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

G. Brilli*, G. Valente†, A. Capotondi*, P. Burgio*, T. Di Mascio†, P. Valente*, A. Marongiu*
** University of Modena and Reggio Emilia, Italy, e-mail: name.surname@unimore.it*
† DISIM Department, University of L'Aquila, Italy, e-mail: name.surname@univaq.it

***Abstract*—Embedded systems are increasingly adopting heterogeneous templates integrating hardware accelerators and application-specific processors, which poses novel challenges. In particular, it is difficult to have accurate control of task activities in Commercial Off-the-shelf (COTS) System on Chips (SoCs), due to complex main memory sharing mechanisms among different computing engines. To address this problem, bandwidth regulation approaches based on monitoring and throttling are widely adopted. Existing solutions, however, are either too coarse-grained, limiting the control over computing engines activities, or platform-dependent, addressing the problem only for specific SoCs. In this paper we propose an innovative, fine-grained and platform-independent approach that can accurately control main memory bandwidth usage in an FPGA-based Heterogeneous System on Chip (HeSoC). Experimental results conducted on the Xilinx Zynq UltraScale+ platform demonstrate that our approach enables solutions not feasible with state-of-the-art bandwidth regulation methods.**

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

I. INTRODUCTION

The current generation of embedded systems widely relies on Heterogeneous Systems-on-Chip (HeSoCs), where general purpose multi-cores are coupled to hardware accelerators and application-specific processors. The adoption of such systems provides sufficient computing power to satisfy the needs of modern software applications, but at the same time it poses novel challenges. In particular, as the number of on-chip compute engines grows, the interference due to main interconnect and memory sharing significantly hampers the tasks' execution time [1], making the system unpredictable from the point of view of timing. Several solutions have been proposed to tackle this problem, ranging from static memory partitioning techniques [2] [3], to task execution models that guarantee predictable memory access [4] and memory bandwidth regulation strategies (e.g., [5]–[7]). The latter in particular are widely available also in commercial products, based on the combination of bandwidth *monitoring* and *throttling* mechanisms. When considering heterogeneous systems based on programmable logic (FPGA), the availability of such mechanisms is typically limited to loosely coupled, coarse-grained components from the point of view of the actuation interval (i.e., the time to monitor and throttle the bandwidth). When co-scheduling numerous HW and SW tasks from real-time operating systems featuring scheduling ticks below the millisecond boundary, fine-grained and tightly-coupled schemes to support bandwidth regulation is mandatory to guarantee Quality of Service (QoS) control. While some fine-grained techniques exist, they are highly platform-dependent, addressing the problem only for specific SoCs [8] [9].

In this paper we propose *an innovative, fine-grained and platform-independent Runtime Bandwidth Regulator (RBR) for disciplined main memory bandwidth usage in Commercial Off-the-Shelf (COTS), FPGA-based HeSoCs*. The RBR performs tightly-coupled monitoring and throttling of main memory bandwidth, effectively delivering the desired QoS levels with high precision. The proposed RBR quickly adapts to dynamically varying QoS levels, and is completely platform independent. Moreover, our system does not interfere with existing running tasks, and introduces a minimal timing overhead.

Our experimental results show that the proposed tightly-coupled bandwidth regulation scheme is able to precisely track and very quickly adapt to dynamically changing QoS requests with a resolution of just 32μs. This approach makes bandwidth regulation effective for applications with timing resolution one to two orders of magnitude smaller than what is possible for state-of-the-art solutions. When evaluated at the whole-system level for the co-scheduling of SW and HW tasks, RBR enables effective bandwidth regulation in presence of much tighter QoS requirements compared to previous work.

