Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC

Francesco Restuccia
Scuola Superiore Sant'Anna
Pisa, Italy
francesco.restuccia@sssup.it

Alessandro Biondi
Scuola Superiore Sant'Anna
Pisa, Italy
alessandro.biondi@sssup.it

Mauro Marinoni
Scuola Superiore Sant'Anna
Pisa, Italy
mauro.marinoni@sssup.it

Giorgio Buttazzo
Scuola Superiore Sant'Anna
Pisa, Italy
giorgio.buttazzo@sssup.it

***Abstract*—Advanced eXtensible Interface (AXI) is an open-standard communication bus interface implemented in most commercial off-the-shelf FPGA System-on-Chips (SoC) to exchange data within the chip. Unfortunately, the AXI standard does not mandate any mechanism to detect possible misbehavior of the connected modules. This work shows that this lack of specification has a relevant impact on popular implementations of the AXI bus. In particular, it is shown how it is easily possible to inject arbitrarily-long delays on modern FPGA system-on-chips under the presence of misbehaving bus masters. To safely solve this issue, this paper presents a general timing analysis to bound the execution of periodically-invoked hardware accelerators in nominal conditions. This timing analysis is then used to configure a latency-free hardware module named AXI Stall Monitor (ASM), also proposed in this paper, capable of detecting and safely solving possible stalls during AXI bus transactions. The ASM leaves a quantified flexibility to the hardware accelerators when deviating from nominal conditions. The contribution is finally supported by a set of experiments on the Zynq-7000 and Zynq Ultrascale+ SoCs by Xilinx.**

I. Introduction

System-on-chips (SoCs) that integrate Field-Programmable Gate Arrays (FPGAs) are becoming very attractive for developing *time-critical* embedded systems due to their high computational power, their flexibility, and the possibility to implement dedicated hardware accelerators on the FPGA available on the same chip [1]–[3]. Just to name an example of their capabilities, they allow implementing on-board inference engines for deep neural networks to increase the intelligence of the system, which is particularly relevant for autonomous/assisted driving applications and advanced robotics [4]–[8].

Nevertheless, a big problem in using such platforms in safety-critical applications is that the interference occurring in accessing shared resources (such as the memory subsystem) may introduce *unbounded and unpredictable delays* in the computational activities, preventing any form of a-priori timing guarantee, required in such systems for *certification* purposes. This is because commercial-off-the-shelf FPGA-based SoCs are generally designed for high throughput (i.e., for best-effort applications), rather than for *time predictability*. In these platforms, data exchange mostly occurs through the AMBA AXI open standard. The AXI standard provides advanced features that makes it highly flexible for different applications, but it does not define any mechanism to supervise the behavior of bus masters. The lack of supervision allows hardware accelerators to behave (or misbehave) in the system without any control. This is especially critical when hardware accelerators are provided as specialized IP blocks developed from external sources so that it is not possible to accurately validate them to verify the absence of misbehavior. To further complicate this issue, in systems using *dynamic partial reconfiguration* (DPR) [9]–[11], misbehaving/malicious hardware accelerators can more likely be programmed on the FPGA. At last, misbehavior in the execution of hardware accelerators can also be caused by a fault of the silicon in the FPGA area. Such misbehaving conditions can compromise the functionality of the entire system, up to requiring a system reset to restore a safe condition. This leads to large recovery delays that may not be acceptable in safety-critical applications and can harm the quality of service in non-critical systems [12].

A. Contribution

This paper makes the following contributions:

1) It studies the impact of the delays introduced by modules on the AXI bus. It also shows that a misbehaving device on the FPGA fabric can easily *stall* the AXI bus on modern FPGA SoC platforms for an unbounded amount of time.
2) It presents an analysis to bound the *worst-case* response time of periodic hardware (HW) tasks that share the bus with other HW-tasks under nominal conditions (no misbehavior).
3) It presents the AXI Stall Monitor (ASM), a minimal hardware module to be deployed on the FPGA fabric that allows monitoring the delays introduced by hardware modules and shielding the system from misbehaving HW-tasks that may stall the bus. The ASM allows for a safe recovery after a misbehavior is detected, keeping the rest of the system operational (i.e., no reset needed). The ASM is easy-to-integrate, and does not introduce any latency.
4) Leveraging the response-time analysis, it presents a method to compute a safe bound on the maximum amount of time that a HW-task can stall while guaranteeing that no task violates its timing constraints. This method has also been implemented in a driver that allows to automatically configure/reconfigure the ASMs (even at runtime).

II. Related work

To adopt heterogeneous SoCs in safety-critical applications, it is crucial to properly address timing predictability issues related to the contention of shared resources within the chip (e.g., bus interconnects and the memory subsystem), especially whenever the behavior of some component cannot be trusted. In the context of multiprocessors, a lot of efforts have been spent in improving timing predictability with hardware and software mechanisms [13], [14], and in bounding contention delays via worst-case timing analysis [15]. While some of these research efforts also apply to heterogeneous platforms [16], [17], new challenges have to be addressed to achieve timing predictability on modern SoCs, especially when hardware acceleration [18], [19] and mixed-criticality applications [20] are concerned. Concerning misbehavior during bus transactions, most FPGA-SoC vendors are generally not explicitly addressing timing predictability, either declaring that no guarantee can be provided [21] or designing their solutions under the assumption that programmable logic always behaves accordingly to standards [22]. Some efforts have been spent in implementing components to monitor the bus traffic and possibly react to critical situations by just performing a system reset. For instance, the Xilinx AXI Performance monitor [23] allows monitoring the performance of the bus in multi-master systems. Also, the research community proposed similar solutions with enriched features to monitor the performance of the bus [24], [25]. However, all these solutions are passive, that is, they do not provide any mechanism to preserve the system operation or to guarantee certain timing constraints in the presence of misbehavior. Despite these represent valuable proposals to monitor and profile the performance of a system, richer solutions are required to adopt FPGA SoCs in safety-critical systems. To the best of our records, this is the first work that explicitly addresses timing predictability in the presence of misbehavior during bus transactions.

