IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 April 2023, accepted 10 May 2023, date of publication 19 May 2023, date of current version 1 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3277959

RESEARCH ARTICLE

The Prism Bridge: Maximizing Inter-Chip AXI Throughput in the High-Speed Serial Era

ROBERT DREHMEL, (Member, IEEE), AND HANS-ULRICH HEISS
Faculty IV Electrical Engineering and Computer Science, Institute of Telecommunication Systems, Technische Universität Berlin, 10623 Berlin, Germany
Corresponding author: Robert Drehmel (drehmel@campus.tu-berlin.de)

This work was supported in part by the German Academic Scholarship Foundation (Studienstiftung des deutschen Volkes).

ABSTRACT In this paper, we present the Prism Bridge, a soft IP core developed to bridge FPGA-MPSoC systems using high-speed serial links. Considering the current trend of ubiquitous serial transceivers with staggeringly increasing line rates, minimizing overhead and maximizing data throughput becomes paramount. Hence, our main design goal is to maximize bandwidth utilization for AXI data, which we realize through an advanced packetization mechanism. We give an overview of the Prism Bridge's design and analyze its half-duplex bandwidth utilization. Additionally, we discuss the results of the experiments we conducted to assess its real-world performance, including measurements of throughput and latency of various combinations of line rates, link-layer cores, and bridge cores. Using a serial link with a 16.375 Gbit/s line rate, the Prism Bridge with an advanced packetizing mechanism achieved an AXI write throughput of 1368.82 MiB/s and an AXI read throughput of 1376.62 MiB/s, an increase of 46.20% and 45.86%, respectively, compared with the de-facto industry-standard core. The advanced packetization mechanism had negligible impact on latency but required 69.15%–73.91% more LUTs and 33.62%–36.19% more flip-flops. We conclude that for most designs that support inter-chip AXI transactions and will not be limited to short transaction lengths, the higher data throughput of the Prism Bridge with an advanced packetization mechanism is worth its cost in additional logic resource utilization.

INDEX TERMS Cluster computing, computer architecture, field-programmable gate arrays, high-speed serial, inter-chip axi communication, protocols.

LIST OF SYMBOLS

$f_{\text{axis}} \in \mathbb{N}$	AXI-Stream clock frequency in Hertz.
$k_{\text{sid}} \in \mathbb{N}$	Data stream ID width.
$k_{\text{msg}} \in \mathbb{N}$	Data stream payload width.
$k \in \mathbb{N}$	Message width ($k = k_{\text{sid}} + k_{\text{msg}} = n - m$).
$l \in \mathbb{N}$	Length of an AXI transaction in transfers (i.e., $\texttt{AxLEN} + 1$).
$m \in \mathbb{N}$	Number of Hamming code and parity bits per word of n data bits.
$n \in \mathbb{N}$	AXI-Stream data width.
$R_{\text{c}}^{(\text{link})} \in \mathbb{R}$	Link layer protocol overhead factor (e.g., $\frac{8}{10}$ for Aurora 8B/10B).
$s_{\text{w}} \in \mathbb{N}$	Data width in each AXI write transfer (i.e., $8 \cdot 2^{\texttt{AWSIZE}}$).

The associate editor coordinating the review of this manuscript and approving it for publication was Ilaria De Munari.

LIST OF SYMBOLS (CONT.)

$s_{\text{r}} \in \mathbb{N}$	Data width in each AXI read transfer (i.e., $8 \cdot 2^{\texttt{ARSIZE}}$).
$t \in \mathbb{N}_0$	Number of AXI transactions.
$\tau \in \{\text{w}, \text{r}\}$	AXI transaction type.
$u \in \{\text{aw}, \text{w}, \text{b}, \text{ar}, \text{r}\}$	AXI channel type.
$v \in \{0, 1, 2, 3\}$	Packetizer behavior.
$w_{\text{aw}} \in \mathbb{N}_0$	AXI AW channel width.
$w_{\text{w}} \in \mathbb{N}_0$	AXI W channel width.
$w_{\text{b}} \in \mathbb{N}_0$	AXI B channel width.
$w_{\text{ar}} \in \mathbb{N}_0$	AXI AR channel width.
$w_{\text{r}} \in \mathbb{N}_0$	AXI R channel width.

I. INTRODUCTION

Chips that integrate programmable logic, general-purpose processors, and serial transceivers have come a long way since Xilinx introduced the Virtex II-Pro device family in

March 2002 [1]. The XC2VP100 device—the top-of-the-line member of the Virtex-II Pro family—includes two IBM PowerPC processors and up to 20 Rocket I/O multi-gigabit transceivers (MGTs), each capable of a maximum line rate of 3.125 Gbit/s [2]. In September 2019, Xilinx introduced the Versal family of devices [3]. Its recently presented, top-end VP2802 chip from the Versal Premium series offers a dual-core Arm Cortex-A72 and a dual-core Arm Cortex-R5F28 processor in addition to 28 GTYP and 140 GTM transceivers that supply the designer with a combined maximum full-duplex serial bandwidth of 17.6 Tbit/s [4].

A variety of FPGA clusters have been proposed, predominantly in the domain of high-performance computing (HPC) [5], [6]. Especially chips featuring the triad of programmable logic, an integrated processing system, and high-speed serial connectivity lend themselves as building blocks for highly interconnected clusters. From single-chip modules that provide serial connectivity through multiple USB-C receptacles [7] to multi-chip modules with intra- and inter-module serial links [8], the high-speed serial era certainly inspires research on FPGA-MPSoC clusters.

The Arm Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol [9] is the dominant standard for address-based intra-chip interconnectivity in FPGA-MPSoCs. It is used to interconnect processors, on-chip memories, input/output devices, other hard IP blocks, and the programmable logic fabric. AXI-Stream is the analogous protocol for streaming data transfers. A straightforward idea for the inter-chip interconnection of FPGA-MPSoCs is to bridge the AXI protocol over serial links.

This work presents the Prism Bridge that allows to forward AXI transactions and interrupt requests to a remote system. Our principal design goal is to utilize as much as possible of the available bandwidth of the underlying link for AXI data while imposing no constraints on the width of individual AXI signals. It is supplemented by a link-layer core, a utility core for link-layer management, and Linux kernel drivers to facilitate research on FPGA-MPSoC chips running Linux.

The main contributions of this paper are:

- an overview of the design and implementation of the Prism Bridge, particularly our gearbox-based packetization mechanism designed to maximize bandwidth utilization (Section III),
- an analysis of its half-duplex bandwidth utilization (Section IV), and
- an evaluation of the performance of its variants and the de-facto industry standard IP core (Section V).

The remainder of this paper is organized as follows: Section II reviews related work, Section III presents the design and implementation overview, Section IV analyses the bandwidth utilization in half-duplex operation, Section V outlines the experimental setup and the methodology and discusses the evaluation results, and Section VI offers a conclusion and identifies prospects for future work.

II. RELATED WORK

The de facto standard core for bridging the AXI protocol is the AXI Chip2Chip IP core [10] from AMD Xilinx. It is used in a wide range of commercial and academic applications. It supports two different interfacing options: the SelectIO (SDR or DDR) interface and the Aurora family of cores using the AXI-Stream protocol. To our knowledge, there are no other competitors to the AXI Chip2Chip core. AMD Xilinx has worked with Arm on the AXI4 specification and uses AXI4 throughout its product line as the interconnect standard [11]. We suppose that—because of AMD Xilinx's commitment to the AXI IP ecosystem—its AXI IP cores are generally expected to deliver optimal performance.

The AMD Xilinx Aurora 8B/10B core [12] is one of the two family members of the Aurora family and implements the Aurora 8B/10B link-layer protocol [13]. Widmer and Franaszek [14] described 8B/10B encoding as early as 1983. Analogously, the other member of the AMD Xilinx Aurora family, the Aurora 64B/66B core [15], implements the Aurora 64B/66B link-layer protocol [16]. Both encodings have been adopted by well-established protocols. For example, USB 3.0 and 1000BASE-X use 8B/10B encoding, while 10GBASE-R uses 64B/66B. Tomori and Osana [17] have implemented the Aurora 64B/66B protocol in their open-source Kyokko core and claim that it exhibits lower latency and utilizes fewer logic resources than AMD Xilinx's Aurora 64B/66B core. They have recently demonstrated Kyokko's channel bonding (i.e., multi-lane channel) capability [18].

Brewer et al. [19] used the AXI Chip2Chip IP core (in combination with the Aurora 64B/66B core) for communication between boards in their RECON architecture in the context of the NASA SpaceCube Intelligent Multi-Purpose System (IMPS). Chimeh [20] presented an architecture for Massive Multi Input Multi Output (Massive MIMO) in 5G radio applications, with examples for 16×16 and 64×64 configurations. Their architecture uses the AXI Chip2Chip core for all communication between boards. Flich et al. [21] used the AXI Chip2Chip core for both inter- and intra-cluster communication in their MANGO project for HPC, their prototype consisting of 96 FPGAs in total. Lettnin and Winterholer [22] described a design that uses a larger and a smaller FPGA, the latter providing a DDR controller for the former, bridged by AXI Chip2Chip cores. In the realm of waveform design and deployment, Wildman et al. [23] split Common-modem Hardware Integrated Library (CHIL) [24] software and hardware onto separate boards and connected them using AXI Chip2Chip cores. Yang et al. [25] proposed a Processing-In-Memory (PIM) architecture. In their setup, the master board accesses the slave board with a bridge built using AXI Chip2Chip cores. Zhu et al. [26] used the AXI Chip2Chip core (with the Aurora 64B/66B core) to access remote memory in their photonic switched optically connected memory system architecture.