II. BACKGROUND AND STATE OF ART

A. 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 [10], GPGPU accelerators [11] [12] and FPGA accelerators [13] [14]. 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) [1]. Several works use this principle [4], [15]–[18] varying the granularity, the scheduling principles or the target architecture. Although functional, these approaches are typically 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. Yao et al. [19] try

to overcome such under-utilization by allowing multiple tasks to access DRAM at the same time. 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 [20]. The main limitation of such approaches resides in their static nature, which is not always practical or altogether feasible. Controlled Memory Request Injection (CMRI) [10] 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. Compiler-level code instrumentation and dynamic task throttling [7] can be used to implement the duty cycling.

Modern HeSoCs indeed typically include HW knobs for controlling QoS and fairness at different levels of the interconnect hierarchy [6], [21]. As it was shown in a recent work by Serrano et al. [9], using these knobs is typically not straightforward, due to the varying degrees of support on different products and to the many different (and in some cases obscure) configurations available. Furthermore, these mechanisms lack generality, as they are typically closed solutions specific to a given vendor or hardware platform (e.g., ARM MPAM [8], QoS400 [22]). Farshchi et al. [5] have studied an approach to mitigate memory contention between CPU cores by implementing a HW throttler as custom FPGA logic. Note that the FPGA is used only as a medium to emulate and prototype a dedicated HW block ideally sitting between the *host* and the shared buses. As such, the focus is not on the interference generated between CPU cores and FPGA accelerators. Moreover, given the slow speed of the FPGA logic compared to the CPU, the granularity at which the throttling can be done is very coarse.

Our proposed solution also focuses on FPGA-based HeSoCs, and overcomes the described problems by tightly coupling a lightweight HW monitoring system [23] with a novel Runtime Bandwidth Regulator (RBR). It operates as a simple, low-overhead block to be integrated in an FPGA accelerator and is capable of dynamically controlling the bandwidth generated by said accelerator in a fine-grained manner.

B. Background

We consider the architectural template of an FPGA-based HeSoC shown in Figure 1, which is composed of a *host* multi-core CPU complex coupled to an FPGA subsystem. The two subsystems communicate via the main DRAM. This template captures the main traits of several existing commercial products. Within the FPGA, one or more *accelerators* are deployed. A generic template for an *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 a fast, local memory. To simplify the development of FPGA applications it is common to also enrich the accelerator template with a *soft core* for local control of the *datapath* and *DMA* operation, without the need for the costly intervention of the main CPU [24]–[27].

In modern HeSoCs, the DMAs inside FPGA accelerators generate much higher DRAM bandwidth request than what happens on the CPU cores, and more than a single master port is typically available to individually attach accelerators to the main interconnect fabric (for example, the Xilinx Ultrascale+ device that we use for our experimental setup features three independent ports). If CPU cores and FPGA accelerators run in parallel without DRAM access control, the execution time of the CPU tasks can slow down by over $10\times$ [13]. On the other hand, enforcing mutually exclusive CPU/FPGA DRAM accesses causes severe under-utilization of the available memory bandwidth. Since the main interface of an *accelerator* to the DRAM is the *DMA*, bandwidth monitoring and throttling should happen at this level. Previous work has explored throttling FPGA *accelerators* by leveraging the *soft cores* for programming the DMA in a duty-cycled loop [14]. Each DMA transfer request is split into several smaller transfers, each of which can be interleaved with a programmable amount of $idle_{cycles}$, computed as shown in (1):

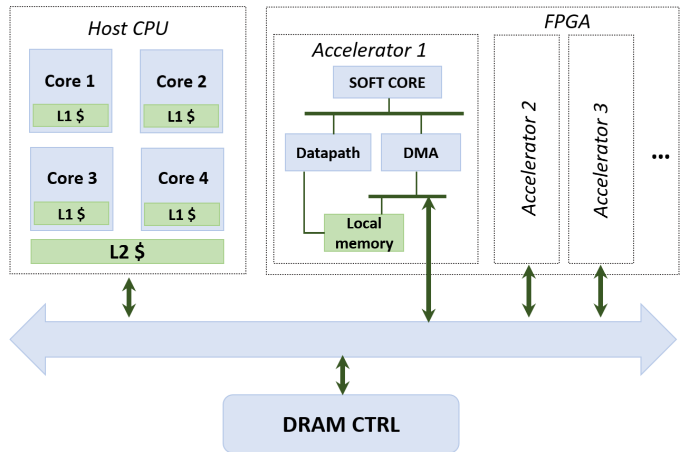

Fig. 1: Architectural template of the target HeSoC.

