A Memory Scheduling Infrastructure for Multi-core Systems with Re-programmable Logic

Authors omitted for review.

***Abstract*—The sharp increase in demand for performance has prompted an explosion in the complexity of modern multi-core embedded systems. This has lead to unprecedented temporal unpredictability concerns in Cyber-Physical Systems (CPS). On-chip integration of programmable logic (PL) alongside a conventional Processing Systems (PS) in modern Systems-on-Chip (SoC) establishes a genuine compromise between specialization, performance, and re-configurability. In addition to typical use-cases, it has been shown that the PL can be used to observe, manipulate, and ultimately manage memory traffic generated by a traditional multi-core processor.**

This paper explores the possibility of PL-aided memory scheduling by proposing a Scheduler In-the-Middle (SchIM). We demonstrate that the SchIM enables transaction-level control over the main memory traffic generated by a set of embedded cores. Focusing on extensibility and reconfigurability, we put forward a SchIM design covering two main objectives. First, to provide a safe playground to test innovative memory scheduling mechanisms; and second, to establish a transition path from software-based memory regulation to provably correct hardware-enforced memory scheduling. We evaluate our design through a full-system implementation on a commercial PS-PL platform using synthetic and real-world benchmarks.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

It is undeniable that the massive increase in expectation on the performance of next-generation cyber-physical systems has deeply impacted the way we design modern embedded and real-time systems. High-resolution, high-bandwidth sensors such as lidars, and depth cameras on the one hand, and data-intensive processing workload such as machine-learning applications on the other hand, have exacerbated the push for high-performance embedded platforms. Following this performance *moving target*, chip manufactures have significantly scaled up clock speeds, CPU count, and heterogeneity. For instance, the on-chip integration of powerful graphic processing units (GPUs) has been the characterizing factor in the NVIDIA Tegra series of embedded systems-on-a-chip (SoC).

In this context, an embedded architectural paradigm that is surging in popularity among manufacturers, researchers, and industry practitioners is the PS-PL organization. This class of embedded platforms integrates on the same die (1) traditional full-speed embedded CPUs and (2) programmable logic constructed using field-programmable gate array (FPGA) technology. This organization naturally defines two macro-domains, namely the Processing System (PS) and the Programmable Logic (PL), hence the name. PS-PL platforms establish a good trade-off between specialization, raw performance, and mission-specific re-configurability. The current generation of commercially available PS-PL platforms is dominated by ARM-based products offered by, most notably, Intel [?] and Xilinx [?]. A pilot large-scale, high-performance PS-PL system is the Enzian platform [?] being rolled out by ETH Zurich[1]. Furthermore, a RISC-V-based solution has been recently made available by Microsemi with their PolarFire SoC [?].

From a real-time perspective, the co-existence of traditional CPUs and a tightly-coupled block of PL has more profound implications than expected. Clearly, it is possible to define custom accelerators in PL and to relieve the main CPUs of some of the heavy data-processing workload. However, more interestingly, recent studies have highlighted the possibility of using the PL also as a way to manage the memory traffic originated from the main CPUs [?], [?]. Such a possibility opens the doors for memory traffic inspection and control at the level of individual transactions; which in turn promises to unlock provable determinism for the real-time workload.

In this paper, we embrace the concept of PL-aided memory traffic management and propose an infrastructure to develop, test and evaluate memory scheduling policies. Specifically, we propose a component, called the Scheduler In-the-Middle—or SchIM, for short—that can be instantiated in the PL to enforce a set of configurable scheduling policies on individual memory transactions generated by the CPUs in the PS.

The overarching goal of the proposed SchIM is twofold. First, we want to provide a playground for researches to test promising novel memory scheduling ideas for multi-core platforms, much like LITMUS^{RT} [?] fostered research on CPU scheduling techniques. Second, we want our SchIM to act as an intermediate stepping stone for industrial applications where strong determinism over memory performance is required. The SchIM can be used to analyze the behavior of realistic workload in a multitude of what-if memory management use-cases. We note that such kind of analysis was previously possible only through full-system simulation or by synthesizing the entire SoC on FPGA—that is, with a soft-core implementation.

In short, this paper makes the following contributions. (1) We demonstrate that a configurable module could be interposed between the cores and the memory controller to perform transaction-level scheduling in commercial PS-PL platforms; (2) we propose a design for a memory scheduling infrastructure that focuses on extensibility and runtime reconfigurability; (3) we address important issues to correctly account and regulate CPU-generated traffic when a shared last-level cache

[1]Also see http://enzian.systems/

is present; (4) we design and implement three pilot memory scheduling policies as a proof-of-concept on the potential of our SchIM; and (5) we perform a full system integration and implementation on a commercial PS-PL embedded platform to evaluate the behavior of the SchIM with synthetic and realistic workload.

II. Related Work

There is a broad consensus that memory resources represent the main performance bottleneck in modern multi-core processors. The observation has sparked a host of research works addressing the problem from multiple angles [?]. In this context, the works representing the inspiration for our SchIM fall in two macro-categories, namely **hardware-based** and **software-based** techniques for main memory traffic management.

The first category includes a large body of works aimed at achieving better and/or more predictable performance by advancing novel hardware redesigns. The works in [?], [?], [?] strive to construct high-performance and fair memory schedulers. The addition of software-controlled memory deadlines and transactional semantics where explored in [?] and [?], respectively. Next, the work by kesson et al. [?], [?] and Paolieri et al. [?] attains timing predictability through careful scheduling of SDRAM commands. Finally, the MEDUSA DRAM controller [?], [?] implements a two-tiers scheduler at the DRAM controller to ensure predictability when accessing memory areas where access time strongly impact application performance. Finally, the hardware designs proposed in [?], [?], [?] put their emphasis on main memory bandwidth partitioning; clever dynamic pipelining is further explored in [?] to better balance average performance and determinism.

Among the software-based techniques are the mechanisms that stemmed from MemGuard, originally proposed in [?] and that rely on broadly available performance counters to regulate the bandwidth extracted by individual CPUs. Later extensions to jointly consider regulation and cache partitioning [?] and to expose control over memory bandwidth as a lockable resource [?] were proposed. Software-based memory throttling has also been implemented at the hypervisor-level [?], [?]. Remarkably, the work in [?] combines regulation mechanisms for CPU and embedded accelerators through the ARM QoS extensions [?].

In addition to the two categories surveyed above, perhaps the most closely related works are those that explored memory isolation techniques in PS-PL platforms. The work in [?] demonstrated that the PL-side can be used to define private memory storage, control, and bus units to strongly isolate high-criticality workload. A number of techniques developed as part of the FRED framework [?] put an emphasis on memory traffic arbitration and management for in-PL accelerators [?], [?]. The AXI HyperConnect [?] is perhaps the component most similar to the SchIM in terms of high-level design. However, both are substantially different as the SchIM is designed to manage embedded CPUs' memory traffic.

