TOP: Towards Open & Predictable Heterogeneous SoCs

Luca Valente, Francesco Restuccia, Davide Rossi, *Member, IEEE*
Ryan Kastner, *Fellow, IEEE* Luca Benini, *Fellow, IEEE*

Abstract—Ensuring predictability in modern real-time Systems-on-Chip (SoCs) is an increasingly critical concern for many application domains such as automotive, robotics, and industrial automation. An effective approach involves the modeling and development of hardware components, such as interconnects and shared memory resources, to evaluate or enforce their deterministic behavior. Unfortunately, these IPs are often closed-source, and these studies are limited to the single modules that must later be integrated with third-party IPs in more complex SoCs, hindering the precision and scope of modeling and compromising the overall predictability. With the coming-of-age of open-source instruction set architectures (RISC-V) and hardware, major opportunities for changing this status quo are emerging. This study introduces an innovative methodology for modeling and analyzing State-of-the-Art (SoA) open-source SoCs for low-power cyber-physical systems. Our approach models and analyzes the entire set of open-source IPs within these SoCs and then provides a comprehensive analysis of the entire architecture. We validate this methodology on a sample heterogenous low-power RISC-V architecture through RTL simulation and FPGA implementation, minimizing pessimism in bounding the service time of transactions crossing the architecture between 28% and 1%, which is considerably lower when compared to similar SoA works.

Index Terms—Heterogeneous SoC, Cyber-Physical-Systems, Timing Predictable Architectures, Open-Source Hardware.

1 Introduction

The exponential growth of cyber-physical systems (CPS) (e.g., self-driving cars, autonomous robots, ...) and related applications has been fueled by the increase in computational capabilities of heterogeneous low-power Systems-on-Chip (SoCs). These SoCs are complex computing platforms composed of a set of different hardware computing units (e.g., CPUs, hardware accelerators), each tailored to a specific target application, sharing a set of resources (memory, sensors) through interconnects [1]–[5]. While integrating multiple computing units on the same platform has enabled efficient scale-up of computational capabilities, it also poses significant challenges when it comes to assessing their *timing predictability*, which is a requirement for CPSs dealing with real-time and safety-critical applications: the primary challenge arises from resource contentions that emerge when multiple active agents within the SoC must access the same shared resources [1]–[7].

Numerous research efforts have focused on enhancing the timing predictability of heterogeneous Systems-on-Chip (SoCs). This includes safely upper bounding execution times for data transfers [8]–[10] or the deadline miss ratio for critical tasks [1]–[3], with the smallest possible pessimism. These efforts have predominantly focused on modeling and analyzing commercial DDR protocols [8], memory IPs [11], and memory controllers [12], but also predictable interconnects [1], [4] and on-chip communication protocols [13]. Regrettably, despite their value, these studies are scattered, with each one focusing on only one of these resources at a time, resulting in being overly pessimistic [5].

Modeling and analysis of communication protocols are done speculatively on abstract models, thus reducing their real-world applicability. Recent works for modeling and analysis of IPs (memories, memory controllers, interconnect, etc.) have to address the unavailability of cycle-accurate RTL descriptions. Many of these IPs are either entirely closed-source [8] or provide loosely-timed behavioral models [5], [12] or just μarchitectural descriptions [1], [3], [4]. In essence, the fragmented and proprietary nature of commercial and research IPs restricts studies to the particular IP, greatly reducing the accuracy achievable through system-level analysis. For example, Restuccia et al. in [9] bound the access times of multiple initiators on FPGA reading and writing from/to the shared DDR memory. The proposed upper bounds' pessimism is between 50% and 90%: even though they finely modeled and analyzed the proprietary interconnect, the authors did not have access to its RTL nor to the memory controller and IP. The same applies to Ditty [10], which is a predictable cache coherence mechanism. In Ditty, even though the caches' timing is finely modeled, the overall execution time can be up to 3× bigger than the theoretical upper bounds, as the authors did not model other components. Another example is AXI-IC^{RT} [1], an advanced AXI interconnect with a sophisticated scheduler which allows transaction prioritization based on importance. While proposing a highly advanced interconnect with a

- *Luca Valente, Luca Benini, and Davide Rossi are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy. Luca Benini is also with the Integrated Systems Laboratory (IIS), ETH Zürich, 8092 Zürich, Switzerland.*
- *Francesco Restuccia and Ryan Kastner are with the Computer Science and Engineering, University of California at San Diego, San Diego, CA 92093 USA.*

This work was supported by Technology Innovation Institute, Secure Systems Research Center, Abu Dhabi, UAE, PO Box: 9639, by the Spoke 1 on Future HPC of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 - Next Generation EU, and by the European Project EuroHPC JU The European Pilot (g.a. 101034126), and by KDT TRISTAN project (g.a.101095947).

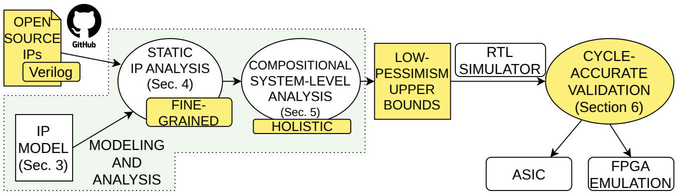
OPEN SOURCE IPs
GitHub
Verilog
STATIC IP ANALYSIS (Sec. 4)
FINE-GRAINED
COMPOSITIONAL SYSTEM-LEVEL ANALYSIS (Sec. 5)
HOLISTIC
LOW-PESSIMISM UPPER BOUNDS
RTL SIMULATOR
CYCLE-ACCURATE VALIDATION (Section 6)
IP MODEL (Sec. 3)
MODELING AND ANALYSIS
ASIC
FPGA EMULATION

Fig. 1: Proposed methodology.

tightly coupled model, the authors do not extend the model to the other components of the SoC, even when assessing the deadline miss ratio and benchmarking the architecture.

The emergence of open-source hardware creates a major opportunity for building accurate end-to-end models for real-time analysis of cutting-edge heterogeneous low-power SoCs [14]–[16]: the openness of the IPs allows for cycle-accurate analysis of the whole architecture from the interconnects to the shared resources. Yet, investigations and successful demonstrations in this direction are still scarce, primarily because open hardware has only very recently reached the maturity and completeness levels required to build full heterogeneous SoCs [17]. In this context, this is the first work to bridge the gap between open-source hardware and timing analysis, demonstrating a methodology that successfully exploits the availability of the source code to provide fine-grained upper bounds of the system-level data transfers. We leverage a set of open-source IPs from the PULP family, one of the most popular open-hardware platforms proposed by the research community [14], [18].

Figure 1 shows the proposed methodology, highlighting the novel contributions in yellow. It consists of (i) a model for standalone IPs composing modern heterogeneous low-power SoCs, (ii) a static analysis of the RTL code of such components, and (iii) a compositional mathematical analysis of the whole system to upper bound the response time of the interactions between managers (initiators) and shared subordinates (targets), considering the maximum interference generated by the interfering managers. Figure 1 highlights the differences with previous studies also based on a static and compositional approach [5], [7], [9]. Previous works typically focus on one IP at a time [9], or rely on loosely-timed models [5] or high-level hardware documentation [8]. On the contrary, our approach leverages the RTL source code to build a precise and detailed description of the hardware components and leverage it to derive an accurate and holistic system-level analysis. This limits the proposed upper bounds' pessimism between 28% and just 1%, in isolation and under interference, which is considerably lower when compared to similar SoA works for closed-source or loosely-timed platforms [1]–[4], [8], [10], as better detailed in Section 7. We demonstrate our methodology on an open-source prototype of a heterogeneous low-power SoC for embedded systems composed of a Linux-capable host core, a parallel accelerator, a set of IOs, and on-chip and off-chip memories.

The manuscript is organized as follows: Section 2 presents the target open-source RISC-V-based SoC architecture, and Section 3 discusses the model we apply to its different components. Section 4 analyzes the components to specialize the generic model to each of them, and Section 5 provides the system-level analysis of the architecture. Finally, Section 6 validates the results with cycle-accurate experiments (on simulation and FPGA), Section 7 compares this work with the SoA. Section 8 concludes the manuscript.

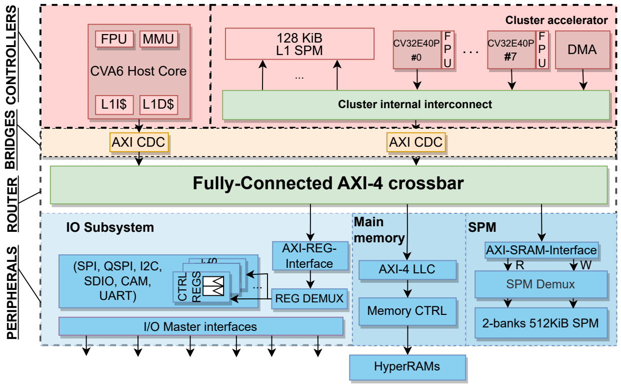

Fig. 2: Sample architecture.

2 Architecture

Fig. 2 shows the architectural template we target and the four classes of hardware modules we identify in the architecture under analysis, namely (i) *controllers*, (ii) the main *crossbar*, (iii) *bridges*, and (iv) *peripherals*, modeled in the next Section. The architecture leverages a set of fully open-source PULP IPs [18]. It is based on Cheshire [15], an open-source host platform consisting of an RV64 Linux-capable CPU, a set of commodity IOs (SPI, UART, ...), and an AXI-based crossbar with a configurable number of subordinate and manager ports for easy integration of accelerators and resources. Our platform includes a parallel accelerator and a low-power lightweight HyperBUS memory controller [19].

The host CPU is CVA6 [20], which is a six stages, single-issue, in-order, 64-bit Linux-capable RISC-V core, supporting the RV64GC ISA variant, SV39 virtual memory with a dedicated Memory Management Unit (MMU), three levels of privilege (Machine, Supervisor, User), and PMP [21]. CVA6 features private L1 instruction and caches, operating in parallel, with the latter being able to issue multiple transactions. When needed, CVA6 can offload computation-intensive tasks to the parallel hardware accelerator, the so-called PULP cluster [22]. It is built around 8 CV32E4-based cores [23] sharing 16×8 kB SRAM banks, composing a 128 kB L1 Scratchpad Memory (SPM). The cluster features a DMA to perform data transfers between the private L1SPM and the main memory: data movement is performed via software-programmed DMA transfers. Once the data are available inside the L1SPM, the accelerator starts the computation.

CVA6 and the cluster are the managers of the systems connected to the main AXI crossbar [24], which routes their requests to the desired subordinates according to the memory map. A manager can access any subordinate in the system. The main subordinates of the systems are, respectively, (i) the on-chip SRAM memory, (ii) the IO subsystem, and (iii) the off-chip main memory with a tightly coupled Last Level Cache (LLC). The on-chip memory is used for low-latency, high-bandwidth data storage. The APB subsystem is used to communicate with off-chip sensors or memories through the commodity IOs. The off-chip main memory is where the code and the shared data are stored. Differently from high-end embedded systems relying on relatively power-hungry and expensive DDR3/4/5 memories, the platform under analysis adopts HyperRAMs as off-chip main memory, which are fully-digital low-power small-area DRAMs with less than 14 IO pins and that provide enough capacity to boot Linux [16] and bandwidth for IoT applications [19], [25].