$$idle_{cycles} = \frac{100 - THR_{\%}}{THR_{\%}} * copy_{cycles} \quad (1)$$

where $THR_{\%} \in [1, 100]$ represents the *throttling factor*[*] to be applied to the transfer, while $copy_{cycles}$ is the time (clock cycles) it took to complete the small transfer. As we will show later on, the main drawback of throttling *accelerators* via software (albeit running on the local *soft core*) is the high programming overhead, which doesn't allow for very fine-grained operation. Furthermore, to retrieve the $copy_{cycles}$ from Equation 1 we need to interact with a *monitoring* component. The looser the coupling with such component, the higher the time to complete the *monitoring* + *throttling* control loop.

In real-time operating systems and applications new tasks can be admitted into the system with μs -scale frequency [28]; with the same frequency co-scheduling decisions related to the maximum allowed bandwidth to each task can be re-evaluated. Consequently, the mechanism for bandwidth *monitoring* and *throttling* needs to be as responsive and tight as possible.

[*]THR=100 means 100% bandwidth granted; THR=1 means 1% bandwidth granted.

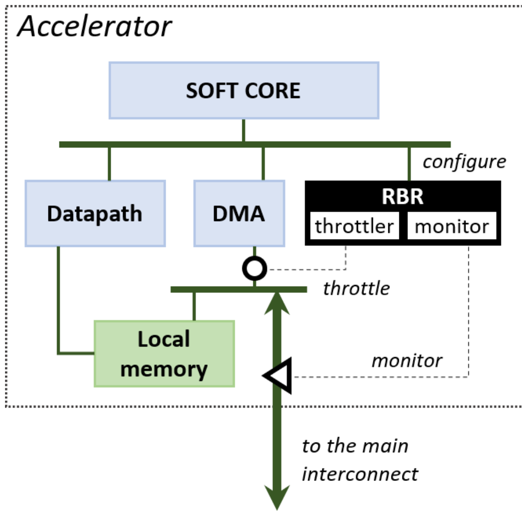

Fig. 2: The FPGA *accelerator* with the RBR.

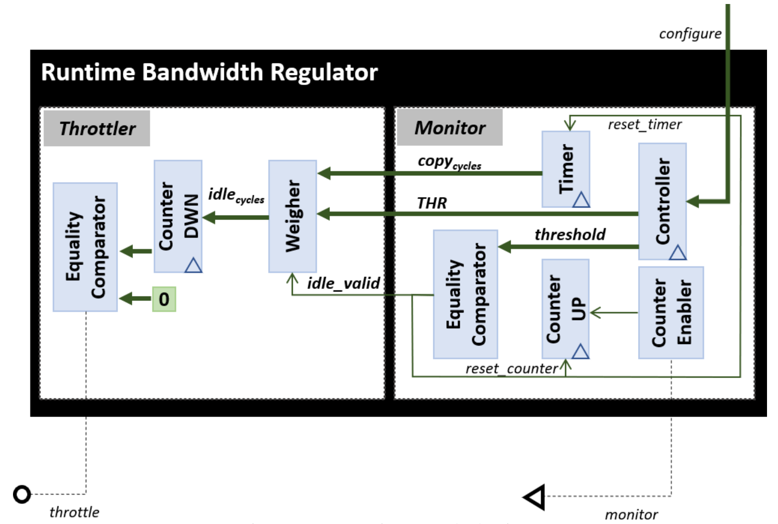

Fig. 3: RBR internal design.