Compared to the literature reviewed above, what sets this work apart are the following aspects. (1) Our SchIM applies to existing PS-PL commercial systems without introducing any hardware modification; (2) it allows management in the PL of memory traffic originated by the embedded CPUs residing in the PS; (3) it provides the framework to test the feasibility and performance of custom memory scheduling policies; and (4) it is designed such that multiple schedulers can coexist, be activated, and configured at runtime.

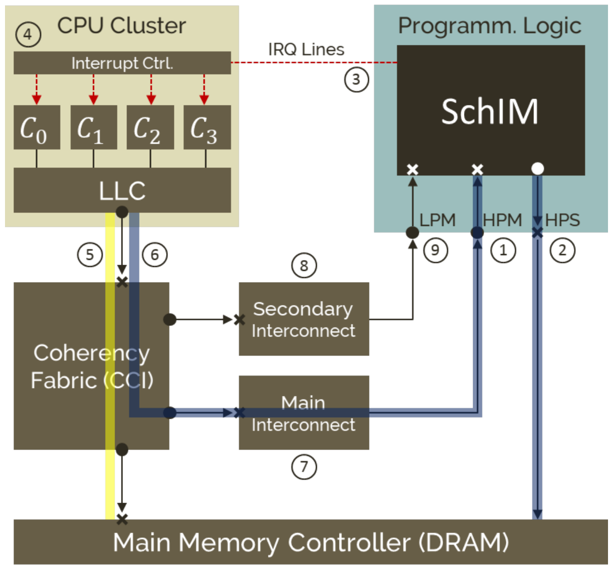


Fig. 1: PS-PL interconnect block diagram

III. Background Concepts

In this section, we introduce some fundamental concepts necessary to understand the overall system design and the class of platforms targeted by this work.

A. Hybrid Multi-Core Platforms with Programmable Logic

This work targets the aforementioned class of embedded multi-core platforms with programmable logic—i.e., PS-PL platforms. In such platforms, the PS encompasses a multi-core processor with a multi-level cache hierarchy and a main memory (DRAM) controller. A simplified block diagram for a reference PS-PL organization is illustrated in Fig. 1. The figure considers a platform with four CPUs denoted as C_0, C_1, C_2, and C_3.

A key feature in PS-PL platforms is the presence of high-performance communication channels between the two domains. These come in the form of data exchange interfaces and interrupt lines. Data exchange channels follow a master-slave paradigm. Specifically, high-performance masters (HPM, Fig. 1①) and high-performance slaves (HPS, Fig. 1②) send and receive transactions to and from the PL, respectively. Additionally, there exist programmable interrupt request (IRQ) lines (see Fig. 1③) that can be driven by the PL and are connected to the interrupt controller (Fig. 1④) inside the PS. As we discuss in Section V-G, the presence of PS-PL interrupt lines is crucial to building PL-assisted memory traffic regulation.

Note also that there might exist PS-PL data ports that are routed through a secondary interconnect (Fig. 1⑧). These can generally sustain less throughput compared to HPS ports; hence we refer to them as low-performance masters (LPM, Fig. 1⑨). LPM ports are useful to perform memory-mapped configuration of PL modules.

B. Programmable Logic In-the-Middle

In this work, we leverage the ability to route main memory traffic originated by the CPUs through the PL. This technique is known as Programmable Logic In-the-Middle, or PLIM for short. PLIM was originally proposed in [**?**]. To fully grasp how PLIM can be achieved, one needs to understand how memory accesses are routed in PS-PL platforms.

Any CPU-generated memory access that results in an LLC miss is routed directly to main memory if its physical address falls within the aperture, say the address range $[A, B]$ handled by the DRAM controller. We refer to this as the *normal route*, depicted in Fig. 1⑤ and highlighted in yellow.

Conversely, generic memory access resulting from an LLC cache miss will be sent on an HPM port if the corresponding physical address falls within another range, say $[C, D]$. One can then insert (1) a lightweight layer of virtualization to map all the physical addresses of a guest OS to the PL, i.e., to fall in the range $[C, D]$; and (2) an address translator in the PL that re-bases request physical addresses to access main memory and relays back the data payload to the requesting CPU(s). In other words, one can find a constant k such that $C = A + k$. Then, the translator in the PL, upon receiving any request at address $x \in [C, D]$ will issue a main memory request at the address $(x - k)$ through the HPS port and provide the response to the CPU. The PLIM technique introduces a secondary memory route for reaching the DRAM, called the *PL loop-back*, or simply *loop-back*, which is highlighted in blue in Fig. 1⑥. Memory transactions on the loop-back route typically traverse the main interconnect, as depicted in Fig. 1⑦. The advantage of PLIM is that transactions on the loop-back route can be inspected, blocked, re-routed, and in general managed by custom re-programmable logic. Importantly, switching from the direct to the loop-back route can be done dynamically at runtime so that the overhead of PLIM can be avoided if deemed detrimental for the application under analysis.

In this paper, we leverage the PLIM approach to perform memory scheduling, hence, we call our module the Scheduler In-the-Middle, or SchIM for short.

C. Advanced eXtensible Interface (AXI)

The vast majority of PS-PL platforms currently available are ARM-based. This is also the case for the platform we used for our evaluation, namely the Xilinx Zynq UltraScale+ MPSoC. Thus, we briefly introduce the communication protocol used for on-chip communication in ARM-based SoCs, namely the Advanced eXtensible Interface (AXI). The AXI is an open specification bus protocol [**?**] used for high-bandwidth data exchanges between on-chip subsystems — such as cache controllers, memory controllers, DMAs, PL modules. It is also used in the PS-PL platforms of reference to exchange data on the aforementioned HPM/HPS ports.

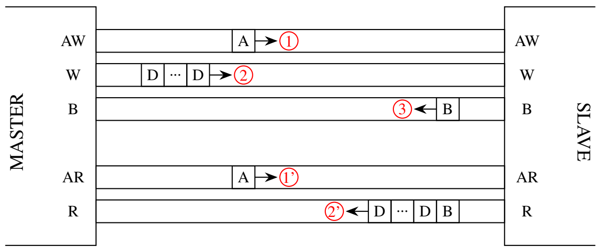


Fig. 2: Caption

The AXI protocol is based on the master-slave duality. A master AXI interface can initiate transactions toward a connected slave interface. The latter responds master-initiated requests. Masters and the slaves communicate with each other through five different channels named AW (address write), W (write), B (write acknowledgment), AR (address read) and R (read), as illustrated in Fig. 2.

