A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs

Marco Pagani
Scuola Superiore Sant'Anna, Pisa, Italy
Université de Lille, CNRS, Centrale Lille, UMR 9189, CRIStAL, Lille, France
marco.pagani@santannapisa.it

Enrico Rossi
Scuola Superiore Sant'Anna, Pisa, Italy
enrico.rossi@santannapisa.it

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

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

Giuseppe Lipari
Université de Lille, CNRS, Centrale Lille, UMR 9189, CRIStAL, Lille, France
giuseppe.lipari@univ-lille.fr

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

Abstract

Hardware platforms for real-time embedded systems are evolving towards heterogeneous architectures comprising different types of processing cores and dedicated hardware accelerators, which can be implemented on silicon or dynamically deployed on FPGA fabric. Such accelerators typically access a shared memory to exchange a significant amount of data with other processing elements. Existing COTS solutions focus on maximizing the overall throughput of the system, rather than guaranteeing the timing constraints of individual hardware accelerators. This paper presents the AXI budgeting unit (ABU), a hardware-based solution to implement a bandwidth reservation mechanism on top of the AMBA AXI standard infrastructure for hardware accelerators deployed on FPGAs. An accurate and tractable model, as well as the corresponding analysis, are also proposed to bound the response time of hardware accelerators in the presence of ABUs, in order to verify whether they can complete before their deadlines. Finally, a set of experiments are reported to evaluate the proposed approach on a state-of-the-art platform, namely the Zynq-7020 by Xilinx. The resource consumption of the ABU has been quantified to be less than 1% of the total FPGA resources of the Zynq-7020.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → System on a chip; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases AXI Bus, Bandwidth Reservation, Hardware Acceleration, FPGA

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.24

1 Introduction

Current computer architectures are evolving towards heterogeneous platforms consisting of different processing elements including general-purpose processing cores, graphics processing cores with general-purpose capabilities, and dedicated hardware accelerators [13]. Moreover, some popular modern SoCs platforms, like Altera's Stratix 10 SX [20] and Xilinx's Zynq

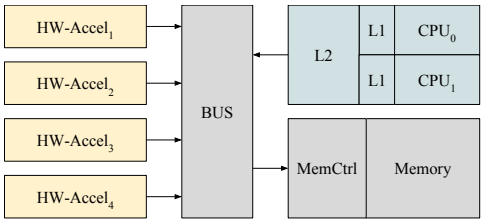

Figure 1 Block diagram of a custom system deployed on a SoC-FPGA platform.

UltraScale+ [39], include a reconfigurable FPGA fabric tightly coupled with general purpose processing elements. This feature vastly extends the capability of these platforms to allow offloading intensive computational activities from the general-purpose processing elements to custom hardware accelerators deployed on the FPGA fabric.

With respect to other types of hardware acceleration, like GPU co-processing, FPGA acceleration allows for precise control of the logic design, resulting in a very predictable behavior of the accelerators and allowing for an accurate estimation of the worst-case execution time [14, 28]. Such characteristics have made FPGA-based acceleration attractive in several safety-critical domains for signal processing and many other computationally-intensive dataflow applications [19, 21]. To name a relevant application of FPGA-based acceleration, these features enable the efficient execution of machine learning algorithms and convolutional neural networks [38] on embedded devices for safety-critical applications, as robotics and automotive.

Hardware accelerators are typically *memory-intensive*, high-performance units capable of autonomously retrieving data from the system memory using direct memory access (DMA) or bus mastering techniques. Each hardware accelerator is implemented using a subset of the FPGA's logic resources that are reserved only to that specific accelerator. Therefore the execution units of accelerators are completely independent from each other and can operate in parallel. For this reason, the execution time of a hardware accelerator depends only on the input data and the available bus and memory bandwidth. Clearly, in the context of a system comprising multiple hardware accelerators, like the one shown in Figure 1, bus/memory contention becomes the dominant factor in determining the response time of the accelerators. If the effects of such a contention are not taken into account, the system execution becomes unpredictable and hardware accelerators may introduce interferences that can jeopardize the entire system.

This scenario is worsened by the fact that often it is not possible for a designer to control the actual bus demand rate of each accelerator deployed on the system. For instance, if the accelerator is available in the form of a closed IP, it may be impossible to tune the actual rate at which bus transactions are issued. Another aspect to consider is the increasing relevance that high-level synthesis (HLS) is gaining in the design of hardware accelerators for FPGAs [26, 11]. While these tools allow for a significant speedup of the hardware design process, they lack the precise control over the design that a register-transfer level (RTL) implementation can achieve. This effectively reduces the possibility for the designer to precisely tune the rate of bus transactions. Finally, hardware accelerators can be plagued by design issues and bugs that may lead to execution overruns or illegal memory accesses.

To mitigate these issues, some hardware vendors typically integrate traditional priority-based arbitration in their interconnect implementations. More recent FPGA platforms also include (limited) mechanisms for QoS-aware arbitration [40]. However, the closed source nature of these implementations, often paired with an opaque description of the internals,

makes it difficult to model such closed IPs and derive formal properties. In fact, the limited flexibility of those mechanisms and the lack of a proper reservation policy make them unsuited for safety-critical environments.

These challenges could be tackled by a methodology that enforces a more predictable environment, allowing for a controlled integration of first and third-party accelerators. As modern operating systems provide isolation and supervision mechanisms for software processes, it is worth providing supervision and reservation mechanisms also for the hardware activities performed by accelerators. This would enhance system predictability and enable the FPGA acceleration paradigm to be effectively used in safety-critical applications.

1.1 Contributions

This paper makes the following contributions.

- First, it proposes the *AXI Budgeting Unit* (ABU), which is a custom hardware component realized in programmable logic that provides bus bandwidth reservation for hardware accelerators deployed on FPGAs. An ABU *shields* a hardware accelerator from possible misbehaviors of other accelerators (in terms of exceeding bus data transfers) by *predictably* enforcing a given bus bandwidth. The ABU is not a bus arbiter but a *traffic shaper* component to be placed between hardware accelerators and a standard AMBA AXI bus infrastructure. ABUs can seamlessly be integrated into any FPGA design on top of the proprietary AXI Interconnect provided by vendors. This approach reduces the development costs and enhances portability and compatibility with any future releases of AXI-compliant IPs. ABUs have been implemented and tested upon state-of-the-art FPGA-based system-on-chips. The resource consumption of an ABU has also been quantified in less than 1% of the total FPGA resources on a Zynq-7020 platform by Xilinx.
- Second, after presenting a model for hardware accelerators based on the characteristics of realistic implementations (from Xilinx IP libraries and OpenCV), the paper proposes an analysis to bound the response times of hardware accelerators. The analysis is performed in the bus bandwidth domain and results to be tractable, as well as accurate to study FPGA-based hardware accelerators.
- Third, the paper reports a set of experimental results conducted on the Zynq-7020 aimed at demonstrating **(i)** the effectiveness of the reservation mechanism implemented by ABUs, even in the presence of misbehaving hardware accelerators, and **(ii)** the validity of the proposed analysis.

The rest of the paper is organized as follows. Section 2 presents the system model and the essential background. Section 3 presents the ABUs. Section 4 illustrates the problem of analyzing hardware tasks in the bandwidth domain and highlights crucial analysis issues. Section 5 shows how ABUs can be leveraged to analyze the system. Section 6 reports on the experimental evaluation. Section 7 reviews the related work and finally Section 8 states our conclusions.

2 System model and Background

This work focuses on FPGA-based system-on-chips and considers an AXI system composed of an interconnect, a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of hardware accelerators, and a shared sink module (e.g., a memory controller). The hardware accelerators are implemented as AXI memory-mapped master modules capable of autonomously accessing data in a shared memory, which

is reachable through the sink. Each accelerator performs a specific computational activity, therefore, from now on, they will be referred to as *hardware tasks* (HW-tasks). All HW-tasks are connected to an Interconnect block, which in turn is connected to the sink module.

The next subsections introduce a model for the Interconnect together with some essential background related to the AXI bus, a model of the HW-tasks, and a model of the sink module. It is important to note that *most of the assumptions reported in this section are only adopted for the purpose of analyzing the system* (Section 4), while the system-level mechanism proposed in this paper (Section 3) – i.e., the ABU – is independent of most of the adopted modeling strategies.

2.1 AXI Interconnect

The central element of an AXI-based system is the AXI Interconnect, which acts like a "switch" connecting one or more AXI master devices to one or more slave devices. The Interconnect performs crucial activities such as protocol conversions and the arbitration of memory transactions. In this work, the Interconnect is assumed to be configured in a N-to-1 mode, i.e., it connects $N \geq 1$ masters to a single slave device such as a memory controller. Under this setting, the Interconnect is in charge of arbitrating the transactions issued by the master modules.