III. Tightly-Coupled Bandwidth Monitoring and Regulation

In the following we present a Tightly Coupled Monitoring and Throttling (TCMT) solution enabling accurate bandwidth regulation of FPGA *accelerators* with minimal overheads and minimal impact on the timing. We assume that all the involved blocks communicate through standard AXI protocols, although the methodology is not specific to any protocol.

The RBR is introduced as a non-intrusive component of the accelerator template, as shown in Figure 2. It contains two main blocks: (i) a *monitor* that probes the outgoing channel to the DRAM to unobtrusively measure the time ($copy_{cycles}$ from Equation 1) to transfer a given amount of bytes (the *threshold*); (ii) a *throttler* that computes the $idle_{cycles}$ from Equation 1 and stops DMA operation for that amount of time. The value of the *threshold* and the throttling factor $THR_{\%}$ are provided during RBR configuration by the *soft core*. This is expected to happen every time the underlying middleware (e.g., the RTOS or the hypervisor) modifies the QoS requirements for the tasks scheduled, for example, a new task has been just admitted.

A close-up of the internals of the RBR is shown in Figure 3. Upon RBR configuration the *threshold* and $THR_{\%}$ values are stored into a *Controller block* in the *monitor*. The *monitor* relies on a *Timer* block to measure the $copy_{cycles}$. The number of bytes read (or written) through the outgoing channel to the DRAM is accumulated in a *Counter Up* block, that stays active until the *Equality Comparator* detects that 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 *throttler* receives the $copy_{cycles}$ and $THR_{\%}$ values from the *monitor* and relies on a *Weigher* block to compute the $idle_{cycles}$. This information is then written inside a down counter (*Counter DWN*) block, which also triggers the operation of a second *Equality 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, the *valid* and *ready* signals of the pending DMA transaction are de-asserted, which blocks DRAM access from this *accelerator*.

The tight coupling between the *monitor* and the *throttler* in the RBR guarantees a total delay of just 2 clock cycles to apply the desired QoS regulation. The granularity of the bandwidth regulation is thus only dependent on the *threshold* value, which is fully in control of the software stack. 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 a FPGA-based HeSoC. Porting the solution to a specific architecture only requires to adapt the outbound monitoring and throttling signals to match the bus protocol. The flexible programmability of *thresholds* and *THR* values allows changing the regulation factor and its granularity at run-time.

IV. Experimental Results

We implement our proposal on a Xilinx Zynq Ultrascale+, *XCZU9EG* HeSoC. The base accelerator template was modeled after the setup described in [14], using Xilinx IPs for the DMA, soft-core and interconnects. As we are only interested in measuring the worst-case interference effects, our accelerators are configured to work as traffic generators[†] [13], and are extended with our RBR. The resulting design was synthesized with a target frequency of 100 MHz. With this setup, on the *XCZU9EG* less than 1% of the FPGA resources (LUTs, FFs and DSPs), are used for the RBR.

Our experimental section is organized in three distinct parts. The first compares the cost for monitoring and throttling operations using the RBR and other standard HW/SW mechanisms available in the target platform. The second focuses on tightly-VS loosely-coupled monitoring and throttling, comparing RBR to other solutions built on top of the available HW mechanisms on the target platform. The third highlights the benefits of fine-grained bandwidth regulation on co-scheduled tasks at the system level.

[†]Note that this is without loss of generality, as well-designed accelerators overlap memory transactions with computation.

Listing 1: Monitoring via APM

```
int APM_monitor (int threshold) {
  while (ReadAPM(BYTES_TRANSFERRED_COUNTER)<
      threshold)
    ;
  return ReadAPM (CYCLE_COUNTER);
}
```

Listing 2: Throttling via SW-controlled DMA

```
void throttle_SW_DMA (int idleCycles) {
  for (int i=0; i < NTRANS; i++) {
    // DMA transfer
    DMA_prog (src, dest, TRSIZE);
    // idle cycles
    Wait (idleCycles);
  }
}
```

Listing 3: Throttling via QoS400

```
void throttle_QoS400 (int idleCycles) {
  qos = idleCycles_to_QoS400_params (idleCycles);
  QoS400_WriteReg (qos);
}
```