III. Essential background

A typical FPGA SoC architecture combines a *Processing system* (PS) (generally based on one or more processors) with a *Field Programmable Gate Array* (FPGA) in a single device. Both the subsystems access a DRAM controller in the PS through which they can access a shared DRAM memory. The de-facto standard interface for interconnections is AMBA AXI [26]. In AXI, the transactions are started by a master, which requests to read/write data from/to a slave. The communication between the FPGA and the PS is allowed by the PS-FPGA interface and the FPGA-PS interface.

1) Multi-Master architecture: There are conditions in which multiple master HW-tasks share the same AXI port. In such conditions, an AXI *interconnect* is in charge of arbitrating conflicting requests and routing the corresponding data. To address the scenario in which contention is maximized, this paper focuses on the case in which all HW-tasks share the same AXI port in the FPGA-PS interface. Hence, the considered architecture is composed of an arbitrary number N of master HW-tasks HW_i, each connected to a slave port of the interconnect I_{AXI}. The (single) master port of I_{AXI} is eventually connected to a slave port in the FPGA-PS interface.

IV. Motivations

The AXI standard does not mandate any mechanism/policy to deal with delays during data sampling. To the best of our knowledge, it is up to the vendor that implements an AXI-compliant bus to decide (i) whether monitoring the bus signals to measure delays during data sampling and/or detect misbehavior, and (ii) how to react to them. As a matter of fact, according to the AXI standard, a module connected to the bus is free to introduce *arbitrary* delays that in turn affect other modules connected to the bus, even if they do not communicate with each other. To further complicate this issue, if a module does not make progress during some phase of the bus protocol, then the AXI standard leaves room for scenarios in which *the entire system can stall for an unbounded amount of time*. This lack of specifications has a relevant impact on commercial, AXI-compliant bus implementations, especially when used to realize time-sensitive systems. Note that, despite some commercial platforms implement mechanisms for isolating bus masters (see [27], p. 366) or managing the quality-of-service of transactions (see [27], p. 375), they unfortunately do not provide any solution to prevent such unbounded delays.

To showcase the problem, we implemented a test-case on two popular FPGA-based SoC, namely the Xilinx ZYNQ-7000 and ZYNQ Ultrascale+, considering the AXI SmartConnect, i.e., the state-of-the-art AXI interconnect for Xilinx platforms. The official documentation of the AXI SmartConnect [22] does not explicitly declare any mechanism that allows monitoring bus transactions and intervene whenever the bus is stalled. Stimulated by this lack of specifications, we were able to generate a stall with both read and write bus transactions. The case of write transactions is discussed next. The one of read transactions is similar and is omitted due to lack of space.

1) Stall with write transactions: The test-case is composed of two HW-tasks implemented in the FPGA fabric, HW_0 and HW_1, which access the memory subsystem in the PS through the AXI interconnect I_{AXI}. HW_0 and HW_1 are custom-developed AXI masters that work as traffic generators programmed by the PS to issue a given number of AXI transactions. An *integrated logical analyzer* (ILA) has been deployed in the FPGA fabric to monitor the AXI channels of both the master port (connected to the FPGA-PS interface) and the slave ports (each connected to a hardware accelerator) of I_{AXI}. When activated, each HW-task issues a request for a single write transaction. The waveform diagram in Figure 1 has been extracted from the ILA to study the bus signals during the execution of the test-case and is described next:

1) HW_0 issues an address request for a write transaction AW_0 on its master AW channel. The address request is sampled by I_{AXI} at the next clock cycle. Same behavior for HW_1, which issues an address request AW_1.
2) HW_1 starts providing the data W_1 to its master W channel, which are sampled by I_{AXI}. Differently, HW_0 does not provide the corresponding data W_0 to I_{AXI}.

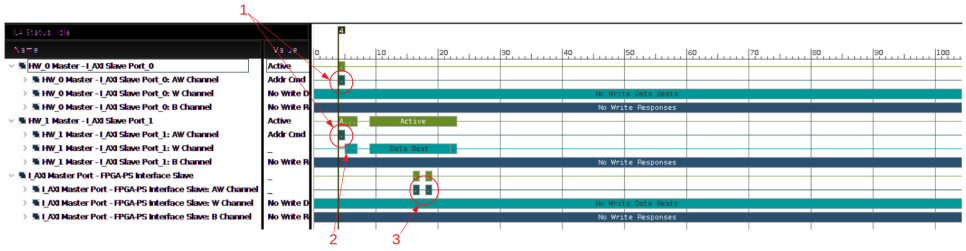

Fig. 1. Bus signals recorded with the ILA for the test-case. HW_0 never provides write data.

3) After a propagation delay, I_{AXI} provides the address requests (AW_0 and AW_1) on the AW channel of its master port. The arbitration phase is won by AW_0, which is propagated before AW_1. Even though the data of AW_1 have already been sampled by I_{AXI}, following the AXI standard specification, I_{AXI} cannot propagate them to the master port until the data corresponding to AW_0 are propagated. This happens because AXI forbids the interleaving of write transactions [26]. In other words, HW_1 cannot access the W channel until the data corresponding to AW_0 are provided by HW_0.

Consequently, *the delay with which HW_0 provides the data directly affects all the other HW-tasks*. Unfortunately, such a delay is not monitored by the interconnect I_{AXI}: indeed, HW_0 is free to delay the data provision for an unbounded time, with the result that the entire system *stalls* (as it happens in Fig. 1).

This example demonstrates that a single HW-task can inject arbitrary delays in the system, possibly causing the violation of the timing constraints of other HW-tasks. This is clearly a critical issue, especially when the system integrates third-party untrustworthy HW-tasks or is exposed to partial reconfiguration (a malicious HW-task can be programmed with the end of stalling the entire bus). Furthermore, also note that incomplete transactions initiated by a HW-task can trigger unpredictable behaviors in other bus components, both in the PS and in the FPGA fabric (e.g., interconnects and buffers), hence requiring a complete system reset to restore a safe condition.

V. Analytical System Model

This section presents an analytical model of the multi-master architecture introduced in Section III. The model is later used in Section VI to derive a worst-case timing analysis.

