators (traffic generators) executing in parallel with APU and RPU cores (executing the same benchmarks described above). Table 4 shows the slowdown, normalized respect the execution time in absence of interference (e.g. 1.2× means an increase in execution time of a 20%). The slowdown measurements are referred to the involved processing units for the VT, T, and M *QoS scenarios*, i.e., the ones for which the hardware QoS knobs could not satisfy the requirement [12]. The accelerators are labeled ACT1 and ACT2 in the experiments. The third accelerator, ACT3, is not included in Table 4, because we aim to directly compare our results with those in [12], where the authors do not consider it in their final results.

To conduct the experiments we exploited our fine-grained RBR to precisely regulate the slowdown of the ACT1 and ACT2 on the maximum tolerated by each scenario (e.g. for the *VT* scenario we imposed a 1.2× for both accelerator templates). This will leave more room to exploit memory bandwidth to the other tasks, without compromising the performance requirements of the accelerator templates. **The results show that RBR can satisfy the requirements for all the actors in QoS scenarios M and T** (the cells shaded in green), **and for most actors also in QoS scenario VT**. This further confirms that tightly-coupled BR enables system-wide scheduling opportunities that are not feasible with state-of-the-art mechanisms.

5.3 Scalability of the Proposed Approach

The experimental results demonstrate the effectiveness of the proposed regulation mechanism in terms of timing resolution, bandwidth redistribution, and flexibility within real-world scenarios.

Evaluating various workloads on the APU confirmed the scalability of the system BR across different types of applications. Additionally, testing different configurations on the Zynq Ultrascale+ – involving one APU with multiple FPGA-based accelerators and multiple configurations involving APU, RPU, and FPGA-based accelerators – highlighted the scalability across different platform setups. This versatility suggests promising performance in systems with additional computing elements like GPUs and DMA-based I/O peripherals, although these setups were not tested in the current experiments.

Adapting the solution to different platforms requires some modifications, specifically aligning the outbound monitoring and throttling signals with the bus protocols of the new architecture. At this stage, there is no precise model for determining the appropriate $THR_{\%}$ value to achieve a specific QoS, which represents an area for future improvement.

6 Conclusion

We introduced a tightly-coupled bandwidth monitoring and throttling solution for FPGA-based HeSoCs. This solution is based on an original IP, the *Runtime Bandwidth Regulator*, that can unobtrusively be integrated in generic FPGA-based accelerator designs. The flexible programmability of the main RBR configuration parameters allows to change the regulation factor and its granularity at run time. This approach makes BR effective for applications with timing resolution one to two orders of magnitude smaller than what is possible for state-of-the-art, SW-controlled solutions.

Compared to fully-HW regulation solutions like ARM CoreLink QoS400, we achieve comparable regulation speed, with a finer throttling resolution and 28.7% better exploitation of the *residual* memory bandwidth. Moreover, changing the granularity at runtime allows for a trade-off between timing resolution and regulation step resolution, enabling the RBR to operate at a timing resolution of up to 0.33 μs as needed. It is also worth noting that HW solutions like QoS-400 are still not widely available across FPGA vendors and products. When evaluated at the whole-system level for the co-scheduling of SW and HW tasks, the RBR enables effective BR in presence of much tighter QoS requirements compared to previous work.

Future activities include integrating the proposed approach with similar on-off control mechanisms for CPU cores (such as [16]) to achieve a more comprehensive system-wide BR that effectively manages both FPGA-based accelerators and CPU cores. Additionally, we aim to develop a quantitative model linking monitoring granularity with accuracy loss, alongside a more structured framework for meeting QoS requirements through $\text{THR}_{\%}$ values. Finally, we plan to validate the approach across diverse platforms as the AMD/Xilinx Versal [1], and within application scenarios in the aerospace domain where multiple workloads share memory resources.