A write transaction begins with an address phase ① where the channel AW is used to transmit the transaction's meta-data, such as the destination address, the transaction ID, and the cacheability attributes the type/length of the burst, and so on. Upon completing this phase, follows the data phase ②, which consists of the transmission of the data payload to be written through the W channel. The response phase ③ concludes a successful write transaction and occurs on the B channel.

The transmission of a read transaction is carried out in a similar way. The address phase 1' is transmitted through the equivalent AR channel and is directly followed by the data phase 2'. A response initiated by the slave follows where the read data is transferred over the R channel. The protocol is asynchronous because different phases of different transactions can interleave on any AXI bus segment. Hence, multiple outstanding transactions can be emitted by a single master and the receipt of out-of-order responses is possible.

IV. Design Goals and Overview

In this section, we introduce the proposed SchIM design and describe the overarching goals of this work. We then provide a bird's-eye view of the SchIM organization and principles of operation.

A. Design Goals

As briefly surveyed in Section II, there have been numerous proposals for better memory controllers and approaches to manage memory traffic in modern multi-core embedded platforms. With respect to the existing literature, the purpose of this work is twofold. First, we want to demonstrate that scheduling CPU-originated memory traffic at the granularity of individual transactions is possible in PS-PL platforms. Second, and more importantly, we want to provide an infrastructure that is generic and extensible enough for the broader research community to adopt and foster a new chapter on PL-assisted

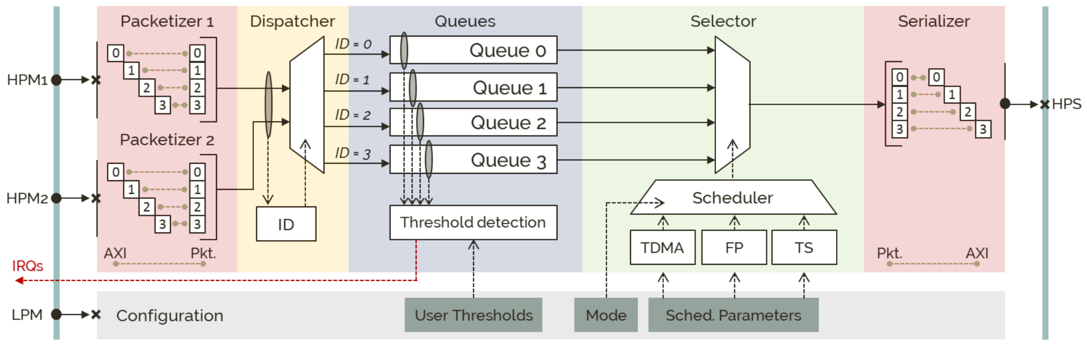


Fig. 3: Internal logic organization of the SchIM connected to the PS through the HPM, LPM and HPS ports.

memory scheduling. With this in mind, we establish the following goals.

Extensible memory scheduling infrastructure. First and foremost, the SchIM has been designed with modularity and extensibility in mind. We separate the functionalities that concern handling, queuing, selection, and forwarding of memory requests inside our infrastructure. Moreover, we design our SchIM to be able to support multiple memory scheduling policies simultaneously. A simple, standardized interface is provided to define new memory scheduling policies without impacting the design of the rest of the SchIM. We discuss in Section V-E the generic interface provided by the SchIM to implement a new memory scheduling policy.

Runtime configuration and transparency. We want the SchIM to be a robust supporting infrastructure to evaluate, compare, and contrast memory scheduling policies. As such, we strive to provide (1) runtime reconfigurability and (2) operational transparency. It is possible to rapidly identify desirable configuration parameters by allowing memory scheduling policies to be switched at runtime. Besides, an adopted policy can be tuned according to the workload criticality and memory intensiveness. For this purpose, the SchIM exposes a memory-mapped configuration interface where all the operational parameters can be changed at runtime. At the same time, we want to ensure that the applications and the (real-time) operating system under analysis need not be modified to use the SchIM. Hence, we propose using a thin virtualization layer to selectively route memory traffic through the SchIM without changes to the binary of OS kernel and applications.

Realistic performance with experimental policies. One of the limiting factors of research on memory scheduling policies is the ability to construct evidence of performance improvements with the realistic workload. Proposing a new memory scheduling policy is traditionally done with either a simulated setup or with a full-system soft-core implementation. Both cases are considerably slower than what it would be in real systems and force the use of an unrealistically lightweight workload. Nonrealistic workloads limit the repeatability of experiments of scale.

Furthermore, the multi-fold drop in performance excludes the possibility of adopting experimental memory scheduling policies for a production-ready system. Conversely, we envision that our SchIM will be an intermediate step in the transition path to the production of research-seeded ideas. Additionally, as PS-PL platforms mature and the interplay of PL and memory resources improves, a SchIM-like design could be the way to go for mission-reconfigurable, upgradable embedded systems.

B. Design Overview

As previously mentioned, the SchIM leverages the PLIM approach. CPU-originated main memory transactions are re-routed through the programmable logic and scheduled by the SchIM according to a flexible and configurable policy. The result is that the timing of memory transactions generated by real-time applications can be carefully determined and reasoned upon. Because the SchIM follows a PLIM approach, transactions can be selectively sent to the SchIM for scheduling. However, it is always possible to dynamically exclude the SchIM and route transactions directly to the main memory. Toward this paper's incentive, we consider a setup in which SchIM handles all the CPU-generated memory transactions.

Fig. 1 provides an overview of the location of the SchIM in the reference platform, while its internal organization is visible in Fig. 3. Application memory requests reach the SchIM the aforementioned HPM ports. Without loss of generality, we consider a SchIM instance with two arrival lanes, which are labeled as $\mathtt{HPM}_1$ and $\mathtt{HPM}_2$ in Fig. 3. The SchIM then forwards the received transactions towards main memory through the `HPS` interface. A more detailed view of the SchIM module is provided in Fig. 3 where the same convention is used to identify input and output ports. In addition, as shown in Fig. 3, a fourth `LPM` port is used to configure the SchIM from the PS.

The SchIM module is composed of a number of sub-modules grouped into three different domains, namely (i) the *interfacing domain*, (ii) the *queuing domain*, and (iii) the *scheduling domain*.

The interfacing domain encompasses the sub-modules to interface the core logic of the SchIM with the rest of the system using the AXI protocol. This is comprised of three sub-modules. These are (i) the *packetizer(s)*, (ii) the *serializer*, and (iii) the previously mentioned *configuration* interface.