Listing 4: LCMT-RBR

```
// Monitoring via APM
copyCycles = APM_monitor (threshold);

// Throttling via RBR
RBR_WriteReg (WEIGHER.time, copyCycles);
RBR_WriteReg (WEIGHER.thr, Thr);
RBR_Assert (IDLE_VALID, 1);
RBR_Assert (IDLE_VALID, 0);
```

Listing 5: LCMT-QoS400

```
// Monitoring via APM
copyCycles = APM_monitor (threshold);

// Throttling via QoS400
idleCycles =(1/Thr)*(100 - Thr)*copyCycles;
throttle_QoS400 (idleCycles);
```

Listing 6: LCMT-SW-DMA

```
// Monitoring via APM
copyCycles = APM_monitor (threshold);

// Throttling via SW-controlled DMA
idleCycles =(1/Thr)*(100 - Thr)*copyCycles;
throttle_SW_DMA (idleCycles);
```

A. Monitoring and Throttling Cost

Bandwidth *monitoring* is typically supported in hardware to some extent on HeSoCs. On the Zynq UltraScale+ this can be done by relying on the *Xilinx AXI Perfomance Monitor (APM)*, a commercial solution to measure AXI performance metrics. On the XCZU9EG HeSoC there is an APM connected to each one of the six ports entering the DRAM controller (three attached to the *host* CPU complex, three to the FPGA). The APMs can be programmed and read by the processing units available on the *host* side of the Zynq UltraScale+, namely the Real-Time Processing Unit (RPU) (ARM Cortex-R family cores) and the Application Processing Unit (APU) (ARM Cortex-A family cores). We execute the software that controls the APMs on the RPU cores, to avoid the involvement of APU cores, where applications typically execute. Listing 1 shows the pseudo code for APM-based monitoring. The APMs can be used to count the number of read/written bytes (`BYTES_TRANSFERRED_COUNT` in our pseudo-code) through the target port and to measure the elapsed clock cycles since activation (`CYCLE_COUNTER`). Our pseudo-code implements the fastest control mode (active polling), which allows to read/write from an APM register (the `ReadAPM` API) in 0.32μs on the RPUs. During this time, the DMA can transfer 512B, which dictates a convenient granularity we can use for the *throttling* (the *threshold*).

Bandwidth *throttling* can be achieved on the target platform by explicitly duty cycling the DMA operation in SW [14]. Listing 2 shows the pseudo-code to be executed on the RPU. The original transfer is split in `NTRANS` smaller transfers, each of size `TRSIZE`. Between one small transfer and the other the `Wait` function is invoked, which stalls the DMA for `idleCycles` cycles. Figure 4 shows the performance penalty to implement such scheme as a single DMA transfer of 512KB is split in increasingly smaller and more numerous ones, when the bandwidth is not throttled (`idleCycles`=0). Transferring 512KB in `NTRANS`=1024 chunks of `TRSIZE`=512B each costs ten times a single transfer of 512KB, requiring around $3ms$. On top of the delay implied by the overhead for programming several DMA transfers, we must consider the idle time to estimate the granularity of the complete duty cycle. As $THR_{\%}$ decreases and approaches 0, the idleness increases. Assuming the worst-case $THR_{\%} = 1$ (1% bandwidth) the system would become ready to process a new request after $(3ms/1024) * 100 = 300\mu s$. For comparison, the HW throttler we propose does not imply overheads for partitioned DMA transfers, and effectively uses the time to transfer 512B (0.32μs) as a baseline for the idle time calculation. Considering the worst-case $THR_{\%} = 1$ request the HW throttling approach would become ready to process a new request after $0.32 * 100 = 32\mu s$.