A. Hardware Tasks

The system comprises a set $\Gamma = \{HW_1, \ldots, HW_n\}$ of HW-tasks. Each HW-task HW_i accesses a private memory buffer in the DRAM memory and is periodically executed every T_i *ms*. Each periodic instance of HW_i (also referred to as *job*): (i) issues $N_i^R \geq 1$ read transactions and $N_i^W \geq 1$ write transactions, both with burst size B_i; (ii) computes for C_i clock cycles; and (iii) has a relative deadline equal to the period T_i (that is, each job must complete before the release of the next one). The number of outstanding transactions is set to ϕ_i, i.e., at any time, HW_i can have at most ϕ_i pending (that is, started but not completed) read transactions and ϕ_i pending write transactions. As long as HW-tasks behave correctly, we assume that they never stall any AXI channel. This statement is equivalent to the following assumptions: (i) read data are immediately sampled by the HW-tasks when provided by the FPGA-PS interface (i.e., no stall on the R channel); and (ii) write data are immediately provided to the W channel when the corresponding address request has been granted (i.e., no stall on the W channel). These nominal conditions are used in Section VI to compute the worst-case response times of HW-tasks when the system behaves correctly.

B. AXI interconnect and bus times

Each HW-task is connected to a corresponding slave input port of an AXI interconnect I_{AXI}, which in turn has a single master port connected to the FPGA-PS interface. The conflicts in address requests of the same type (read or write) and issued by different HW-tasks are managed by round-robin arbiters in I_{AXI}. The granularity of the round-robin arbiters is ϕ_I, that is, each HW-task is granted to issue at most ϕ_I read requests and ϕ_I write requests for each round-robin turn. Conversely, since I_{AXI} has a single master port, the traffic coming from it and directed to the slave ports does not experience any conflict. The AXI interconnect introduces the following delays in the propagation of address requests and data: (i) d_I^{addr} is the maximum latency to traverse I_{AXI} for an address request; (ii) d_I^{data} is the maximum latency to traverse I_{AXI} for a word of data (read or write); and (iii) d_I^{bresp} is the maximum latency to traverse I_{AXI} for a write response. These propagation delays can be derived from the specifications of the AXI interconnect (if available) or by means of experimental profiling.

Address and data occupation times are modeled with the following delay terms: (i) t_{addr} is the time occupied by the address request; (ii) t_{data} is the time occupied by a single data word; and (iii) t_{bresp} is the time occupied by the write response. The latter three delays are typically one clock cycle.

C. Processing System and Memory Controller

On many commercial platforms, the documentation of the internals of the DDR physical core block (located in the PS), which includes the memory controller, is not publicly available or not detailed. For these reasons, a fine-grained modeling of the DDR physical core block goes beyond the scope of this paper and is hence not addressed here. Therefore, this paper is based on a coarse-grained modeling by assuming that all the DDR-related logic in the PS introduces the following latencies, which refer to the conditions of maximum interference in accessing the DDR memory: (i) d_{PS}^{read} is the maximum latency introduced by the PS on a read transaction. It corresponds to the maximum time elapsed between the sample at the FPGA-PS interface of the address read request and the availability

at the FPGA-PS interface of the first word of corresponding data. (ii) d_{PS}^{write} is the latency introduced by the PS on a write transaction. It corresponds to the maximum time elapsed between the sample at the FPGA-PS interface of the last word of data of a write transaction and the availability of the corresponding write response at the FPGA-PS interface.

These parameters depend on the internal structure of the PS and can be quantified from the documentation of the SoC platform (if available) or by means of experimental profiling.

VI. Bounding the response time of HW-tasks

This section presents a timing analysis to bound the worst-case response time for each HW-task in the system in nominal condition (i.e., no delay due to stall in providing/reading the data is introduced). The worst-case response times are then used at the end of this section to check the schedulability of the HW-tasks, i.e., to verify whether the HW-tasks violate their deadlines, and to configure the ASM modules such that the schedulability is preserved even in the presence of misbehaving HW-tasks (Section VII). The reference architecture for the analysis is the one described in Section III. Response times are first bounded by studying the memory access time of a single transaction. Subsequently, an upper bound on the maximum number of interfering transactions is derived. Finally, we combine all together to bound the maximum time needed to complete a job of each HW-task.

A. Worst-case access time of a single transaction

Note that, due to the parallel channels offered by the AXI bus, write and read transactions can be studied independently at the stage of analysis.

1) Read Transactions: A read transaction R issued by a HW-task under analysis HW_{ua} begins by issuing an address read request R_{addr} to the AR channel of HW_{ua}'s master port, which is sampled by I_{AXI}. The time needed to issue an address request is t_{addr}. Request R_{addr} is then propagated through I_{AXI} and is sampled by the FPGA-PS interface. At this point, the PS routes R_{addr} to the memory controller. The first word of data is available at the FPGA-PS interface after at most d_{PS}^{read} time units. Now, read data cross I_{AXI} in reverse order with respect to R_{addr}, until HW_{ua} is reached. The propagation time on the data to cross I_{AXI} is d_I^{data}. Since data words are sequentially propagated (i.e., one data word after the other, each occupying t_{data} clock cycles), the propagation time d_I^{data}, and the latency d_{PS}^{read} are just paid once on all the data burst. It follows that the time to propagate a data burst of size B_{ua} words through I_{AXI} is equal to $d_I^{\text{data}} + B_{ua} \cdot t_{\text{data}}$. Overall, the memory access time d_{single}^R for a single read transaction is given by

$$d_{\text{single}}^R = t_{\text{addr}} + d_I^{\text{addr}} + d_{PS}^{\text{read}} + d_I^{\text{data}} + B_{ua} \cdot t_{\text{data}}. \quad (1)$$