The PS-facing end of the **packetizer** offers an AXI slave port to accept new incoming transactions. Upon receipt, this module transforms each transaction into an equivalent *packet* that can be queued and scheduled by SchIM. Packetization of AXI transactions is necessary to be able to store transactions that are serial by nature. A standard AXI transaction is composed of one address phase (AR or AW channel) followed by a data phase (R or W channel), which can be itself composed of multiple successive bursts.

In many ways, the **serializer** is the dual module of the packetizer. Its purpose is to transform the packets that encode CPU-generated memory requests back into AXI-compliant transactions. As such, the serializer offers a master port to the rest of the system to be routed to the main memory controller.

The queuing domain handles how packets are stored between receipt and re-trasnmission. This domain is comprised of (i) the *dispatcher* module, (ii) the *transaction queues*, and (iii) the *selector* module.

The use of **multiple transaction queues** is necessary to differentiate the traffic of the CPUs and perform scheduling. As such, the SchIM associates a queue to each of the active cores — four in the platform of reference. The queues implemented in the SchIM not only act as a holding space for in-flight memory transactions. They also (a) provide information to the scheduling domain regarding their current state, and (b) they can generate a congestion control signal to the associated CPU core.

Congestion control is vital because memory transactions originated at the LLC controller follow the same route to the SchIM regardless of the originating CPU. Typically, the total number of outstanding transactions that the cores can emit exceeds the queuing elements' capacity on the loop-back route. Hence, priority inversion arises if a low-priority CPU's memory traffic is (temporarily) held. Latter is due to the uncontrolled queue buildup, which provokes a head-of-line blockage. Importantly, what described is true also for the normal route and it is a direct consequence of the best-effort nature of traditional multi-core memory buses. The SchIM allows the user to specify a configurable threshold on the occupancy of the queues that, when reached, issues a regulation signal to the corresponding CPU. We describe in greater detail how congestion control was implemented on the target platform in Section V-G.

As suggested by Fig. 3, transactions are categorized and enqueued based on the source of traffic. The **dispatcher** module performs the matching between an incoming transaction and the destination queue. Similarly, transactions are dequeued by the **selector** module and sent directly to the output of the SchIM following the scheduling domain's resolutions.

The scheduling domain encompasses all the sub-modules that enable arbitration of transactions issued by the different cores of the PS. The modules in this domain are intended to be generic for extensibility, albeit the first set of three template schedulers is provided as a proof of concept. The scheduling policies currently implemented in the SchIM are Fixed Priority (FP), Time Division Multiple Access (TDMA), and Budget-based Traffic Shaping (TS). Each of the parameters required by the implemented policies — such as the priorities, the periods, and the budgets — can be adjusted at runtime via the configuration interface.

The FP scheduler allows associating a priority value to each of the transaction queues. Pending transactions at the queues are then forwarded out of the SchIM following the user-defined priority order. The TDMA scheduler allows associating a transmission time slot to each of the queues expressed in PL clock cycles. The module then builds a schedule by concatenating the per-core slots so that only pending transactions from one queue at a time are forwarded by the SchIM. Finally, with the TS scheduler, it is possible to associate a maximum rate at which transactions from each queue are forwarded by the SchIM.

V. SchIM Design and Implementation

A full-system implementation was carried out on a Xilinx ZCU102 development system, which is based on a Xilinx Zynq UltraScale+ XCZU9EG PS-PL SoC. The PS comprises includes four ARM Cortex-A53 CPUs that share a unified 1 MB LLC. The PS includes a DDR4-2666 controller connected to a 4 GB DDR4 memory module. There are two high performance master interfaces (HPM1 and HPM2); and a third interface routed through the low power domain (LPM). The PL is capable of driving up to 16 interrupt requests lines towards the PS interrupt controller. We hereby provide key details on the operation of our SchIM in the target platform. These include complementary software stack, memory traffic accounting, regulation to prevent head-of-line blocking, and programming model.

A. Software Stack

As mentioned in Section IV-A, we want to ensure that the SchIM can be used with no modification to the OS and the applications under analysis. For this reason, we rely on a thin virtualization layer that can be used to redirect memory traffic from the direct route to the loop-back route (see Section III-B). For this purpose we use the open-source Jailhouse [**?**] partitioning hypervisor[2] Jailhouse does not boot the target machine. Instead, it relies on a standard Linux kernel to perform the initial boot sequence. When enabled form a Linux driver, Jailhouse dynamically virtualizes the original OS. In line with its partitioning-only philosophy, Jailhouse has a small footprint and enforces virtualization-aided partitioning of essential resources like CPUs, interrupts, main memory, I/O devices. It does not perform any virtual-CPU scheduling.

Following Jailhouse's nomenclature, a resource partition is called a *cell*, while guest OS's are referred to as *inmates*. An inmate can be either a bare-metal application, an RTOS or a full-fledged OS like Linux. Jailhouse uses ARM hardware Virtualization Extensions (VE) to offer a set of Intermediate Physical Address (IPA) to its inmates that is compatible with the way they have been compiled. Jailhouse then maps IPA ranges of different cells to configurable Physical Addresses

[2]The source code is available at https://github.com/siemens/jailhouse.git.

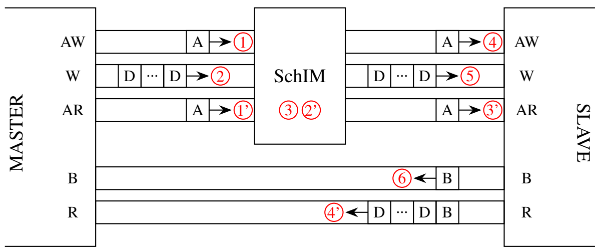


Fig. 4: AXI communication with mediated by the SchIM.

(PAs) — stage-2 translation. By changing the configured stage-2 mapping, it is possible to dynamically re-route via the loop-back the memory traffic generated by each inmate.

As described below, some modifications were necessary to the mainline Jailhouse code for our full system implementation[3].

B. Altered communication scheme

In order to achieve the objective of re-ordering transactions, one must alter the standard AXI communication scheme explained in the Section III-C. To this end, the SchIM is interposed between the master (HPM) and the slave (HPS) as depicted in Fig. 4. As shown in Fig. 4, only the phases initiated by the masters (i.e. address phase on AW and AR and the data phase on W) are intercepted for re-ordering by the SchIM. The introduction of the SchIM has a direct consequence on the overall communication scheme. Unlike the response phases on channels R and B that remain unchanged, the address and write data phases are handled following a store-and-forward scheme. Consequently, a write transaction will start exactly as in the standard AXI scheme with its address phase ① and data phase ②. These two transactions are buffered within the SchIM's queues (③) and only relayed following the internal memory scheduler's logic. This release of the transaction leads to the initialisation of two new address and data phase ④ and ⑤. Finally, the response phase ⑥ goes directly from the slave to the master without being intercepted. For read transactions, the same modifications apply to the address phase ①’ which is buffered (②’) for some time before being re-emitted in ③’. Just like for write acknowledgments writing, the read response phase ④’ is not intercepted by the SchIM.