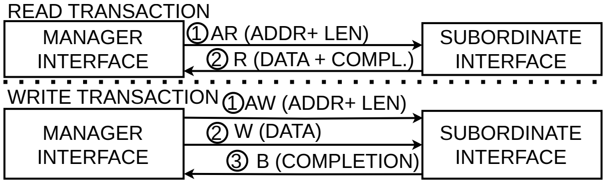

Fig. 3: AXI Channel architecture

3 Model

This section presents the model we construct for the different components of our SoC. Our aim is to propose a general model that describes the characteristics of the components and that can be re-targeted to different IPs and novel architectures, regardless of the number of integrated controllers and peripherals. This work is also an effort to provide base support to stimulate further studies in predictability improvements and analysis for open hardware architectures.

3.1 Communication model

We identify four classes of hardware modules in the architecture under analysis, shown in Fig. 2, namely (i) *controllers*, (ii) the main *crossbar*, (iii) *bridges*, and (iv) *peripherals*. As the AXI standard is the main communication standard used to implement non-coherent on-chip communications [24], we discuss here its main features. It defines a manager-subordinate interface enabling simultaneous, bi-directional data exchange and multiple outstanding transactions. Fig. 3 shows the AXI channel architecture and information flow. Bus transactions are initiated by a *controller* (exporting a manager interface), submitting a transaction request to read/write data to/from a subordinate interface through AR or AW channels, respectively. A request describes the starting target address and a *burst length*. After the request phase, in case of a read, data are transmitted through the R channel. In case of a write, data are provided by the *controller* to the target *peripheral* through the W channel. Upon completing a write transaction, the *peripheral* also sends a beat on the B channel to acknowledge the transaction's completion. For multiple in-flight write transactions, the standard enforces strict in-order access to the W channel: the data on the W channel must be propagated in the same order as the AW channel requests. Even though the standard does not require it, many commercial and open-source platforms apply the same policy for reads, typically to limit the system's overall complexity, as reported in their documentation [26], [27].

3.2 Controller model

Controllers have an active role on the bus. Each *controller* exports an AXI manager interface, through which it initiates requests for bus transactions directed to the *peripherals*. A generic *controller* C_i can be described through two parameters: the maximum number of outstanding read/write transactions that it can issue in parallel, denoted with $\phi^{C_i}_{R/W}$, and their relative burst length β_i. While our model and analysis can be applied to a generic architecture, the system under analysis features as *controllers* a CVA6 core [20] and a cluster accelerator [22] (see Section 2). Bus transactions issued by the cluster interfere with those issued by CVA6 and vice-versa. CVA6 is assumed to compute a critical periodic workload, running on top of a Real-time Operating System (RTOS). The PULP cluster executes computation-intensive tasks and issues bus transactions through its DMA. Contention internal to the PULP cluster has been profiled in detail in [28]. However, our analysis provides the worst-case data transfer time in accessing the shared *peripherals* to support the safe scheduling and execution of critical tasks within their deadline. We specifically focus on interference in accessing the shared resources. Modeling the internal effects of *controllers*, such as pipeline stalls in the core or contention within the accelerator, is beyond the scope of this work.

3.3 Peripheral model

Peripherals export a *subordinate* interface through which they receive and serve the bus transactions. The *peripherals* deployed in the system are heterogeneous. Nonetheless, our model offers a set of parameters representative of a generic peripheral, and it is not tied to a specific communication protocol. It works as the baseline for the analysis of any *peripheral* deployed in the system under analysis. The generic *peripheral* P_j is characterized with two sets of parameters: (i) the maximum number of supported outstanding reads ($\chi^{P_j}_R$) and write ($\chi^{P_j}_W$) transactions; (ii) the maximum number of cycles incurred from the reception of the request to its completion, for a read ($d^{P_j}_R$) and a write ($d^{P_j}_W$) transaction in isolation. $d^{P_j}_R$ and $d^{P_j}_W$ are composed of two contributions: (i) the *data time*, defined as the time required for the *peripheral* to send or receive one word of data (t_{DATA}) multiplied by the burst length of the transaction in service (β_i) and (ii) the *control overhead* t_{CTRL}, defined as the maximum time elapsing between accepting the request and the availability of the first word of data (reads) or availability to receive data (writes). From the previous considerations, $d^{P_j}_{R/W} = t^{P_j}_{\text{CTRL}} + t^{P_j}_{\text{DATA}} \cdot \beta$. We define two extra parameters ρ^{P_j} and θ^{P_j}. The first indicates the level of pipelining in serving multiple transactions. $\rho^{P_j} = 1$ means that each stage of P_j does not stall the previous, and transactions are served in a pipelined fashion, while $\rho^{P_j} = 0$ indicates that no pipeline is implemented. $\theta^{P_j} = 0$ indicates that read and write transactions interfere with each other. $\theta^{P_j} = 1$ indicates that read and write transactions can be handled in parallel by P_j.

3.4 Main crossbar model

We provide here the model of the main *crossbar*, the routing component enabling communication among *controller*s and *peripheral*s. Each *controller* has its manager port connected to a subordinate port of the *crossbar*. Each *peripheral* has its subordinate port connected to a manager port of the *crossbar*. We model the *crossbar* R_0 with two sets of parameters: (i) the maximum amount of outstanding read and write transactions that a subordinate port can accept ($\chi_R^{R_0}$ and $\chi_W^{R_0}$, respectively); and (ii) the maximum overall latency introduced by R_0 on each read ($d_R^{R_0}$) and write transaction ($d_W^{R_0}$). $d_R^{R_0}$ and $d_W^{R_0}$ are composed of two contributions: (i) the overall delay introduced by the *crossbar* on a transaction in isolation (t_{PROP}); (ii) the maximum time a request is delayed at the arbitration stage due to the contention generated by interfering transactions ($t_{\text{CON}}^{R_0}$). From the previous considerations, the propagation latency is modeled as $d_{R/W}^{R_0} = t_{\text{PROP}}^{R_0} + t_{\text{CON}}^{R_0}$. Such parameters depend on the arbitration policies and routing mechanisms, as we investigate in detail in Section 4.

3.5 Bridge model

Bridges export a single manager interface and a single subordinate interface. They perform protocol/clock conversion between a *controller* and the *crossbar*. Bridges require a certain number of clock cycles to be crossed but do not limit the number of in-flight transactions and do not create any contention. We model the bridges with two parameters: the overall maximum delay introduced over a whole transaction for (a) read ($d_R^{Q_j}$) and (b) write ($d_W^{Q_j}$) transactions.

4 Analysis of the hardware modules

This Section analyzes the worst-case behavior of the *peripherals*, *bridges*, and the *crossbar* present in the platform under analysis through careful evaluation of the RTL code, aided by cycle-accurate simulation. Our approach is compositional: in this Section, we derive the IP micro-architecture from the corresponding RTL code and bound the service times at the IP level in isolation, according to the model introduced in Section 3. In Section 5, we provide a worst-case analysis at the system level, in isolation and under interference. We define $t_{\text{CK}}^{P_j}$ as the period of the clock fed to P_j.

4.1 AXI CDC FIFO queues

AXI CDC FIFOs are leveraged to perform clock-domain crossing between two AXI-based devices. The generic AXI CDC FIFO F_i is a *bridge*: we apply here the model presented in Section 3.5. It exports a manager interface and a subordinate interface. It is composed of five independent CDC FIFOs, each serving as a buffer for an AXI channel, having depth D_{CDC}^i (design parameter for the IP under analysis).

4.1.1 RTL IP structure

Figure 4 shows the block diagram of a CDC FIFO in the platform under analysis. They are structured following established clock domain crossing (CDC) principles [24]. The design is split into two parts, the transmitter (TX) and the receiver (RX), having different clock domains. TX and RX interface through asynchronous signals, namely a counter for data synchronization (synchronized with two-stage Flip-Flops (FFs)) and the payload data signal.

4.1.2 Delays analysis

As mentioned earlier, CDC FIFOs are *bridges*: we apply the model presented in Section 3.5. The CDC FIFO under analysis behaves as follows: TX samples the payload data into an FF. In the following cycle, the TX counter is updated. The TX counter value gets then through two synchronizations FFs – the updated pointer value is observed by the RX after two clock cycles. At that point, RX samples the data in one clock cycle to then propagate it in the following one. It follows that crossing the CDC FIFO introduces a fixed delay of one clock cycle of the TX domain ($t_{\text{CK}}^{\text{TX}}$) and four clock cycles of the RX domain ($t_{\text{CK}}^{\text{RX}}$). This means that the delay in crossing the CDC FIFO is equal to $t_{\text{CDC}}(t_{\text{CK}}^{\text{TX}}, t_{\text{CK}}^{\text{RX}}) = t_{\text{CK}}^{\text{TX}} + 4 \cdot t_{\text{CK}}^{\text{RX}}$.We leverage this baseline delay to build the overall latency introduced by F_i, interposed between a manager (clocked at t_{CK}^{C}) and a subordinate (clocked at t_{CK}^{P}).

Read transaction: A read transaction AR_k is composed of two phases: (i) the address propagation phase and (ii) the data phase. This means that F_i is crossed twice to complete AR_k: during phase (i), the manager is on the TX side, propagating the request. In phase (ii), the subordinate is on the TX side, propagating the data. Hence, the propagation latency is $t_{\text{CDC}}(t_{\text{CK}}^{C}, t_{\text{CK}}^{P})$ in phase (i) and $t_{\text{CDC}}(t_{\text{CK}}^{P}, t_{\text{CK}}^{C})$ in phase (ii). Adding them together, the propagation latency introduced by F_i on AR_k is equal to:

$$d_R^{\text{CDC}} = t_{\text{CDC}}(t_{\text{CK}}^{C}, t_{\text{CK}}^{P}) + t_{\text{CDC}}(t_{\text{CK}}^{P}, t_{\text{CK}}^{C}) = 5(t_{\text{CK}}^{C} + t_{\text{CK}}^{P}) \quad (1)$$

Write transaction: A write transaction is composed of three phases: (i) an address phase (manager on the TX side), (ii) a data phase (manager on the TX side), and (iii) a write response phase (subordinate on the TX side). Phases (i) and (ii) happen in parallel (see [29] p. 45). Thus, $t_{\text{CDC}}(t_{\text{CK}}^{C}, t_{\text{CK}}^{P})$ is incurred for phases (i) and (ii), and $t_{\text{CDC}}(t_{\text{CK}}^{P}, t_{\text{CK}}^{C})$ for phase (iii). The delay introduced by F_i on AW_k is equal to the delay introduced in Equation 1, $d_W^{\text{CDC}} = d_R^{\text{CDC}}$.

4.2 AXI SRAM scratchpad memory (SPM)

The AXI SPM is a high-speed, low-latency memory component used for temporary data storage – a block design representation is reported in Figure 5. The SPM memory is a *peripheral*: we apply here the model presented in Section 3.3.

4.2.1 RTL IP structure

The first stage of the SPM architecture is represented by a protocol converter (AXI-SRAM-Interface), translating the read and write AXI channels into SRAM-compatible transactions. Following the converter, an internal demux directs the SRAM transactions to the desired SRAM bank, where the data is stored. Each SRAM bank provides two independent SRAM ports, one for reads and one for writes, as from the specification of industry-standard SRAM resources [30].