As FPGAs are a mainstay in particle physics experiments, particularly at the Large Hadron Collider (LHC) at CERN,

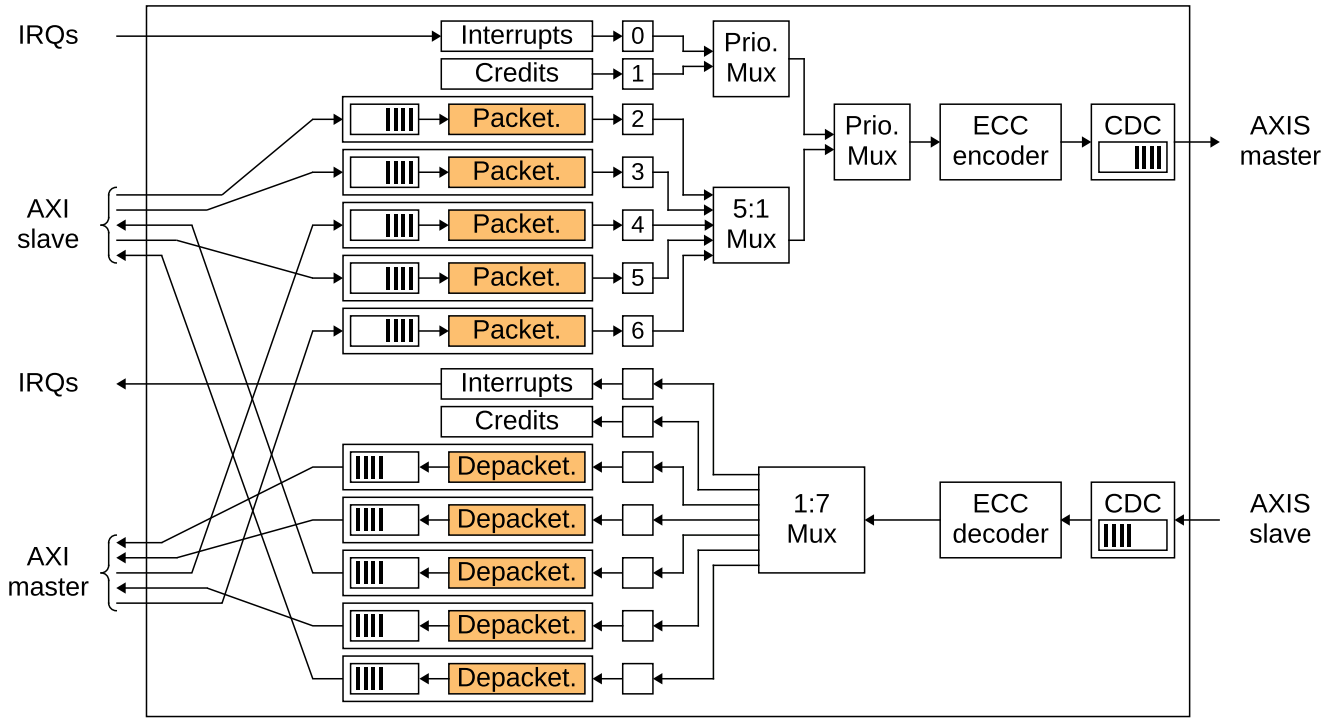


FIGURE 1. Bridge block diagram (full type). Arrows coming from an AXI interface are pointing to AXI input modules that contain a FIFO (with its input connected to the AXI interface) and a packetizer. Arrows going to an AXI interface are coming from AXI output modules that contain a depacketizer and a FIFO (with its output connected to the AXI interface).

applications for AXI bridging are found in that realm as well. Mehner et al. [27] presented board-management mezzanine boards for the Serenity family of Advanced Telecommunications Computing Architecture (ATCA) blades. Each of these mezzanine boards controls the main board it is connected to using the IPBus protocol [28] over an AXI Chip2Chip bridge. Similarly, Byszuk et al. [29] used the AXI Chip2Chip core to bridge IPBus communication between the control and processing FPGAs on the MTF7 board [30]. On the PIxel detector high Luminosity UPgrade (πLUP) board [31], [32], the AXI Chip2Chip core (using SelectIO DDR) is instantiated on both FPGAs to enable communication. Loukas [33] presented the Barrel Calorimeter Processor demonstrator (BCPd) ATCA blade that hosts an Embedded Linux Mezzanine (ELM) [34] mezzanine board. The ELM controls the FPGA on the main board using AXI over an AXI Chip2Chip bridge. Spiwoks et al. [35] presented an implementation of the Muon-to-Central-Trigger-Processor-Interface (MUCTPI) ATCA blade. Its Control Processor uses an AXI Chip2Chip core to manage the two Muon Sector Processors and the Trigger Readout TTC.

The AXI Chip2Chip core is not identified as a bottleneck in the works cited above. However, Ioannou et al. [36] at one point identified the ''inter-FPGA links'' based on the AXI Chip2Chip core as the bottleneck of their UniLogic HPC architecture design. They report latency reduction by replacing the Aurora core but do not elaborate on the Aurora core replacement. They state

> As data transfer, or throughput availability, is a common cause for bottlenecks, additional research that targets optimizations for inter-FPGA communication throughput ought to prove beneficial. [36]

and suggest increasing the bandwidth with link bonding to increase throughput. We expect that our approach of increasing bandwidth utilization and the approach of increasing bandwidth would have a synergistic effect on AXI throughput. We hope that our work serves as an impulse for designers in academia and industry to evaluate whether their designs could benefit from the throughput increase provided by the Prism Advanced Bridge.

III. DESIGN

The Prism Bridge presented in this paper is part of the Prism project that aims at developing an ecosystem for cluster computing with Unix-like operating systems. The Prism Bridge is an IP core written in SystemVerilog. It comes in three primary types: The unidirectional *slave* type that provides interrupt request input ports and an AXI slave interface to serialize from, the unidirectional *master* type that provides interrupt request output ports and an AXI master interface to deserialize to, and the bidirectional *full* type that combines the master and slave types. In this paper, we focus on the two unidirectional types. All primary types also provide one AXI-Stream master interface, one AXI-Stream slave interface, and one AXI-Lite slave interface. The AXI-Stream interfaces provide the connection to the link-layer protocol core. The AXI-Lite slave interface allows access to memory-mapped registers for control, status, configuration, and debugging. A Linux kernel driver provides a sysfs-based interface to these registers.

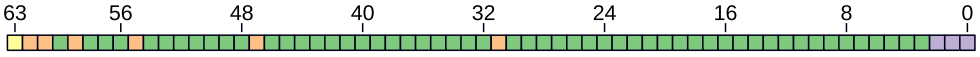


FIGURE 2. Prism Bridge AXI-Stream data word layout. Data stream payload Data stream ID Hamming code Parity bit.

To assist designers in handling the AMD Xilinx Aurora cores with a Linux system on an FPGA-MPSoC, the Prism ecosystem supplies the Prism Aurora Control core. This core also provides an AXI-Lite slave interface for memory-mapped register access and is accompanied by a Linux kernel driver that allows link-layer status management using a sysfs-based interface.

Furthermore, we used the Kyokko source code as the base for our Prism Kyokko link-layer core. Prism Kyokko works with the Zynq UltraScale+ GTH transceivers connected to the SFP+ connectors on the AMD Xilinx ZCU102 board. In contrast with Kyokko, which works in framing mode (using the AXI-Stream `TLAST` signal), Prism Kyokko works in streaming mode. We improved the code, added `ASYNC_REG` attributes [37] where appropriate to potentially strengthen CDC synchronization, and packaged it as an IP core usable in AMD Xilinx Vivado IP Integrator Block Designs [38], [39]. In its current form, one Prism Kyokko core provides two bidirectional communication channels, while one Aurora core provides a single communication channel. We did not, however, change any clock-related parts of Kyokko. Kyokko uses the asynchronous 64B/66B gearbox of the transmitter and the receiver of the GTH transceivers [40] and different clock domains for the transmit (TX) and receive (RX) interfaces. It includes an asynchronous 66-bit-wide FIFO to synchronize the RX signals with the clock of the TX clock domain. Kyokko's AXI-Stream master and slave interfaces are synchronous with the TX clock as well.

From here on, we use the term *AXI channel vector* to refer to the signals of an AXI channel (excluding the handshake signals) seen as one consecutive string of bits. The block diagram of the Prism Bridge is shown in Figure 1. At its core, the Prism Bridge transports up to seven independent *data streams* in each direction. Five of these data streams carry AXI channel vectors, one is used for interrupts, and one is used for the flow-control subsystem. Each AXI-Stream data word transferred by the Prism Bridge consists of m Hamming code and parity bits, k_{sid} data stream ID bits, and k_{msg} data stream payload bits. We refer to the data stream ID bits as the *data stream ID vector* and to the data stream payload bits as the *data stream payload vector*. The k-bit *data stream vector* is made up of a data stream payload vector concatenated with a data stream ID vector. The data stream ID vector specifies to which data stream the data stream payload vector belongs. Figure 2 shows the layout of an AXI-Stream data word.

An *input module* is the source, and an *output module* is the sink of a data stream. The input module for an AXI-related data stream contains a FIFO and a packetizer. This input module's inwards-facing FIFO stores an AXI channel vector and passes it to the input module's packetizer, which encodes it into the data stream. Correspondingly, the output module for an AXI-related data stream contains a depacketizer and a FIFO. This output module's depacketizer decodes the AXI channel vector from the data stream and stores it in the output module's outwards-facing FIFO. Input and output modules for the other data streams are more integrated and do not have separate packetization and depacketization steps. The interrupt data stream input module checks every clock cycle whether interrupt request input signals are asserted and sends a message on the interrupt data stream to the remote bridge if necessary. The remote bridge's interrupt data stream output module receives this message and asserts the interrupt request output signals for one clock cycle. A data stream payload vector coming from an input module is tagged with a data stream ID, multiplexed with other data streams, equipped with error correction bits, and stored in the egress clock domain crossing (CDC) FIFO for AXI-Stream transfer. Upon reception by the remote bridge, the AXI-Stream data word is stored in its ingress CDC FIFO, error-checked and possibly error-corrected, demultiplexed, stripped of its data stream ID, and the remaining data stream payload vector is passed to the output module associated with the data stream. In the following subsections, we provide details on some major components of the Prism Bridge.

A. PACKETIZER AND DEPACKETIZER

We differentiate two packetization mechanisms supported by the Prism Bridge: a simple, padding-based mechanism and an advanced, gearbox-driven one.

The simple packetizer slices an AXI channel vector into as many data stream payload vectors as necessary and keeps the remaining bits of the last data stream payload vector unused. That makes the work of the simple depacketizer easy: It simply needs to concatenate the incoming data stream payload vectors and drop the excess bits. The simple packetizer produces $\left\lceil \frac{x}{k_{\text{msg}}} \right\rceil$ data stream payload vectors for an x-bit-wide AXI channel vector.