C. Queueing Domain

At the heart of the queueing domain, lies the queues. They work as FIFOs. However, instead of inserting the new data at the back of the queue, the new data is always inserted as close as possible to the front of the queue. This mechanism helps avoiding gaps within the queues prevent the lost of few clock cycles that would be required to move the data from the back to the front. Form the authors experiments, saving clock cycles in SchIM is vital to keep the final bandwidth as high as possible.

[3]The modified Jailhouse sources are available at *URL OMITTED FOR BLIND REVIEW.*

Furthermore, the queues have been designed to deal with three constrains. Firstly, the queues store both read and write packets such that the order at which transactions arrived is guaranteed. This implies that all the queue slots have the same size regardless of whether they contain read or write packets. Secondly, due to the altered communication scheme (see Section V-B), each slot needs to be large enough to store both the address phase payload and the corresponding data of an AXI write transaction (678 bits). The depth of each queue is determined by considering the worst-case scenario. The latter consists in having to handle simultaneously the maximum number of outstanding read and write transactions. Our SchIM instance on the considered Xilinx UltraScale+ platform was configured with queues that are 16 slots in depth. Indeed, the HPM ports in this platform cannot handle more than 8 transactions of each type [**?**].

D. LLC-SchIM Interface and Traffic Accounting

As illustrated in Fig. 1, the considered system features an LLC shared between the four cores of the PS. For a non-cacheable read (resp., write) memory access, which CPU represents the source of the traffic is carried in the ID bits of the corresponding AR (resp., AW) AXI transaction. But for cacheable memory accesses, which is the norm for application workload, this is not the case. This is mainly because cache controllers typically use a write-back strategy. In this case, a read or write cache miss causes up to two events: (1) a cache refill and (2) a cache eviction. The cache refill is carried out with a read AXI transaction. If the line being evicted was previously written (dirty), then the eviction causes a write AXI transaction. It follows that, while read AXI transaction have an easily identifiable source, write transactions do not. Indeed, a CPU x might be causing the eviction of a line previously allocated and modified by CPU y. Hence, accounting (and scheduling) the resulting write transaction as if it originated from CPU x would be incorrect.

To ensure fair accounting for both read and write traffic, we rely on cache partitioning through coloring. As studied in a number of previous works, cache coloring is easy to implement at the hypervisor level [**?**], [**?**], [**?**]. In our system setup, we leverage the support Jailhouse already provides. The standard support has been extended to support booting a Linux inmate over colored memory. Cache partitioning allows us to establish a 1-to-1 relationship between any read/write transaction traversing the SchIM and the originating CPU. Moreover, with cache coloring in place, the SchIM uses the color bits in the address of the memory transactions (AR and AW channels) — instead of the AXI ID bits — to differentiate between the traffic of the various cores.

Finally, recall that the SchIM forwards transactions between HPM and HPS ports. These ports follow the asynchronous AXI protocol that allows issuing multiple outstanding AR and AW transactions. The protocol dictates that any outstanding transaction must have a unique AXI ID. This property is crucial to be able to match received responses with outstanding requests. Unfortunately, there might exist a mismatch between the bit-width of the AXI ID emitted at the HPM ports and the bit-width of AXI ID accepted by the HPS ports. For instance, in the platform of reference, the HPMs emit 16-bit AXI IDs, while the HPS AXI ID bit-width is 6 bits. Therefore, the SchIM also acts as an AXI ID translator.

E. Scheduling Interface and Implemented Policies

All the memory schedulers included in the scheduling domain share a common interface, to ease the integration of a new scheduler. In terms of input signals a generic scheduler module must define (1) a manual reset signal that can be triggered through the configuration port; (2) a vector of bits where each bit indicates whether the associated queue is empty; and (3) a signal indicating if the last scheduled transaction as been consumed. Alongside these inputs, the scheduling modules have also access to all the configuration registers listed in Table I. In terms of outputs a SchIM scheduler must define (1) a signal to the selector indicating the queue considered for scheduling; and (2) a signal stating whether the current scheduling decision is valid. We hereby review the initial set of memory scheduling policies implemented in the SchIM.

1) Fixed Priority: The FP scheduling module aims at enforcing strict prioritization of cores' memory traffic. The priority ordering is explicitly defined by the user through the configuration port. While the SchIM only has four queues, 16 different levels of priority are offered. The ordering must be strict, meaning that two cores cannot be assigned with the same priority.

The FP scheduling module only needs two pieces of information. That is (1) the priority associated with each queue and (2) whether a given queue contains at least one buffered transaction. The module logic always selects the queue with the highest priority. Lower priority queues are considered when higher priority queues do not have transactions. This is done by internally setting the user defined priority of a queue as 0 when the corresponding queue is empty.

2) Time Division Multiple Access: The TDMA memory scheduler is a non-work conserving policy that operates by defining a per-core time *slot* during which the core has exclusive access to main memory. The slots are expressed in PL clock cycles, to maximize granularity. The configuration port can be used to specify and change the slots specifications at runtime.

The implementation of the module uses a counter register to track the time elapsed in the current TDMA major frame — defined as the sum of all the cores' slots. It is reset to 0 at the beginning of a new major frame. Using the time-tracking register, the module determines to which core the current slot belongs, and forwards the information to the queue selector. This is done by summing up the length of all the previous slots, and determining if the current time falls within the interval of the considered core's slot.

3) Traffic Shaping (TS): The proposed TS transaction scheduling policy operates by defining a minimum inter-arrival time (MIT) that needs to elapse between any two consecutive transactions from the same CPU. A transaction that arrives before sufficient time has elapsed since the previous one is held by the SchIM until the aforementioned property is respected.

This scheduling module is implemented as follows. For each of the SchIM queues, the module defines a register counting the time elapsed since the last forwarded transaction. Once this counter has reached the period set by the user (through the configuration port), the module checks if the queue corresponding to the core contains any transaction. In the case where a transaction is available in the corresponding queue, the latter is forwarded to the output of SchIM (i.e., the serializer) and the counting register is reset to 0. Otherwise, the counting register is blocked to the desired period until a transaction is available for scheduling in the corresponding queue. Any tie between two cores is solved using a fixed priority arbitration defined by the user.