Acknowledgments

This work was supported by project AI4CSM, which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 101007326. This work was supported by FRACTAL EU project, under ECSEL JU and grant agreement No 877056. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Austria, Belgium, Czech Republic, Italy, Netherlands, Lithuania, Latvia, Norway. The work was also supported by the European Union under the NextGenerationEU Programme within the Plan "PNRR - Missione 4 "Istruzione e Ricerca" - Componente C2 Investimento 1.1 "Fondo per il Programma Nazionale di Ricerca e Progetti di Rilevante Interesse Nazionale (PRIN)" by the Italian Ministry of University and Research (MUR)". Project Title: "Simplifying Predictable and energy-efficient Acceleration from Cloud to Edge (SPACE)", Project code: E53D23007800006. MUR D.D. financing decree n. 959, 30th June 2023. The work was also supported by the European Union under the NextGenerationEU Programme under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY - CUP E13C22001060006.

References

[1] Xilinx, "VERSAL ACAP," https://www.xilinx.com/products/silicon-devices/acap/versal.html, 2022, accessed: 05-June-2023.

[2] NVIDIA, "NVIDIA Jetson AGX Orin Industrial module," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2023, accessed: 05-June-2023.

[3] Qualcomm, " Heterogeneous Computing for your Demanding Apps ," https://developer.qualcomm.com/blog/heterogeneous-computing-your-demanding-apps, 2020, accessed: 05-June-2023.

[4] J. Zhao, H. Cui, J. Xue, and X. Feng, "Predicting cross-core performance interference on multicore processors with regression analysis," *IEEE Transactions on Parallel & Distributed Systems*, vol. 27, no. 05, pp. 1443–1456, may 2016.

[5] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "Bliss: Balancing performance, fairness and complexity in memory access scheduling," *IEEE Transactions on Parallel & Distributed Systems*, vol. 27, no. 10, pp. 3071–3087, oct 2016.

[6] CAST, *Position Paper CAST-32A Multi-core Processors*, 2016, Accessed: November 21st, 2021. [Online]. Available: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf

[7] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, " Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Comput. Surv.*, vol. 48, no. 2, nov 2015. [Online]. Available: https://doi.org/10.1145/2830555

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

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

[10] M. Zini, G. Cicero, D. Casini, and A. Biondi, "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms," *Software: Practice and Experience*, 11 2021.

[11] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A Closer Look at Intel Resource Director Technology (RDT)," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 127–139. [Online]. Available: https://doi.org/10.1145/3534879.3534882

[12] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021, pp. 3:1–3:26.

[13] ARM, "ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service," https://developer.arm.com/documentation/dsu0026/latest/, 2016, accessed: 13-April-2023.

[14] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 133–145.

[15] Giulio Corradi, "Tools, Architectures and Trends on Industrial all Programmable Heterogeneous MPSoC," URL: http://archives.ecrts.org/fileadmin/files_ecrts17/Giulio_Corradi_Presentation.pdf, 6 2017.

[16] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mempol: polling-based microsecond-scale per-core memory bandwidth regulation," *Real-Time Systems*, 2024.

[17] Accelerat, *Clare Software-Stack*, 2024. [Online]. Available: https://accelerat.eu/clare

[18] Minervasys, *The Minerva Tool*, 2024. [Online]. Available: https://www.minervasys.tech/downloads

[19] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2016.

[20] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A Real-Time virtio-based Framework for Predictable Inter-VM Communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 27–40.

[21] G. Brilli, A. Capotondi, P. Burgio, and A. Marongiu, "Understanding and Mitigating Memory Interference in FPGA-based HeSoCs," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022, pp. 1335–1340.

[22] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[23] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "AXI HyperConnect: A Predictable, Hypervisor-level Interconnect

for Hardware Accelerators in FPGA SoC," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[24] Z. Jiang, K. Yang, N. Fisher, I. Gray, N. C. Audsley, and Z. Dong, "AXI-IC$^{\mathrm{RT}}$ RT : Towards a Real-Time AXI-Interconnect for Highly Integrated SoCs," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 786–799, 2023.