2.1.1 Arbitration policy

The AXI specification [5] does not mandate any specific arbitration protocol for the Interconnect. Some implementations of the Interconnect, such as the Xilinx standard Interconnect IP [41], provide two arbitration modes: (i) fixed-priority scheduling, in which the user configures static priorities for the slave ports, and (ii) a fair allocation using round robin. In recent releases of the Vivado suite, Xilinx provides the new SmartConnect IP [45] (meant to replace the current Interconnect IP in new designs) in which the fixed-priority arbitration has been dropped retaining the round-robin arbitration only. Hence, to match realistic modern designs, this work only focuses on round-robin arbitration. In addition, it is assumed that the Interconnect **(i)** implements *ideal round-robin scheduling* with reclaiming, i.e., the unused bandwidth is fairly re-distributed by the contenders that demand more than the fair bandwidth share, and **(ii)** does not introduce any overhead. Note that the actual implementation of the round-robin policy is typically not known, e.g., as it is the case of the Xilinx IPs, which are closed-source and lack of a proper detailed documentation concerning arbitration policies. As a result, a more accurate modeling of the arbitration may be difficult to obtain and may introduce inconsistencies among different versions of the IPs. Nevertheless, the experimental results carried out in this work surprisingly revealed a *marginal* deviation of behavior of the Xilinx Interconnects with respect to the ideal case (see Section 6).

2.1.2 AXI Links

An AXI link provides a bidirectional connection between a master and a slave interface. Each AXI link comprises five independent transaction channels: two channels (read address and read data) for read transactions, and three channels (write address, write data, and write response) for write transactions. Each channel implements a two-way handshake mechanism by using a pair of VALID and READY signals. The producer generates the VALID signal to indicate when the address or data are available. The consumer generates the READY signal to indicate that it can accept the information. The actual transfer occurs only when both

the VALID and READY signals are asserted. In this paper, to distinguish between READY and VALID signals of read and write transactions, the letters R and W are appended before their names (e.g., RREADY and WREADY).

Read and write channels of a link can operate independently one from each other, i.e., each HW-task may perform read and write transactions concurrently. However, the AXI specification [5] does not mandate how the Interconnect should manage such a level of concurrency among channel groups. In this work, it is assumed that the Interconnect arbitrates read and write channel groups independently, thus permitting concurrent read and write transactions from master modules. For instance, note that both the standard and smart Interconnect IPs provided by Xilinx can operate in this mode [41], [45].

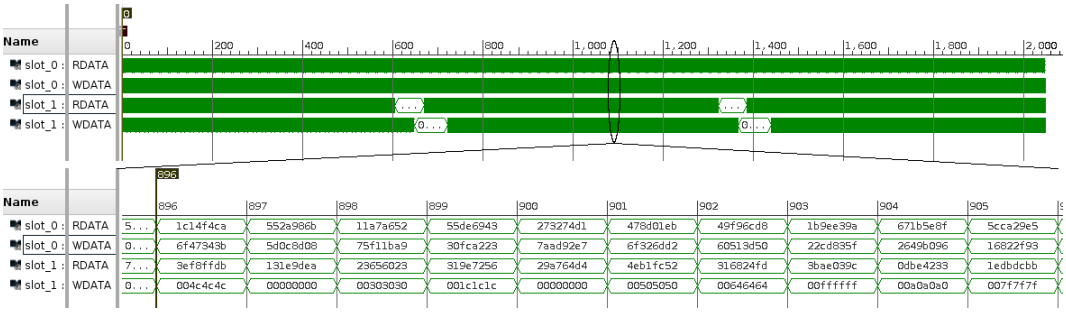

Figure 2 Screenshots of bus signals for read and write memory transactions of two HW-tasks (FIR and SOBEL filters), taken from the Vivado tool by Xilinx. The HW-tasks are implemented with High Level Synthesis upon a Xilinx Zynq-7020 platform. The figure also reports a zoom of about 10 clock cycles.

2.2 HW-tasks

All HW-tasks are periodically activated, and thus generate a potentially-infinite sequence of execution instances (also referred to as *jobs*). Each HW-task operates like a DMA module, generating an equal number of read and write transactions. The transactions issued by each HW-task are assumed to be uniformly distributed during its execution and hence issued at a fixed rate. Please observe that, despite this modeling strategy may seem coarse, many real-world hardware accelerators that perform data-parallel operations (e.g., video, image, and signal processing on raw data) *present regular memory access patterns that can be modeled with a uniform demand.* As a representative example, Figure 2 reports the bus signals for memory transactions of two state-of-the-art HW-tasks, namely a FIR filter (slot0 in the figure) and a Sobel filter from the OpenCV library (slot1 in the figure). The trace at the top of the figure reports the execution of the 0.76% and the 0.6% of a job of the two HW-tasks, respectively. The HW-tasks have been implemented with high-level synthesis (HLS) upon a Xilinx Zynq-7020 platform. As it can be noted from the figure, the FIR filter exhibits a uniform pattern of transactions (one 32-bit word per clock cycle); the same holds for the Sobel filter, with the exception of a few clock cycles every about 600 clock cycles (the stop is attributed to the end of the processing of a row of the input image). Across all its execution, the amount of clock cycles in which the Sobel filter does not issue bus transactions corresponds to less than the 10%. Nevertheless, please observe that for the purpose of analysis the Sobel filter can still be pessimistically modeled by assuming that bus transactions are issued even in the last 600 clock cycles: further details on this strategy are discussed in Section 6.3.

Formally, each HW-task τ_i is characterized by the following three parameters: **(i)** a *demand rate* D_i, which represents the rate of memory transactions (both reads and writes), **(ii)** the maximum number N_i of memory transactions issued by each job, and **(iii)** its period T_i. Due to the presence of separate channels for reads and writes, the demand rate of each HW-task is bounded by two transactions per clock cycle. Demand rates are typically expressed as number of transactions per clock cycle; when needed, a word size (such as 32-bit) may also be used in place of the number of transactions. It is very important to note that HW-tasks have very *different characteristics with respect to classical software tasks*. Indeed, HW-tasks have an intrinsic parallel execution and are usually implemented such that they can perform computations while issuing memory transactions (i.e., computations and memory accesses are overlapped in time). For instance, this fact can also be observed from Figure 2, as the hardware accelerators issue memory transactions at (almost) every clock cycle. For this reason, computations times are not modeled and HW-tasks are assumed to be completed when they complete all their N_i memory transactions.

2.3 Sink module

The sink module models an endpoint block like a memory controller or a downstream AXI Interconnect (e.g., in the presence of multiple Interconnects that are connected in a hierarchical manner). Formally, the sink module is modeled with a *supply bandwidth* S that denotes the total rate of transactions it can accept, i.e., the maximum ratio of read and write transactions served per clock cycle.

It is worth mentioning that the size, in bytes, of a single transaction may vary even on the same system depending on how the AXI logic has been implemented on each module. Actually, the AXI standard allows connecting multiple hardware modules with different transaction word sizes, or even protocol version; the Interconnect is then responsible to convert the format of transactions. For instance, the High-Performance ports included in the Zynq platforms by Xilinx to access DDR memories dispose of a supply rate of two double-word (64-bit) transactions per clock cycle, while the default configuration of AXI master ports for hardware accelerators uses single-word transactions. In this paper, when it is necessary to avoid possible inconsistencies, demand and supply rates are always expressed by using the smallest word in the system.

3 AXI Budgeting Unit

This work proposes an infrastructure that comprises a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of ABU modules controlled by a central unit named *ABU controller*. Each ABU module is conceived to be placed between a hardware accelerator and the remainder of the bus infrastructure. A sample setup is shown in Figure 3. The purpose of each ABU module is to *supervise* the bus traffic generated by the corresponding hardware accelerator providing both temporal and spatial isolation. Specifically, the objectives of ABUs are:

- implementing a memory bandwidth reservation mechanism by **(i)** keeping track of the number of bus transactions issued by HW-tasks, and **(ii)** enforcing a maximum *budget* of transaction within periodic time windows; and
- as a side feature, implementing a memory protection mechanism that restricts the address space accessible by HW-tasks to a set of configurable regions.

The ABU controller serves as a central control point that allows programming the ABU modules by means of memory-mapped registers exposed through a single AXI slave interface. In its typical usage, such memory-mapped registers are controlled by the CPU (e.g., by

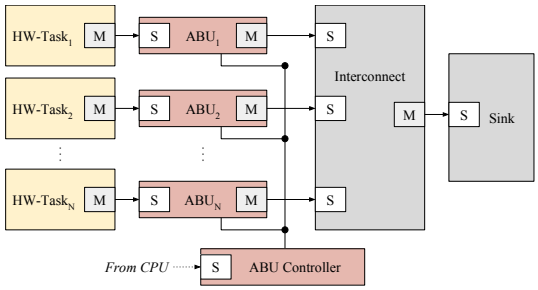

Figure 3 Illustration of an AXI system with hardware accelerators protected by ABUs. The boxes labeled with M and S denote master and slave AXI ports, respectively.

a driver at the level of the operating system or a hypervisor). The ABU modules are in turn connected to the ABU controller through a custom bus, which is used to transfer configuration parameters and signals. As it is illustrated in Figure 3, each ABU module also exports one AXI master and one AXI slave interface. The AXI slave interface serves as the access point for the hardware accelerator, while the AXI master port is meant to be connected to the remainder of the bus. These components are implemented in VHDL using a RTL behavioral description and deployed onto the FPGA fabric.