The AXI protocol allows flexibility regarding the width of some of its signals (e.g., `WUSER`), resulting in variably-sized AXI channel vectors. Parts of the advanced packetizer's and depacketizer's logic vary based on the input data width and the output data width (i.e., the width of the AXI channel vector and k_{msg}, respectively). Therefore, tailor-made advanced packetizers and depacketizers are generated by the Prism Bridge Generator. The Prism Bridge Generator is a program that uses a custom C++ template engine and appropriate template files to produce SystemVerilog source code. Dynamically generated modules create a challenge for a potential ASIC (application-specific integrated circuit)

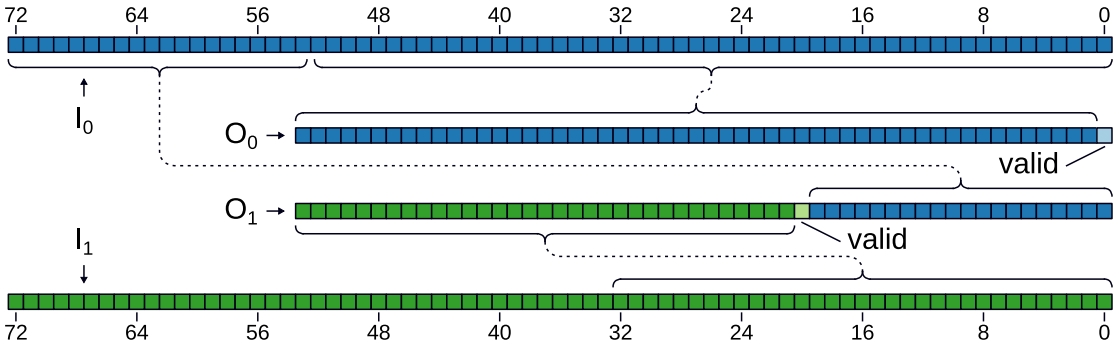


FIGURE 3. Advanced packetizer data stream payload vector layout. I_0 valid bit I_0 data bit I_1 valid bit I_1 data bit I_0: Input word 0 I_1: Input word 1 O_0: Output word 0 O_1: Output word 1.

implementation. One obvious solution would be to restrict the ASIC to a specific set of AXI signal widths. Another possible and more flexible solution would be to redesign the advanced packetizer and depacketizer to be nonspecific components that operate based on specific information stored in some form of configuration memory.

The advanced packetizer has a pipeline consisting of shift-left modules connected in series to meet timing requirements. The advanced packetizer pipeline has an output signal to notify the subsequent gearbox circuit whenever the pipeline is empty. A shift-left module instance is configured by two major parameters: an offset $y \in \mathbb{N}_0$ and a width $z \in \mathbb{N}$. The Bridge Generator appropriately parameterizes the instantiated shift-left modules to minimize the necessary logic resources.

The merger circuit at the end of the pipeline constructs a data stream payload vector from the shifted input data and the data currently in its buffer. If there is no data in the pipeline, it resets its state and forwards the current contents of its buffer. A valid bit is set in front of every valid AXI channel vector to distinguish between valid and invalid data in outgoing data stream payload vectors, as depicted in Figure 3. The advanced depacketizer uses this valid bit to control the state of its pipeline and merger. Concluding, the advanced packetizer needs $\left\lceil \frac{l(x+1)}{k_{\text{msg}}} \right\rceil$ AXI-Stream transfer cycles for l consecutive x-bit-wide AXI channel vectors.

Where an explicit differentiation is necessary, we call the Prism Bridge a Standard Bridge when it uses simple packetization for all AXI channels and an Advanced Bridge when it uses the simple packetization for the AXI `AW`, `B`, and `AR` channels and advanced packetization for the AXI `W` and `R` channels.

B. MULTIPLEXER AND DEMULTIPLEXER

Our design uses four multiplexer/demultiplexer instantiations. The first 2:1 priority multiplexer prioritizes data stream vectors coming from the interrupt data stream and only forwards a data stream vector from the credit data stream if no data stream vector from the interrupt data stream is available. The 5:1 multiplexer forwards data stream vectors in a round-robin fashion; it selects a data stream vector from the AXI-related data streams and passes it on to the second 2:1 priority multiplexer. The second 2:1 priority multiplexer forwards either the output from the first 2:1 priority multiplexer or the output from the 5:1 multiplexer, prioritizing the former. The 1:7 demultiplexer forwards each incoming data stream vector to the correct output module.

C. ERROR CORRECTION CODE (ECC) ENCODING AND DECODING

An Error Correcting Code (ECC) encoding module adds m bits to the incoming data stream vector, and a complementary ECC decoding module removes these bits and performs error correction and detection. Together, these modules implement SECDED (single error correction, double error detection) [41], a Hamming(63,57) [42] scheme with an additional parity bit. The Prism Bridge provides a signal that signifies a correctable error (i.e., a single-bit error) and a signal that signifies a detected but uncorrectable error (i.e., a double-bit error). Both of these signals are asserted for one clock cycle after the clock cycle in which the respective error occurs. The designer employing the Prism Bridge can utilize these signals as interrupt request sources or connect suitable counter cores to them for diagnosis purposes, for example.

D. CLOCK DOMAIN CROSSINGS

The Prism Bridge core is in the same clock domain as its one or two AXI interfaces. Its two AXI-Stream interfaces can—and are assumed to—be in a different clock domain. We call the former and the latter clock domain the AXI and the AXIS clock domain, respectively. In the most likely usage scenario we envisioned, the signals in the AXI clock domain are synchronized with one of the Programmable Logic (PL) clocks, and the signals in the AXIS clock domain are synchronized with the clock signal provided by the link-layer core. Crossing clock domains is an intricate topic involving metastability [43]. When designing with Vivado, designers can delegate the handling of such intricacies to Xilinx Parameterized Macros (XPMs) [44]. The Prism Bridge uses the AXI-Stream FIFO XPM for the ingress and the egress CDC

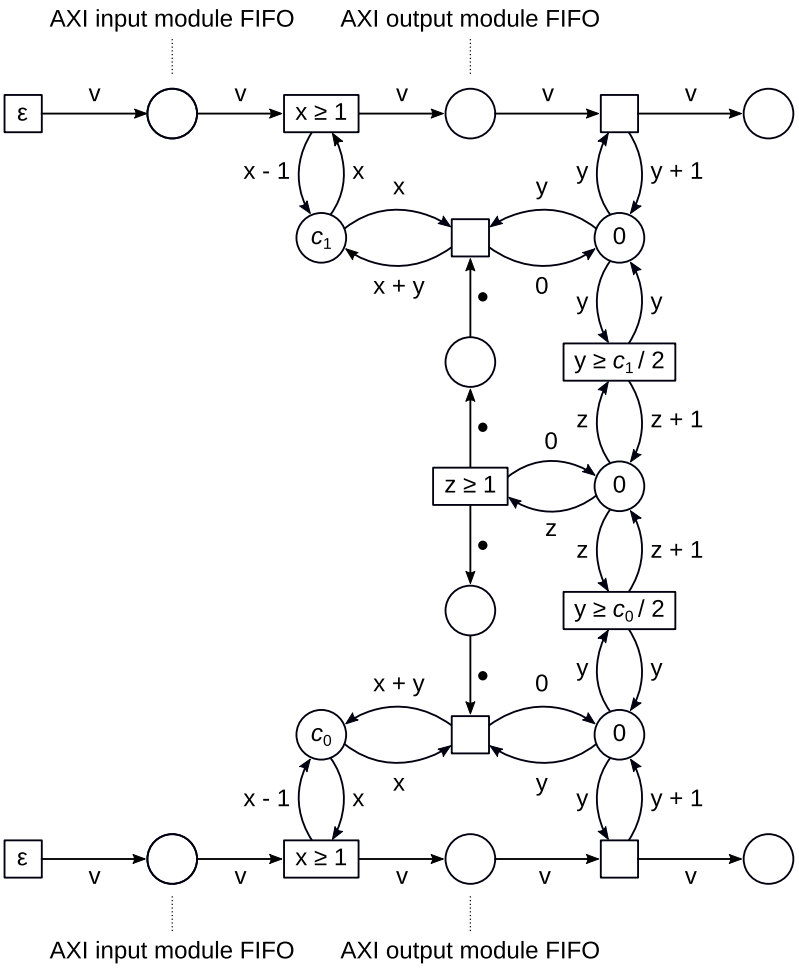


FIGURE 4. Abstract Petri net model of the flow-control mechanism for two channels. *v* represents an AXI channel vector. c_0 and c_1 represent the initial number of credits for data stream 0 and 1, respectively.

FIFO. It is parameterized in both instances to work with independent clocks and to have the minimum possible depth of 16 elements. Note that a CDC FIFO XPM can easily be replaced with a regular FIFO XPM to optimize latency and logic resource utilization in an alternative usage scenario with a single clock domain. For example, both the signals in the AXI clock domain and the signals in the AXIS clock domain could be synchronized with the clock signal provided by the link-layer core, forming a single, all-encompassing clock domain.

E. FLOW CONTROL

The AXI protocol uses a *valid*/*ready* handshaking mechanism for each of its five channels, so both sides of an AXI channel can stall the transmission. Forwarding the *ready* signal over the Prism Bridge would induce an unacceptable performance penalty. Therefore, a handshake can occur on an incoming AXI channel with an asserted output *ready* signal (on Prism Bridge A) independently from the state of the input *ready* signal of the corresponding outgoing AXI channel (on Prism Bridge B). Consequently, flow control is needed to avoid overflowing the output module's FIFO when the AXI channel vector is subsequently serialized and forwarded to be deserialized on Prism Bridge B. The Prism Bridge solves this using a credit-token-based flow control system that involves a bookkeeping module used to acquire tokens for incoming AXI channels and a bookkeeping module to release tokens for outgoing AXI channels. For each AXI channel, an independent pool of credit tokens is maintained in both bookkeeping modules. The credit token acquiring module keeps track of available tokens, and the credit token releasing module keeps track of tokens that have been released and are to be made available. A credit token must be available for the incoming AXI channel vector to be passed on from the FIFO to the packetizer. When the AXI channel vector is passed to the packetizer, the credit token acquiring module removes one token from the pool of the corresponding channel. In the same manner, when an AXI handshake occurs to transfer an outgoing AXI channel vector, the credit token releasing module adds one token to its pool for the corresponding channel. When a certain threshold of credit tokens for a channel in the credit token releasing module has been reached, the current

state of its credit token pools is sent on the credit data stream to the remote bridge, and all credit token pools in the credit pool releasing module are reset to zero—the credit tokens pools are, figuratively, transferred to the remote bridge. Upon reception of the credit token pools, the remote bridge's credit token acquiring module adds the tokens of each of these pools to its corresponding own pool, and, thus, the credit tokens are available again. In the default configuration, each channel's threshold to trigger the transfer of all released credit tokens to the remote bridge is half the maximum number of available tokens for that channel. Figure 4 shows an abstract view of this system as a Petri net based on the modeling presented in [45].