[25] ARM, "CoreLink QVN-400 Network Interconnect Advanced Quality of Service using Virtual Networks," https://developer.arm.com/documentation/dsu0027/latest/, 2016, accessed: 13-April-2023.

[26] A. Pellegrini, "Arm Neoverse N2: Arm's 2 nd generation high performance infrastructure CPUs and system IPs," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–27.

[27] G. Brilli, G. Valente, A. Capotondi, P. Burgio, T. Di Masciov, P. Valente, and A. Marongiu, "Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[28] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.

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

[30] S. Yamagiwa and K. Wada, "Performance study of interference on GPU and CPU resources with multiple applications," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–8.

[31] K. Manev, A. Vaishnav, and D. Koch, "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 179–187.

[32] G. Valente, T. D. Mascio, L. Pomante, and G. D'Andrea, "Dynamic partial reconfiguration profitability for real-time systems," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 102–105, 2021.

[33] G. Valente, V. Muttillo, F. Federici, L. Pomante, and T. Di Mascio, "Analysis of reconfiguration delay in heterogeneous systems-on-chip via traffic injection," *IEEE Embedded Systems Letters*, vol. 16, no. 2, pp. 162–165, 2024.

[34] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[35] J. Martinez, I. Sañudo, and M. Bertogna, "Analytical Characterization of End-to-End Communication Delays With Logical Execution Time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2244–2254, 2018.

[36] B. Forsberg, L. Benini, and A. Marongiu, "HePREM: A Predictable Execution Model for GPU-based Heterogeneous SoCs," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 17–29, 2021.

[37] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global Real-Time Memory-Centric Scheduling for Multicore Systems," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2739–2751, 2016.

[38] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 109–118.

[39] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms," *Real-Time Systems*, pp. 1–40, 04 2022.

[40] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, "Dynamic memory bandwidth allocation for real-time gpu-based soc platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3348–3360, 2020.

[41] R. Cavicchioli, N. Capodieci, M. Solieri, M. Bertogna, P. Valente, and A. Marongiu, "Evaluating Controlled Memory Request Injection to Counter PREM Memory Underutilization," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2020, pp. 85–105.

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

[43] ARM, *AMBA 4 AXI and ACE Protocol Specification*, 2013, Accessed: May 5,2022. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e

[44] Xilinx, *Zynq UltraScale+ Technical Reference Manual*, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual

[45] Intel, *Agilex 7 Technical Reference Manuel*, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683458/current/overview-of-the-fpgas-and-socs.html

[46] H. Omidian, N. Ivanov, and G. G. Lemieux, "An Accelerated OpenVX Overlay for Pure Software Programmers," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 290–293.

[47] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, "Agile SoC Development with Open ESP : Invited Paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[48] X. Ling, T. Notsu, and J. Anderson, "An Open-Source Framework for the Generation of RISC-V Processor + CGRA Accelerator Systems," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 35–42.

[49] G. Bellocchi, A. Capotondi, F. Conti, and A. Marongiu, "A RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 9–17.

[50] G. Valente, T. Fanni, C. Sau, T. D. Mascio, L. Pomante, and F. Palumbo, "A Composable Monitoring System for Heterogeneous Embedded Platforms," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, jul 2021. [Online]. Available: https://doi.org/10.1145/3461647

[51] U. of California Los Angeles, "PolyBench/C the Polyhedral Benchmark suite," http://web.cs.ucla.edu/~pouchet/software/polybench/, 2012, accessed: 03-April-2023.

Giacomo Valente received the MS. Degree in Electronic Engineering in 2014 and the Ph.D. degree in Information and Communication Technology in 2018 from University of L'Aquila. His primary research activities are in electronic design automation, reconfigurable computer architectures, and real-time systems. Since 2022, he has been an Assistant Professor in Computer Architecture at the Department of Information Engineering, Computer Science, and Mathematics of the University of L'Aquila. He is the author or co-author of more than 30 research articles in peer-reviewed journals and international conference proceedings. He has been also a reviewer and member of several TPCs related to his research topics.