Another way of throttling bandwidth on the *XCZU9EG* is that of relying on the ARM QoS-400 regulators [9], an extension to AMBA AXI4 interconnect that provides additional dynamic QoS regulation mechanisms. Although this type of support is not always available, and thus relying on it makes the solution very platform dependent, we still deem it relevant to compare against this type of state-of-the-art HW knobs. Listing 3 shows the pseudo-code to be

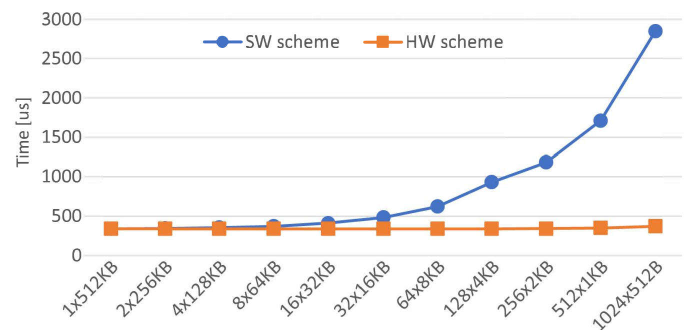

Fig. 4: SW and HW Throttling Costs (in μs) to partition a DMA transfer in multiple smaller transfers.

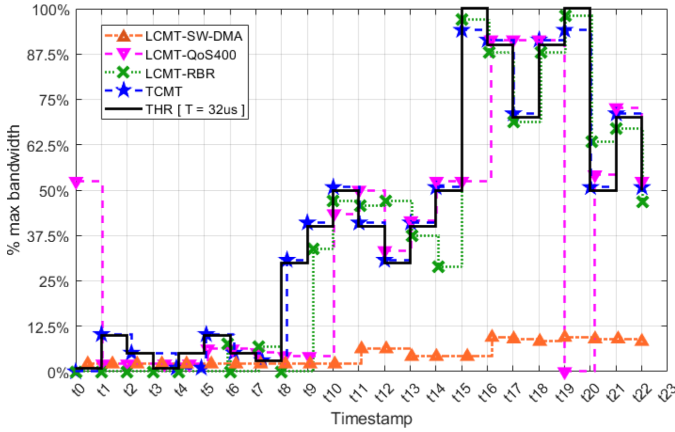

Fig. 5: Comparison of our TCMT with loosely-coupled SW mechanism, to follow a temporized bandwidth profile.

executed on the RPU. Programming the QoS-400 regulators can be achieved with a single write into a memory-mapped register (in our pseudo-code, via the `QoS400_WriteReg` API). However, since QoS-400 also splits DMA transfers into small ones of unknown size, the value written inside the register (the `qos` parameter) is not a direct representation of the desired `idleCycles` value, but rather a custom bitmask that can be retrieved via a look-up table indexed with `idleCycles`. For this reason, we first need to execute a `idleCycles_to_QoS400_params` function that performs the look-up. Other than the usual 0.32μs to write into the QoS-400 register, 1.89μs are needed for the look-up.

B. Tightly-coupled monitoring and throttling

This experiment is aimed at comparing the speed at which various bandwidth regulation (monitoring+throttling) approaches adapt to a trace of dynamically evolving QoS settings (i.e., $THR_{\%}$ settings). We expect QoS requirements to change as new (co-)scheduling decisions are taken by the OS; this happens when new tasks are admitted into the system, or when periodic tasks start/end their execution. For the bandwidth regulation mechanism to be capable to adapt to the μs -scale task frequency of certain real-time applications [28], tight coupling of the *monitoring* and *throttling* operations is fundamental. To show this, we compare four approaches:
1) TCMT: Our tightly-coupled regulation solution, entirely based on the RBR;
2) LCMT-RBR: Loosely-coupled regulation implemented by coupling APM-based monitoring and RBR-based throttling;
3) LCMT-QoS400: Loosely-coupled regulation by coupling APM-based monitoring and QoS400-based throttling;
4) LCMT-SW-DMA: Loosely-coupled regulation implemented by coupling APM-based monitoring and explicit SW-based DMA throttling.