Working principle. According to the AXI standard, the master modules are the ones in charge of initiating bus transactions. Consequently, the HW-tasks drive the system by concurrently performing requests for bus transactions to the Interconnect, which in turn selects which pending transactions need to be propagated to the sink. The main idea behind the budgeting mechanism of ABUs is to act as a *proxy* between HW-tasks and the Interconnect by monitoring and altering the AXI signals. An example of a ABU in action is shown in Figure 4 for the case of a HW-task that performs a set of write transactions. The figure reports the state of the AXI signals that are relevant for the considered examples, namely WVALID in output from the HW-task and the ABU (first and second rows, respectively), WREADY in output from the Sink and the ABU (third and fourth rows, respectively), and WDATA to show the data traffic on the bus (last row). The evolution of the ABU budget over time is also reported at the top of the figure. As it can be observed from the figure, when the ABU budget ends at time t_1, write transactions are blocked despite the HW-task is ready to transmit data (WVALID in output from the HW-task is up) and the Sink is ready to receive it (WREADY in output from the Sink is up). This is accomplished by masking signals WVALID and WREADY forcing their logic state to zero, as it is illustrated in the second and fourth rows in the figure within time interval $[t_1, t_2]$. Note that, when no budget exhaustion occurs, the ABU has a transparent behavior mirroring all signals (see time window $[t_2, t_3]$ in the figure).

Budgeting mechanism. For each ABU A_i, the proposed solution allows configuring **(i)** a maximum budget B_i of number of transactions, and **(ii)** a period P_i with which the budget is replenished. Each ABU also keeps track of a variable parameter denoted as *instantaneous budget* b_i. At the system startup, $b_i = B_i, \forall i = 1, \ldots, n$. Then, as a HW-task performs bus transactions, the instantaneous budget is decremented until it reaches zero (budget depletion). As long as its instantaneous budget is zero, an ABU forbids bus transactions by acting on (R/W)VALID and (R/W)READY data and address signals. The instantaneous budget is recharged in a periodic and synchronous manner, i.e., if the system startup corresponds to

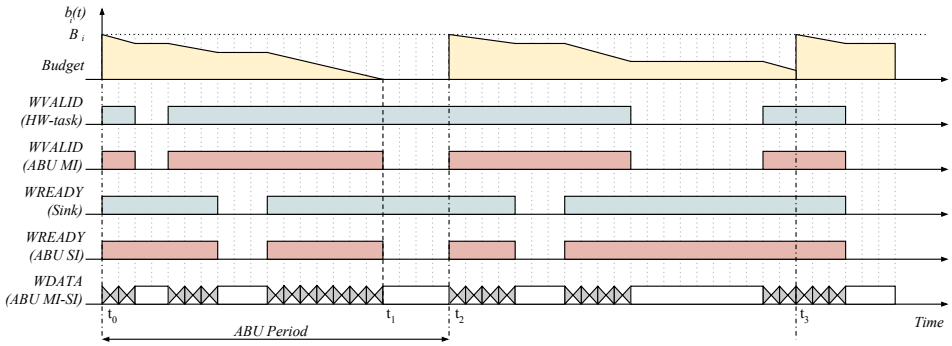

Figure 4 Example of ABU in action: impact on the AXI bus signals.

time $t = 0$, the instantaneous budget of A_i is set to $b_i = B_i$ at every time $t = kT_i,\ k \in \mathbb{N}$. From the perspective of memory bandwidth, note that each ABU enforces a transaction rate B_i/P_i for the corresponding HW-task *independently of the behavior of the latter*.

Memory protection. The ABU controller allows configuring X memory address regions for each ABU A_i to which the corresponding HW-task is allowed to access. Each of such regions $r_{i,j}$ (with $j = 1, \ldots, X$) is identified with a base memory address and a size, which are configurable by means of memory-mapped registers offered by the controller. Whenever a HW-task τ_i performs an access outside one of the regions $r_{1,1}, \ldots, r_{i,X}$, the corresponding ABU A_i blocks all memory transactions of τ_i, as it would be disconnected from the bus; consequently, the ABU controller raises an interrupt signal. The HW-task that triggered the fault can be identified by reading a status register of the controller. The normal operation of the ABU can be restored by acting on another control register offered by the controller. This feature is particularly useful in the context of virtualized systems, where a hypervisor running on the CPU of the system-on-chip can configure the memory regions and react to illegal memory accesses.

ABU internals. The internal architecture of an ABU module is illustrated in Figure 5. The communication channels on the AXI link between the master and the slave interfaces are routed through a *decoupler* block that can stop the master from issuing transactions. The decoupler works by acting on the *ready* and *valid* signals to temporarily suspend the handshake procedure. The budgeting mechanism is implemented by means of a transaction counter that keeps track of each read/write transaction and, when the budget is exhausted, sends a signal to activate the decoupler block. The ABU controller provides a pair of registers for configuring the budget and the period of each ABU. Such registers are accessible as memory-mapped via the AXI slave interface of the controller. The memory protection function is implemented by comparing the values on the read and write address channels with the range of addresses specified for each region $r_{i,j}$.

Note that the core logic of ABUs is implemented with lightweight mechanisms (counters, comparators, and switches) and hence no extra clock cycles are needed to traverse ABUs. Therefore, ABUs *do not introduce delays*: the cost of using them is only attributed to the additional FPGA resources required to be deployed. The resource utilization of one ABU and the ABU controller when implemented upon a Xilinx Zynq-7020 platform is reported in Table 1. The table also reports the percentage of resources occupied by the two modules with respect to the total amount resources available on the Zynq-7020. As it can be noted from the table, ABUs have a very marginal impact on resource consumption.

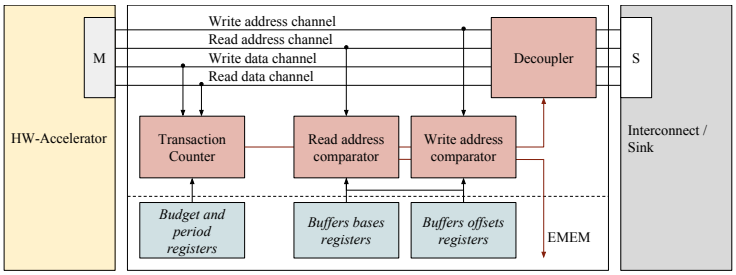

Figure 5 Internal functional block diagram of an ABU.

Table 1 Resource utilization for an ABU unit and the ABU controller on a Zynq-7020 platform.

Resource type	One ABU	ABUs Controller
LUT	436/53200 (0.82%)	279/53200 (0.56%)
FF	379/106400 (0.36%)	529/106400 (0.50%)
DSP	0/140 (0%)	0/140 (0%)
BRAM	0/220 (0%)	0/220 (0%)

4 Bandwidth-driven response-time analysis

This section studies the effect of bandwidth contention on HW-tasks under the considered modeling strategy, and presents a methodology to guarantee the system predictability using the ABU. Differently from most proposals in the literature, *the analysis proposed in this paper does not aim at accounting for possible interleaves of bus transactions over time* (e.g., like the analysis of classical periodic real-time tasks), but *aims at studying the contention incurred by HW-tasks in the bandwidth domain*, i.e., considering the actual rates at which the transactions make progress in the presence of other interfering tasks. **Why this approach?** As mentioned in Section 2.2, real-world hardware accelerators typically perform uniformly-distributed bus transactions at a constant rate, and, in particular, they even issue transactions at every clock cycle (see Figure 2). These characteristics make possible to treat HW-tasks as *fluid* computational activities that make progress at a given rate (e.g., similarly to fair multiprocessor scheduling [4]), and hence allow studying the system in bandwidth domain.

To better illustrate this peculiarity of the problem studied in this work, a simple example is firstly reported to show the effect of the contention introduced by round-robin arbitration (Sec. 4.1) in the bandwidth domain. Then, an observation concerning the critical instant for a set of HW-tasks is presented together with an illustrative example (Sec. 4.2). Finally, a strategy to enhance the system predictability by making HW-tasks prone for worst-case response-time analysis is presented (Sec. 5).