The AXI-SRAM-Interface is structured in two submodules, independently managing read and write transactions. The first stage of each submodule is a FIFO queue (of depth $D_{\text{FIFO}}^{\text{SPM}}$) buffering the AXI AW or AR channel, respectively. Each submodule features the logic for protocol translation, consisting of (i) saving transaction metadata (starting address and length) and (ii) producing the output SRAM requests. For writes, the incoming data on the W channel are directly propagated towards the banks. The logic operating the protocol conversion generates the address for each W beat. For reads, the data coming from the SRAM banks are directly driven on the R channel. The logic keeps compliance with the AXI standard, adding the last signal or generating write responses when required. *The demux* is fully combinatorial

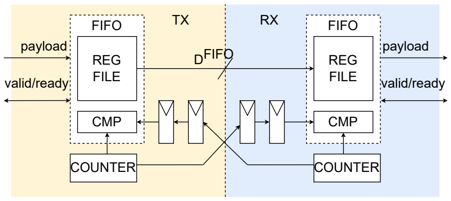

Fig. 4: CDC FIFO block diagram.

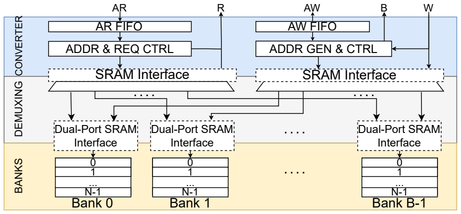

Fig. 5: AXI SPM block diagram

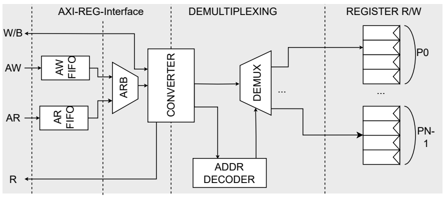

Fig. 6: IO subsystem block diagram.

and selects the target bank according to the request's address. *The SRAM banks* are technology-specific macros instantiated at design time. Each SRAM bank's port exports an enable signal, an address signal, and a signal to determine if a transaction is a read or a write. The SRAM interface expects simultaneous propagation of data and commands for writes; for reads, the data are sent the cycle following the command.

4.2.2 Delays and parallelism analysis

AXI-SRAM-Interface: the FIFOs in the converter are only in charge of data buffering – each FIFO introduces a fixed delay of one clock cycle ($t^{\text{SPM}}_{\text{CK}}$). After the FIFOs, the control logic requires at most one clock cycle ($t^{\text{SPM}}_{\text{CK}}$) to set up the propagation of a burst transaction – the direct connection over the W and R channels makes the data streaming in a pipeline fashion, adding no further latency. At the end of a write transaction, the converter takes two clock cycles ($2t^{\text{SPM}}_{\text{CK}}$) to generate the write response: one to acknowledge that the last W beat has been accepted and one to provide the B response. The same applies to reads, to generate the AXI last signal. Summing up the contributions, the control latency introduced by the AXI-SRAM-Interface to each transaction is upper bound by $4t^{\text{SPM}}_{\text{CK}}$ for both reads and writes.

Demux: The demultiplexing is combinatorial: it connects the transaction to the SRAM bank in one clock cycle ($t^{\text{SPM}}_{\text{CK}}$).

Banks: As by the definition of the SRAM interface [30], an SRAM bank serves one transaction per clock cycle, which makes $t^{\text{SPM}}_{\text{DATA,R/W}} = t^{\text{SPM}}_{\text{CK}}$. For write transactions, the protocol guarantees that the SRAM bank samples the data in parallel with the request (in the same clock cycle). For read transactions, the data are served the clock cycle after the bank samples the request. So, it contributes to $t^{\text{SPM}}_{\text{CTRL,R}}$ with one clock cycle ($t^{\text{SPM}}_{\text{CK}}$). Summing up the contributions, the service time of the SPM in isolation is upper bound by:

$$t^{\text{SPM}}_{\text{CTRL,W}} = 5 \cdot t^{\text{SPM}}_{\text{CK}}; t^{\text{SPM}}_{\text{CTRL,R}} = 6 \cdot t^{\text{SPM}}_{\text{CK}}; t^{\text{SPM}}_{\text{DATA,R/W}} = t^{\text{SPM}}_{\text{CK}}; \tag{2}$$

Consider now the parallelism supported by the SPM. The maximum number of accepted outstanding transactions at the SPM χ^{SPM}_{R} is defined by the depth $D^{\text{SPM}}_{\text{FIFO}}$ of the input buffers implemented in the AXI-SRAM-Interface. Thus,

$$\chi^{\text{SPM}}_{R} = \chi^{\text{SPM}}_{W} = D^{\text{SPM}}_{\text{FIFO}} \tag{3}$$

The *SPM* module under analysis is aggressively pipelined, operations are executed in one clock cycle, and no stall sources are present in the design. Also, as mentioned earlier, read and write transactions do not interfere with each other. From the previous considerations, $\rho^{\text{SPM}} = 1$ and $\theta^{\text{SPM}} = 1$.

4.3 IO Subsystem

The IO subsystem is the *peripheral* in charge of writing/reading data to/from the off-chip I/Os. We apply here the model presented in Section 3.3. It is composed of a set of memory-mapped peripheral registers that are accessed through a demux and that manage the datapaths issuing the transactions on the I/O interfaces (e.g., SPI, I2C, etc.).

4.3.1 RTL IP structure

Figure 6 shows the block diagram of the IO subsystem. It is composed of an AXI-REG-Interface, a demux, and a set of registers. The first stage of the *AXI-REG-Interface* is composed of two FIFOs (of depth $D^{\text{IO}}_{\text{FIFO}}$), buffering read and write transactions, respectively. After the FIFOs, a round-robin arbiter manages read and write transactions, allowing only one at a time to pass to the protocol conversion. Since the IO subsystem is meant for low-power reads and writes, registers' transactions share the same set of signals for reads and writes and are limited to single-word accesses. For such a reason, the IO subsystem does not support burst transactions (requests having $\beta_i > 1$ are suppressed). *The demux* stage decodes the request and directs it to the proper register destination, where it is finally served as a register read or write.

4.3.2 Delays and parallelism analysis

The IO subsystem is a *peripheral*, thus, we apply the model proposed in Section 3.5. Considering the maximum service delays, overall, the IO subsystem is composed of four stages: (i) the FIFOs, (ii) the protocol conversion, (iii) demultiplexing, and (iv) target register access. The first three stages, contributing to the control overhead, introduce a fixed delay of one clock cycle ($t^{\text{IO}}_{\text{CK}}$) each for a total of $3 \cdot t^{\text{IO}}_{\text{CK}}$ clock cycles. Consider now stage (iv). In the case of a write, the request and the corresponding data are propagated in parallel in one clock cycle. In the case of a read, the register provides the data in the clock cycle following the request – $t^{\text{IO}}_{\text{CTRL}}$ requires one extra clock cycle. Summing all the contributions, the service time of the I/O subsystem is upper bounded by:

$$t^{IO}_{\text{CTRL,W}} = 3 \cdot t^{\text{IO}}_{\text{CK}}; \quad t^{IO}_{\text{CTRL,R}} = 4 \cdot t^{\text{IO}}_{\text{CK}}; \quad t^{IO}_{\text{DATA,W/R}} = t^{\text{IO}}_{\text{CK}} \tag{4}$$

Consider now the parallelism. Similarly to the SPM module, the IO subsystem is capable of buffering up to $D^{\text{IO}}_{\text{FIFO}}$ of each type in its input FIFO queues. Thus, the maximum number of outstanding transactions supported by the IO subsystem is equal to:

$$\chi^{\text{IO}}_{W} = \chi^{\text{IO}}_{R} = D^{\text{IO}}_{\text{FIFO}} \tag{5}$$

The IO subsystem serves read and write transactions one at a time, and no pipelining is implemented among the different stages. This means that $\rho^{\text{IO}} = 0$ and $\theta^{\text{SPM}} = 0$.

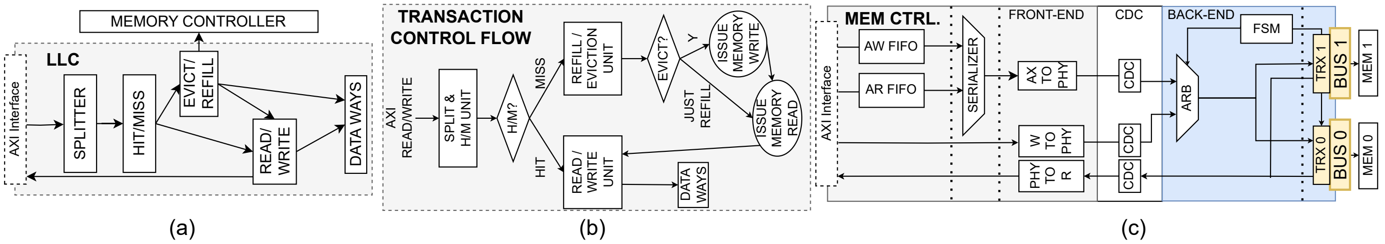

Fig. 7: Block diagrams of the components of the main memory subsystem. (a) LLC block diagram, (b) Transaction control flow diagram, (c) Memory controller block diagram.

4.4 The main memory subsystem

The main memory subsystem is a *peripheral*: we apply here the model presented in Section 3.3. It is composed of three macro submodules: (i) the *AXI Last-level Cache (LLC)*; (ii) the *HyperRAM memory controller (HMC)*; and (iii) the *HyperRAM memory (HRAM)*. It is based on HyperRAM memories leveraging the HyperBUS protocol [25]. HyperRAMs are optimized for low-overhead data storage while offering up to 3.2Gbps bandwidth. HyperRAMs expose a low pin count, a fully digital 8-bit double-data-rate (DDR) interface used for commands and data. HyperRAMs serve transactions in order, one at a time, as required by the protocol [25]. While a pure in-order strategy is simpler than those deployed by high-end commercial memory controllers, it is important to note that these controllers are typically complex closed-source IPs, making detailed analysis extremely challenging. Notably, our analysis is the first to explore this level of detail. Furthermore, the memory subsystem under analysis has shown to be effective in tape-outs of Linux-capable chips [16]. We model the service times of a single transaction in case of an LLC hit and miss. By doing so, we provide upper bounds that can be leveraged by future studies focusing on LLC interference between different *controllers* at the application level. For example, advanced cache management studies for real-time applications (e.g., cache coloring) could leverage the upper bounds provided here to bound overall task execution times.