Gianluca Brilli is a postdoctoral researcher in computer engineering at the University of Modena and Reggio Emilia, within the High-Performance Real-Time Laboratory (HiPeRT-Lab) located in Modena, Italy. His expertise lies in the field of software and hardware acceleration using reconfigurable embedded systems. His main research interests are main memory QoS regulation and memory interference mitigation on FPGA-based heterogeneous systems.

Tania Di Mascio is an Associate Professor in the Department of Information Engineering, Computer Science, and Mathematics (DISIM) at the University of L'Aquila. She teaches courses on Methods and Methodologies for ICT and Advanced Database Systems. At the University of L'Aquila, she is responsible of the Interaction and Computational Systems Laboratory and oversees the L'Aquila node of CINI Assisting Technology. In addition to her academic roles, she serves as an Innovation Manager for MIMIT and is a co-founder and CXO of Project Innovation srl. Prof. Di Mascio has participated in numerous EU projects, assuming significant scientific management roles. Currently, she coordinates the Assistive Technology and Embedded System research group (ATES@AQ). Her primary research interests include human-computer interaction, assistive technologies and embedded systems, and technology-enhanced learning. She has authored or co-authored over 150 research articles in peer-reviewed journals and conference proceedings. Prof. Di Mascio is a member of several steering committees for international and national conferences. She acts as a referee for numerous international journals and frequently serves on organizing and program committees for conferences and workshops. Additionally, she is an active member of SIGCHI, the Center of Excellence of DEWS, and the HiPEAC (High Performance, Edge, and Cloud Computing) European Organization.

Alessandro Capotondi (Member, IEEE) is Assistant Professor at the University of Modena and Reggio Emilia. He received a PhD in Electronics, Telecommunications and Information Technology at the University of Bologna in 2016. He has been a research assistant and postdoctoral researcher at the University of Bologna and the University of Modena and Reggio Emilia. His research interests focus mainly on embedded systems and heterogeneous computing devices, architectures for programmable and reconfigurable logic (FPGA), and HW-SW co-design of embedded systems. In these areas, he has published more than 30 papers in international peer-reviewed conferences and journals, with more than 900 citations and an h-index of 15 [Google Scholar].

Paolo Burgio got a Ph.D in Electronics Engineering jointly between the University of Bologna and the University of Southern-Brittany, in 2013. His research topics are next-generation predictable systems based on heterogeneous many-cores and GP-GPUs, with an eye on compilers, and parallel programming models. Since 2014 he joined HiPeRT Lab at Univ. of Modena, where he currently coordinates the activities on autonomous vehicles and drones, and smart cities. He is co-founder of the HiPeRT srl startup.

Paolo Valente (male) is Assistant Professor at the Department of Computer Science of the University of Modena and Reggio Emilia, Italy. He previously was a Research Collaborator at the University of Pisa. His research activity is mainly focused on the design and analysis of real-time and proportional-share scheduling algorithms for CPU, disk and network. Notable contributions include the QFQ packet scheduler, providing quasi-optimal guarantees at O(1) cost, and the BFQ proportional-share I/O scheduler. Both are now part of the Linux kernel. In addition, he has contributed new mathematical results, and new paradigms, for guaranteeing both real-time constraints and a high utilization in multiprocessor systems. He was and is involved in national and European research projects.

Andrea Marongiu received the PhD degree in Computer and Electronic Engineering from the University of Bologna, Italy, in 2010. He has been a postdoctoral reserch fellow at ETH Zurich, Switzerland. He currently is an associate professor at the University of Modena and Reggio Emilia. His research interests focus on programming models and architectures in the domain of heterogeneous multi- and many-core systems-onchip. In this field, he has published more than 120 papers in peer-reviewed conferences and journals.