4.1 Illustrative example

To illustrate the effect of bandwidth contention incurred by HW-tasks subject to round-robin arbitration, consider a system composed of **(i)** a sink module providing a supply of $S = 6$, **(ii)** an interconnect directly connected to the sink module, and **(iii)** three HW-tasks, namely τ_1, τ_2, and τ_3, directly connected to the interconnect. The HW-tasks have the same demand $D_1 = D_2 = D_3 = S/2 = 3$ corresponding to half of the supply. The first HW-task (τ_1) needs to perform $N_1 = 6$ transactions and has a period of $T_1 = 9$ time units. The second HW-task (τ_2) performs $N_2 = 24$ transactions within a period of $T_2 = 11$ time units. Finally, the third HW-task (τ_3) performs $N_3 = 30$ transactions within a period of $T_3 = 15$ time units.

To avoid possible misunderstanding, please bear in mind that HW-tasks are statically allocated onto the FPGA area and hence do not contend the logical resources of the FPGA. For this reason, HW-tasks operate in a *parallel* fashion using their own (private) logic resources and can incur in contention only when issuing bus transactions.

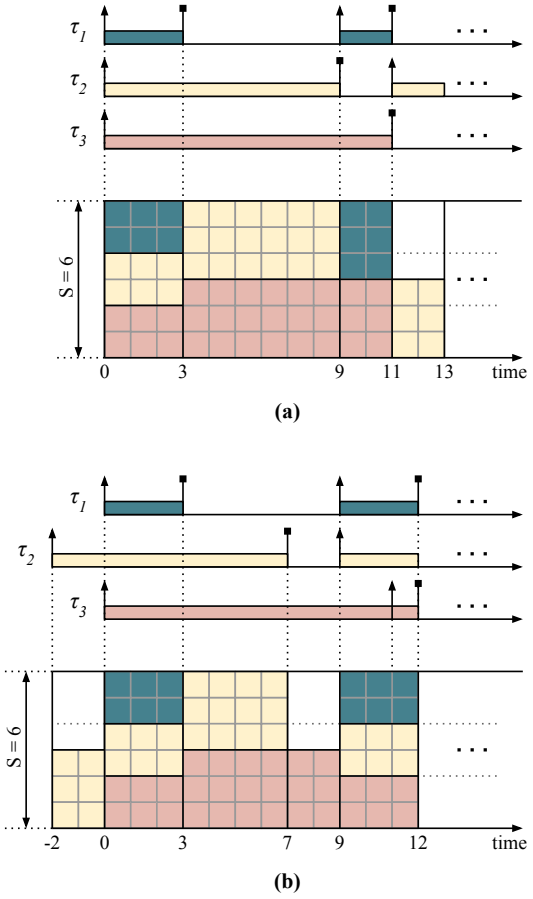

Figure 6 Examples of HW-task scheduling in the bandwidth domain with **(a)** synchronous release and **(b)** without synchronous release. In **(b)**, HW-task τ_3 experiences a longer response time with respect to the schedule in **(a)**.

Consider the case in which all HW-tasks are synchronously released at the same instant $t = 0$. Figure 6(a) illustrates the resulting schedule of the three HW-tasks by showing the intervals of time in which they are operating (on the top of the figure) and the repartition of the bandwidth over time (on the bottom of the figure). Each square unit of the bandwidth supply in the figure represents a transaction unit. At time $t = 0$, since the total bandwidth demanded by all HW-tasks $D_1 + D_2 + D_3 = 9$ exceeds the available bandwidth supply $S = 6$, the Interconnect limits the bandwidth of the three HW-tasks to a fair share of $S/3 = 2$. This bandwidth allocation continues up to $t = N_1/(S/3) = 3$, when τ_1 finishes its execution. Once τ_1 completes, τ_2 and τ_3 can proceed at their full rate of $S/2 = 3$ without suffering any contention. At time $t = 9$, τ_2 completes but a new periodic instance of τ_1 is also released. Again, both τ_1 and τ_3 can progress at their full rate without contention. At time $t = 11$, τ_1 and τ_3 complete at the same time and a new instance of τ_2 is activated. The latter can then proceed to operate while no other HW-task is active. Since τ_2 demands a bandwidth of $D_2 = 3$, half of the supply is left unused up to the next activation of τ_3 (which will occur at time $t = 15$).

4.2 Analysis issues

As it can be noted from Figure 6(a), HW-tasks are "slowed down" only when the total bandwidth demanded by active HW-tasks exceeds the supply (as it happens in $[0, 3)$ in the figure), i.e., when they make progress at a rate that is lower than their demand. Clearly, this phenomenon affects the worst-case response times of the HW-tasks.

Unfortunately, a bandwidth-driven response-time analysis cannot be accomplished by leveraging classical techniques for periodic real-time tasks. In particular, when studying the problem, we identified a set of issues (in some way similar to those identified in the analysis of multiprocessor real-time systems under global scheduling [16]) that prevent to analyze the system by looking at a single scheduling scenario.

To provide a taste of the identified issues, this section demonstrates that the classical critical instant theorem for periodic real-time tasks under uniprocessor scheduling does not hold for the problem studied in this work. Indeed, the longest response time of a HW-task *may not occur when it is synchronously released together with all other HW-tasks.*

To this end, consider the same system setup used for the previous example (Sec. 4.1). This time, assume that τ_2 is released before τ_1 and τ_3 at time $t = -2$, as shown in Figure 6(b). In this way, the first job of τ_2 can issue six transactions without suffering contention before τ_1 and τ_3 are activated at time $t = 0$. Hence the first job of τ_2 completes early (time $t = 7$) with respect to the case of synchronous release, leaving half of the bandwidth supply unused in time interval $[7, 9)$. Since τ_2 has been released earlier, also its next instance will be released earlier at time $t = 9$. The second job of τ_2 interferes with both τ_1 and τ_3 causing τ_3 to finish at time $t = 12$, i.e., one unit of time later than in the case of synchronous release. Hence τ_3 misses its deadline at time $t = 11$.

Proving a correct critical instant for the general case resulted a challenging problem that is still open for the authors. Nevertheless, as it is shown in the following section, ABUs can be extremely useful to make the system far more prone to analysis, hence increasing its predictability.

5 Response-time analysis with ABUs

Besides ABUs implement resource reservation, hence protecting the system from misbehaving HW-tasks, they can also be leveraged at the stage of analysis to help bounding the response times of the HW-tasks. Indeed, under the assumption that the ABU periods are orders of magnitude smaller then the periods of the HW-tasks, i.e., $P_i \ll \min_{i=1,\dots,n} \{T_i\}$, ABUs can act as *bandwidth regulators* limiting the maximum demand rate of HW-tasks.

Differently to software-based reservation techniques, for which a short reservation period determines a high overhead, the assumption on ABUs' periods is practical because ABUs are realized in hardware and hence do not introduce relevant issues when adopted with short reservation periods. Specifically, as mentioned in Section 3, ABUs are built with counters and signal switches that do not introduce delays and do not represent bottlenecks for the logic circuits deployed onto the FPGA such that the operating frequency of the latter has to be limited.

Under this setting, each ABU offers to the corresponding HW-task a virtual, *dedicated* supply of bus bandwidth B_i/P_i, which is independent of the behavior of the other HW-tasks as long as the ABU budgets are guaranteed. Therefore, the problem of analyzing a set of HW-tasks protected by ABUs can be decomposed into two independent steps:

1. guaranteeing that a set of ABUs can provide the corresponding bandwidths in the worst case, i.e., their entire budgets can be safely provided in every period; and
2. guaranteeing that the bandwidth provided by each ABU is sufficient for the corresponding HW-task to meet its deadline.

These steps are addressed in the following two sub-sections, respectively.

5.1 Analyzing ABUs

As long as the sum of the bandwidths provided by a set of ABUs does not exceed the total supply S, i.e., $\sum_{i=1}^{n} B_i/P_i \leq S$, no contention can occur; therefore, it is guaranteed that their budgets can be provided within every periodic instance. However, in the general case, this condition may not hold, and hence the analysis of ABUs must account for contention exactly as discussed in the example of Section 4.1.

Nevertheless, differently from a direct analysis of HW-tasks, two observations can be leveraged to make the analysis of ABUs tractable. First, as mentioned in Section 3, ABUs are synchronously activated at the system startup. Second, due to the assumption on the ABUs' periods ($P_i \ll \min_{i=1,\ldots,n}\{T_i\}$), there is no particular advantage in assigning heterogeneous periods to ABUs, and hence to act as fluid bandwidth regulators they can be all configured with the same period P. How to configure a suitable value for the period P is discussed in the experimental evaluation reported in Section 6.

Under this setting, it is then sufficient to study the case of synchronously released ABUs by analyzing a single problem window of length P that contains a single periodic instance of each ABU. In other words, it is enough to verify that all ABUs can provide their budget before time $t = P$ assuming that they are all released at time $t = 0$.