2) Write Transactions: A write transaction W issued by a HW-task under analysis HW_{ua} begins by issuing an address write request W_{addr} to the AW channel of HW_{ua}'s master port, which is sampled by I_{AXI}. The time for the address request is t_{addr}. The address request W_{addr} is then propagated through I_{AXI}, until reaching the FPGA-PS interface. The written data W_{data} are sequentially propagated with W_{addr} through I_{AXI} the very next clock cycle after the sampling of W_{addr} by I_{AXI}, one word after the other. As data can be propagated only after the corresponding address is propagated, the overall propagation latency introduced by I_{AXI} on W_{addr} and W_{data} is given by the maximum between d_I^{addr} and d_I^{data}. After traversing I_{AXI}, W_{addr} and W_{data} can reach the FPGA-PS interface. In the PS, W_{addr} and W_{data} are then routed to the memory controller and a write response W_{resp} is finally issued after at most d_{PS}^{write} time units. Finally, W_{resp} is propagated through I_{AXI} to HW_{ua}, experiencing a latency of d_I^{bresp}. Recall that the data time for each word of data is denoted by t_{data} and the burst size is B_{ua} words (see Sec. V). Overall, it follows that the memory access time d_{single}^W for a single write transaction is given by

$$d_{\text{single}}^W = t_{\text{addr}} + \max(d_I^{\text{addr}}, d_I^{\text{data}}) + b_{ua} \cdot t_{\text{data}} + d_{PS}^{\text{write}} + t_{\text{bresp}} + d_I^{\text{bresp}}. \quad (2)$$

As a final remark, it is worth noting that the bounds for read and write transactions proposed in this section do not properly correspond to the case in which they are served in isolation: this is because, due to the reasons explained in Section V-C, the model always considers in d_{PS}^{read} and d_{PS}^{write} the worst-case latency at the memory controller, independently of the actual transactions issued by the HW-tasks.

B. Bounding the number of interfering transactions

To bound the contention delay incurred by a HW-task HW_{ua} (under analysis), it is required to bound the *number of transactions issued by other HW-tasks that can interfere with those issued by* HW_{ua}. We bound such a number in two different ways. First, we bound the maximum number of transactions that can interfere with a transaction issued by HW_{ua} at the AXI interconnect: the resulting bound multiplied by the total number of transactions issued by HW_{ua} yields a safe contention bound for HW_{ua}. Second, we bound the number of transactions that other HW-tasks can issue within HW_{ua}'s period (i.e., the largest time window in which HW_{ua} can be pending without missing its deadline). Both the bounds are safe but incomparable, and are hence alternative approaches that we combine to get a tighter bound. Note that the following results hold for both read and write transactions: therefore, to simplify the notation, we denote the number of transactions (either read or write) issued by any HW-task HW_i by just N_i (i.e., removing the superscript).

1) First bound: Consider a HW-task under analysis HW_{ua} that issues a single request for transaction r_{ua}. In the worst-case scenario, the AXI interconnect I_{AXI} grants r_{ua} at the last turn of the round-robin arbitration, i.e., after the requests for the transaction of the $n-1$ interfering HW-tasks HW_j in the system. From the model in Section V-A, the round-robin arbiters of I_{AXI} grants at most ϕ_I requests for transaction per HW-task for each round-robin cycle. However, still from the model, each HW_j can have at most ϕ_j outstanding transactions. Hence, I_{AXI} grants at most $\min(\phi_I, \phi_j)$ requests for transactions issued by HW_j for each round-robin cycle. Overall, summing up the contribution of all the HW_j, the total

number of interfering transactions granted before r_{ua} is upper bounded by $\sum_{HW_j \in \Gamma \setminus \{HW_{ua}\}} \min\{\phi_I, \phi_j\}$. Finally, since the latter equation holds for any transaction issued by HW_{ua}, the total number of interfering transactions with a job of HW_{ua} is bounded by

$$Y_{ua}^{(1)} = \sum_{HW_j \in \Gamma \setminus \{HW_{ua}\}} \min\{\phi_I, \phi_j\} \cdot N_{ua}. \quad (3)$$

2) Second bound: Consider a HW-task HW_{ua} (under analysis) and its corresponding period T_{ua}, and assume all jobs never execute after their deadlines. Without loss of generality, suppose that a periodic instance of HW_{ua} begins at time 0. Clearly, to interfere with HW_{ua}, a job of another HW-task HW_j must be released after time $-T_j$, otherwise it would be completed when HW_{ua} is released. Similarly, an interfering job of HW_j must be released before time T_{ua}, otherwise HW_{ua} would already be completed and hence no contention can be generated. As a result, the time window of interest to study the contention generated by HW_j to HW_{ua} is $(-T_j, T_{ua}]$ with length $T_j + T_{ua}$. In this window there are at most $\left\lceil \frac{T_{ua}+T_j}{T_j} \right\rceil$ jobs of HW_j. As each job of HW_j issues at most N_j transactions, there are at most $\left\lceil \frac{T_{ua}+T_j}{T_j} \right\rceil \cdot N_j$ transactions that can interfere with HW_{ua}. Summing up the contribution of all other HW-tasks $\neq HW_{ua}$, the maximum total number transactions which interfere with a job of HW_{ua} is upper bounded by

$$Y_{ua}^{(2)} = \sum_{HW_j \in \Gamma \setminus \{HW_{ua}\}} \left\lceil \frac{T_{ua}+T_j}{T_j} \right\rceil \cdot N_j. \quad (4)$$

3) Combining the two bounds: As discussed above, the two bounds are incomparable. Indeed, there are scenarios in which one can provide better results than the other, and vice-versa. For instance, the first bound would be quite pessimistic if HW_{ua} is interfered by a HW-task issuing just a few transactions, then it may be possible that not all HW_{ua}'s transactions incur in contention at the AXI interconnect. On the other hand, the first bound is very effective when the other HW-tasks issue a lot of transactions that can overlap with HW_{ua}'s execution, but HW_{ua} issues just a very few transactions. To take the best of the bounds they can be combined as follows:

$$Y_{ua} = \min\left\{Y_{ua}^{(1)}, Y_{ua}^{(2)}\right\}. \quad (5)$$

C. Response-time bounds

This section leverages the results of the two previous sections to compute an upper bound on the worst-case response times of HW-tasks. It bears repeating that the bounds in Equations (3) and (4) hold for both read and write transactions. Therefore, to ease the presentation of the following formulas, we denote by Y_{ua}^R and Y_{ua}^W the bound of Equation (5) where N_i is replaced with N_i^R and N_i^W ($\forall HW_i \in \Gamma$), respectively, in both Equations (3) and (4). The total delay incurred by a HW-task HW_{ua} (under analysis) in performing *read* transactions is given by (i) the time needed to complete its read transactions, plus (ii) the contention delay suffered by the latter. Following Section VI-A, the first time is bounded by $N_{ua}^R \cdot d_{\text{single}}^R$, whereas the contribution that each interfering transaction provides to the contention delay is upper bounded by d_{single}^R. Hence, the contention delay is bounded by $Y_{ua}^R \cdot d_{\text{single}}^R$. Overall, it follows that the total delay due to read transactions is bounded by

$$\Delta_{ua}^R = (N_{ua}^R + Y_{ua}^R) \cdot d_{\text{single}}^R. \quad (6)$$

Following the same reasoning, the total delay incurred by HW_{ua} in performing *write* transactions is bounded by

$$\Delta_{ua}^W = (N_{ua}^W + Y_{ua}^W) \cdot d_{\text{single}}^W. \quad (7)$$

Finally, the response time for each job of HW_{ua} is upper bounded by

$$RT_{ua} = \Delta_{ua}^R + C_{ua} + \Delta_{ua}^W. \quad (8)$$

A set of HW-tasks Γ is said *schedulable* if $\forall HW_{ua} \in \Gamma$, $RT_{ua} \leq T_{ua}$. As a final note, it is important to remark that the model presented in Section V-C considers the worst-case latency introduced on each single transaction in the terms d_{PS}^{read} and d_{PS}^{write}. Hence, even if read and write transactions can experience mutual interference at the memory controller when issued in parallel [15], the corresponding delay is already accounted in d_{PS}^{read} and d_{PS}^{write}. Consequently, with the respect to the modeling strategy adopted here, the worst-case response time of HW_{ua} occurs when read transactions, write transactions, and execution phases are serialized (i.e., no intra-task parallelism). This also allows obtaining a safe response-time bound that is general enough to cope with any possible release order and interleaving of the transactions at the memory controller. The analysis can be applied to any set of HW-tasks by setting the parameters of the above formulas, which have been implemented in an analysis tool within a driver (see next section). Sec. VIII shows how the analysis is applied to a case study.

VII. AXI Stall Monitor

This section presents the AXI Stall Monitor (ASM) module, a solution developed in HDL language to monitor the behavior of HW-tasks and intervene whenever they introduce dangerous delays during a bus transaction. The ASM has been exported as a standard IP module, and is hence easy to integrate in both existing and ex-novo applications. Each ASM module defines an AXI master interface and an AXI slave interface, and is placed between the HW-task (or a slot in a reconfigurable setting) to be supervised and the AXI interconnect. The ASM also exports an interrupt signal and a control AXI slave interface to set the ASM parameters. A sample architecture

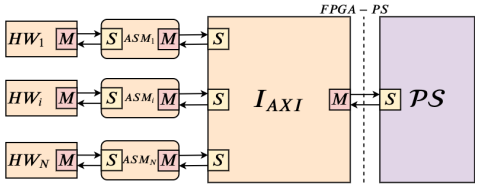

Fig. 2. A sample architecture with three AXI Stall Monitor modules.

comprising three ASM modules is shown in Figure 2. The

ASM behavior is based on the notion of *stalled clock cycle*, which occurs when at the rising edge of the clock at least one of the following conditions happens on the AXI channels monitored by the ASM: (i) there is at least one pending read transaction from the monitored HW-task, signal RVALID is high, and signal RREADY is low; (ii) there is at least one pending write transaction from the monitored HW-task, signal WREADY is high, and signal WVALID is low; or (iii) there is at least one pending write transaction from the monitored HW-task, signal BVALID is high, and signal BREADY is low. The ASM comprises an internal *counting logic* to keep track of stalled clock cycles, a *monitoring logic*, and a *decoupling logic*. A counter is *periodically* replenished to a given value, referred to as *budget*. Both the budget and the replenishment period are parameters that can be configured by the PS via memory-mapped registers. All the ASM modules share a common period: its synchronicity is given by a common external signal coming from the PS. Conversely, the budget is individually configured for each ASM. The ASM works according to two operating modes:

1) Monitor mode: The ASM is completely transparent to both the supervised HW-task and the AXI interconnect, and does not introduce any latency on the AXI channels. In this mode, the ASM supervises the AXI channel and, for each clock cycle stalled by the supervised HW-task, it decrements of one unit the internal counter. When the counter reaches zero (i.e., the budget is depleted), the ASM immediately switches to the decouple mode by taking the following actions: (i) raises an interrupt to the PS, signaling a misbehavior condition related to the supervised HW-task; (ii) decouples the supervised HW-task from the AXI channel (this shields the system from any further misbehavior of the monitored HW-task until the PS reacts to the interrupt); (iii) solves the eventual stall on read transactions by dropping the pending data at I_{AXI} directed to the supervised HW-task; (iv) solves the eventual stall on write transactions by completing the pending transaction(s) previously-issued by the supervised HW-task and never completed (according to the burst length of the stalled transaction(s)). In this case, the data provided by the ASM are dummy (the PS is notified) and the corresponding write response(s) are dropped by the ASM. Note that, since the memory buffers are private, a HW-task can compromise only the status of its private buffer when misbehaving. In other words, the misbehavior of a HW-task cannot corrupt the data of the other HW-tasks.

2) Decouple mode: In this mode, the ASM keeps the supervised HW-task physically decoupled from I_{AXI}. In decouple mode, the periodic replenishment of the counter does not take effect. The PS can let the ASM return to the monitor mode by acting on its memory-mapped control registers.

The interrupt signal raised during the monitor-to-decouple transition also acknowledges the PS that the work performed by the HW-task is not valid — as the HW-task is decoupled from the bus, its functional behavior is corrupted. The decision of whether readmitting the misbehaving HW-task in the system or not is left to the PS. Once the PS acknowledges the ASM to return to the monitor mode, the mode switch is delayed until the next replenishment time of the counter. Leaving the control to the PS makes the ASM versatile for different applications. For instance, in case a misbehavior is detected, the PS can reconfigure the slot hosting the misbehaving HW-task with another HW-task (possible in a reconfigurable setting), exclude a portion of the FPGA area (e.g., if the silicon is deemed bugged), or take other application-dependent actions.