All the LCMT approaches leverage APM for the *monitoring* phase (the `APM_monitor` API). LCMT-RBR, shown in Listing 4, requires four writes into the RBR *throttler* registers to configure its operation. Specifically, we first, need to initialize the *Weigher* with the `copyCycles` and the $THR_{\%}$ value, then we need to assert and deassert the `idle_valid` signal. LCMT-QoS400 and LCMT-SW-DMA, respectively highlighted in Listings 5 and 6, require to determine via SW the `idleCycles` value using Equation 1, then they rely on the throttling APIs introduced in the previous section.

TABLE I: BW regulation granularity (minimum period T [μs])

	System			
	TCMT	**LCMT-RBR**	**LCMT-QoS400**	**LCMT-SW-DMA**
T[us]	32	192	320	3645

Figure 5 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 handles a worst-case 1% bandwidth throttling request, whose magnitude matches the task admission frequency for control-oriented real-time applications [28]. 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 a trace of $THR_{\%}$ setting requests (e.g., coming from the OS scheduler). The other curves show how the various bandwidth regulation approaches adapt to the black curve over time.

TCMT precisely follows the $THR_{\%}$ profile in every operating condition, as per its nominal latency. All the LCMT approaches are in general not capable of adjusting to a QoS trace evolving this fast, even those that rely on very fast HW throttling mechanisms (LCMT-RBR and LCMT-QoS400). Table I shows the minimum period T for which the various approaches adapt to the QoS request profile. LCMT-RBR, LCMT-QoS400, LCMT-SW-DMA are respectively $6\times$, $10\times$, $114\times$ slower than TCMT. In conclusion, the tightly-coupled approach makes bandwidth regulation effective for applications with timing resolution one to two orders of magnitude smaller than what is possible for loosely-coupled approaches.

C. System-wide interference mitigation

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 [9]. Here, three FPGA accelerators (Xilinx traffic generators) are considered, each attached to a different DRAM controller port (with a dedicated QoS-400 regulator). 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 QoS knobs 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 (exe-

TABLE II: Real-world benchmarks

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×

cuting the same benchmarks described above). Table II shows the slowdown (compared to the execution time in absence of interference) experienced by the involved processing units for the VT, T and M *QoS scenarios*, i.e., the ones for which the QoS-400 and other HW QoS knobs could not satisfy the requirement [9]. The results show that RBR can satisfy the requirements for all the actors in *QoS scenarios* M and T, and for most actors also in *QoS scenario* VT (except the cells shaded in red). This further confirms that tightly-coupled bandwidth regulation enables system-wide scheduling opportunities that are not feasible with state-of-the-art mechanisms.

V. Conclusion

We introduced a tightly-coupled bandwidth monitoring and throttling solution for FPGA-based HeSoCs. This innovative solution is based on an original IP, the *Runtime Bandwidth Regulator*, that can unobtrusively be integrated in generic FPGA accelerator designs. This approach makes bandwidth regulation effective for applications with timing resolution one to two orders of magnitude smaller than what is possible for state-of-the-art solutions. When evaluated at the whole-system level for the co-scheduling of SW and HW tasks, RBR enables effective bandwidth regulation in presence of much tighter QoS requirements compared to previous work.

VI. Acknowledgements

The authors have received funding from ECSEL JU projects AI4CSM (GA N.101007326) and FRACTAL (GA N.877056).

References

[1] 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

[2] 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

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

[4] 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 Tech. and Applications Symposium*, 2011, pp. 269–279.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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, p. 85.

[11] H. Wen and W. Zhang, "Interference Evaluation In CPU-GPU Heterogeneous Computing," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[12] 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.

[13] M. Mattheeuws, B. Forsberg, A. Kurth, and L. Benini, "Analyzing Memory Interference of FPGA Accelerators on Multicore Hosts in Heterogeneous Reconfigurable SoCs," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1152–1155.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 345–357.

[21] 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.

[22] 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.

[23] 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

[24] 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.

[25] 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 Inter. Conf. On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[26] 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.

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

[28] 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.