4.4.1 RTL IP structure

The AXI Last-Level Cache is the interface of the memory subsystem with the platform. The LLC under analysis has configurable cache line length, defined as LW_{LLC}. Figure 7(a) shows the LLC's block diagram, composed of 5 pipelined units: (i) burst splitter, (ii) hit-miss detection, (iii) eviction/refill, (iv) data read/write, and (v) data ways. Figure 7(b) shows how these units cooperate to serve the requests. The burst splitter buffers and splits the incoming AXI requests into multiple sub-requests that have the same length of the cache line, and it calculates the tags of the sub-transactions. A β_i-word AXI burst request is split internally into $\lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil$ requests of length LW_{LLC}. The tags are the input to the hit-miss detection unit, which analyzes them to determine if any sub-request will be a (a) hit or (b) miss. In case (a), the transaction is directed to the read/write unit: if it is a (a.i) read, the read response is generated and immediately sent through the AXI subordinate port, completing the transaction. In the case of a (a.ii) write, the locally cached value is updated, and a write response is generated and sent back to the AXI interface to complete the transaction. In case (b), the transaction is submitted to the eviction/refill unit. Refill is performed on every miss and consists of issuing a read to the memory controller to fetch the missing data and update the data way. Eviction is performed when a cache set is full to free the necessary spot before a refill. A Least Recently Used (LRU) algorithm is used in the module under analysis.

The HyperRAM memory controller [31] is depicted in Figure 7(c). It consists of two tightly coupled modules working in two separated frequency domains: (i) the AXI *front-end* and (ii) the *back-end* PHY controller. The front-end handles and converts the AXI transactions into data packets for the PHY controller; it runs at the same clock as the LLC ($t_{\text{CK}}^{\text{HMC}}$). The back-end features a Finite State Machine (FSM) to send/receive the data packets and keep compliance with the HyperBUS protocol timings and data flow; it runs at the same clock as the HyperRAMs ($t_{\text{CK}}^{\text{HRAM}}$). The back-end handles two off-chip HyperRAMs in parallel, configured with interleaved addresses. As each HyperRAM arranges data as 16-bit words, the word size of the back-end is $DW_{\text{HYPER}} = 32$ bits.

The first stage of the front-end is composed of two FIFOs buffering incoming AXI read and write requests. Then, a serializer solves conflicts among reads and writes, allowing only one AW or AR request at a time. Following, three modules translate between AXI and the back-end protocol: (i) AXTOPHY, translating the AXI AW or AR requests into commands for the back-end; (ii) PHYTOR converting the data words from the back-end into AXI read beats for the AXI interface; and (iii) WTOPHY, converting AXI W data beats into data words and generating write response at the end of the transaction. Three CDC FIFOs are deployed between the AXTOPHY, WTOPHY, and PHYTOR and the back-end. The back-end deploys an internal FSM arranging the requests coming from the front-end into 48-bit vector requests, as required in the HyperBUS protocol, and propagating the data packets to/from the two physical HyperRAM memories through two *transceivers* (TRX).

The HyperRAM memory is an off-chip memory IP [25]. It is provided with a cycle-accurate model, fundamental for our analysis purposes [32]. Each HyperRAM is organized as an array of 16-bit words and supports one outstanding burst transaction, up to 1kB long. As two HyperRAM are interleaved, the overall burst can be up to 2kB long [19].

4.4.2 Delays and parallelism analysis

We now bound the worst-case service time of the main memory subsystem, analyzing its components one at a time. Starting with the LLC, we follow the control flow diagram reported in Figure 7(b) to guide the explanation. The LLC collects the requests incoming to the main memory. Three scenarios can happen: (i) LLC cache hit, (ii) LLC cache miss with refill, and (iii) LLC cache miss with eviction and refill.

In case (i), the LLC directly manages the request, and no commands are submitted to the HMC. The request proceeds through the LLC splitter, hit/miss unit, read/write unit, and data way stages. By design, each stage of the LLC requires a fixed number of clock cycles. The burst splitter executes in one clock cycle ($t_{\text{CK}}^{\text{LLC}}$). The hit/miss detection stage takes two clock cycles ($2t_{\text{CK}}^{\text{LLC}}$): one for tag checking and one to propagate the request to the read/write unit or the evict/refill unit. The read/write unit requires one clock cycle ($t_{\text{CK}}^{\text{LLC}}$) to route the transaction to the data ways. The data ways accept the incoming request in one clock cycle ($t_{\text{CK}}^{\text{LLC}}$) to then access the internal SRAM macros (same as the SPM, Section 4.2). The internal SRAM takes one clock cycle to provide the read data ($t_{\text{CK}}^{\text{LLC}}$), but no further latency is required on writes. Once it gets the response, the read/write unit routes the read channel to the AX interface, whereas it takes one clock cycle ($t_{\text{CK}}^{\text{LLC}}$) to generate the write B response at the end. Thus, read/write unit and data ways take together three clock cycles ($3t_{\text{CK}}^{\text{LLC}}$). Summing up the contributions, the service time in case of a hit is upper bound by:

$$t_{\text{CTRL,R/W}}^{\text{MS-HIT}} = 6 \cdot t_{\text{CK}}^{\text{LLC}}; \quad t_{\text{DATA,R/W}}^{\text{MS-HIT}} = t_{\text{CK}}^{\text{LLC}}; \tag{6}$$

Consider now cases (ii) and (iii): the eviction and refill stage is also involved, and a read (for refill) and, optionally, a write (for eviction) is issued to the main memory. Eviction and refill are run in parallel. Each operation performs two steps, each taking one clock cycle: (a) generating a transaction for the main memory and (b) generating a transaction for the data way. Thus, summing the latency introduced by the eviction and refill stage ($2t_{\text{CK}}^{\text{LLC}}$) with the ones from the other stages, the LLC's contribution to the overall control time in case of a miss is upper bound by:

$$t_{\text{CTRL,R/W}}^{\text{LLC-MISS}} = t_{\text{CTRL,R/W}}^{\text{MS-HIT}} + 2t_{\text{CK}}^{\text{LLC}} \tag{7}$$

Consider now the delay introduced by the HMC on a generic request. Later, we will use it to bound the service time for the batch of transactions issued by the LLC. As described earlier, the HMC is composed of (a) the front-end, (b) the CDC FIFOs, and (c) the back-end. Consider (a): each one of the front-end's submodules takes one clock cycle to sample and process the transaction, except for the serializer, which takes two. As transactions pass through 4 modules (FIFOs, serializer, AXITOPHY, and either WTOPHY or PHYTOR), the overall delay contribution of the front-end is equal to $5t_{\text{CK}}^{\text{HMC}}$. Consider now (b): these are the CDC FIFOs composing the AXI CDC FIFOs introduced in Section 4.1. For writes, the transmitter (TX) is the front-end, sending data to the back-end from the AXTOPHY and the WTOPHY. As both transfers happen in parallel, the delay introduced by the CDC on a write is upper bound by $t_{\text{CDC}}(t_{\text{CK}}^{\text{HMC}}, t_{\text{CK}}^{\text{HRAM}})$. For reads, first, the front-end transmits (TX) the AXTOPHY request, and then the back-end transmits the data beats: the delay introduced by the CDC on a read is upper bound by $t_{\text{CDC}}(t_{\text{CK}}^{\text{HMC}}, t_{\text{CK}}^{\text{HRAM}}) + t_{\text{CDC}}(t_{\text{CK}}^{\text{HRAM}}, t_{\text{CK}}^{\text{HMC}})$. Consider now (c): the back-end's FSM parses the incoming request into a HyperRAM command in one cycle ($t_{\text{CK}}^{\text{HRAM}}$). Following this, an extra cycle is required for the data to cross the back-end. Summing up the contributions just described, the control time of the HMC on a generic transaction is upper bound by:

$$\begin{aligned} t_{\text{CTRL,R}}^{\text{HMC}} &= 5 \cdot t_{\text{CK}}^{\text{HMC}} + t_{\text{CDC}}(t_{\text{CK}}^{\text{HMC}}, t_{\text{CK}}^{\text{HRAM}}) + t_{\text{CDC}}(t_{\text{CK}}^{\text{HRAM}}, t_{\text{CK}}^{\text{HMC}}) + 2 \cdot t_{\text{CK}}^{\text{HRAM}} \\ t_{\text{CTRL,W}}^{\text{HMC}} &= 5 \cdot t_{\text{CK}}^{\text{HMC}} + t_{\text{CDC}}(t_{\text{CK}}^{\text{HMC}}, t_{\text{CK}}^{\text{HRAM}}) + 2 \cdot t_{\text{CK}}^{\text{HRAM}} \end{aligned} \tag{8}$$

Consider now the delays introduced by the HyperRAM memories on a generic request. The control overhead time to access the HyperRAM memory is defined by the HyperBUS protocol [25]. First, the 48-bit HyperRAM command vector is sent over the two memories in $3 \cdot t_{\text{CK}}^{\text{HRAM}}$ clock cycles, as the HyperBUS command bus is 16 bits. Following, the HyperBUS provides a fixed latency for the maximum time to access the first data word, accounting for refresh effects and crossing row boundaries. The specifications [33] bound such a delay between 7 and 16 clock cycles. In our case, this is set to $12 \cdot t_{\text{CK}}^{\text{HRAM}}$. Thus, the total control latency of the HyperRAM memory is upper bound by:

$$t_{\text{CTRL,R/W}}^{\text{HRAM}} = 15 \cdot t_{\text{CK}}^{\text{HRAM}} \tag{9}$$

At this point, data are ready to be propagated. As the AXI domain and the HyperRAM have different data widths, the number of cycles to send/receive an AXI word is:

$$t_{\text{DATA,R/W}}^{\text{HRAM}} = DW_{\text{HYPER}} \cdot \lceil \frac{DW_{\text{AXI}}}{DW_{\text{HYPER}}} \rceil \cdot t_{\text{CK}}^{\text{HRAM}} \tag{10}$$

We now have all the elements to bound the overall service time of the whole main memory subsystem in case of a miss (ii) with refill and (iii) eviction and refill. First, we bound the service time to serve a refill (read) request. A β_i-long transaction is split by the LLC into $\lceil \beta_i / LW_{\text{LLC}} \rceil$ sub-transactions to the memory, each LW_{LLC}-long. Therefore, by multiplying the control time of each sub-transaction ($t_{\text{CTRL,R}}^{\text{HMC}} + t_{\text{CTRL,R}}^{\text{HRAM}}$) by the number of transactions issued ($\lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil$), we bound the control time introduced by the memory controller and the off-chip memories. To this, we sum the control time of the LLC in case of a miss ($t_{\text{CTRL,W/R}}^{\text{MS-MISS}}$) and obtain the whole control overhead. The same reasoning applies to the data time: the total number of values requested by the LLC to the memory will be equal to $LW_{\text{LLC}} \cdot \lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil$ and the overall time spent reading $LW_{\text{LLC}} \cdot \lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil t_{\text{DATA,R/W}}^{\text{HRAM}}$. It follows that the time to serve one word is $\frac{LW_{\text{LLC}}}{\beta_i} \cdot \lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil \cdot t_{\text{DATA,R/W}}^{\text{HRAM}}$. Summing it with the data time of the LLC ($t_{\text{DATA,R/W}}^{\text{MS-HIT}}$), we obtain the following upper bounds for case (ii):

$$\begin{aligned} t_{\text{CTRL,R/W}}^{\text{MS-MISS-REF}} &= t_{\text{CTRL,R}}^{\text{LLC-MISS}} + \left\lceil \frac{\beta_i}{LW_{\text{LLC}}} \right\rceil \cdot (t_{\text{CTRL,R}}^{\text{HMC}} + t_{\text{CTRL,R}}^{\text{HRAM}}); \\ t_{\text{DATA,R/W}}^{\text{MS-MISS-REF}} &= t_{\text{DATA,R/W}}^{\text{MS-HIT}} + \frac{LW_{\text{LLC}}}{\beta_i} \cdot \left\lceil \frac{\beta_i}{LW_{\text{LLC}}} \right\rceil \cdot t_{\text{DATA,R}}^{\text{HRAM}}; \end{aligned} \tag{11}$$