A. Putting all together: configuring the ASM parameters

We present here a method, which leverages the analysis in section VI, to configure the ASM parameters (period and budgets) such that (i) the schedulability of all the HW-tasks is preserved, i.e., deadlines cannot be missed due to stalled clock cycles, and (ii) leave some flexibility to the HW-tasks such that they are free to introduce a *quantifiable* delay in providing/reading data during transactions. Differently from fixed timers to monitor stalls, this approach allows intervening only when a stall can effectively harm the timing constraints of the HW-tasks. To begin, we note that the delay related to stalled clock cycles can directly affect the worst-case response times of HW-tasks. Indeed, it can both increase the contention delay (when an interfering HW-task introduces some stalling) and the duration of the transactions of a HW-task under analysis. The HW-task with the shortest *slack* S_{min}, i.e., spare clock cycles between its worst-case response time and its deadline, is the one that can tolerate the shortest delay before harming the system schedulability. Hence, to preserve schedulability, the total delay due to stalled clock cycles must be lower than $S_{min} = \min_{HW_i \in \Gamma}\{T_i - RT_i\}$. Furthermore, it is worth observing that the impact of such a delay on worst-case response times is dependent on the replenishment period of the ASMs, i.e., fixing a budget, the shorter the period, the more stalled clock cycles can be generated without triggering the decouple mode. To control the impact of the ASMs on worst-case response times, we set the ASM period T to the maximum period among all HW-tasks, i.e., $T = \max_{HW_i \in \Gamma}(T_i)$. In this way, we are sure that, in the worst-case, at most one budget replenishment event e overlaps with the period of each HW-task. Consequently, a HW-task monitored by an ASM that is configured with a budget β can generate at most 2β stalled clock cycles within the period of another HW-task under analysis (β cycles before e and other β cycles afterwards).
Let β_i be the budget of the ASM that protects HW-task HW_i and let HW_{ua} be the (or one of the) HW-task(s) with the shortest slack S_{min}. In the worst-case, all HW-tasks can introduce the largest delays due to stalling while HW_{ua} is pending, which corresponds to their ASM budgets. Therefore, the system schedulability is preserved as long as $\sum_{HW_i \in \Gamma} 2\beta_i \leq S_{min}$. This implies that any ASM budget configuration that satisfies the following inequality is safe:

$$\sum_{HW_i \in \Gamma} \beta_i \leq \lfloor S_{min}/2 \rfloor .$$

The designer is free to assign the individual budget of each ASM in an arbitrarily manner as long as the above inequality

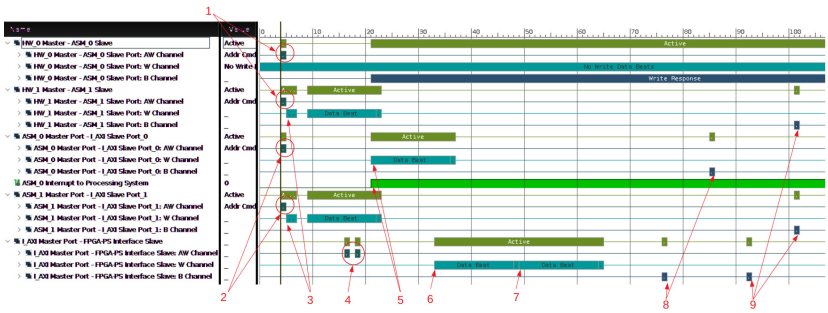

Fig. 3. Bus signals recorded with the ILA to show the behavior of the ASM on a write transaction.

is satisfied. The configuration of the ASM parameters can also be automatically managed via its driver (running in PS), which computes them by means of the results of Section VI. This also allows an easy reconfiguration of the ASMs when some HW-task changes its configuration (e.g., issue more transactions or change period) or is replaced due to partial reconfiguration.

B. The ASM in action

The test-case discussed in Section IV has been extended by introducing an ASM module for each HW-task (analogously as illustrated in Figure 2, with $N = 2$). Here, the two HW-tasks HW_0 and HW_1 are monitored by two ASM modules ASM_0 and ASM_1, respectively. The same experimental setting described in Section IV has been used to record the AXI signals with the ILA and the configuration in which HW_0 never provides data has been tested again. The resulting bus signals are illustrated in Figure 3, which is explained next:

1) HW_0 issues an address request for a write transaction AW_0 on its master AW channel. Same behavior for HW_1, which issues an address request AW_1.
2) AW_0 and AW_1 pass through ASM_0 and ASM_1, respectively, without any latency and are sampled by I_{AXI}.
3) HW_1 starts providing the data W_1 to its master W channel, which are propagated through ASM_1 (with no latency) and sampled by I_{AXI}. Differently, HW_0 does not provide the corresponding data W_0 to ASM_0. Hence, ASM_0 starts counting the stalled clock cycles introduced by HW_0.
4) After a propagation delay, I_{AXI} provides AW_0 and AW_1 to its master port, which are then sampled at the AXI slave port in the FPGA-PS interface. The arbitration phase is won by AW_0, which is propagated before AW_1.
5) HW_0 exhausted the budget of ASM_0. Therefore, ASM_0 switches to decouple mode and takes the control of the bus. ASM_0 starts providing the W_0 (dummy) data to correctly end the pending write transaction, bringing back the system to a safe condition. HW_0 is now decoupled from the bus until the PS sets the ASM_0 to switch back to monitor mode.
6) W_0 data are provided by I_{AXI} to its master port W channel. Data are sampled by the slave port in the FPGA-PS interface.
7) The stall on the bus is resolved. The data W_1 can finally be provided by I_{AXI} to its master port (W channel). Data are hence sampled by the slave port in the FPGA-PS interface.
8) The write response for AW_0 reaches the master port of I_{AXI} and is propagated to ASM_0. Since the transaction has been aborted, it is dropped by ASM_0 (i.e., it is not propagated to HW_0, which is decoupled from the bus).
9) The write response for AW_1 reaches the master port of I_{AXI} and is propagated to HW_1, completing the write transaction (no latency introduced by the ASM).