When contention occurs, it is not straightforward to compute how the available bandwidth supply is distributed between a set of active HW-tasks. In fact, considering n arbitrary HW-tasks and a supply S, they can be classified in **(i)** those that demand less (or the same) bandwidth than the fair share S/n, and **(ii)** those that demand more bandwidth than S/n, with the result that the spare bandwidth left by HW-tasks of type (i) is fairly re-distributed between the HW-tasks of type (ii). Algorithm 1 is presented to account for this phenomenon and computes the actual share of bandwidth of a supply S for each HW-task in a set $\mathcal{C}_{HW}$ of contending HW-tasks.

Algorithm 1: Computing bandwidth shares.

Input: *A set of HW-tasks:* $\mathcal{C}_{HW} = \{\tau_1, \ldots, \tau_m\}$
Input: *Sink supply:* S
Output: *A set of bandwidth shares:* $\overline{\mathcal{D}} = \{\overline{D}_1, \ldots, \overline{D}_m\}$

```
1  begin
2      S_rem ← S
3      M ← |C_HW|
4      for τ_i^hw ∈ C_HW by increasing D_i do
5          D̄_i ← min(D_i, S_rem/M)
6          S_rem ← S_rem − D̄_i
7          M ← M − 1
8      end
9      return D̄
10 end
```

The correctness of the algorithm is stated by the following lemma.

▶ **Lemma 1.** *Given a sink with supply S and a set of HW-tasks $\mathcal{C}_{HW}$ that contend for the supply, Algorithm 1 computes the correct share of bandwidth $\overline{D}_i$ assigned to each HW-task $\tau_i \in \mathcal{C}_{HW}$ under a fair arbitration.*

Proof. The proof is by induction on the iterative steps of the algorithm. *Base case (first iteration, $M = |\mathcal{C}_{HW}|$):* Let τ_i be the HW-task considered at the first iteration. If $D_i \geq S/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S/M$, which is correct, as it corresponds to the fair share. Since the set of HW-tasks is explored in order of increasing D_i (see line 4), then all the following iterations will consider HW-tasks with $D_i \geq S/M$ and, for the same reason, will be assigned a bandwidth equal to the fair share. Otherwise, if $D_i < S/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$. Note that this cannot affect the bandwidth assignment of the other HW-tasks as D_i is lower than the fair share S/M. *Inductive case ($M < |\mathcal{C}_{HW}|$):* Suppose that the algorithm assigned a correct bandwidth to the first $|\mathcal{C}_{HW}| - M + 1$ HW-tasks and that it remains to distribute a supply bandwidth S_{rem} to $M < |\mathcal{C}_{HW}|$ HW-tasks. Let τ_i be the HW-task considered at the current iteration. Similarly to the base case, if $D_i \geq S_{rem}/M$, then by line 5 τ_i is assigned a bandwidth share $\overline{D}_i = S_{rem}/M$, which is correct, as it corresponds to the fair share with respect to the remaining $M - 1$ HW-tasks. Again, since the set of HW-tasks is explored in order of increasing D_i, the same will hold for all the following iterations. Otherwise, if $D_i < S_{rem}/M$, then the HW-task will be assigned a bandwidth share equal to the required demand $\overline{D}_i = D_i$, which again cannot affect a fair distribution for the following $M - 1$ HW-tasks. Hence the lemma follows. ◀

Leveraging Algorithm 1, it is finally possible to build a schedulability test that verifies whether a set of ABUs can provide their budget within their period P. This is accomplished by Algorithm 2, which unrolls the execution of a set of HW-tasks protected by ABUs within an analysis window $[0, P]$.

The algorithm inputs the set of HW-tasks $\mathcal{T}_{HW}$ and the corresponding set of ABUs $\mathcal{A}$ (i-th ABU is connected to the i-th HW-task), and returns a boolean predicate that indicates whether the ABUs are schedulable or not. The algorithm keeps track of the analysis time t (initialized to $t = 0$) and the instantaneous budget b_i available for each ABU A_i, which is initialized to B_i (full budget). At the system startup ($t = 0$), all ABUs have available budget and hence all HW-tasks are considered active, i.e., they can generate transactions. Consequently, at line 4, the set of active HW-tasks, denoted with $\mathcal{C}_{HW}$, is initialized to $\mathcal{T}_{HW}$. Then, the procedure enters a loop at line 5. At each iteration, the algorithm computes the distribution of the supply S among the active HW-tasks by means of Algorithm 1, so obtaining the share of bandwidth $\overline{D}_i$ for each HW-task $\tau_i \in \mathcal{C}_{HW}$. Subsequently, it computes the amount of time Δ needed by at least one ABU A_i to provide all the available budget b_i, which is given by $\Delta = \min(b_i / \overline{D}_i)$. If a HW-task is not able to complete within the period P, then the system is deemed unschedulable and the algorithm terminates (lines 8-9). Otherwise, the algorithm proceeds by updating the budget of each ABU accounting for a lower-bound on the transactions performed in an interval of length Δ (line 12). Also, if the budget of an ABU is depleted ($b_i = 0$), then the corresponding HW-task is prevented to issue transactions and hence is removed from the set of active HW-tasks $\mathcal{C}_{HW}$ (line 14). Finally, the algorithm advances the time t by Δ and continues to iterate until the set $\mathcal{C}_{HW}$ is empty. If the algorithm completes without never detecting a deadline miss at lines 8-9, then the system is deemed schedulable.

Algorithm 2: Analysis of ABUs.

Input: *A set of HW-tasks:* $\mathcal{T}_{HW} = \{\tau_0, \ldots, \tau_n\}$
Input: *A set of ABUs:* $\mathcal{A} = \{A_0, \ldots, A_n\}$
Output: *Result of the schedulability test (true/false)*

```
1  begin
2      t ← 0
3      b_i ← B_i  ∀i = 1, ..., n
4      C_HW ← T_HW
5      while C_HW ≠ ∅ do
6          D̄ ← Algorithm 1(C_HW, S)
7          Δ ← min_{τ_i ∈ C_HW} (b_i / D̄_i)
8          if Δ + t ≥ P then
9              return false
10         end
11         for τ_i ∈ C_HW do
12             b_i ← b_i − ⌊D̄_i · Δ⌋
13             if b_i = 0 then
14                 C_HW ← C_HW \ {τ_i}
15             end
16         end
17         t ← t + Δ
18     end
19     return true
20 end
```

Finally, the following lemma states that the analysis of Algorithm 2 is sustainable, i.e., increasing the ABU budgets can only worsen the schedulability of a set of ABUs (and, vice versa, a set of schedulable ABUs remains schedulable if the budgets are decreased).

▶ **Lemma 2.** *The schedulabiliy test provided by Algorithm 2 is sustainable with respect to budgets* B_i*.*

Proof. Suppose that a set of ABUs is not schedulable according to Algorithm 2. Hence, there exists a certain time t at which the condition at line 8 holds. Consider an arbitrary ABU A_i (associated to task τ_i) and let $[0, t')$ be the interval of time in which τ_i is in set $\mathcal{C}_{HW}$ during $[0, t)$, i.e., $t' \leq t$. There are two cases: **(i)** τ_i is still in set $\mathcal{C}_{HW}$ at time t (i.e., $t' = t$), **(ii)** τ_i left set $\mathcal{C}_{HW}$ before time t (i.e., at time $t' < t$).

Case (i): In $[0, t)$, τ_i always contributed to the bandwidth distribution by means of Algorithm 1. Hence, if the budget B_i is increased, the bandwidth shares $\overline{D}_i$ assigned during $[0, t)$ are the same and therefore the schedulability result cannot change. If $\Delta = b_i/\overline{D}_i$ (i.e., at time t, τ_i is the task detected to miss its deadline), then, by increasing the budget B_i, Δ can only increase and hence the condition at line 8 would hold too.

Case (ii): Similarly to the previous case, τ_i always contributed to the bandwidth distribution in $[0, t')$ and hence, if B_i is increased, the execution of Algorithm 2 cannot change up to time t'. If the budget B_i is increased to $B_i + \epsilon$, at time t' it can be either that the value of Δ remains the same, or that it increases too by ϵ. Consequently, τ_i will remain for more time into set $\mathcal{C}_{HW}$, contributing to the bandwidth distribution also after time t', or still leaves set $\mathcal{C}_{HW}$ at time t'. In both these cases the schedulability result cannot change.