The described TS policy is similar to the ARM QoS regulation mechanism that is available at the level of interconnect for hardware accelerators [**?**], [**?**]. It is also similar, at least in principle, to software-based memory regulation techniques such as MemGuard [**?**]. Yet, TS operates differently from the hardware implementations of MemGuard-like regulation that have been proposed so far in [**?**], [**?**]. Indeed, our TS scheduler does not rely on memory budgets and replenishment periods. Instead, it provides memory bandwidth enforcement at the granularity of individual memory transactions. This also differentiates the TS policy in the SchIM from the *transaction supervisor* proposed in [**?**].

F. Programming Model

The parameters that compose the programming interface of the SchIM are summarized in Table I. The `base` address referenced in the table can be set when the SchIM is deployed in the PL. By default, this is set to `0x800000000`. All the parameter registers are 32 bit wide, except for the priorities of the FP scheduler. In this case, the priority values are encoded using 8 bits. The last "Mode" register allows a user to select the active memory scheduler.

TABLE I: Available SchIM configuration registers.

Parameter	Associated Core	Address
TDMA slots	C_0	`base+0x00`
	C_1	`base+0x04`
	C_2	`base+0x08`
	C_3	`base+0x0C`
User Thresholds	C_0	`base+0x10`
	C_1	`base+0x14`
	C_2	`base+0x18`
	C_3	`base+0x1C`
FP Priorities	C_0 | C_1 | C_2 | C_3	`base+0x20`
TS MITs	C_0	`base+0x24`
	C_1	`base+0x28`
	C_2	`base+0x2C`
	C_3	`base+0x30`
	Reserved	
Mode	N/A	`base+0x38`

G. PL-to-PS Feedback

Each of the HPM ports interfacing the PS and the PL sides (HPM1 and HPM2) have two dedicated queues for read and write transactions. Since transactions are being buffered inside SchIM as well as in these port buffers, head-of-line blocking can happen. Head-of-the-line blocking is harmful for performance; and can cancel out the benefits of transaction scheduling performed by the SchIM. For instance, in the case of a non work-conserving policy (e.g., TDMA), if the HPM port queue gets filled with transaction coming for the same core, no other transaction will be able to reach the SchIM and thus be considered for scheduling. This implies that no transaction would be scheduled until the end of the active core's TDMA slot. On the other hand, for work-conserving policies (e.g., FP) in the presence of head-of-line blocking, the decisions being taken by SchIM would directly depend on the order at which transactions are emitted by the HPM port buffer.

In both cases, one must prevent the cores from saturating the HPM port buffers. In order to avoid such situation, we implemented a feedback scheme aimed at slowing down the cores when necessary. As we mentioned in the context of Fig. 3, the SchIM's queues are associated a programmable threshold. Whenever the queue occupancy reaches (or exceeds) the associated threshold, a per-core interrupt line is asserted from the PL to the PS side. When received, the interrupt is treated by the platform software as a *fast interrupt request* (FIQ) and directly handled by the hypervisor—invisible to any guest OS. The advantage of using FIQs instead of regular IRQs is the significantly reduced handling latency [?]. Minor modifications to the TrustZone monitor were necessary to correctly configure FIQ handling. To minimize overhead, the installed FIQ handler only executes two assembly instructions. These are (1) a `dsb` memory barrier that stops the core until all the outstanding memory transactions have been completed, and (2) a `eret` instruction to exit the FIQ context. There is not need to save/restore any register because FIQs have banked syndrome/status registers and because no general purpose register is modified in the handler.

Ideally, the available space in the HPM buffers should be shared evenly between the cores. Since each HPM port has a buffer with a depth of 8+8 transactions, each core should occupy at most 2 slots in each buffer. Unfortunately, our experiments highlighted that the control over amount of transactions buffered by each core is imperfect. Often times, the selected threshold is exceeded by up to two transactions. This is the main reason why we propose a dual-ported SchIM which uses both the available HPM ports. Indeed, by assigning two cores on each of the ports, the ideal threshold on maximum amount buffered transactions can be doubled. The increase provides enough room to compensate for imperfections in the micro-regulation performed with PL-to-PS FIQ delivery.

VI. Evaluation

The present section aims at evaluating the behavior of the SchIM on the target platform, it overhead and benefits. First, in subsection VI-A, we review our experimental setup. Thereafter, we assess the overhead introduced by the SchIM in Section VI-B. In Section VI-C, an in-depth analysis of the SchIM behaviour is presented. Finally, a demonstration of the memory isolation enabled by the SchIM using real-world benchmarks is provided in Section VI-D.

A. Experimental Setup

The SchIM has been evaluated using synthetic benchmarks (or *Memory Bombs*), real benchmarks selected from the San Diego Vision Benchmark Suite (SD-VBS) [?] and a combinations of the two. Specifically, the 7 most memory-intensive benchmarks have been selected, i.e. *stitch*, *texture synthesis*, *disparity*, *tracking*, *localization*, *mser* and *sift*. For all our runs we have considered the VGA input size (640×480 pixels). When running any benchmark, we use the cache coloring mechanism provided by Jailhouse to partition the LLC evenly amongst the cores and to prevent our measurements to be affected by inter-core cache line eviction. As a result, each benchmark operates on 1/4 of the total cache space—256 KB.

To evaluate the capabilities of the SchIM, three memory routes for the traffic generated by the cores are compared. The first two serve as baselines, whereas, the last one is the one under analysis and involves the SchIM module. The first path consists in the cores directly accessing the main memory. As illustrated in Fig. 1, the traffic simply goes through the *Main Interconnect* before arriving at the DDR controller. This path is referred to as the *normal route*. In the second path, traffic is routed through the PL but without being subject to scheduling. In this case, a simple one-to-one connection between one of the HPM ports and the HPS port is programmed in the PL. No other manipulation on the traversing memory transactions is performed inside the PL; we refer to this case as the *simple loop-back*. Finally, we consider a third case where the SchIM module is deployed and in use to schedule memory traffic generated by the CPUs in the PL. Cores 0 and 1 target HPM1 aperture, while cores 2 and 3 target HPM2. In our analysis, SchIM is used in all the available modes, i.e., FP, TDMA and TS.

B. Platform Capabilities and performance degradation

Intuitively and as discussed in [?], redirecting the traffic coming from the cores to the PL side incurs a performance hit. In fact, the PL operates at a lower frequency compared to the PS. As such, larger memory latency and a smaller throughput should be expected when routing memory traffic through the PL. On the other hand, by redirecting and scheduling traffic in the PL, we can achieve higher system predictability. In order to weight up the pros and the cons of these two orthogonal objectives, we have computed the throughput of one *core under analysis*, here core 0 (noted C_0), for each of the aforementioned paths under all the possible levels of contention. The result of this experiment is display in Fig. 5.