If the eviction is also required, $\lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil$ extra write transactions of length β_i are performed to save the evicted data. Following the same reasoning as earlier, this batch of transactions will introduce $\lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil (t_{\text{CTRL,W}}^{\text{HMC}} + t_{\text{CTRL,W}}^{\text{HRAM}})$ clock cycles to the control time and $\frac{LW_{\text{LLC}}}{\beta_i} \cdot \lceil \frac{\beta_i}{LW_{\text{LLC}}} \rceil \cdot t_{\text{DATA,W}}^{\text{HRAM}}$ to the data time. We sum these numbers to eq. 11 to upper bound the overall control and data time as follows:

$$\begin{aligned} t_{\text{CTRL,W/R}}^{\text{MS-MISS-REF-EV}} &= t_{\text{CTRL,W/R}}^{\text{MS-MISS-REF}} + \left\lceil \frac{\beta_i}{LW_{\text{LLC}}} \right\rceil (t_{\text{CTRL,W}}^{\text{HMC}} + t_{\text{CTRL,W}}^{\text{HRAM}}); \\ t_{\text{DATA,W/R}}^{\text{MS-MISS-REF-EV}} &= t_{\text{DATA,W/R}}^{\text{MS-MISS-REF}} + \frac{LW_{\text{LLC}}}{\beta_i} \cdot \left\lceil \frac{\beta_i}{LW_{\text{LLC}}} \right\rceil \cdot t_{\text{DATA,W}}^{\text{HRAM}}; \end{aligned} \tag{12}$$

Consider now the parallelism of the main memory subsystem. This is defined by the LLC, which acts as an interface with the rest of the platform, buffering up to $D^{\text{LLC}}_{\text{FIFO}}$ read and write transactions. This means that the maximum number of supported outstanding transactions is as follows:

$$\chi_R^{MS} = \chi_W^{MS} = D^{\text{LLC}}_{\text{FIFO}} \tag{13}$$

The LLC is pipelined: in the case all the enqueued accesses are hits, there is no stalling. However, the memory controller handles only one transaction at a time, stalling the preceding ones, and only serves one read or one write at a time. Hence, as soon as one access is a miss, $\rho^{\text{MS}} = 0$ and $\theta^{\text{MS}} = 0$.

4.5 AXI host crossbar

The AXI host crossbar under analysis is a consolidated AXI crossbar already validated in multiple silicon tapeouts [16], [15], [24]. We apply here the generic model for the *crossbar* proposed in Section 3.4. The crossbar is referred as R_0.

4.5.1 RTL IP structure

As detailed in Figure 8, the crossbar exports a set of input subordinate ports (S) and output manager ports (M). Each S port is connected to a demultiplexer, which routes the incoming AW and AR requests and W data to the proper destination. Each M port is connected to a multiplexer, which (i) arbitrates AW and AR requests directed to the same *peripheral*, (ii) connects the selected W channel from the *controller* to the *peripheral*, and (iii) routes back the R read data and B write responses. The crossbar under analysis can be configured for a fully combinatorial (i.e., decoding and routing operations in one clock cycle) or pipelined structure with up to three pipeline stages. In the platform under analysis, it is configured to be fully combinatorial.

4.5.2 Delays and parallelism analysis

To analyze the maximum propagation delays introduced by the crossbar, we upper bound the overall latency on a transaction by combining the delays introduced on each AXI channel. We provide two upper bounds, one for transactions in isolation (i.e., $t^{R_0}_{\text{PROP,R/W}}$ as defined in Section 3) and the other for transactions under contention (i.e., $t^{R_0}_{\text{PROP,R/W}} + t^{R_0}_{\text{CON,R/W}}$ as defined in Section 3). We will use both of them in our architectural analysis reported in Section 5.

Maximum delays in isolation: Thanks to the combinatorial structure, it is guaranteed by design that a request for a transaction, a data word, or a write response crosses the crossbar in one clock cycle ($t^{R_0}_{\text{CK}}$). Consider a whole AXI transaction. For a read transaction, the crossbar is crossed twice: on the AR and R AXI channels, respectively. For each AXI write transaction, the crossbar is crossed two times: the first time is crossed by the AW and W beats (propagated in parallel), and the second time by the B response. Thus, the propagation delays in isolation are equal to:

$$t^{R_0}_{\text{PROP,R/W}} = 2 \cdot t^{R_0}_{\text{CK}}; \tag{14}$$

Maximum delays under contention: Under contention, multiple *controllers* connected to the crossbar can attempt to concurrently send requests to the same *peripheral*, generating interference. The arbiters deploy a round-robin scheme capable of granting one AW and one AR request for each clock cycle. In the worst-case scenario, the request under analysis loses the round-robin and is served last, experiencing a delay of $M_{R_0} - 1$ clock cycles (with M_{R_0} the number of *controller* capable of interfering with the request under analysis). From the previous considerations, the maximum propagation time introduced by the crossbar is upper bound by:

$$t^{R_0}_{\text{CON,R}} = t^{R_0}_{\text{CON,W}} = M_{R_0} - 1 \tag{15}$$

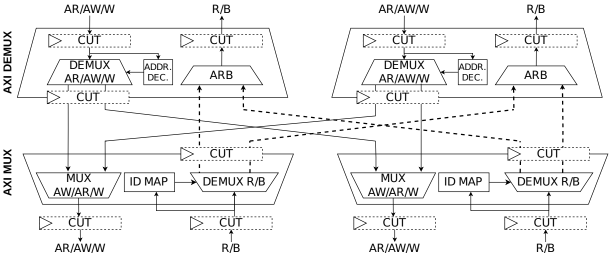

Fig. 8: AXI Crossbar block diagram

Consider now the parallelism. Concerning reads, the crossbar does not keep track of the inflight transactions. To route the responses back, it appends information to the AXI ID. Doing so does not limit the maximum number of outstanding transactions. The behavior is different for writes: AXI enforces a strict in-order execution of write transactions (see [29] p. 98). This requires the crossbar to implement a table to know the order of granted transactions. The maximum number of outstanding write transactions per S port is limited by the depth of such tables, refereed as $D^{R_0}_{\text{TAB}}$. From the previous consideration: $\chi^{R_0}_W = D^{R_0}_{\text{TAB}}$. In the architecture under analysis, $\chi^{R_0}_W$ is set to be bigger than the parallelism supported by the *peripherals* so that the crossbar does not limit the overall parallelism of the system.

5 System-level worst-case response time analysis

This section introduces our system-level end-to-end analysis to upper bound the overall response times of read and write transactions issued by a generic *controller* and directed to a generic *peripheral*, considering the maximum interference generated by the other *controllers* in the system. Our approach is static [34] and compositional [35]: we leverage the model from Section 3 and the component-level static analysis from Section 4 to compose, step-by-step, the system-level worst-case service time of transactions traversing the architecture.

We make an assumption aligned with the SoA [3], [4], [8], [11], [12], [36] to ensure independence among *peripherals* while not compromising the generality of the analysis. It is assumed that multiple outstanding transactions of the same type (either read or write) issued by the same *controller* target the same *peripheral*: before issuing a transaction targeting a *peripheral* P_j, a *controller* completes the pending transactions of the same type targeting a different *peripheral* P_z. Without such an assumption, due to the strict ordering imposed by the AXI standard [29] on the W channel, and the structure of some *peripherals* generating interference between reads and writes (i.e., $\rho^{P_j} = 0$), transactions issued by C_k and directed to P_j might interfere with transactions issued by C_i and directed to P_z, if C_i also issues in parallel transactions to P_j, and vice-versa. This assumption allows us to relax our analysis, removing such pathological cases. It is worth noticing that it does not enforce any relationship between read and write transactions. Such an assumption can either be enforced at the software level or at the hardware level. The results of our analysis can be extended to such corner cases if required. We leave this exploration for future works.

The first step of the analysis is to bound the overall response time of a transaction in isolation (Lemma 1). Secondly, we bound the maximum number of transactions that can interfere with a transaction under analysis, either of the same type (e.g., reads interfering with a read, Lemma 2) or of a different type (e.g., write interfering with a read, and vice versa, Lemma 3). Lemma 4 bounds the maximum temporal delay each interfering transaction can delay a transaction under analysis. Finally, Theorem 1 combines the results of all the lemmas to upper bound the overall worst-case response time of a transaction under analysis under interference. We report the lemmas in a general form. $AX_{i,j}$ can represent either a read or write transaction issued by the generic *controller* C_i and directed to the generic *peripheral* P_j. The *crossbar* is referred to as R_0. To make our analysis general, we assume that $\Psi_j = [C_0, ..., C_{M-1}]$ is the generic set of interfering *controllers* capable of interfering with C_i issuing transactions to P_j and that that a generic set of *bridges* $\Theta_i = \{Q_0, ..., Q_{q-1}\}$ can be present between each *controller* C_i and the crossbar R_0. The cardinality of Ψ_j is referred to as $\mid \Psi_j \mid$ and corresponds to the number of *controllers* interfering with $AX_{i,j}$.

Lemma 1. *The response time in isolation of $AX_{i,j}$ is upper bounded by:*

$$d^{X}_{i,j} = d^{P_j}_{\text{R/W}} + \sum_{Q_l \in \Theta_{i,j}} d^{Q_l}_{\text{R/W}} + d^{R_0}_{\text{R/W}} \tag{16}$$

Proof. Section 4 upper bounds the worst-case delays in isolation introduced by each component in the platform. According to their definition, such delays account for all of the phases of the transaction. The components in the platform are independent of each other. Thus, the delay introduced by each traversed component is independent of the behavior of the other components. It derives that the overall delay incurred in traversing the set of components between C_i and P_j is upper bounded by the sum of the worst-case delays introduced by all of the components in the set. Summing up the maximum delay introduced by the target *peripheral* P_j ($d^{P_j}_{\text{R/W}}$), by the set of traversed *bridges* Θ_i, and by the *crossbar* R_0 ($d^{R_0}_{\text{R/W}}$), the lemma derives. □

Lemma 2. *The maximum number of transactions of the same type that can interfere with $AX_{i,j}$ is upper bounded by:*

$$S^{X}_{i,j} = min \left(\sum_{C_y \in \Psi_j} \phi^{C_y}_{X}, \chi^{P_j}_{X} + \mid \Psi_j \mid \right) \tag{17}$$

Proof. The $\min$ in the formula has two components. As from the AXI standard definition, an interfering *controller* C_k cannot have more than $\phi^{C_k}_{X}$ pending outstanding transactions. This means that summing up the maximum number of outstanding transactions for each interfering *controller* in Ψ_j provides an upper bound on the number of transactions of the same type interfering with $AX_{i,j}$ – the left member of the $\min$ derives. From our *peripheral* analysis reported in Section 4, P_j and R_0 can limit the maximum amount of transactions accepted by the system: P_j accepts overall at most $\chi^{P_j}_{R/W}$ transactions – when such a limit is reached, any further incoming transaction directed to P_j is stalled. After P_j serves a transaction, R_0 restarts forwarding transactions to the *peripheral* following a round-robin scheme (see Section 4). In the worst-case scenario, C_i loses the round-robin arbitration against all of the $\mid \Psi_j \mid$ interfering *controllers* in Ψ_j, each ready to submit an interfering request. Summing up the contributions, also $\chi^{P_j}_{R} + \mid \Psi_j \mid$ upper bounds the maximum number of transactions interfering with $AX_{i,j}$ – the right member of the $\min$ derives. Both of the bounds are valid – the minimum between them is an upper bound providing the least pessimism – Lemma 2 derives. □