IV. BANDWIDTH ANALYSIS

For the following analysis, we use an AXI write address channel width (w_{aw}) of 99 bits, the AXI write channel (w_{w}) has a width of 73 bits, the AXI write response channel (w_{b}) has a width of 8 bits, the AXI read address channel (w_{ar}) has a width of 99 bits, and lastly, the AXI read channel (w_{r}) is 73 bits wide. This represents a signal width configuration with `AxADDR` and `xDATA` signals that have a width of 64 bits, `AxID`/`xID` signals that are 6 bits wide and without `AxUSER`/`xUSER` signals.

We define the $\mathcal{W}_{u,v}$ function that returns the number of AXI-Stream transfer cycles necessary to transfer t AXI transactions of length l with indices for the AXI channel type and packetizer behavior:

for the simple packetizer

$$\begin{aligned}
\mathcal{V}_{\mathrm{simple}}(x,t,l) &= t\cdot\left\lceil\frac{x}{k_{\mathrm{msg}}}\right\rceil\\
\mathcal{W}_{\mathrm{aw},v}(t,l) &= V_{\mathrm{simple}}(w_{\mathrm{aw}},t,l)\\
\mathcal{W}_{\mathrm{b},v}(t,l) &= V_{\mathrm{simple}}(w_{\mathrm{b}},t,l)\\
\mathcal{W}_{\mathrm{ar},v}(t,l) &= V_{\mathrm{simple}}(w_{\mathrm{ar}},t,l)\\
\mathcal{V}_0(x,t,l) &= tl\cdot\left\lceil\frac{x}{k_{\mathrm{msg}}}\right\rceil\\
\mathcal{W}_{\mathrm{w},0}(t,l) &= \mathcal{V}_0(w_{\mathrm{w}},t,l)\\
\mathcal{W}_{\mathrm{r},0}(t,l) &= \mathcal{V}_0(w_{\mathrm{r}},t,l)\,,
\end{aligned}$$

for the advanced packetizer if its gearbox is never reset

$$\begin{aligned}
\mathcal{V}_1(x,t,l) &= \left\lceil\frac{tl\,(x+1)}{k_{\mathrm{msg}}}\right\rceil\\
\mathcal{W}_{\mathrm{w},1}(t,l) &= \mathcal{V}_1(w_{\mathrm{w}},t,l)\\
\mathcal{W}_{\mathrm{r},1}(t,l) &= \mathcal{V}_1(w_{\mathrm{r}},t,l)\,,
\end{aligned}$$

if its gearbox is reset after each AXI transaction

$$\begin{aligned}
\mathcal{V}_2(x,t,l) &= t\cdot\left\lceil\frac{l\,(x+1)}{k_{\mathrm{msg}}}\right\rceil\\
\mathcal{W}_{\mathrm{w},2}(t,l) &= \mathcal{V}_2(w_{\mathrm{w}},t,l)\\
\mathcal{W}_{\mathrm{r},2}(t,l) &= \mathcal{V}_2(w_{\mathrm{r}},t,l)\,,
\end{aligned}$$

and finally, if its gearbox is reset after each AXI transfer

$$\begin{aligned}
\mathcal{V}_3(x,t,l) &= tl\cdot\left\lceil\frac{x+1}{k_{\mathrm{msg}}}\right\rceil\\
\mathcal{W}_{\mathrm{w},3}(t,l) &= \mathcal{V}_3(w_{\mathrm{w}},t,l)\\
\mathcal{W}_{\mathrm{r},3}(t,l) &= \mathcal{V}_3(w_{\mathrm{r}},t,l)\,.
\end{aligned}$$

We define the $\mathcal{B}^{(a)}_{\tau,v}$ function that gives the number of bits transferred for the traffic class a. The decorations $\rightarrow$ and $\leftarrow$ signify a transfer to and from the bridge with the AXI master interface, respectively. We define the function for

AXI data

$$\begin{aligned}
\overrightarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{w},v}(t,l) &= tls_{\mathrm{w}}\\
\overleftarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{w},v}(t,l) &= 0\\
\overrightarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{r},v}(t,l) &= 0\\
\overleftarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{r},v}(t,l) &= tls_{\mathrm{r}}\,,
\end{aligned}$$

AXI metadata

$$\begin{aligned}
\overrightarrow{\mathcal{B}}^{(\mathrm{meta})}_{\mathrm{w},v}(t,l) &= t\,(w_{\mathrm{aw}}+l\,(w_{\mathrm{w}}-s_{\mathrm{w}}))\\
\overleftarrow{\mathcal{B}}^{(\mathrm{meta})}_{\mathrm{w},v}(t,l) &= tw_{\mathrm{b}}\\
\overrightarrow{\mathcal{B}}^{(\mathrm{meta})}_{\mathrm{r},v}(t,l) &= tw_{\mathrm{ar}}\\
\overleftarrow{\mathcal{B}}^{(\mathrm{meta})}_{\mathrm{r},v}(t,l) &= tl\,(w_{\mathrm{r}}-s_{\mathrm{r}})\,,
\end{aligned}$$

packetizer overhead

$$\begin{aligned}
\overrightarrow{\mathcal{B}}^{(\mathrm{oh})}_{\mathrm{w},v}(t,l) &= k_{\mathrm{msg}}\left(\mathcal{W}_{\mathrm{aw},v}(t,l)+\mathcal{W}_{\mathrm{w},v}(t,l)\right)\\
&\quad - tw_{\mathrm{aw}} - tlw_{\mathrm{w}}\\
\overleftarrow{\mathcal{B}}^{(\mathrm{oh})}_{\mathrm{w},v}(t,l) &= k_{\mathrm{msg}}\mathcal{W}_{\mathrm{b},v}(t,l) - tw_{\mathrm{b}}\\
\overrightarrow{\mathcal{B}}^{(\mathrm{oh})}_{\mathrm{r},v}(t,l) &= k_{\mathrm{msg}}\mathcal{W}_{\mathrm{ar},v}(t,l) - tw_{\mathrm{ar}}\\
\overleftarrow{\mathcal{B}}^{(\mathrm{oh})}_{\mathrm{r},v}(t,l) &= k_{\mathrm{msg}}\mathcal{W}_{\mathrm{r},v}(t,l) - tlw_{\mathrm{r}}\,,
\end{aligned}$$

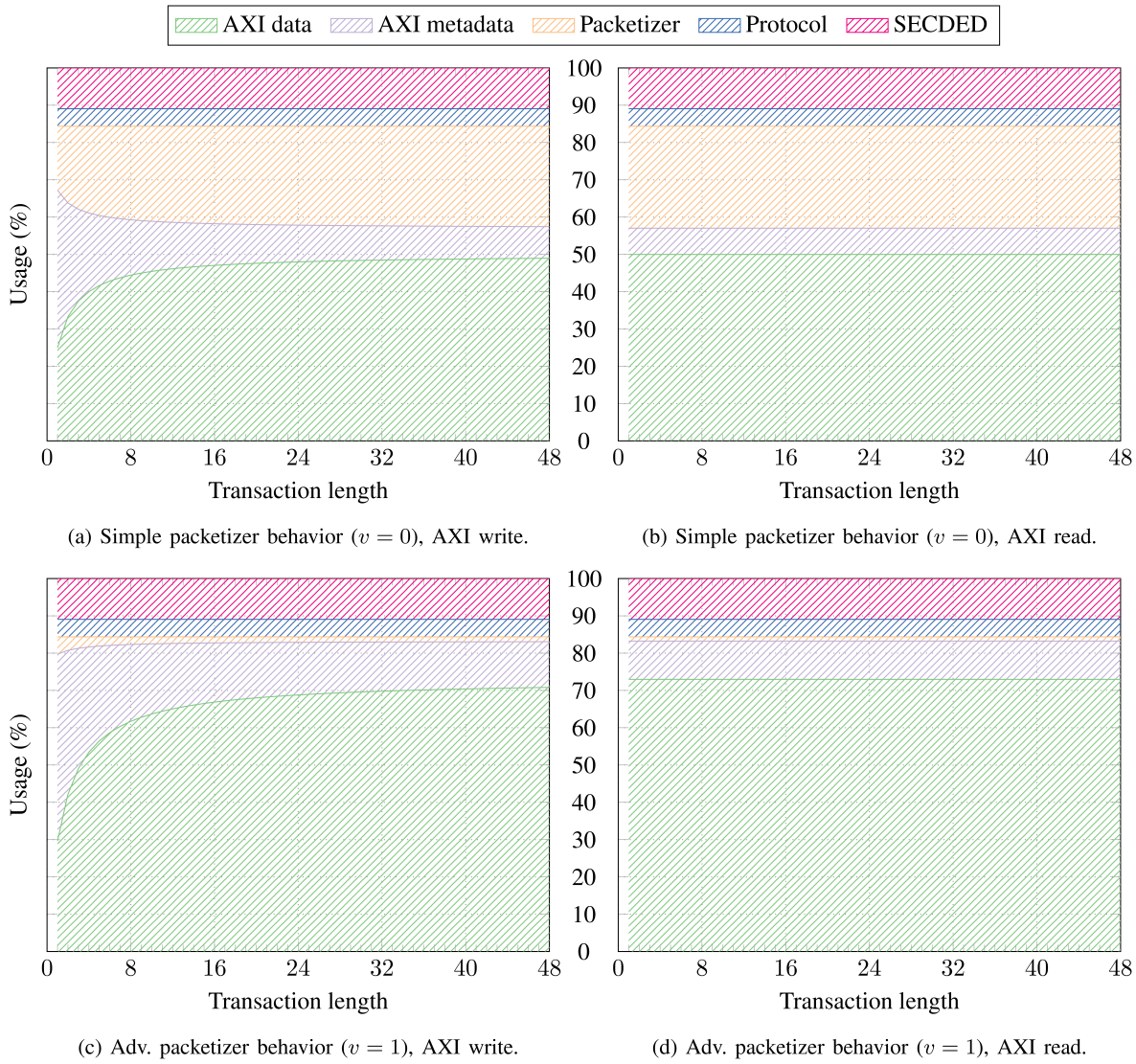
protocol overhead

$$\begin{aligned}
\overrightarrow{\mathcal{B}}^{(\mathrm{proto})}_{\mathrm{w},v}(t,l) &= k_{sid}\left(\mathcal{W}_{\mathrm{aw},v}(t,l)+\mathcal{W}_{\mathrm{w},v}(t,l)\right)\\
\overleftarrow{\mathcal{B}}^{(\mathrm{proto})}_{\mathrm{w},v}(t,l) &= k_{sid}\mathcal{W}_{\mathrm{b},v}(t,l)\\
\overrightarrow{\mathcal{B}}^{(\mathrm{proto})}_{\mathrm{r},v}(t,l) &= k_{sid}\mathcal{W}_{\mathrm{ar},v}(t,l)\\
\overleftarrow{\mathcal{B}}^{(\mathrm{proto})}_{\mathrm{r},v}(t,l) &= k_{sid}\mathcal{W}_{\mathrm{r},v}(t,l)\,,
\end{aligned}$$