From Fig. 5, one can observe that in general, the throughput experienced by the core under analysis (i.e., C_0) is high and that, regardless of the contention level on the normal route, the bandwidth remains high. As mentioned earlier, redirecting the cores' traffic through the PL side has a cost. In fact, in the case with no contention (the left-most bar cluster), the bandwidth experienced by C_0 when going through the simple loop-back,

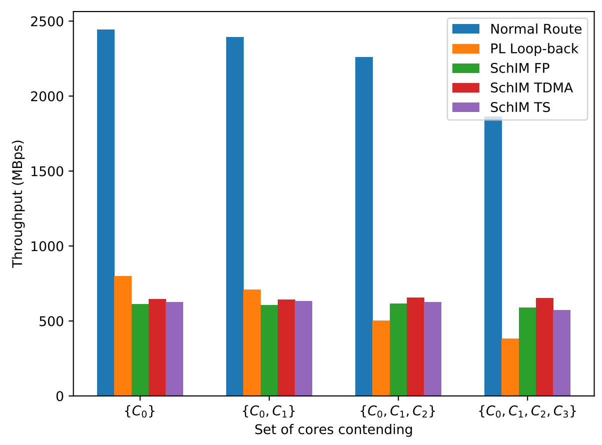


Fig. 5: Bandwidth in MBps for different path under increasing set of cores contending.

is reduced by around 75% compared to the normal route. Looking at the left-most bar cluster, one can also observe the overhead introduced by having the presence of the SchIM on the memory loop-back. The additional drop in bandwidth is approximately 185 MBps. While the throughput is larger in the case of the *simple loop-back*, it also drops linearly as soon as the contention level increases. On the other hand, the SchIM manages to preserve the same bandwidth, with negligible interference being observed in the most challenging scenario (the right-most bar cluster). In other words, the SchIM guarantees performance isolation between the cores with respect to the bus usage.

C. Internal Behaviour of SchIM

The next objective is to verify the correct behavior of our scheduling policies at the granularity of a clock cycle by observing the inputs, the outputs and the internal signals and registers of the SchIM module. This is made possible thanks to the *Integrated Logic Analyser* (or ILA) provided by Xilinx [**?**]. The latter IP can be directly implemented on the PL side, alongside the SchIM, and is able to probe the signals and to store them in a local memory. For this experiment, a group of relevant internal signals have been probed and captured during a window of 16384 contiguous clock cycles. Then, the information has been extracted by post-processing the data. To characterize the behavior of the three different policies, the ILA has been instrumented to collect (i) the amount of transactions being buffered in the queues at each clock cycle (inset 1 in Fig. 6a, Fig. 6b, and Fig. 6c), (ii) the rate at which queues receive new transactions from the cores cluster (inset 2 in Fig. 6a, Fig. 6b, and Fig. 6c), and (iii) the queues ID of each transaction forwarded by the SchIM module (inset 3 in Fig. 6a, Fig. 6b, and Fig. 6c).

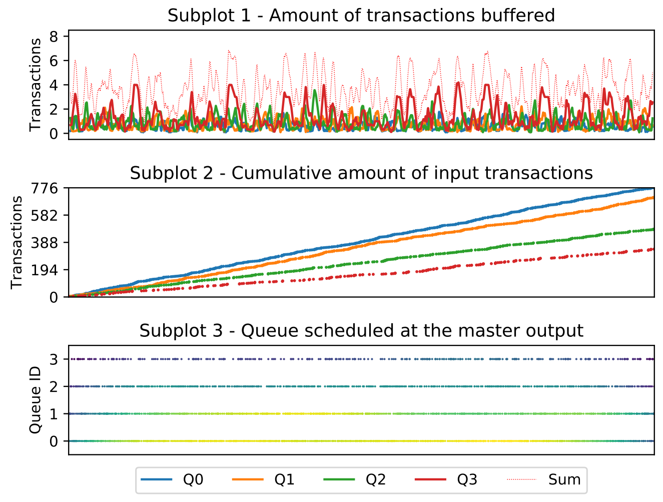
For the Fixed Priority trace snapshot displayed in Fig. 6a, the following strict priority ordering has been considered: $C_0 \succ C_1 \succ C_2 \succ C_3$ where the $\succ$ operator means that the left argument has a strictly higher priority than the right argument. In this experiment, a regulation threshold of 2 for each core has been used. As emphasized by the inset 2 in Fig. 6a, the FP scheduler is able to prioritize the traffic of one core at the expense of the others according to the priorities assignment. Furthermore, one can observe that the rate at which the queues receive new transactions from their associated core is proportional to the priority place in the priority ordering. Finally, the third inset in Fig. 6a confirms the correct behaviour of the FP policy. Thanks to the heat map, one can clearly see that the cores with the highest priority also feature the highest density of transactions at the output of the SchIM.

The trace snapshot displayed in Fig. 6b has been obtained by configuring the SchIM module in TDMA mode. For the sake of clarity, a slot of 512 clock cycles have been set for each core. In addition, the threshold of each core has been set to 1 to create sharp transitions. The insets 2 and 3 of Fig. 6b clearly show the behaviour expected from a TDMA schedule. In fact, one can clearly see in the latter that transactions originating from one core are only being repeated out of the SchIM module during a well defined 512 clock cycles time slot. Moreover, queues are scheduled one after the other in a periodic fashion. In the inset 2 of Fig. 6b, we can observe a similar pattern, with transactions arriving only during the TDMA slot associated to their queue (and indirectly core). Globally, the rate at which queues receive transactions is steady and constant. The variations can be explained by different factors such as the coarseness of the FIQ feedback regulation and the scheduling happening in the OS.

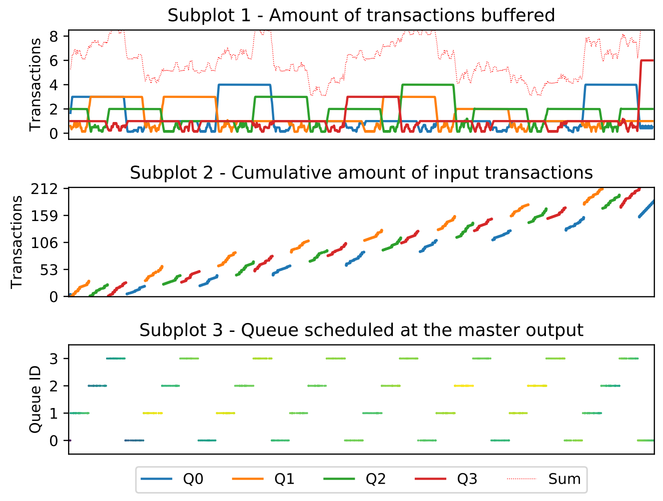
The trace snapshot for the TS policy has been obtained with the minimal inter-arrival time set to 256 clock cycles for all the cores. Similarly to the TDMA experiment, such period has been set arbitrarily in order to improve the clarity of the trace snapshot displayed in Fig. 6c. Thanks to the inset 3 in Fig. 6c, we can see that under a significant traffic load, the TS mode of the SchIM is able to shape the output traffic. Roughly, the same pattern can be observed in inset 2 of Fig. 6c. However, there are two exceptions. First, one queue can receive more than one transaction. This is due to the coarseness of the FIQ feedback regulation. Secondly, some queues seem to be prioritized. This can be explained by the fact that in the TS scheduler implementation break ties between valid ready transactions using a FP policy. In this experiment, since the SchIM module is constantly under pressure, there are always ties between the queues and FP is frequently applied.