Lemma 3. *The maximum number of transactions of a different type (represented here as Y, i.e., write transactions interfering with a read under analysis, and vice versa) interfering with $AX_{i,j}$ is upper bounded by:*

$$U^{Y}_{i,j} = (S^{X}_{i,j} + 1) \cdot (1 - \theta^{P_j}) \tag{18}$$

Proof. According to Section 4.5, R_0 manages transactions of different types independently – thus, no interference of this type is generated at the R_0 level. From Section 3, $\theta^{P_j} = 1$ represents the case in which the *peripheral* is capable of serving read and write transactions in parallel (e.g., the SPM *peripheral*, Section 4.2). Thus, no interference is generated among them – the second equation derives. From Section 3, $\theta^{P_j} = 0$ represents the case in which P_j does not feature parallelism in serving read and write transactions (i.e., also write transactions interfere with reads, e.g., main memory subsystem, Section 4.4). Considering lemma 2, at most $S^{X}_{i,j}$ transactions of the same type can interfere with $AX_{i,j}$. With $\theta^{P_j} = 0$, and assuming a round-robin scheme arbitrating between reads and writes at the *peripheral* level, each one of the $S^{X}_{i,j}$ interfering transaction of the same type can be preceded by a transaction of the opposite type, which can, therefore, create interference. The same applies to $AX_{i,j}$, which can lose the arbitration at the *peripheral* level as well. Summing up the contribution, it follows that $S^{X}_{i,j} + 1$ can overall interfere with $AX_{i,j}$ – the first equation derives. □

Lemma 4. *The maximum time delay that a transaction of any kind $AX_{k,j}$ issued by the generic interfering* controller *C_k can cause on $AX_{i,j}$ is upper bounded by:*

$$\Delta_{k,j} = d^{R_0}_{\text{R/W}} + (1 - \rho^{P_j}) \cdot t^{P_j}_{\text{CTRL,R/W}} + t^{P_j}_{\text{DATA,R/W}} \cdot \beta_k \tag{19}$$

Proof. In traversing the path between C_k and P_j, $AX_{k,j}$ shares a portion of the path with $AX_{i,j}$, i.e., the target *peripheral* P_j and the crossbar R_0 – no *bridges* from Θ_k belongs to the shared path, thus the delay propagation of $AX_{k,j}$ do not contribute in delaying $AX_{k,j}$. Considering the delay generated by $AX_{k,j}$ at R_0, this is upper bounded by $d^{R_0}_{\text{R/W}}$ in Section 3.4. As from Section 3.3, $t^{P_j}_{\text{CTRL,R/W}} + t^{P_j}_{\text{DATA,R/W}} \cdot \beta_k$ is the maximum service time of P_j for the transaction $AX_{k,j}$ and upper bounds the maximum temporal delay that $AX_{k,j}$ can cause on $AX_{i,j}$ at P_j. As from the definition of an interfering transaction, $AX_{k,j}$ is served by P_j before $AX_{i,j}$. As defined by the model in Section 3.3, when $\rho^{P_j} = 1$, the *peripheral* works in a pipeline fashion. This means that for $\rho^{P_j} = 1$, the control time $t^{P_j}_{\text{CTRL,R/W}}$ of an interfering transaction is pipelined and executed in parallel with the transaction under analysis. Differently, when $\rho^{P_j} = 0$, no pipeline is implemented, and the control time of the interfering transaction can partially or totally interfere with the transaction under analysis. From the previous considerations, the contribution $(1 - \rho^{P_j}) \cdot t^{P_j}_{\text{CTRL,R/W}} + t^{P_j}_{\text{DATA,R/W}} \cdot \beta_k$ derives. Summing up the contributions, the lemma follows. □

Theorem 1. *The overall response time of $AX^{i,j}$ under the interference generated by the other* controllers *in the system is upper bounded by:*

$$H^X_{i,j} = d^X_{i,j} + (S^X_{i,j} + U^Y_{i,j}) \cdot \Delta_{k,j} \tag{20}$$

Proof. Summing up the contribution in isolation for $AX_{i,j}$ (Lemma 1) with the sum of the maximum number of interfering transactions of the same type (Lemma 2) and of a different type (Lemma 3) multiplied by the maximum delay generated by each interfering transaction (Lemma 4), Theorem 1 derives. □

The results presented in this Section represent analytical upper bounds derived through static code analysis and the formulation of mathematical proofs. Section 6 will validate them through a comprehensive set of cycle-accurate experiments and measurements, and it will discuss the model's limitation and dependency on the analyzed IPs.

6 Experimental validation

This Section describes the experimental campaign we conducted to validate the methodology and models. The aim of the experimental campaign is to assess that the results presented in the previous Sections correctly upper bound the maximum delays and response times at the component level and the architectural level. We follow a hierarchical approach: at first, Section 6.1 aims to validate the results at the component level we proposed in Section 4. Following, in Section 6.2, we experimentally validate the system-level analysis we proposed in Section 5. The experiments are conducted in a simulated environment (leveraging the Siemens QuestaSIM simulator) and by deploying the design on an FPGA platform. In the simulated experiments, we deploy custom AXI managers for *ad-hoc* traffic generation and cycle-accurate performance monitors. The generic custom manager represents a generic configurable *controller* C_i issuing requests for transactions – we will refer to that as GC_i. In the FPGA, we leverage CVA6 and the PULP cluster to generate the traffic with synthetic software benchmarks and rely on their performance-monitoring registers to collect the measurements. The experimental designs are deployed on the AMD-Xilinx VCU118, using the Vitis 2022.1 toolchain. Similar State-of-the-Art works upper bounding the execution time of a single transaction leverage synthetic benchmarks to measure the worst-case access times since generic applications fail to do so [8]–[10]. For this reason, we concentrate on synthetic benchmarks at the IP and the system level.

6.1 Component-level hardware modules

6.1.1 Delays analysis

This subsection presents the tests run to measure the worst-case access latency time in isolation for the *peripherals* ($d^{P_j}_{R/W}$), the *crossbar* ($d^{R_0}_{R/W}$) and the *bridges* ($d^{Q_j}_{R/W}$) from Section 4. We connect the generic controller CG_i to the IP under analysis for these experiments. We let CG_i issue 100'000 transactions, one at a time, with random burst length (β_i). We monitor the service times and then pick the longest ones for different β_i.

Figure 9 compares the maximum measured experimental delays with the upper bound proposed in Section 4. Figure 9(a) reports the maximum service time of the main memory subsystem in case of a miss as a function of the burst length of the transaction under analysis, either when (i) only a refill is necessary and (ii) both refill and eviction are necessary, compared with the bounds proposed in Section 4.4. The measured service times are lower than the bounds. The pessimism is between 3% and 10.1%; the larger β_i, the higher the pessimism. Higher pessimism on longer transactions is due to the internal splitting at the LLC. As from our analysis, the memory subsystem is not fully pipelined ($\rho^{MS} = 0$). However, in practice, the control and data phases of consecutive sub-transactions might be partially served in parallel by the LLC and the memory controller. This means that the longer the transaction, the higher the number of sub-transactions and their overlap, and the lower the service time compared to our model. Thus, the pessimism increases. Figure 9(b) reports the measured results on the main memory subsystem but in case of a hit, compared with the bounds proposed in Section 4.4. As we consider an LLC hit, the access to the HyperRAM is not performed: this test analyzes the service time of the LLC. Our bounds are always upper bounds for the maximum measured results. The trend here is reversed w.r.t. Figure 9(a) – as β_i increases, the relative pessimism decreases from 7.7% down to 0.4%. In this case, the source of the pessimism comes only from the control time, which does not depend on β_i, while there is no pessimism on the data time. Hence, this pessimism gets amortized as the burst length and the overall service time increase. We conduct the same experimental campaign also on the AXI SPM – the measured results, compared with the bounds proposed in Section 4.2, are reported in Figure 9(c). The trends are similar to the ones reported in Figure 9(b) for LLC hits – the pessimism of our analysis is limited to 1 and 2 clock cycles for reads and writes on the control time, respectively. As in the case of the LLC HITs, the upper bound on the control overhead gets amortized for longer transactions, and the pessimism reduces from 8.8% to 0.5%.

Figure 9(d) reports the maximum measured latency to cross an AXI CDC FIFO as a function of the manager clock period (the subordinate clock period is fixed to 30 ns) and compared with the bounds proposed in Section 4.1. The results are independent of the length of the transaction. To stimulate the highest variability, the phases of the clocks are randomly selected on a uniform distribution. The first bar reports the crossing delays from the manager to the subordinate side, corresponding to the delays introduced on the AW, W, and AR AXI channels. The second bar reports the crossing delays from the subordinate to the manager side, corresponding to the overall delays on the AXI R and B channels. The third bar shows the overall delay on a complete transaction, corresponding to the sum of the two previously introduced contributions (see Section 4.1). The pessimism of our bounds is, at most, one clock cycle of the slowest clock between manager and subordinate.

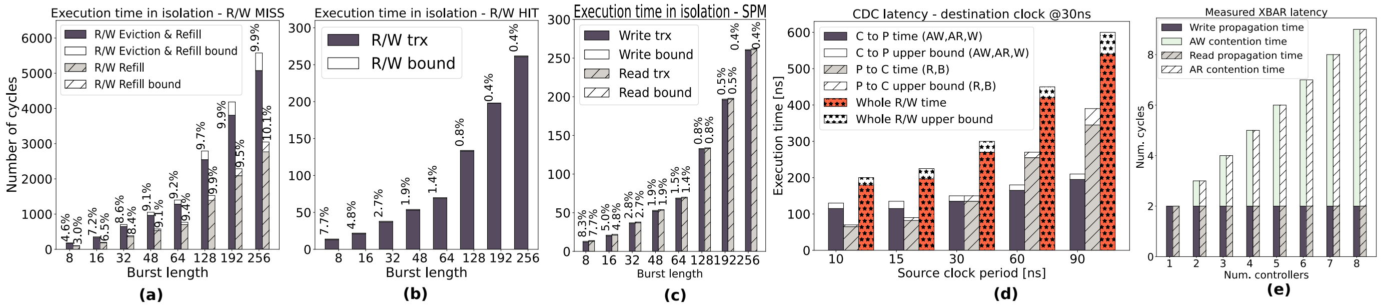

Fig. 9: Services time in isolation.

Figure 9(e) reports the measured propagation delays introduced by the crossbar over an entire write and read transaction, compared with the bounds of Section 4.5, varying the number of *controllers*. As explained in Section 4.5, the propagation delay is the sum of the propagation latency without interference (eq. 14) and the additional contention latency (eq. 15), which depends on the number of *controllers*. Thanks to the simplicity of the arbitration operated by the crossbar (pure round-robin), our proposed bounds exactly match the measurements. We conducted the experimental campaign also on the IO subsystem. We measured the maximum service time and compared it with the upper bounds of Section 4.3, which we do not show for space reasons: such IP supports only single-word transactions. Our upper bounds exceed the maximum measured service time with pessimism of down to 2 clock cycles, with service times of 4 (write) and 5 (read) clock cycles.