and SECDED overhead

$$\begin{aligned}
\overrightarrow{\mathcal{B}}^{(\mathrm{secded})}_{\mathrm{w},v}(t,l) &= m\left(\mathcal{W}_{\mathrm{aw},v}(t,l)+\mathcal{W}_{\mathrm{w},v}(t,l)\right)\\
\overleftarrow{\mathcal{B}}^{(\mathrm{secded})}_{\mathrm{w},v}(t,l) &= m\mathcal{W}_{\mathrm{b},v}(t,l)\\
\overrightarrow{\mathcal{B}}^{(\mathrm{secded})}_{\mathrm{r},v}(t,l) &= m\mathcal{W}_{\mathrm{ar},v}(t,l)\\
\overleftarrow{\mathcal{B}}^{(\mathrm{secded})}_{\mathrm{r},v}(t,l) &= m\mathcal{W}_{\mathrm{r},v}(t,l)\,.
\end{aligned}$$

With the widths w_u defined in the beginning of this section, this function is used to visualize the bandwidth usage in



(a) Simple packetizer behavior ($v = 0$), AXI write.

(b) Simple packetizer behavior ($v = 0$), AXI read.

(c) Adv. packetizer behavior ($v = 1$), AXI write.

(d) Adv. packetizer behavior ($v = 1$), AXI read.

FIGURE 5. Half-duplex bandwidth utilization.

Figure 5, wherein the increased bandwidth used for AXI metadata and data and the decreased bandwidth used for packetizer overhead are prominently visible.

Assuming an egress CDC FIFO (on the bridge that transmits the AXI `W` or `R` channel) that never underflows, and using

$$\begin{aligned}\mathcal{W}'_{\mathrm{w},v}(l) &= \mathcal{W}_{\mathrm{aw},v}(k_{\mathrm{msg}}, l) + \mathcal{W}_{\mathrm{w},v}(k_{\mathrm{msg}}, l)\\ \mathcal{W}'_{\mathrm{r},v}(l) &= \mathcal{W}_{\mathrm{r},v}(k_{\mathrm{msg}}, l)\\ \mathcal{B}'^{(\mathrm{data})}_{\mathrm{w},v}(l) &= \overrightarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{w},v}(k_{\mathrm{msg}}, l)\\ \mathcal{B}'^{(\mathrm{data})}_{\mathrm{r},v}(l) &= \overleftarrow{\mathcal{B}}^{(\mathrm{data})}_{\mathrm{r},v}(k_{\mathrm{msg}}, l)\,,\end{aligned}$$

the AXI data throughput (in bit/s) from bridge to bridge can be approximated by

$$\mathcal{X}_{\tau,v}(l) = \frac{f_{\mathrm{axis}} \cdot R_{\mathrm{c}}^{(\mathrm{link})} \cdot \mathcal{B}'^{(\mathrm{data})}_{\tau,v}(l)}{\mathcal{W}'_{\tau,v}(l)}\,.$$

This approximation considers traffic on the AXI write address channel `AW` and the AXI write channel `W` in case of write transactions and traffic on the AXI read channel `R` in case of read transactions. It is implied here that f_{axis} is equal to the line rate divided by n, and not every clock cycle can transfer an AXI-Stream word, because the link-layer core has to account for encoding overhead ($R_{\mathrm{c}}^{(\mathrm{link})}$). Alternatively, depending on the design of the link-layer core, f_{axis} may already be multiplied by $R_{\mathrm{c}}^{(\mathrm{link})}$.

The reason we use k_{msg} as the number of transactions in the function $\mathcal{W}'_{\tau,v}$ is that we need to insure that

$$\forall i \in \mathbb{N} : \mathcal{W}_{u,v}(it, l) = i\mathcal{W}_{u,v}(t, l)\,.$$

In other words, the number of transactions must not have a negative impact on throughput.

V. EVALUATION

A. EXPERIMENTAL SETUP

We implemented all designs for the evaluation with Vivado 2022.2 and conducted all measurements using the AMD

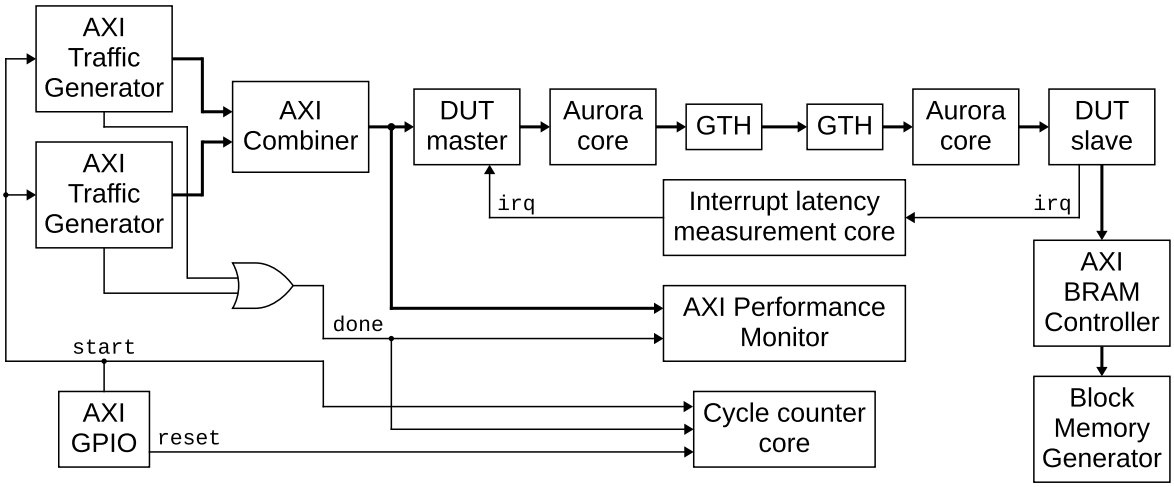
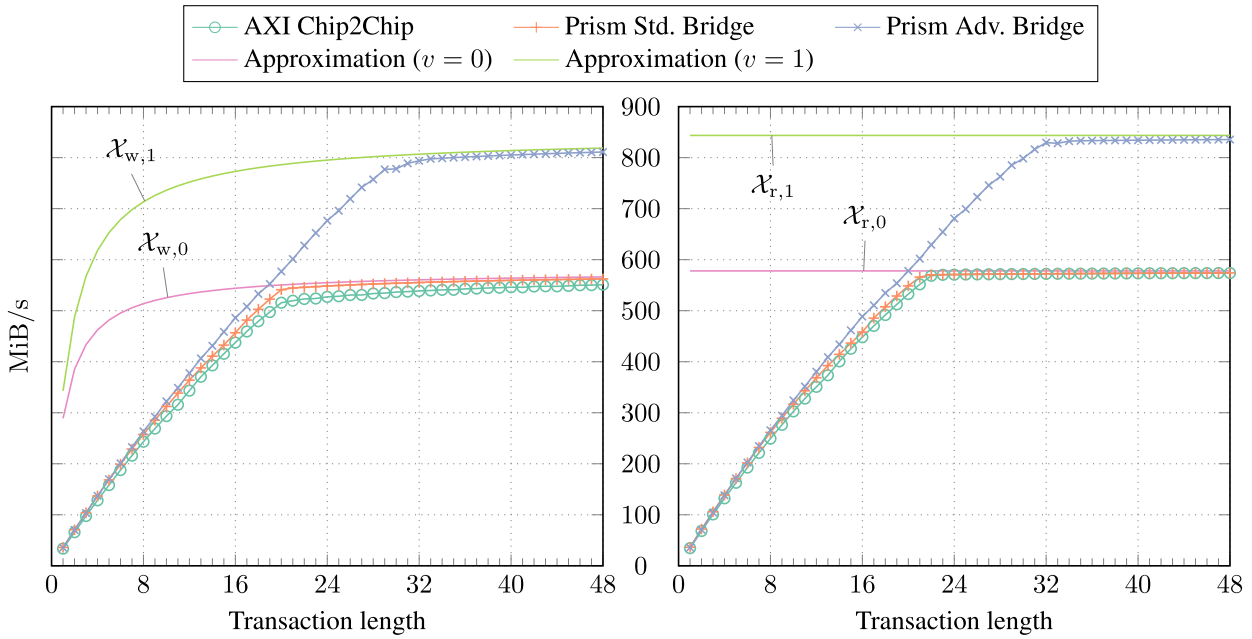


FIGURE 6. Experimental setup for AXI throughput and latency measurements.

TABLE 1. FPGA post-implementation resource utilization (No URAM and DSP blocks have been used.).