D. Memory Isolation

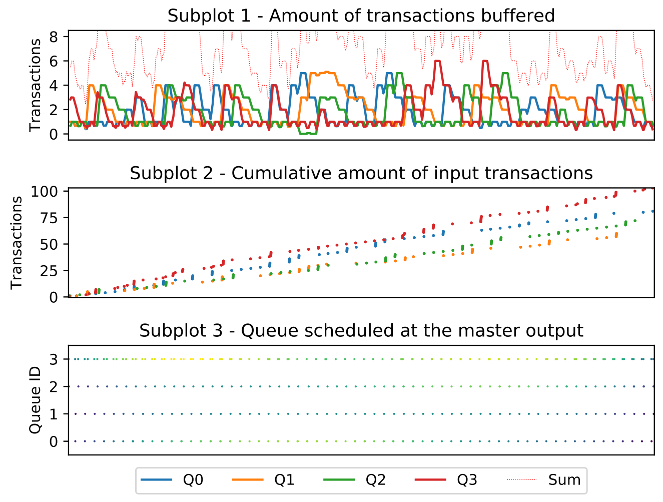
To evaluate the capability of SchIM with respect to its ability to ensure performance isolation between the cores, a set of experiments involving SD-VBS benchmarks were designed. Here, we compare the execution time of an application on a given core when running alongside interfering synthetic benchmarks (memory bombs) on all the other cores. The slowdown compared to the case in which the observed benchmark runs alone in the system is computed. It follows that a ratio of 1 denotes the ideal isolation. The results obtained on the



(a) FP with ordering $C_0 \succ C_1 \succ C_2 \succ C_3$



(b) TDMA with slots of 512 clock cycles



(c) TS with min. period of 256 clock cycles

Fig. 6: Trace snapshots of SchIM for FP (6a), TDMA (6b) and TS (6c)

TABLE II: Inter-core Interference Ratios

Benchmark	Memory Route & Scheduler			
	Loop-back	SchIM TDMA	SchIM TS	SchIM FP
stitch	1.14	1.06	0.98	1.16
texture synth.	1.03	1.02	1.0	1.03
disparity	1.46	1.07	0.96	1.46
tracking	1.13	1.05	0.97	1.13
localization	1.02	1.01	1.0	1.02
mser	1.59	1.06	0.98	1.56
sift	1.14	1.04	0.97	1.14

considered benchmarks are listed in Table II. All the results in the table are the aggregation (geometric average) of 100 different runs in the same configuration.

The simple loop-back path is used as a baseline for this experiment because scheduling is performed in this configuration. The results for the simple loop-back are displayed in the left-most column in Table II. We highlight the sensitivity of both *disparity* and *mser* to inter-core interference, with ratios of respectively 1.46 and 1.59. On the other hand, *texture synthesis* and *localization* do not seem to suffer significantly from inter-core interference. In comparison, the TDMA scheduler manages to guarantee isolation of the core under analysis. In fact, the ratio of all the benchmarks we tested lie in the proximity of 1.0, with a maximum inter-core interference of around 7%. Similarly, the TS scheduler is also capable of ensuring a sound isolation of the core under analysis. Unfortunately, the FP scheduler is unable to guarantee the isolation of the core under analysis despite having been assigned the highest priority. Even worst, the scheduler has little-to-none impact since its slowdown is comparable to that observed on the loop-back path. This result can be explained by the fact that, despite the enforcement of the FP policy, and its proven correct behaviour (see Fig. 6a), the enforced ordering of transaction is likely overridden by the transaction re-ordering mechanisms present in the DDR controller. The problem specifically occurs because the access pattern of the interfering bombs is sequential, and hence preferably treated by the DDR controller. Conversely, the FP scheduler works correctly in scenarios like those in Fig. 6a because the memory access pattern of the core under analysis is sequential as well. It follows that the FP policy is mainly useful when performing the movement of large, sequential data chunks.

VII. Discussion

When comparing the throughput that is experienced by the cores, the normal route always provides a higher throughput in comparison to the the loop-back based path. However, when only comparing those loop-back based paths, we can see that the price to pay for having SchIM is largely compensated by the memory isolation the latter provides. In general, redirecting the cores traffic toward the PL side with SchIM is interesting in combination with techniques such as *address bleaching* and *zero-copy recoloring*.

The PL-to-PS feedback is an interesting regulating mechanism. Moreover, it is totally transparent to the hypervisor's inmates and uses the fastest route possible to communicate with the PS side. Nonetheless, this feedback mechanism is coarse. The subfigure 1 of Figure 6b highlights this problem. In fact, even though all the queues have been assigned a threshold of 1, this threshold is often exceeded. The worst case being queue 3 exceeding the threshold by 5 in the right side of the plot.

The thresholds used for the FIQ regulation require to be fine tuned manually by the user. Future extension of the SchIM should include schedulers capable of dynamically adapting the thresholdd in order to maximize the performance and improve the cores isolation.

VIII. Conclusion

In the present article we introduced the SchIM, a memory transactions scheduler framework that can be integrated in commercially available platforms featuring a tightly coupled processing system and programmable logic. A full-system implementation in a commercially available PS-PL platform has been detailed, which encompasses the accompanying software stack and the platform-specific integration steps has been detailed in as well as advanced scheduling techniques are few of many possible future directions.

Through a set of experiments, we assessed the capabilities of the framework and demonstrated the correct behaviour of the proposed scheduling policies, namely Fixed Priority, Time Division Multiple Access and Traffic Shaping. Finally, we showed using a suite of real-world benchmarks that the SchIM is capable of enforcing strong temporal isolation in spite of heavy memory contention.

The authors see the proposed SchIM as a stepping stone to propose, test, and validate novel memory scheudling policy to be tested on embedded platforms with realistic performance and complex workload. For this reason, the SchIM has been designed to be open-source and with extensibility in mind. Especially, we strongly envision that the SchIM could represent a stepping-stone toward profile-based memory traffic scheduling.