VIII. Experimental results

The experiments are focused on a case study in which temporal guarantees must be provided. The target platforms are two common off-the-shelf FPGA SoC: the Xilinx Zynq-7000 (Z-7020) and the Xilinx Zynq Ultrascale+ (ZU9EG). Since the results on the two SoC are comparable, due to lack of space we just report the data from the ZYNQ-7000 platform.

A. Experimental setup

The experimental architecture is composed of three common hardware accelerators IP from the Xilinx IP library: the AXI Fast Fourier Transform (HW_{FFT}) [28], the AXI Direct Memory Access (HW_{DMA}) [29], and the AXI Finite Implulse Response filter (HW_{FIR}) [30]. These hardware accelerators are the considered HW-tasks for this experiment. The three HW-tasks are monitored by ASM modules named ASM_{FFT}, ASM_{DMA} and ASM_{FIR}, respectively. The HW-tasks are connected to a Xilinx AXI SmartConnect [22] (I_{AXI}), which provides them the access to the PS. The architecture corresponds to the one reported in Figure 2, with $N = 3$.
Each HW-task is managed by a periodic software (SW) task (SW_{FFT}, SW_{DMA}, and SW_{FIR}, respectively), running in the PS, on top of the FreeRTOS operating system. Each SW-task is in charge of configuring the corresponding HW-task and activating it (i.e., triggering a job).

The experimental architecture is completed with a Xilinx System Integrated Logic Analyzer (ILA) [31], which is used to profile: (i) the propagation latency of the AXI SmartConnect (ii) d_{PS}^{read} and d_{PS}^{write} (due to the lack of detailed information about the DDR Controller in the documentation [32] [27]); and (iii) the parameters ϕ and C (that depend on intrinsic properties of the Xilinx IPs). Table I reports the configuration of the HW-tasks for the experiments[1] Note that parameters N_R, N_W, and ϕ are quantified in number of transactions,

[1] HW_{FFT} transforms four data windows of 64KB each. Hence, C_{FFT} is obtained by profiling the per-window execution time and multiplying it by four. C_{DMA} and C_{FIR} have been quantified by profiling the largest time needed by HW_{DMA} and HW_{FIR} to perform the computation required by one memory transaction and then multiplying it by the number of per-job transactions.

TABLE I
CONFIGURATION OF THE HW-TASKS

	N_R	N_W	C	ϕ	B	T
HW_{FFT}	4096	4096	$4 \cdot 201$	6	16	50
HW_{DMA}	256	256	$256 \cdot 101$	6	16	20
HW_{FIR}	8192	8192	$8192 \cdot 103$	6	16	30

parameters C and B in clock cycles, and parameters T in milliseconds. Table II reports the profiled value of ϕ_I and the maximum propagation latency measured on the platform. Finally, table III reports the resource consumption of an ASM module IP on the two platforms considered in this work, compared against the total amount of available resources.

TABLE II
PROFILED VALUE FOR AXI SMARTCONNECT AND PS.

ϕ_I	d_I^{addr}	d_I^{data}	d_I^{bresp}	d_{PS}^{read}	d_{PS}^{write}
1	12	9	9	50	40

B. Finding the ASM overall budget and period

This section describes the steps performed by the driver to configure the ASM parameters by leveraging the analysis in Section VI and the considerations made in Section VII-A. By Equation (5), the maximum number of interfering transactions for each HW-task are $Y^R_{\text{FFT}} = 5120$, $Y^W_{\text{FFT}} = 5120$, $Y^R_{\text{DMA}} = 512$, $Y^W_{\text{DMA}} = 512$, $Y^R_{\text{FIR}} = 8960$, and $Y^W_{\text{FIR}} = 8960$. Hence, according to Section VI, the upper bound on the response time of each HW-task are $RT_{\text{FFT}} = 12.6ms$, $RT_{\text{DMA}} = 1.3ms$, and $RT_{\text{FIR}} = 24.2ms$. Since $RT_{\text{FFT}} < T_{\text{FFT}}$, $RT_{\text{DMA}} < T_{\text{DMA}}$, and $RT_{\text{FIR}} < T_{\text{FIR}}$, the set of HW-tasks is deemed schedulable. Now, according to Section VII, the safe overall stall budget is equal to the minimum between the slacks $S_{\text{FFT}}/2$, $S_{\text{DMA}}/2$, and $S_{\text{FIR}}/2$, and hence equal to $17.4/2 = 8.7ms$. The replenishment period of the ASMs is the maximum period of the HW-tasks in the system, i.e., $50ms$.

C. Spreading the ASM overall budget

Two strategies for assigning the ASM budgets were tested: (a) linearly spreading the overall stall budget as a function of the HW-tasks' periods (the higher the HW-task period, the higher the assigned ASM budget) and (b) spreading the overall budget according to the criticality of the HW-tasks, giving 90% of the overall budget to HW_{DMA} (here assumed to be the most critical) while linearly splitting the remaining 10% between HW_{FFT} and HW_{FIR}, according to the length of their period. The experiments have been conducted at first in nominal conditions, i.e., all the HW-tasks well-behave and the ASM modules are turned off (experiment (i)). Then, keeping the ASMs off, HW_{DMA} is made misbehaving (experiment (ii)). In this condition, HW_{DMA} issues requests for write transactions and never provides the corresponding data. Subsequently, the ASM modules are switched on and the case in which all the HW-tasks in the system well-behave (experiment (iii)) and the case in which HW_{DMA} misbehaves (experiment (iv)) are tested. The results are reported in Figures 4(a) and 4(b),

TABLE III
RESOURCE UTILIZATION OF AN ASM.