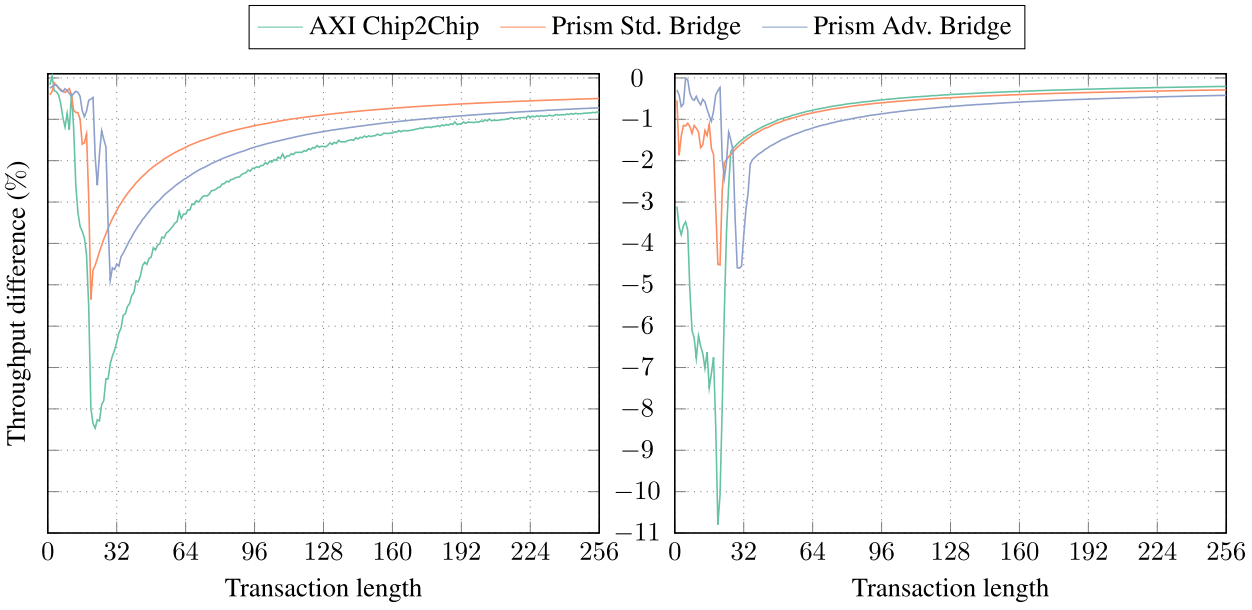
Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18
Aurora 8B/10B (clock master)	≤521	≤508	0	13	≤1155	0	0
Aurora 8B/10B (clock slave)	≤518	≤505	0	13	≤1140	0	0
Aurora 64B/66B (clock master, < 10 Gbit/s)	647	612	0	35	≤2001	1	0
Aurora 64B/66B (clock slave, < 10 Gbit/s)	≤647	≤613	0	34	≤1982	1	0
Aurora 64B/66B (clock master, ≥ 10 Gbit/s)	≤459	≤424	0	35	≤1587	1	0
Aurora 64B/66B (clock slave, ≥ 10 Gbit/s)	≤457	≤423	0	34	≤1568	1	0
Prism Kyokko (all line rates)	≤951	≤945	0	6	1501	2	0
Aurora Control	≤70	≤70	0	0	≤228	0	0
AXI Chip2Chip (AXI slave, Aurora 8B/10B)	705	685	16	4	1111	3	1
AXI Chip2Chip (AXI slave, Aurora 64B/66B)	≤976	≤932	40	4	1434	4	4
AXI Chip2Chip (AXI master, Aurora 8B/10B)	651	631	16	4	1140	3	1
AXI Chip2Chip (AXI master, Aurora 64B/66B)	≤898	≤854	40	4	1554	4	2
Prism Std. Bridge (AXI slave, Aurora 8B/10B)	1045	965	80	0	1964	2	2
Prism Std. Bridge (AXI slave, Aurora 64B/66B)	≤989	≤909	80	0	1829	2	2
Prism Std. Bridge (AXI slave, Prism Kyokko)	≤994	≤914	80	0	1829	2	2
Prism Std. Bridge (AXI master, Aurora 8B/10B)	1077	997	80	0	1837	8	1
Prism Std. Bridge (AXI master, Aurora 64B/66B)	≤1021	≤941	80	0	1702	8	1
Prism Std. Bridge (AXI master, Prism Kyokko)	≤1028	≤948	80	0	1702	8	1
Prism Bridge simple packetizer	6	6	0	0	76	0	0
Prism Bridge simple depacketizer	31	31	0	0	76	0	0
Prism Adv. Bridge (AXI slave, Aurora 8B/10B)	1773	1693	80	0	2579	2	2
Prism Adv. Bridge (AXI slave, Aurora 64B/66B)	≤1720	≤1640	80	0	2444	2	2
Prism Adv. Bridge (AXI slave, Prism Kyokko)	≤1724	≤1644	80	0	2444	2	2
Prism Adv. Bridge (AXI master, Aurora 8B/10B)	1789	1709	80	0	2453	8	1
Prism Adv. Bridge (AXI master, Aurora 64B/66B)	≤1727	≤1647	80	0	2318	8	1
Prism Adv. Bridge (AXI master, Prism Kyokko)	≤1736	≤1656	80	0	2318	8	1
Prism Bridge adv. packetizer	342	342	0	0	357	0	0
Prism Bridge adv. depacketizer	324	324	0	0	410	0	0

Xilinx Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC on a ZCU102 evaluation board [46]. The components of the experimental setup and their relations are depicted in Figure 6. We used an AMD Xilinx AXI Traffic Generator [47] as the AXI master (i.e., transaction source) and configured it for custom traffic generation in *advanced* mode to allow for maximum flexibility. We used an AMD Xilinx AXI Block RAM (BRAM) Controller [48] connected to a True Dual-Port Block RAM generated by an AMD Xilinx Block Memory Generator [49] as the AXI slave (i.e., transaction sink). We conducted measurements of AXI-related metrics using the AMD Xilinx AXI Performance Monitor [50], and we developed and used custom cores to measure IRQ latency and count clock cycles. The AXI channel widths and individual AXI signal widths used in all evaluation designs were identical to those in Section IV. We configured each AXI Chip2Chip core combined with the Aurora 64B/66B core to use the appropriate AXI data width, address width, ID width, and `WUSER` width. However, we configured each AXI Chip2Chip combined with the Aurora 8B/10B core to use an AXI address width of 32 bits and an AXI data width of 32 bits. This was done



(a) AXI write. (b) AXI read.

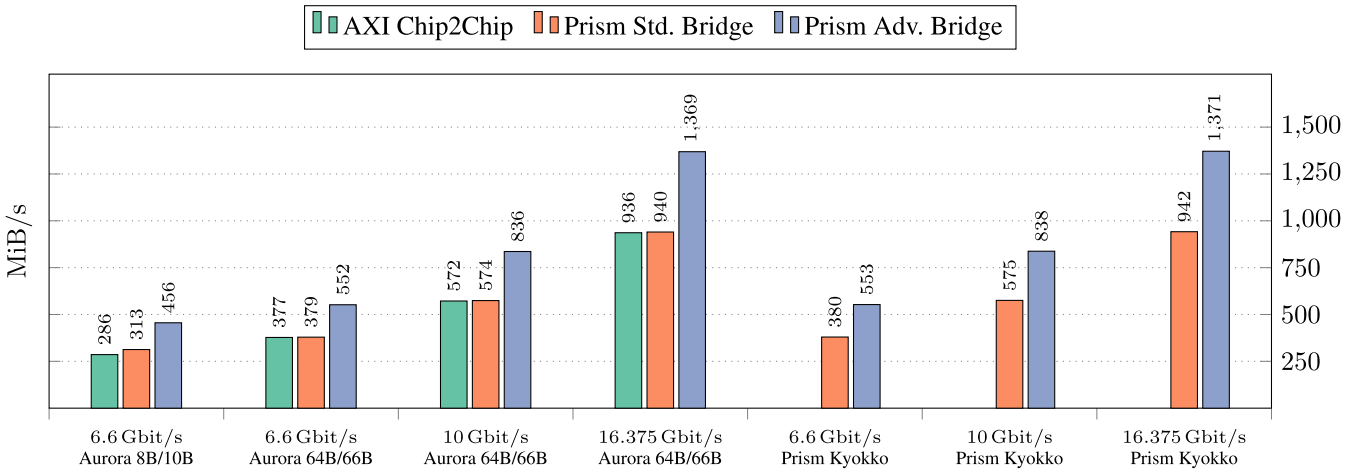
FIGURE 7. Half-duplex AXI throughput (10 Gbit/s line rate, Aurora 64B/66B).



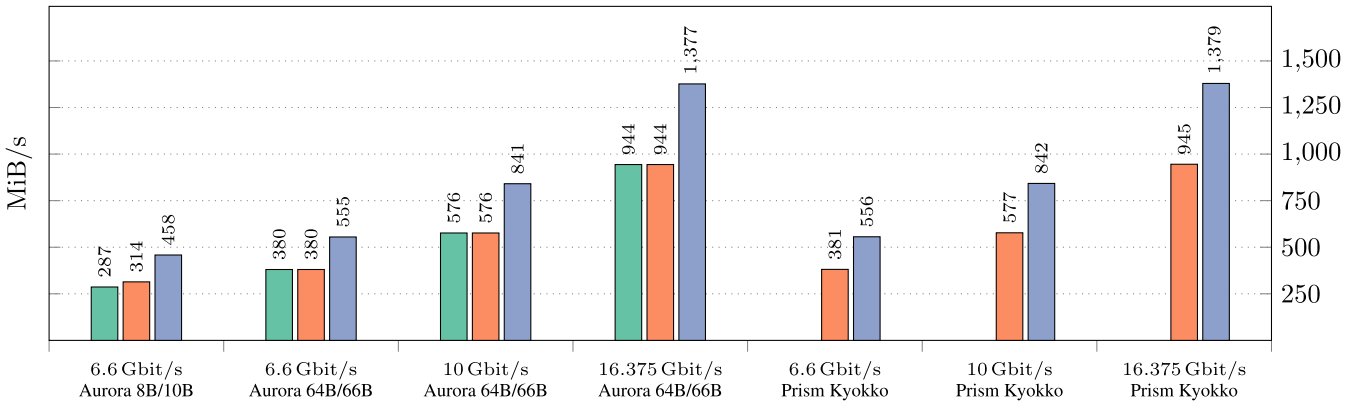
(a) AXI write. (b) AXI read.

FIGURE 8. Difference of full-duplex AXI throughput compared to half-duplex AXI throughput (10 Gbit/s line rate, Aurora 64B/66B).

because the AXI Chip2Chip core does not support single-lane operation if used with the Aurora 8B/10B core and either the AXI address width or the AXI data width is set to 64 bits. Hence, we used an AMD Xilinx AXI Data Width Converter to convert the 64-bit data width of the AXI Traffic Generator's AXI master interface to the 32-bit data width of the AXI Chip2Chip core's AXI slave interface for this case. A disadvantage of the AXI Data Width Converter is that it removes the AXI `AxID` signal, thereby reducing the number of outstanding transactions to one. Note that we left the AXI address width set to 32 bits for the AXI Chip2Chip and Aurora 8B/10B combination. All AXI Chip2Core parameters not concerned with AXI or PHY configuration remained set to their default values. We evaluated the AXI Chip2Chip core,



(a) AXI write.