Hence the lemma follows. ◀

5.2 Assigning ABU budgets

As ABUs act as bandwidth regulators for HW-tasks, they enforce a specific rate at which transactions are issued. Specifically, a HW-task τ_i protected by ABU A_i issues transactions at rate B_i/P as long as the ABU is guaranteed to be schedulable according to the analysis presented in the previous section. Therefore, to guarantee that τ_i is capable of performing N_i transactions within its implicit deadline T_i, it is sufficient that the following inequality is satisfied: $\frac{N_i}{B_i/P} \leq T_i$. By rewriting the latter equation, it is possible to derive a constraint on the ABU budgets to ensure the schedulability of a set $\mathcal{T}_{HW}$ of HW-tasks, i.e.,

$$\forall \tau_i \in \mathcal{T}_{HW}, \quad B_i \geq \frac{N_i \cdot P}{T_i}. \tag{1}$$

Note that the same constraint can be generalized to the case of constrained deadlines by simply replacing T_i with the relative deadline of the HW-task.

▶ **Lemma 3.** *If a set of HW-tasks $\mathcal{T}_{HW} = \{\tau_0, \ldots, \tau_n\}$ respectively protected by a set of ABUs $\mathcal{A} = \{A_0, \ldots, A_n\}$ is not schedulable (according to Algorithm 2) by setting the ABU budgets as $B_i = \frac{N_i \cdot P}{T_i}$, then it is not schedulable with any other budget assignment.*

Proof. Given the constraint of Equation (1), $B_i = \frac{N_i \cdot P}{T_i}$ is the *minimum* budget for each ABU A_i such that the schedulability of τ_i can be guaranteed. Hence, feasible budget configurations can include only budget values larger than $\frac{N_i \cdot P}{T_i}$. By Lemma 2, if a set of ABUs is not schedulable by assigning such minimum budgets, then it is also not schedulable by assigning larger budgets. Hence the lemma follows. ◀

6 Experimental evaluation

To assess the effectiveness of the ABUs on a real hardware system, an experimental evaluation has been conducted on the Zynq-7020 SoC platform by Xilinx. The Zynq-7020 belongs to the Zynq-7000 SoCs family, which comprises a collection of SoCs mainly differing for the size and class of the FPGA fabric. Almost all SoCs of the Zynq-7000 family include a dual-core ARM Cortex-A9 processor with a set of integrated peripherals (PS subsystem) tightly coupled with a 7-series FPGA fabric (PL subsystem) that can be used to extend the system with custom hardware modules. The experimental evaluation is structured in two parts: the first part aims at evaluating the effectiveness of the reservation mechanism enforced by the ABUs using DMA-like HW-tasks; the second part evaluates the ABUs with a case study application that comprises a finite impulse response (FIR) HW-task for signal processing and a Sobel HW-task for image processing from OpenCV.

All HW-tasks used in this evaluation have been designed with the Vivado high-level synthesis (HLS) tool by Xilinx. The choice of utilizing HLS comes from the steadily increasing relevance that high-level synthesis is assuming in the design of hardware accelerators. For instance, a HLS tool can also be used to synthesize a HW-task implementing a custom compute unit for executing an OpenCL kernel. The hardware-level interface of the HW-tasks used in this evaluation consists of **(i)** two AXI4 master interfaces for accessing the system memory; **(ii)** an AXI4-lite slave control interface, to expose a set of memory-mapped registers through which the software can control the HW-task; and **(iii)** an interrupt signal to notify the processor when the computation of the HW-task is completed.

Each HW-task is controlled by a periodic software task running on top of the FreeRTOS kernel, which in turn runs upon one the Cortex-A processors of the Zynq-7020. The software task relies on a device driver for managing the HW-task, feeding the addresses of the source

and destination memory buffers as arguments. The driver controls the HW-tasks through the set of control registers exported via the AXI4-lite slave interface. Each job of each software task starts the corresponding HW-task and then self-suspends waiting for the HW-task to complete the execution. When the HW-task has completed, it sends an interrupt signal, which is caught by the interrupt service routine included in the driver. The service routine, in turn, wakes up the software task, which can then complete its job. This evaluation is focused on the timing properties of HW-tasks only.

Evaluation of the reservation mechanism. The first part of the experimental evaluation aims at validating the effectiveness of the reservation mechanism when one or more HW-tasks deviate from the nominal behavior by demanding a higher transaction rate and issuing more transactions than expected. Note that, from the perspective of bus contention, the bus transactions issued by HW-tasks are the only relevant aspect. Therefore, this evaluation employs a set of DMA-like HW-tasks, which allows for an almost-arbitrary control of the bus transactions that are generated. Nevertheless, also note that several hardware accelerators for FPGAs, including those of the Xilinx's IPs library such as FFT [43], FIR filter [44], and Convolution Encoder [42], require the support of a DMA for accessing the system memory.

Variants of HW-tasks. To simulate the effect of a misbehaving HW-task, three variants of the same DMA-like HW-task have been designed. Each variant differs by the amount of data N_i and the demand rate D_i. The parameters of these variants, referred to as modes, are summarized in Table 2. The demand value in MB/s is calculated by considering that each bus transaction involves a 32-bit word and that the clock rate of the FPGA is set to 100 MHz. All the HW-tasks issue 16-word burst transactions. On the Zynq-7020, the maximum supply bandwidth S available to access the memory from the PS through a high performance (HP) port is four transactions per clock for each port, as they operate in 64-bit mode (the DRAM clock is set to 525 MHz).

Table 2 Configuration of HW-tasks. The demand D_i is expressed in both transactions per clock cycle and in megabytes per second.

HW-task mode	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[MB]
1	2	763	524288	2
2	1	381	262144	1
3	2/3	254	131072	0.5

Description of the experimental setting. The system setup used for this evaluation comprises four DMA-like HW-tasks allocated on the Zynq's PL and connected to a single HP port through an AXI Interconnect. The Interconnect is set in performance mode to maximize the bandwidth available to the HP port. An ABU module is placed between each HW-task and the Interconnect. The baseline configuration includes two HW-tasks, τ_1 and τ_2, set in mode 1, a HW-task, τ_3, operating in mode 2, and the last HW-task τ_4 set in mode 3. This configuration represents the system operating in *nominal conditions*, i.e., when all the HW-tasks respect their nominal demand D_i and data length N_i values, and is referred to as *nom*. To study the effect of misbehaving HW-tasks, two additional variants of the baseline configuration have been defined. In the first misbehaving configuration, referred to as *misb-3*, τ_3 operates in mode 1 instead of mode 2. This configuration, represents the case in which a

single HW-task exceeds its nominal values, demanding a higher transaction rate and length. In the second misbehaving configuration, named *misb-3-4*, τ_3 and τ_4, normally operating in mode 2 and mode 3 respectively, now operate in mode 1. This configuration aims at reproducing the scenario in which two HW-tasks exceed their nominal values.

6.1 Profiling HW-tasks

The first set of experiments has been carried out to characterize the system configurations without ABUs. To this end, a separate profiling experiment has been conducted for each configuration of the system: the base configuration *nom*, and two misbehaving configurations *misb-3* and *misb-3-4*. These experiments allow evaluating the impact of one or more misbehaving HW-tasks on the response times of the other HW-tasks when using the default round-robin arbitration policy of the Interconnect. For this set of experiments, τ_1 is activated every 10 ms, τ_2 every 15 ms, τ_3 every 25 ms, and τ_4 every 50 ms. Measurements on the hardware have been conducted with multiple runs by testing random activation offsets of the HW-tasks, for a total of about 30 minutes of execution (collecting data for hundreds of thousands of jobs). Figure 7 presents the results of these experiments by reporting the longest-observed response times on the real hardware as solid color bars. The results corresponding to the misbehaving HW-tasks are highlighted with different colors and patterns. Comparing the response times observed under nominal conditions (*nom*) with the response times obtained under misbehaving configurations, it is evident that even a single misbehaving HW-task (*misb-3*) could have a significant impact on the response time of the other HW-tasks. This effect becomes even more tangible when taking into account the configuration *misb-3-4* in which two HW-tasks misbehave. For instance, the response time of τ_1 in *misb-3-4* increases by more than 50% with respect to nominal conditions.