SoC	Resources		
	LUT	**FF**	**BRAM/DSP**
Z-7020	185/53200 (0.35%)	203/106400 (0.19%)	0
ZU9EG	183/274080 (0.07%)	203/548160 (0.04%)	0

respectively, also compared against the theoretical bounds derived in Section VIII-B (note that the y-axis is in log scale). In experiment (ii), as expected (see Section IV), the misbehavior of HW_{DMA} stall the entire system, letting all the other HW-tasks experiencing an *unbounded* delay in the response time. Similarly, in experiment (iv), HW_{DMA} misbehaves and thus does not complete its execution. Hence, no data are reported in Figure 4 for exp. (ii) and for HW_{DMA} in exp. (iv).
The measured response times in experiments (i) and (iii) for all the HW-tasks are equivalent. This confirms that the ASMs do not introduce latency. In experiment (iv), the misbehavior of HW_{DMA} increases the response times of both HW_{FFT} and HW_{FIR}. The measured response times in experiment (iv) for both HW_{FFT} and HW_{FIR} are larger in Figure 4(b) than in Figure 4(a). This is because in configuration (b) HW_{DMA} disposes of a larger stall budget and hence is free to delay the bus for more time with respect to configuration (a). However, in both the configurations, thanks to ASM_{DMA}, the schedulability of HW_{FFT} and HW_{FIR} is not endangered: indeed, they can both complete within their corresponding deadlines, regardless of the HW_{DMA}'s misbehavior.

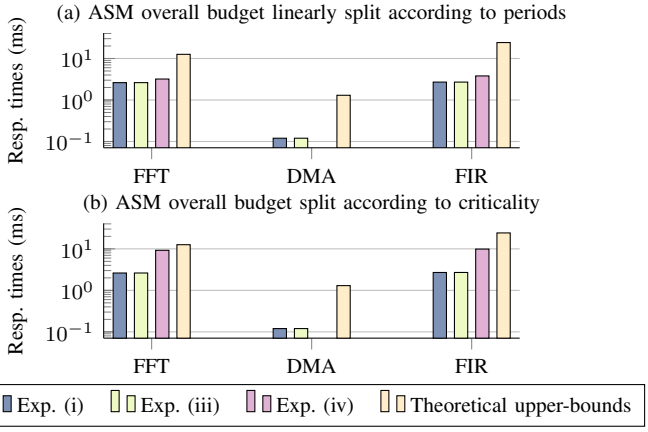

Fig. 4. Maximum measured response times.

IX. CONCLUSIONS

Modern FPGA-based SoCs rely on the AXI open standard, which has been conceived under the assumption that all the modules connected to the bus behave correctly. This paper showed how this assumption can lead to unpredictable delays or even to starvation. To address this issue, a timing analysis is proposed to bound the worst-case response time of HW-tasks. The analysis is then implemented in a driver that configures the AXI Stall Monitor (ASM) modules to control delays during bus transactions while preserving the system schedulability. Experimental results demonstrated the practical applicability and the effectiveness of the ASM on real platforms.

REFERENCES

[1] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[2] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69–76, 2008.

[3] S. R. Faraji, M. H. Najafi, B. Li, D. J. Lilja, and K. Bazargan, "Energy-efficient convolutional neural networks with deterministic bit-stream processing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1757–1762.

[4] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65–74.

[5] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 33–42. [Online]. Available: https://doi.org/10.1145/3289602.3293904

[6] Q. Gautier, A. Althoff, and R. Kastner, "Fpga architectures for real-time dense slam," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 83–90.

[7] G. G. Lemieux, J. Edwards, J. Vandergriendt, A. Severance, R. De Iaco, A. Raouf, H. Osman, T. Watzka, and S. Singh, "TinBiNN: Tiny binarized neural network overlay in about 5,000 4-LUTs and 5mw," *arXiv preprint arXiv:1903.06630*, 2019.

[8] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 15–24.

[9] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in *Real-Time Systems Symposium (RTSS), 2016 IEEE*. IEEE, 2016, pp. 1–12.

[10] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, 2006.

[11] T. Dörr, T. Sandmann, F. Schade, F. K. Bapp, and J. Becker, "Leveraging the partial reconfiguration capability of FPGAs for processor-based fail-operational systems," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2019, pp. 96–111.

[12] A. M. Johnson Jr and M. Malek, "Survey of software tools for evaluating reliability, availability, and serviceability," *ACM Computing Surveys (CSUR)*, vol. 20, no. 4, pp. 227–269, 1988.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2015.

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

[15] 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.

[16] B. Forsberg, A. Marongiu, and L. Benini, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 318–321.

[17] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: FPGA nocs with provably stall-free FIFOs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 222–231.

[18] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[19] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: Weak behaviours and programming assumptions," in *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 577–591.

[20] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling mixed-criticality in SoC-based real-time embedded systems," in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 235–244.

[21] *Avalon Interface Specifications*, Intel FPGA, 2018, mNL-AVABUSREF.

[22] *SmartConnect, LogiCORE IP Product Guide*, Xilinx, 2018, pG247.

[23] *AXI Performance Monitor v5.0*, Xilinx, 2017, pG037.

[24] H.-m. Kyung, G.-h. Park, J. W. Kwak, T.-j. Kim, and S.-B. Park, "Design and implementation of performance analysis unit (pau) for axi-based multi-core system on chip (soc)," *microprocessors and microsystems*, vol. 34, no. 2-4, pp. 102–116, 2010.

[25] A. Moro, F. Federici, G. Valente, L. Pomante, M. Faccio, and V. Muttillo, "Hardware performance sniffers for embedded systems profiling," in *2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)*. IEEE, 2015, pp. 29–34.

[26] *AMBA AXI and ACE Protocol Specification*, ARM, 2012.

[27] *Zynq UltraScale+ Device - Reference Manual*, Xilinx, 12 2017, uG1085.

[28] *Fast Fourier Transform, LogiCORE IP Product Guide*, Xilinx, 2018, pG109.

[29] *AXI Central Direct Memory Access, LogiCORE IP Product Guide*, Xilinx, 2018, pG034.

[30] *FIR Compiler, LogiCORE IP Product Guide*, Xilinx, 2018, pG149.

[31] *System Integrated Logic Analyzer v1.0*, Xilinx, 2017, pG261.

[32] *Zynq-7000 - Reference Manual*, Xilinx, 9 2016, uG585.