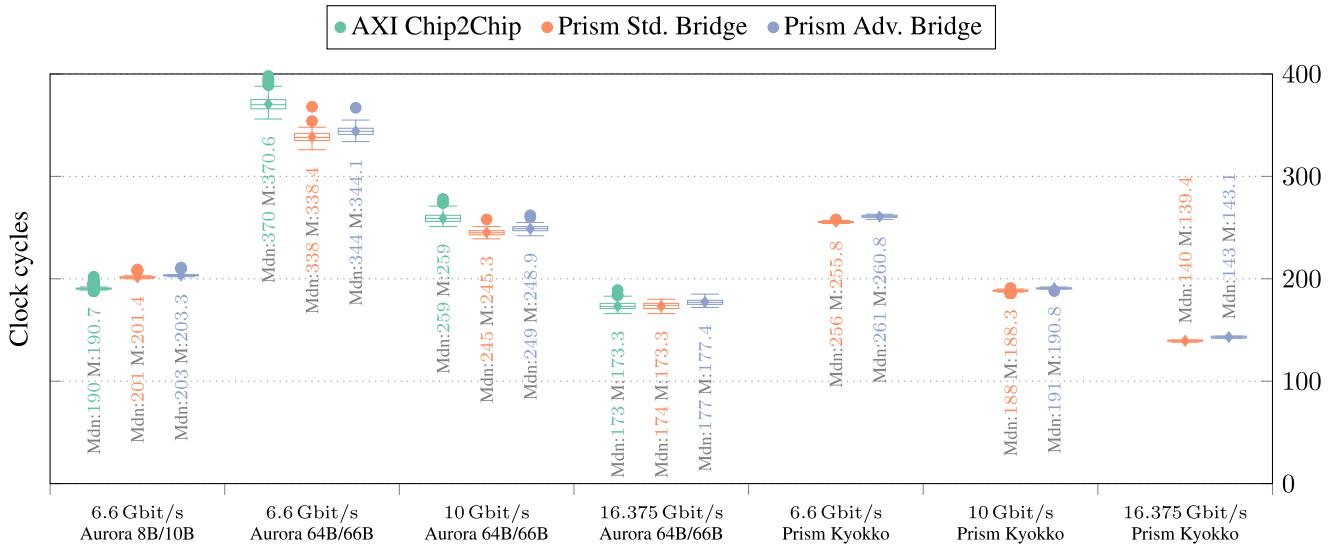
(b) AXI read.

FIGURE 9. Half-duplex AXI throughput of transactions with a length of 256 transfers.

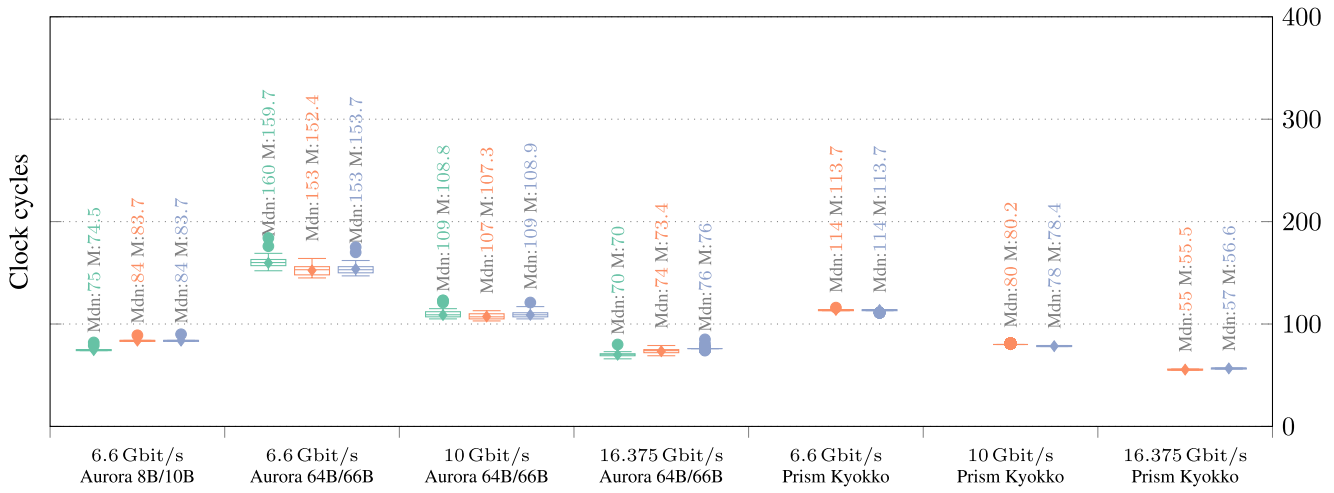
Prism Standard Bridge, and Prism Advanced Bridge in combination with the Aurora 8B/10B and the Aurora 64B/66B core. Additionally, we evaluated the Prism Standard Bridge and Prism Advanced Bridge in combination with the Prism Kyokko core. Although the AXI Chip2Chip core uses AXI-Stream to communicate with an Aurora core, it needs additional signals not provided by the Prism Kyokko core. Hence, we did not evaluate the AXI Chip2Chip core with the Prism Kyokko core. We performed the evaluations with the Aurora 8B/10B core with a line rate of 6.6 Gbit/s and the evaluations with the Aurora 64B/66B core and the Prism Kyokko core with line rates of 6.6 Gbit/s, 10 Gbit/s, and 16.375 Gbit/s. We configured the Aurora 8B/10B and Aurora 64B/66B core to include shared logic when serving as the link-layer core for a bridge with an AXI slave interface and to use external shared logic when serving as the link-layer core for a bridge with an AXI master interface. We call an Aurora core with the former configuration a clock master and one with the latter configuration a clock slave.

Most cores and all AXI interfaces in the designs used the PL clock PL0 generated by the RPLL with a frequency of ∼280 MHz. The Aurora cores used the PL clock PL1 generated by the RPLL with a frequency of ∼50 MHz as the initialization clock.

The four SFP+ (Enhanced Small Form-factor Pluggable) ports on the ZCU102 board are connected to the GTH Quad 230. The ZCU102 board has two SiliconLabs Si570 chips, each producing a differential clock signal with a programmable frequency. One of these clock signals is buffered by a Si53340 chip and distributed to the GTH Quad 230 as `MGTREFCLK0`. Si570 chips are programmable via I^2C; the Linux driver for the Si570 uses the frequency specified in the device tree to set the frequency when booting. We used `MGTREFCLK0` of GTH Quad 230 as the GT reference clock with a frequency of 132 MHz for a line rate of 6.6 Gbit/s and with a frequency of 125 MHz for line rates of ≥ 10 Gbit/s. We used the GT lanes corresponding to the SFP+ ports 0 and 1, which we connected with a direct-attach copper (DAC) cable. To ensure the measurement results of the higher level protocols were not skewed by the bit error rate (BER) of the physical link, we evaluated FS SFP-H10GB-CU50CM cables specified to 10 Gbit/s with a length of 0.5 m.



(a) AXI write.



(b) Interrupt request.

FIGURE 10. Latency.

We used the AMD Xilinx IBERT core for UltraScale GTH Transceivers [51] for BER tests and tested the cables with the line rates 6.6 Gbit/s, 10 Gbit/s, and 16.375 Gbit/s. All of the available bit patterns for BER tests were tested for one hour, namely PRBS-7, PRBS-9, PRBS-15, PRBS-23, PRBS-31, slowclk, and fastclk. The specific cable we selected for all tests achieved a BER of zero for all combinations of line rate and bit pattern.

The application that controls the measurement process was compiled with AMD Xilinx Vitis 2022.2 and ran on a Linux system we created with AMD Xilinx PetaLinux 2022.2. The AXI Traffic Generator has a command RAM that is separated into two arrays, each containing 256 commands, each command comprised of four 32-bit words. The first array is used for commands describing read transactions, and the second one is used for commands describing write transactions; both are processed simultaneously. There are two AXI Traffic Generators in each of our designs: an AXI Traffic Generator handling the writing-related AXI channels (XATG-W) and one handling the reading-related AXI channels (XATG-R). For half-duplex AXI throughput measurements, the AXI Traffic Generator handling the channels of interest ran a program of 255 commands while the other AXI Traffic Generator remained idle. Two separate runs were necessary to conduct full-duplex AXI throughput measurements. To observe AXI write throughput in full-duplex mode, we generated a write command array and ran it on ATG-W, and we generated a read command array and ran it looped on ATG-R—vice-versa to observe AXI read throughput. Using two unlooped command arrays would have stopped the cycle counter as

soon as the processing of any one of the two command arrays finished. The Loop Enable bit in the Master Control register to enable looped processing affects the processing of both write and read command arrays; this is the reason we used two separate AXI Traffic Generator instances. We configured the commands to drive the AXI signals as follows:

- `WSTRB`=FF_{16}
- `AxPROT`=0
- `AxID`=$i \mod j$ (where i is the command number and j is the number of supported outstanding transactions)
- `AxSIZE`=3
- `AxBURST`=1 (`INCR`)
- `AxLOCK`=0
- `AxLEN` is set corresponding to the currently tested transaction length
- `AxQOS`=0
- `AxCACHE`=3

A consequential limitation of the AXI Traffic Generator is that it supports a maximum of four outstanding transactions. We conducted every AXI-related measurement using the following sequence:

1) generate appropriate command arrays,
2) configure and program the AXI Traffic Generators,
3) reset the clock cycle counter via a dedicated GPIO pin,
4) start both AXI Traffic Generators (using their external trigger input) and the clock cycle counter simultaneously via a dedicated GPIO pin, and
5) wait for the Master Completion bit of the relevant AXI Traffic Generator's Error Status register to be set.

The signal coming from the AXI Traffic Generator's `done` port stops the clock cycle counter, resulting in an accurate measurement of the number of clock cycles the processing of the command arrays took. The AXI throughput measurements were repeated 100 times and averaged, while the latency measurements were repeated 1000 times. We conducted AXI latency measurements using an AXI Traffic Generator command array containing a single valid command, producing a transaction with a single transfer (`AxLEN` $= 0$). Because the processing of a write command array is not completed until the last AXI write response (`B`) channel handshake occurs, our write transaction latency measurements are effectively round-trip measurements.

B. RESULTS

1) LOGIC RESOURCES

Table 1 shows the post-implementation logic resource utilization for the AMD Xilinx Aurora and AXI Chip2Chip cores, the Prism Kyokko, Aurora Control and Prism Bridge cores, and the latter's packetizer and depacketizer modules. Note that the logic utilization of the same instantiated module in different designs often differs. Hence, the values shown in the table and referred to in the following discussion are the worst-case (i.e., highest) values from the designs used for the evaluation.