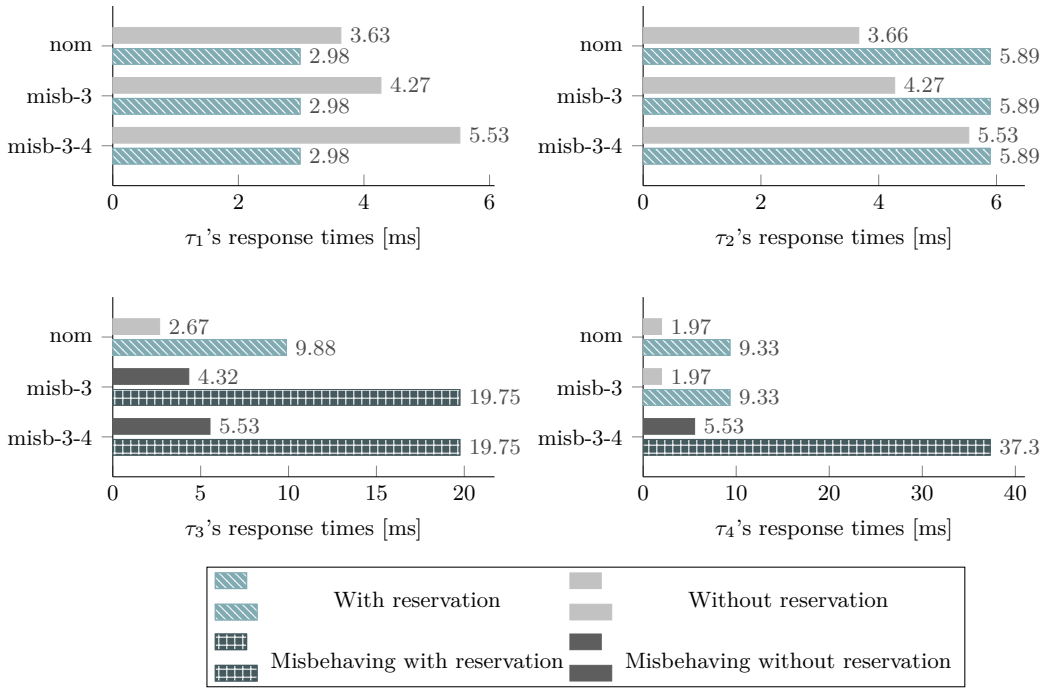

Figure 7 Response times of four HW-tasks without and with ABUs under multiple configurations.

6.2 Evaluating the reservation mechanism

The following set of experiments analyzes what happens when the ABUs are present. These experiments serve two purposes: first, to test the effectiveness of temporal isolation between HW-tasks; second, to confirm that the assumptions made in Sections 2 and 4 to model and analyze the system are realistic. To this end, the longest-observed response times on the hardware have been compared with the response-time bounds computed by the analysis of Section 4. The ABUs have been configured according to the minimum budgets provided by Lemma 3 under nominal conditions. The period of ABUs has been selected according to the following rationale. Since the ABUs count integer transactions, the period must be chosen as the smallest value that can can ensure that all the minimum budgets provided by Lemma 3 are integers. Furthermore, to avoid splitting transaction bursts, it is worth choosing a period such that the budget is a multiple of the burst size. Such a period can be easily obtained with a binary search. The resulting ABU configuration for this experimental setting is reported in Table 3. The table also reports the response times, both observed on the hardware and obtained by the analysis proposed in this work, under configuration *nom*.

As it can be noted from Figure 7, ABUs allow controlling the longest-observed response times (e.g., fixed to 2.98 for τ_1) independently of the behavior of the other HW-tasks; indeed, the response times are the same even in the misbehaving configurations *misb-3* and *misb-3-4*. Clearly, this improvement is achieved at the expenses of the misbehaving tasks (τ_3 and τ_4): in fact, their response times in misbehaving configurations is penalized.

Table 3 Configuration parameters for the ABUs and response times for the corresponding HW-tasks under the nominal configuration.

HW-Task	ABU		Response times [ms]	
	B_i [tr]	P [clk]	Longest observed	By analysis
τ_1	224	128	2.982	2.995
τ_2	112		5.893	5.991
τ_3	32		9.876	10.485
τ_4	16		9.328	10.485

6.3 A case study

The second part of the experimental evaluation considers a case-study application that comprises a FIR filter HW-task for signal processing, a Sobel HW-task for image processing, and two DMA-like HW-tasks operating in mode 1. The FIR filter implements a 12th order low-pass filter designed to process 16kHz audio samples with a cutoff frequency of 4 kHz. Internally, the FIR filter uses fixed-point representations to take advantage of the FPGA's DSP blocks. Each instance of the FIR filter processes 1 MB of samples. The Sobel filter processes 640x480 RGB images with 24-bit color depth, resulting in a size of 1200 KB. Table 4 summarizes the characteristics of these accelerators, which both issue 16-word burst transactions.

As visible from the trace shown in Figure 2, the access pattern generated by the Sobel filter HW-task is not strictly uniform due to a short pause occurring between two image lines. Such a signal analysis has been performed on the real hardware by instrumenting the design with an integrated logic analyzer (ILA) module. Clearly, the access pattern of the Sobel HW-task violates the uniform transaction hypothesis made in Section 2 to model the system. However, by performing the pessimistic assumption that the Sobel HW-task continues issuing

Table 4 Parameters of the Sobel and FIR hardware accelerators.

HW-task	D_i		N_i	
	[tr/clk]	[MB/s]	[tr]	[KB]
Sobel	1.9	725	614400	2400
FIR	2	763	524288	2048

transactions even during the brief pause between a line and the next, it is still possible to safely model it as a uniform access accelerator. Such a model can be used to assign the ABU budget and compute safe upper bounds on the response time of the Sobel HW-task. The case study application has been tested with a set of four experiments considering different HW-task periods and ABU budgets. Table 5 summarizes the parameters used for the experiments. The ABU period P is set to 128 clock cycles in all of the experiments. The results are reported in Figure 8, which compares the response times calculated using the response-time analysis presented in this paper, plotted as solid bars, with the longest-observed response times obtained on the real hardware, illustrated with striped bars. Measurements on the hardware have been performed as described in the previous section.

The experimental results show that the ABU is indeed effective even considering a case-study application comprising a realistic hardware workload suited for signal and image processing. The response times bounds obtained with the analysis are close to the longest-observed values with a maximum relative error of 3% in the case of HW-tasks with uniform demand. As expected, the maximum difference between the bound and the measurements (13%) occurs for the Sobel HW-task, since it has been pessimistically modeled by assuming a continuous bus access at its maximum rate.

Table 5 Configuration parameters for the case study (HW-task periods and ABU budgets).

Task	Experiment							
	1		2		3		4	
	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]	T_i [ms]	B_i [tr]
FIR	6	176	6	160	8	96	6	160
Sobel	7	160	8	144	9	112	12	96
DMA-2	10	80	12	64	6	144	7	128
DMA-1	12	64	7	112	7	128	10	80

7 Related work

Resource reservation techniques have been introduced in the context of real-time systems for CPUs scheduling [31, 1, 8] and applied to share other computational resources like programmable GPUs [23, 22]. Essentially, the idea is assigning to each entity (e.g., task) a fraction of a shared resource under contention (e.g., processor) in order to provide temporal isolation. Similarly, this work adjusts the same approach to the contention of the AMBA AXI bus in the context of hardware-programmable SoC FPGA platforms.

Many research efforts have been dedicated to the problem of bus contention in real-time systems. Schliecker et al. [33] use an event-based model to estimate delays for communications and computation activities on a multicore SoC platform. Pellizzoni and Caccamo [29] analyzed the interaction between CPU and peripherals while contending a shared main memory within a theoretical framework and proposed a conceptual solution based on a hardware server to control the unpredictable behavior of COTS peripherals. Betti et al. [6] presented a

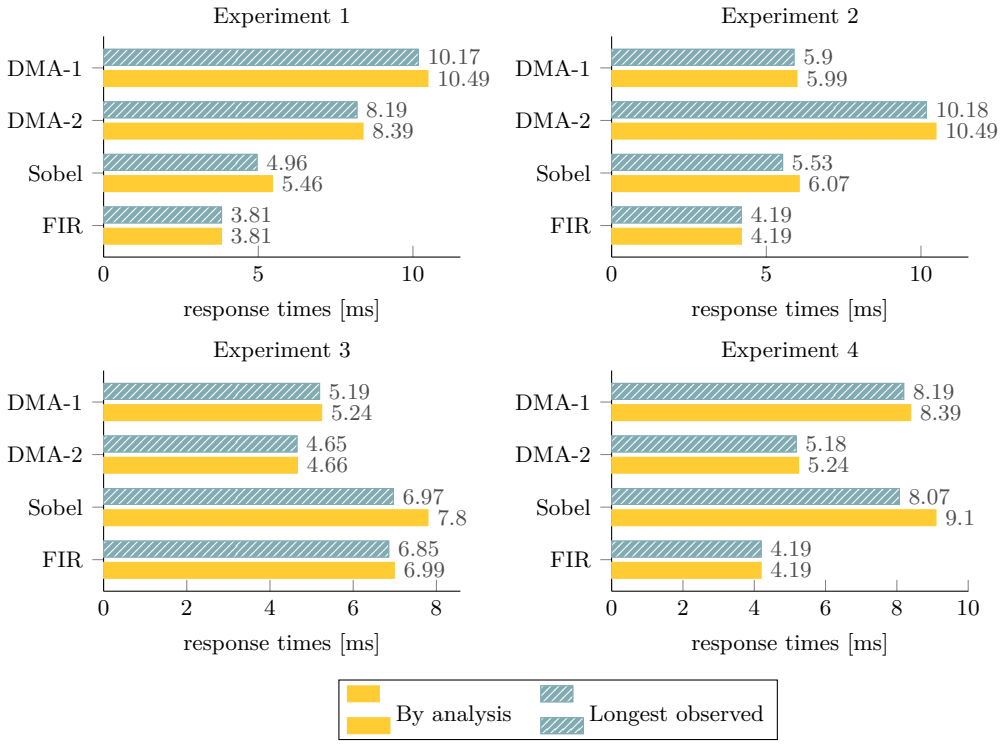

Figure 8 Response times for the case study.

framework for providing real-time guarantees in a COTS platform. Each peripheral within the platform is supervised by a "real-time bridge" controlled by a system-wide peripheral scheduler. Their framework has been developed and evaluated on PC platforms with PCI Express bus while our approach considers on-chip buses for integrated SoC-FPGA platforms.