6.1.2 Parallelism

We also demonstrate our analysis of parallelism of the *peripherals* ($\chi^{P_j}_{R/W}$) and the *crossbar* ($\chi^{R_0}_{R/W}$) analyzed in Section 4. To do so, we configured CG_i to issue unlimited outstanding transactions to the *peripheral* under test. In parallel, we monitor the maximum number of accepted outstanding transactions. Our measurements match our analysis: the maximum number of outstanding transactions is defined by the maximum parallelism accepted at the input stage of the peripherals and the crossbar.

6.2 System-level experiments

While the previous experiments focused on the evaluation at the IP level, this set of experiments aims to evaluate the system-level bounds proposed in Section 5. We first validate our analysis in simulation. We developed a System Verilog testbench with two configurable AXI synthetic *controllers* CG_i connected to the target architecture (see Figure 2) stimulating overload conditions to highlight worst-case scenarios. We also validate our results on FPGA, generating traffic with CVA6 and the PULP cluster.

At first, we evaluate the results in isolation *at the system level* as a function of the burst length, leveraging the same strategy used for the previous experiments. Namely, these tests are meant to validate Lemma 1 (eq. 16). To measure the service time at the system level in isolation, we let one GC_i issue 100’000 transactions, one at a time, with different β_i, while the other GC_k is inactive. We monitor the service times and then pick the longest ones for each β_i. Figures 10 (a) and (b) report the maximum measured system-level response times in isolation for completing a transaction issued by the generic *controller* GC_i and directed to (a) the main memory subsystem (case of cache miss, causing either refill or both refill and eviction) and (b) to the SPM memory, compared with the bounds proposed in Lemma 1. The measured service times are smaller than the bounds in all the tested scenarios. The measure and the trends reported in Figure 10(a) are aligned with the ones found at the IP level and reported in Figure 9(a). This is because the overhead introduced by the crossbar (in isolation) and the CDC FIFOs is negligible compared to the memory subsystem’s service time. Figure 10(b) shows a trend aligned with the results at the IP-level reported in Figure 9(c): the lower β_i, the higher the pessimism. It is worth mentioning that the analysis shows higher pessimism at the system level than at the IP level. This is due to the extra pessimism from the crossbar and the CDC, which is nevertheless amortized on longer transactions, down to 1.9%.

We now analyze the results under maximum interference, to verify the results of Lemma 2 and 3 and Theorem 1. For these tests, the execution of GC_i (100’000 transactions, one at a time) receives interference by *controller* GC_k. β_k is fixed and equal to β_i, while we vary the amount of outstanding transactions GC_k can issue ($\phi^{CG_k}_{R/W}$). Figures 10 (c), (d), and (e) report the maximum measured system-level response times for completing a transaction issued by the generic *controller* GC_i and directed to (c) the main memory with an LLC miss considering $\beta_i = 16$, (d) the SPM memory, considering $\beta_i = 16$, and (e) the SPM memory, considering $\beta_i = 256$, and compare them with the upper bounds proposed in equation 20. Figures 10 (c), (d), and (e) verify the results of Lemma 2 and 3: when $\phi^{CG_k}_{R/W} > \chi^{MS}_{R/W}$ (two bars on the right), the total service time is defined by the parallelism of the peripheral itself – as expected, after saturating the number of interfering transactions accepted by the peripheral, the measured results are the same regardless of the increase of $\phi^{CG_k}_{R/W}$. Differently, when $\phi^{CG_k}_{R/W} \leq \chi^{MS}_{R/W}$, a reduced value of $\phi^{CG_k}_{R/W}$ corresponds to lower interference and response times. Figure 10(c) refers to the case of an LLC miss under interference when $\beta_k = 16$. The results confirm the safeness of our analysis, which correctly upper bounds the overall response times with a pessimism around 15%, which is slightly higher than the pessimism of a transaction in isolation at the system level. As explained in the previous subsection, when multiple transactions are enqueued, the memory subsystem can partially serve their data and control

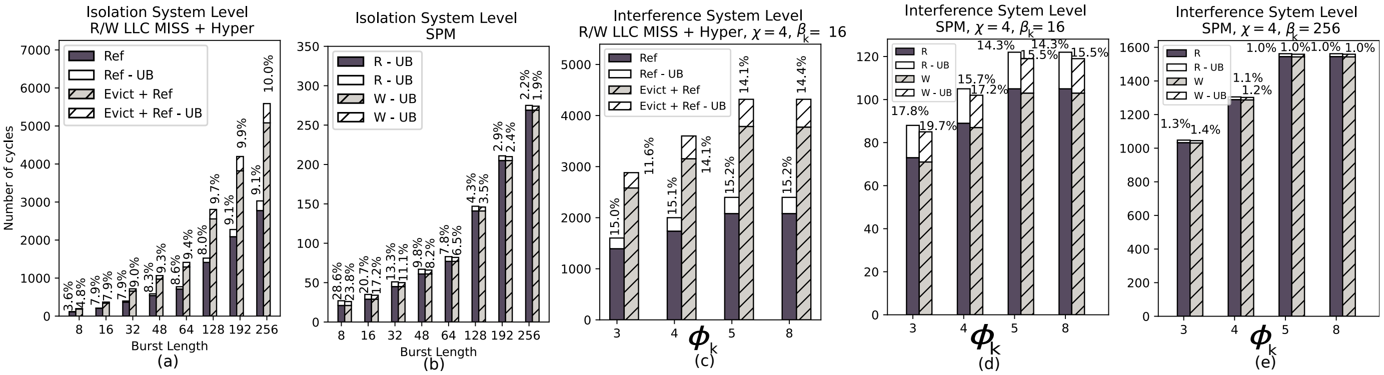

Fig. 10: Services times under interference.

phases in parallel. However, our model only allows $\rho^{MS} = 1$ or $\rho^{MS} = 0$, i.e., either the *peripheral* is fully pipelined or not pipelined at all. Since $\rho^{MS} = 0$, the pessimism is slightly higher when more transactions are enqueued (and partially served in parallel) as equation 19 counts the service time of a transaction fully when $\rho^{MS} = 0$. Varying β_k of GC_k gives comparable results – we do not report such results for briefness and lack of space. We provide two charts for the SPM, in Figure 10(d) and Figure 10(e). The comparison of the two charts highlights how the interfering transactions' length impacts the analysis's pessimism, ranging between 19.7% for $\beta = 16$ to 1% for $\beta = 256$. The trend here is aligned with the service time at the system level in isolation: the pessimism comes from the control times of SPM and propagation latency of the crossbar and the CDC, which are amortized as the data time increases with β_k.

6.3 Discussion

In this Section, we validated the analysis of Sections 4 and 5 through an extensive set of tests. We demonstrated how the proposed approach enables detailed explanations of the analysis's pessimism and facilitates iterative refinement. This allows us to derive upper bounds that are safe yet not overly pessimistic, particularly when compared to similar state-of-the-art works based on closed-source or loosely-timed IPs. Nevertheless, while the methodology is promising, the resulting analysis may seem limited in comparison to other works that model more sophisticated closed-source IPs. Here, we discuss the limitations of our analysis, focusing on its dependence on the underlying characteristics of the available open-source hardware.

It is noteworthy how the analysis leverages the round-robin policy of the main interconnect and the in-order nature of *peripherals* in Lemmas 2 and 3. The absence of internal reordering allows to derive the number of transactions preceding the one under interference directly from the arbitration policy. As long as the *peripherals* serve the transactions in order, extending the analysis to support other arbitration policies is expected to require minimal effort. Instead, supporting *peripherals* with internal transaction reordering can lead to *timing anomalies* [7] and make the proposed model unsafe, as previously demonstrated in [5]. Our analysis focuses on the available *peripherals* within the target architecture, as out-of-order *peripherals* are not available open-source to us. We envision expanding the analysis to match higher-performance platforms as open-source hardware evolves.

Lastly, it is important to note that the analysis bounds only a single transaction issued by C_i – this limitation is not imposed on the interfering controllers. Lemma 2 does not consider C_i to have more pending transactions, except for the ones already accepted by P_j. In other words, Lemma 2 assumes that there is not a queue of transactions buffered in the *bridges* between C_i and R_0, which could exist when P_j is full. We could potentially extend the model to define a batch of enqueued transactions and then modify Lemma 2 to analyze this scenario. Such an extension would further build upon the proposed model and analysis, which is limited to bound the access time of a single transaction.

7 Related Work

In this Section, we provide a thorough comparison with previous works focusing on enhancing the timing predictability of digital circuits. Traditionally, the majority of these works leverage commercial off-the-shelf devices [34], [38] or predictable architectures modeled with a mix of cycle-accurate and behavioral simulators [39]. Also, they focus on bounding the execution times for predefined specific software tasks rather than the individual transaction service times [7], [38]–[40]. Furthermore, they build the models from dynamic experiments rather than from static analysis, largely due to the dearth of detailed hardware specifications [35], limiting the generality of their approach. More recent works advocate for static modeling and analysis of protocols [8], [13], interconnect [1], [3], [9], and shared memory resources [5], [10] to provide more generic and comprehensive models. While their value is undeniable, due to the unavailability of the source RTL, each one focuses on only one of these resources, resulting in a significant penalty to the pessimism of the upper bounds [5]. Our work breaks from this convention, presenting a holistic static model of an entire open-source architecture rigorously validated through RTL cycle-accurate simulation and FPGA emulation. As Table 1 shows, this is the first work to analyze and model the open-source silicon-proven RTL of all the IPs composing a whole SoC to build the least pessimistic upper bounds for data transfers within the architecture when compared to similar SoA works.

Biondi et al. [13] developed a model of the memory-access regulation mechanisms in the ARM MPAM and provided detailed instantiations of such mechanisms, which they

TABLE 1: Comparison with State-of-the-Art works for predictability. IC = Interconnect. DMR = Deadline miss ratio.

	Target	Analysis on	Pessimism	Technology	Open RTL
Biondi et. al. [13]	ARM MPAM Protocol	Protocol specs (Model)	No HW	✗	✗
Hassan et. al. [8]	JEDEC DDR3 Protocol	Protocol specs (Model)	$0\% - 200\%$	✗	✗
Abdelhalim et.al. [5]	Whole mem. hier.	IPs & System (C++ Model)	$16\% - 50\%$	✗	✗
BlueScale [3]	Hier. mem. IC	IC uArch (Black-box)	DMR	FPGA	✗
AXI-RT-IC [1]	AXI SoC IC	IC uArch (Black-box)	DMR	FPGA	✗
Restuccia et. al. [9]	AXI Hier. mem. IC	IC uArch (Black-box)	$50\% - 90\%$	FPGA	✗
AXI-REALM [37]	AXI traffic regulator	No analysis	No model	FPGA & ASIC	✓
Ditty [10]	Cache coher. mechanism	IP (Fine-grained RTL)	$100\% - 200\%$	FPGA	✓
This Work	SoC IC, peripherals & system-level	IP & System (Fine-grained RTL)	$1\% - 28\%$	FPGA & ASIC	✓

then evaluated with IBM CPLEX, a decision optimization software for solving complex optimization models. While elegant, this approach is not validated on hardware and, therefore, is limited in terms of applicability and precision. A more practical and adopted approach is the one proposed by Hassan and Pellizzoni [8]. The authors develop a fine-grained model of the JEDEC DDR3 protocol, validated with MCsim [12], a cycle-accurate C++ memory controller simulator. Unfortunately, not having access to the RTL prevents fine-grained modeling and analysis and mandates over-provisioning, strongly impacting the overall pessimism of the system, which can be as high as 200%. Abdelhalim et al. in [5] present a study bounding the access times of memory requests traversing the entire memory hierarchy and propose μarchitectural modifications to the arbiters in such hierarchy. Their modifications result in very low pessimism (down to 16%) on synthetic and real-world benchmarks. However, the results are validated on C++ models of the cores, interconnect, and memory controllers, not RTL code targeting silicon implementation.