In the context of memory contention on multicore platforms, Agrawal et al. [2] presented a technique to perform the analysis both WCETs and schedulability of real-time activities under dynamic memory scheduling. Yum et al. [47] proposed a memory bandwidth reservation mechanism named MemGuard. The system provides memory performance isolation employing a bandwidth regulator for each core. The bandwidth regulators enforce a budgeting mechanism and are implemented using performance counters. Our approach is somehow related to this work since both consider bandwidth regulation of bus master agents. However, while MemGuard considers inter-core interference on an Intel chip multiprocessor, our work considers bus interference generated by hardware accelerators on the AMBA AXI bus.

In the domain of packet switching networks, many efforts have been dedicated to the modeling and the analysis of traffic scheduling algorithms to provide quality of service (QoS) guarantees [15, 37]. Such methodologies have also been employed on SoCs platforms to develop and analyze arbiters for heavily-contented resources like the system memory [3, 17]. The ABU can be improved by leveraging the results of these works. Concerning the development of on-chip communication infrastructures for SoC platforms, transaction-based buses and packet-based networks on chip (NoC) remain the dominant approaches [32]. Typically, arbitration for on-chip interconnects is performed using Fixed Priority, Round Robin, and Time-Division Multiple Access (TDMA). Poletti et al. presented a performance analysis comparing different arbitration policies for SoCs platforms in [30]. A TDMA-based arbitration scheme with dynamic timeslot allocation is employed in [32, 10] to improve system predictability while

providing good average-case performance. Lahiri et al. [24] proposed a statistical approach to arbitration using a ticket-based random selection which was further extended by other works [12, 25] to improve predictability. Steine et al. [36] proposed a TDMA budget based scheduler for data flow applications, which has been used by Staschulat et al. [35] for memory arbitration. However, while the latter work is explicitly targeted at embedded systems, it is still limited to dataflow applications. Bourgade [9] proposed a bus arbitration scheme for multicore platforms designed to ease the estimation of the tasks' worst-case execution times. Reconfigurable bus arbiters [46, 34] can be dynamically configured to change the arbitration policy depending on the application requirements. Likewise, several papers in the literature addressed the problem of designing predictable memory controllers for multi-core architectures. Guo et al. [18] presented a comparative analysis of predictable DRAM controllers.

8 Conclusions

This paper presented the ABU, a hardware-based reservation mechanism for the AMBA AXI bus aimed at isolating hardware accelerators implemented on FPGAs. After describing the internal architecture of the ABU, a response-time in the bandwidth domain has been presented to verify the schedulability of a set of hardware accelerators under real-time constraints. The proposed mechanism has been implemented and validated on the Xilinx Zynq-7020 platform to demonstrate its practical applicability. An substantial experimental evaluation confirmed the effectiveness of the proposed solution, showing that it can efficiently be implemented by consuming less than 1% of the total FPGA resources. As a future work, we plan to evaluate the possibility of including a reclaiming mechanism for the unused supply and extend the analysis to support for dynamic workloads by taking advantage of partial reconfiguration [7, 27].

References

1 Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 4–13. IEEE, 1998.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, December 2018.

3 Benny Akesson, Liesbeth Steffens, and Kees Goossens. Efficient service allocation in hardware using credit-controlled static-priority arbitration. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 59–68. IEEE, 2009.

4 James H. Anderson, Philip Holman, and Anand Srinivasan. Fair Scheduling of Real-Time Tasks on Multiprocessors. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

5 ARM. *AMBA AXI and ACE Protocol Specification*, 2011.

6 E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-Time I/O Management System with COTS Peripherals. *IEEE Transactions on Computers*, 62(1):45–58, January 2013. doi:10.1109/TC.2011.202.

7 Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

8 Alessandro Biondi, Alessandra Melani, and Marko Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37. IEEE, 2014.

9 Roman Bourgade, Christine Rochange, and Pascal Sainrat. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4. IEEE, 2011.

10 Paolo Burgio, Martino Ruggiero, Francesco Esposito, Mauro Marinoni, Giorgio Buttazzo, and Luca Benini. Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 187–194. IEEE, 2010.

11 Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. From software to accelerators with legup high-level synthesis. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 18. IEEE Press, 2013.

12 Chien-Hua Chen, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

13 Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

14 Ben Cope, Peter YK Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on computers*, 59(4):433–448, 2010.

15 Rene L Cruz et al. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131, 1991.

16 Robert I. Davis and Alan Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4), 2011.

17 Manil Dev Gomony, Jamie Garside, Benny Akesson, Neil Audsley, and Kees Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, 2017.

18 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):53, 2018.

19 Dominik Honegger, Helen Oleynikova, and Marc Pollefeys. Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 4930–4935. IEEE, 2014.

20 Intel. *Stratix 10 GX/SX Device Overview*, October 2017.

21 Jan Moritz Joseph, Morten Mey, Kristian Ehlers, Christopher Blochwitz, Tobias Winker, and Thilo Pionteck. Design space exploration for a hardware-accelerated embedded real-time pose estimation using vivado HLS. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pages 1–8. IEEE, 2017.

22 Shinpei Kato, Karthik Lakshmanan, Yutaka Ishikawa, and Ragunathan Rajkumar. Resource sharing in GPU-accelerated windowing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 191–200. IEEE, 2011.

23 Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

24 Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. The LOTTERYBUS on-chip communication architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(6):596–608, 2006.

25 Bu-Ching Lin, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. A precise bandwidth control arbitration algorithm for hard real-time SoC buses. In *Proceedings of the 2007 Asia*

and South Pacific Design Automation Conference, pages 165–170. IEEE Computer Society, 2007.

26 Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

27 Marco Pagani, Alessio Balsini, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 96–101. IEEE, 2017.

28 Karl Pauwels, Matteo Tomasi, Javier Diaz Alonso, Eduardo Ros, and Marc M Van Hulle. A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers*, 61(7):999–1012, 2012.

29 R. Pellizzoni and M. Caccamo. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. *IEEE Transactions on Computers*, 59(3):400–415, March 2010. doi:10.1109/TC.2009.156.

30 Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo. Performance analysis of arbitration policies for SoC communication architectures. *Design Automation for Embedded Systems*, 8(2-3):189–210, 2003.

31 Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1998*, volume 3310, pages 150–165. International Society for Optics and Photonics, 1997.

32 Thomas D Richardson, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Yuan Xie, Chita Das, and Vijay Degalahal. A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 8–pp. IEEE, 2006.

33 Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 161–166. ACM, 2008.

34 Éricles Sousa, Deepak Gangadharan, Frank Hannig, and Juergen Teich. Runtime reconfigurable bus arbitration for concurrent applications on heterogeneous MPSoC architectures. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 74–81. IEEE, 2014.

35 Jan Staschulat and Marco Bekooij. Dataflow models for shared memory access latency analysis. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 275–284. ACM, 2009.

36 Marcel Steine, Marco Bekooij, and Maarten Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 37–44. IEEE, 2009.

37 Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on networking*, 6(5):611–624, 1998.

38 Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Computing Surveys (CSUR)*, 51(3):56, 2018.

39 Xilinx. *Zynq UltraScale+ Device - Technical Reference Manual*, December 2017. UG1085.

40 Xilinx Inc. *Using Quality of Service (QoS) Capabilities in Zynq-7000 AP SoC Devices*, July 2015. XAPP1266.

41 Xilinx Inc. *AXI Interconnect, LogiCORE IP Product Guide*, 2018. PG059.

42 Xilinx Inc. *Convolutional Encoder, LogiCORE IP Product Guide*, 2018. PG026.

43 Xilinx Inc. *Fast Fourier Transform, LogiCORE IP Product Guide*, 2018. PG109.

44 Xilinx Inc. *FIR Compiler, LogiCORE IP Product Guide*, 2018. PG149.

45 Xilinx Inc. *SmartConnect, LogiCORE IP Product Guide*, 2018. PG247.

46 Ching-Chien Yuan, Yu-Jung Huang, Shih-Jhe Lin, and Kai-hsiang Huang. A reconfigurable arbiter for SOC applications. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 713–716. IEEE, 2008.

47 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.