More recently, different researchers proposed models of hardware IPs that they could validate through cycle-accurate experiments [1], [4], [9]. In [9], Restuccia et al. focused on upper bounding the response times of AXI bus transactions on FPGA SoCs through the modeling and analysis of generic hierarchical interconnects arbitrating the accesses of multiple hardware accelerators towards a shared DDR memory. In this work, the interconnect under analysis is a proprietary Xilinx IP, which had to be treated as a black box. Also, due to the unavailability of the RTL code, the authors did not model the other IPs composing the target platform, limiting the precision of the proposed upper bounds, which achieve a pessimism between 50% and 90%. Jiang et al. modeled, analyzed, and developed AXI-IC^{RT} [1] and Bluescale [3], two sophisticated interconnects providing predictability features and coming with a comprehensive model. However, the model and analysis proposed in AXI-IC^{RT} [1], and Bluescale [3] are not as fine-grained as ours: the authors do not provide upper bounds of the access times but rather focus on the deadline miss ratio given a fixed workload for the different controllers in the system. Moreover, the authors do not provide the RTL of such solutions. AXI-REALM [37] proposes completely open-source IPs supporting predictable communications. However, it misses a holistic model and analysis. In Ditty [10], researchers propose an open-source predictable directory-based cache coherence mechanism for multicore safety-critical systems that guarantees a worst-case latency (WCL) on data accesses with almost cycle-accurate precision. However, Ditty's model only covers the coherency protocol latency and the core subsystem, overlooking system-level analysis and achieving very pessimistic boundaries. In this landscape, it emerges clearly that our work is the first one covering both modeling and analysis of the interconnects and the shared memory resources, with an in-depth analysis of silicon-proven open-source RTL IPs and achieving the lowest pessimism when compared to similar SoA works.

8 Conclusions

In conclusion, this is the first work to bridge the gap between open-source hardware and predictability modeling and analysis. It presented (i) a fine-grained model and analysis for the typical building blocks composing modern heterogeneous low-power SoCs directly based on the source RTL, and (ii) a full mathematical analysis to upper bound data transfer execution times. Namely, we demonstrated a methodology that successfully exploits the availability of the source code to provide safe, but not overly pessimistic, upper bounds for the execution times of data transfers when compared to similar SoA works based on closed-source IPs.

As discussed in Section 6, after this thorough evaluation, we envision extending our results to other popular open-source IPs and different arbitration policies. To hopefully stimulate novel research contributions, we open-source a guide to replicate the results shown in Section 6 at https://github.com/pulp-platform/soc_model_rt_analysis, comprehensive of the simulated environment and the software benchmarks to run on a sophisticated Cheshire-based SoC targeting automotive applications.

References

[1] Z. Jiang *et al.*, "AXI-IC^{RT} RT : Towards a Real-Time AXI-Interconnect for Highly Integrated SoCs," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 786–799, 2022.

[2] A. Biondi *et al.*, "SPHERE: A multi-SoC architecture for next-generation cyber-physical systems based on heterogeneous platforms," *IEEE Access*, vol. 9, pp. 75 446–75 459, 2021.

[3] Z. Jiang *et al.*, "BlueScale: a scalable memory architecture for predictable real-time computing on highly integrated SoCs," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1261–1266.

[4] F. Restuccia *et al.*, "AXI HyperConnect: A Predictable, Hypervisor-level Interconnect for Hardware Accelerators in FPGA SoC," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[5] S. Abdelhalim *et al.*, "A Tight Holistic Memory Latency Bound Through Coordinated Management of Memory Resources," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, vol. 262, 2023, pp. 17:1–17:25.

[6] G. Fernandez *et al.*, "Contention in multicore hardware shared resources: Understanding of the state of the art," in *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, 2014, pp. 31–42.

[7] S. Hahn, M. Jacobs, and J. Reineke, "Enabling Compositionality for Multicore Timing Analysis," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, 2016, p. 299–308.

[8] M. Hassan and R. Pellizzoni, "Bounding DRAM Interference in COTS Heterogeneous MPSoCs for Mixed Criticality Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[9] F. Restuccia *et al.*, "Bounding Memory Access Times in Multi-Accelerator Architectures on FPGA SoCs," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 154–167, 2022.

[10] Z. Wu, M. Bekmyrza, N. Kapre, and H. Patel, "Ditty: Directory-based Cache Coherence for Multicore Safety-critical Systems," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[11] M. Hassan, "On the Off-Chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?" in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 495–505.

[12] R. Mirosanlou, D. Guo, M. Hassan, and R. Pellizzoni, "Mcsim: An extensible dram memory controller simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 105–109, 2020.

[13] M. Zini, D. Casini, and A. Biondi, "Analyzing Arm's MPAM From the Perspective of Time Predictability," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 168–182, 2023.

[14] A. Herrera, "The Promises and Challenges of Open Source Hardware," *Computer*, vol. 53, no. 10, pp. 101–104, 2020.

[15] A. Ottaviano, T. Benz, P. Scheffler, and L. Benini, "Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2023.

[16] L. Valente *et al.*, "Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs," in *2023 IEEE Hot Chips 35 Symposium (HCS)*, 2023, pp. 1–12.

[17] M. B. Taylor, "Your Agile Open Source HW Stinks (Because It Is Not a System)," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.

[18] PULP, "PULP Platform Github," https://github.com/pulp-platform, 2022.

[19] L. Valente *et al.*, "HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[20] OpenHW-Group, "CVA6," https://github.com/openhwgroup/cva6, 2022.

[21] M. Schneider *et al.*, "Composite Enclaves: Towards Disaggregated Trusted Execution," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, p. 630–656, Nov. 2021.

[22] P. Platform, "PULP cluster," https://github.com/pulp-platform/pulp_cluster, 2022.

[23] OpenHW-Group, "CV32E40P," https://github.com/openhwgroup/cv32e40p, 2023.

[24] A. Kurth *et al.*, "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[25] B. John, "HyperRAM as a low pin-count expansion memory for embedded systems," https://www.infineon.com/dgdl/Infineon-HyperRAM_as_a_low_pin-count_expansion_memory_for_embedded_systems-Whitepaper-v01_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0fb28970272c&da=t, 2020.

[26] AMD, "Zynq-7000 - Technical Reference Manual, UG585," https://docs.xilinx.com/r/en-US/ug585-zynq-7000-SoC-TRM.

[27] A. Noami, B. Pradeep Kumar, and P. Chandrasekhar, "High Performance AXI4 Interface Protocol for Multi-Core Memory Controller on SoC," in *Data Engineering and Communication Technology*, K. A. Reddy, B. R. Devi, B. George, and K. S. Raju, Eds. Singapore: Springer Singapore, 2021, pp. 131–140.

[28] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-low-latency lightweight dma for tightly coupled multi-core clusters," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2597917.2597922

[29] ARM, "AMBA AXI Protocol Specification," https://developer.arm.com/documentation/ihi0022/j/?lang=en, 2022.

[30] Xilinx-AMD, "Dual Port SRAM specifications," https://docs.xilinx.com/r/2022.1-English/ug1483-model-composer-sys-gen-user-guide/Dual-Port-RAM.

[31] PULP, "HyperRAM Controller RTL," https://github.com/pulp-platform/hyperbus, 2022.

[32] Infineon, "HyperRAM RTL," https://www.infineon.com/dgdl/Infineon-S27KL0641_S27KS0641_VERILOG-SimulationModels-v05_00-EN.zip?fileId=8ac78c8c7d0d8da4017d0f6349a14f68, 2022.

[33] Infineon, "HyperBUS specifications," https://www.infineon.com/dgdl/Infineon-HYPERBUS_SPECIFICATION_LOW_SIGNAL_COUNT_HIGH_PERFORMANCE_DDR_BUS-AdditionalTechnicalInformation-v09_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ed619b05663, 2022.

[34] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008. [Online]. Available: https://doi.org/10.1145/1347375.1347389

[35] T. Mitra, J. Teich, and L. Thiele, "Time-critical systems design: A survey," *IEEE Design & Test*, vol. 35, no. 2, pp. 8–26, 2018.

[36] F. Restuccia *et al.*, "Modeling and analysis of bus contention for hardware accelerators in FPGA SoCs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[37] B. Thomas *et al.*, "AXI-REALM: A Lightweight and Modular Interconnect Extension for Traffic Regulation and Monitoring of Heterogeneous Real-Time SoCs," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

[38] J. P. Cerrolaza *et al.*, "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, aug 2020. [Online]. Available: https://doi.org/10.1145/3398665

[39] M. Schoeberl *et al.*, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[40] G. Fernandez *et al.*, "Increasing confidence on measurement-based contention bounds for real-time round-robin buses," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015.

Luca Valente received his PhD from the University of Bologna in the Department of Electrical, Electronic, and Information Technologies Engineering (DEI) in 2024. His main research interests are hardware-software co-design of heterogeneous SoCs.

Francesco Restuccia received a PhD degree in computer engineering (cum laude) from Scuola Superiore Sant'Anna Pisa, in 2021. He is a post-doctoral researcher at the University of California, San Diego. His main research interests include hardware security, on-chip communications, timing analysis for heterogeneous platforms, cyber-physical systems, and time-predictable hardware acceleration of deep neural networks on commercial FPGA SoC platforms.

Davide Rossi received the Ph.D. degree from the University of Bologna in 2012. He has been a Post-Doctoral Researcher with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi," University of Bologna, since 2015, where he is currently an Associate Professor. His research interests focus on energy-efficient digital architectures. In this field, he has published more than 100 papers in international peer-reviewed conferences and journals.

Ryan Kastner is a professor in the Department of Computer Science and Engineering at UC San Diego. He received a PhD in Computer Science at UCLA, a masters degree in engineering and bachelor degrees in Electrical Engineering and Computer Engineering from Northwestern University. His current research interests fall into three areas: hardware acceleration, hardware security, and remote sensing.

Luca Benini holds the chair of Digital Circuits and Systems at ETHZ and is Full Professor at the Università di Bologna. He received a PhD from Stanford University. His research interests are in energy-efficient parallel computing systems, smart sensing micro-systems and machine learning hardware. He has published more than 1000 peer-reviewed papers and 5 books. He is a Fellow of the ACM and a member of the Academia Europaea.