The Aurora 64B/66B core configured for a line rate of $<$10 Gbit/s needed up to 188 look-up tables (LUTs) and 414 flip-flops more than when configured for a line rate of $\geq$10 Gbit/s. This is because the AMD Xilinx Transceivers Wizard [52] embedded in the Aurora 64B/66B core instantiates CPLL calibration logic when the CPLL is used for clocking. The Aurora 64B/66B core uses the CPLL for the 6.6 Gbit/s line rate, and a QPLL for the higher line rates. The Prism Kyokko core currently uses a QPLL for all line rates. For line rates of $\geq$10 Gbit/s, the Prism Kyokko core needed 35 LUTs (3.82%) more than what two complementary Aurora 64B/66B cores needed. However, two Aurora 64B/66B cores required 1654 flip-flops (110.19%) more than the Prism Kyokko core.

The Prism Bridge uses more logic resources when linked to the Aurora 8B/10B core because additional logic is needed to change the AXI data width to the internally used 64 bits.

Comparing bridges with an AXI slave interface, combined with the Aurora 64B/66B core, the AXI Chip2Chip core used up to 976 LUTs and 1434 flip-flops, the Prism Standard Bridge used up to 989 LUTs and 1829 flip-flops, and the Prism Advanced Bridge used up to 1720 LUTs and 2444 flip-flops. While the Prism Advanced Bridge needed 76.23% more LUTs and 70.43% more flip-flops than the AXI Chip2Chip core, it needed 73.91% more LUTs and 33.62% more flip-flops than the Prism Standard Bridge. The latter comparison gives a better idea of the logic resource cost premium induced by the advanced packetization required for the higher throughput, as that is the only difference between both variants.

Similarly, comparing bridges with an AXI master interface, combined with the Aurora 64B/66B core, the AXI Chip2Chip core used up to 898 LUTs and 1554 flip-flops, the Prism Standard Bridge used up to 1021 LUTs and 1702 flip-flops, and the Prism Advanced Bridge used up to 1727 LUTs and 2318 flip-flops. We see that the Prism Advanced Bridge caused a 92.32% increase in LUT utilization and a 49.16% increase in flip-flop utilization compared with the AXI Chip2Chip core and a 69.15% increase in LUT utilization and a 36.19% increase in flip-flop utilization compared with the Prism Standard Bridge.

To put the logic resource requirements into context: The Prism Advanced bridge consumed up to 1789 LUTs (0.65%) and 2579 flip-flops (0.47%) of the 274 080 LUTs and 548 160 flip-flops provided by the ZU9EG [53] chip on our test board.

2) THROUGHPUT

The half-duplex AXI throughput we measured for a line rate of 10 Gbit/s and with the Aurora 64B/66B core is visualized in Figure 7. The curves of the measurement results for the other line rates look similar. Each measurement curve aligns visibly with its corresponding approximation curve after its knee point. The steep increase of the measurement curves observed ahead of their knee point is a consequence of the AXI Traffic Generator's limitation to four outstanding transactions. For example, when the AXI Traffic Generator has submitted the last transfer of the write transaction

№109, №110 will not be started and therefore no data will be transferred until the AXI Traffic Generator has received the response from write transaction №106. Depending on the transaction length, the AXI Traffic Generator may receive the write response for №106 before or after the last transfer of №109 occurred. In the latter case, idle time accrues, resulting in the throughput gap observed in the curve before the knee point. A similar issue arises when the AXI Traffic Generator has to wait for a read transfer with the `RLAST` signal asserted to complete a previous read transaction in order to be able to start the next one. On the other hand, the response for a transaction with a sufficient transaction length is received during the transfers of a subsequent transaction, precluding the idle time.

Figure 8 shows by what percentage AXI throughput differs in full-duplex operation compared to half-duplex operation when using a line rate of 10 Gbit/s and the Aurora 64B/66B core. Each curve appears turbulent before and gradually increasing after crossing a certain transaction length threshold. Once again, this point occurs earlier on the x-axis for the AXI Chip2Chip core and the Prism Standard Bridge. The AXI Chip2Chip AXI write throughput difference curve has a small amount of unsteadiness sprinkled in—an effect not shown by the AXI read throughput difference curve of the AXI Chip2Chip core. The transaction length at which the gradual increase starts is, not coincidently, the same transaction length at which the knee point in the corresponding curve in Figure 7 occurs. The highest AXI write throughput difference is −8.47% (at transaction length 22) for the AXI Chip2Chip core, −5.35% (at transaction length 20) for the Prism Standard Bridge, and −4.89% (at transaction length 29) for the Prism Advanced Bridge. The highest AXI read throughput difference is −10.80% (at transaction length 20) for the AXI Chip2Chip core, −4.52% (at transaction length 21) for the Prism Standard Bridge, and −4.60% (at transaction length 30) for the Prism Advanced Bridge.

Figure 9 presents the maximum achievable half-duplex AXI throughput of write and read transactions. It shows the measurement results for the maximum transaction length: transactions made up of 256 transfers (`AxLEN` = 255). Notably, the design using the Aurora 8B/10B core and the AXI Chip2chip core achieved an AXI write throughput of 286.57 MiB/s and an AXI read throughput of 285.87 MiB/s. In contrast, the design using the same Aurora core and the Prism Standard Bridge achieved an AXI write throughput of 314.07 MiB/s (9.60% more) and an AXI read throughput of 312.85 MiB/s (9.44% more). This discrepancy is due to the former design's limitation to a single outstanding transaction, as described earlier.

We will now compare the results of designs with a 16.375 Gbit/s line rate and the Aurora 64B/66B core handling the link layer. Firstly, the AXI Chip2Chip core could transfer 936.29 MiB/s in write transactions and 943.80 MiB/s in read transactions. The maximum throughput achieved by the AXI Chip2Chip core was very similar to the throughput achieved by the Prism Standard Bridge. This observation strongly suggests that the AXI Chip2Chip core uses a packetization mechanism similar to the simple packetization mechanism found in the Prism Standard Bridge. Secondly, with the Prism Advanced Bridge, we measured an AXI write throughput of 1368.82 MiB/s and an AXI read throughput of 1376.62 MiB/s. Compared to the AXI Chip2Chip core, this represents a 46.20% increase in AXI write throughput and a 45.86% increase in AXI read throughput. Lastly, the Prism Advanced Bridge performed slightly better with the Prism Kyokko core than with the Aurora 64B/66B core and achieved an AXI write throughput of 1371.38 MiB/s (0.19% more) and an AXI read throughput of 1379.19 MiB/s (0.19% more).

The measured AXI throughput of transactions with a length of 256 transfers differs from the function $\mathcal{X}_{\tau,v}$ by −0.08% (Prism Standard Bridge, 6.6 Gbit/s, Prism Kyokko, AXI write throughput approximation) to −0.68% (AXI Chip2Chip, 16.375 Gbit/s, Aurora 64B/66B, AXI write throughput approximation). This excludes the measurements for the combination of a 6.6 Gbit/s line rate, Aurora 8B/10B core, and Chip2Chip core, again because of the aforementioned limitation concerning outstanding transactions.

In summary, the Prism Advanced Bridge combined with the Prism Kyokko core provided the highest AXI throughput in our tests. Therefore, we consider this combination to be the best choice for typical designs that need inter-chip AXI communication. There is one caveat: In designs that will be used to process mainly short AXI transactions, a bridge with a simple packetizer mechanism may suffice.

3) LATENCY

Figure 10 shows the results of the latency measurements. In the following discussion, we refer to and compare the median values. On the one hand, combined with the Aurora 64B/66B core and with a line rate of 6.6 Gbit/s, it took the Prism Advanced Bridge 7.03% fewer clock cycles to complete the write transaction than the AXI Chip2Chip core needed. On the other hand, with a line rate of 16.375 Gbit/s, the Prism Advanced Bridge completed the write transaction in 2.31% more clock cycles than the AXI Chip2Chip core. The interrupt request latency measurements show a concordant scenario. Considering both write and interrupt request latency, the Prism Advanced Bridge needed more clock cycles than the AXI Chip2Chip core in four cases, fewer in three cases, and the same number of clock cycles in one case. Conducting measurements without a link-layer core (e.g., AXI-Stream only) may reduce the latency variability enough for a more evident pattern to emerge.

Comparing the Prism Advanced Bridge to the Prism Standard Bridge when using the Aurora 64B/66B core, the former needed up to 1.78% more clock cycles to complete an AXI write transaction and up to 2.70% more clock cycles to forward an interrupt request than the latter.

It took the Prism Advanced Bridge combined with the Prism Kyokko core up to 24.13% fewer clock cycles to complete an AXI write transaction and up to 28.44% fewer clock

cycles to forward an interrupt request than when combined with the Aurora 64B/66B core.

Interestingly, the AXI write latency of the designs using the Aurora 8B/10B core was 40.53%–48.65% lower than the latency of the corresponding design using the Aurora 64B/66B core with the same line rate and still 17.96%–26.64% lower than the latency of a corresponding design using the Aurora 64B/66B core and a 51.52% higher line rate (10 Gbit/s). Again, interrupt request latencies behaved similarly. We measured the lowest interrupt request latency (55 clock cycles) with the Prism Standard Bridge combined with the Prism Kyokko core and with a line rate of 16.375 Gbit/s.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have reviewed related work, presented an overview of the design and implementation of the novel, throughput-optimized Prism Bridge, and given a formal analysis of its half-duplex bandwidth utilization. Furthermore, we have outlined the experimental setup and methodology of our performance evaluation of two variants of the Prism Bridge, along with one competitor bridge core.

With a 16.375 Gbit/s line rate and the Aurora 64B/66B link-layer core, the AXI Chip2Chip core achieved a throughput of 936.29 MiB/s in write transactions and 943.80 MiB/s in read transactions whereas the Prism Advanced Bridge attained an AXI write throughput of 1368.82 MiB/s and an AXI read throughput of 1376.62 MiB/s. This amounts to a 46.20% increase in AXI write throughput and a 45.86% increase in AXI read throughput. Our experiments indicate that the advanced packetization mechanism has negligible impact on AXI or interrupt request latencies. Possibly obscured by variability introduced by the link layer, clock domain crossings, or both, no clear pattern regarding latency differences among the tested bridge cores emerged. As expected, higher line rates resulted in lower latencies. The Prism Advanced Bridge combined with the Prism Kyokko core needed up to 24.13% and 28.44% fewer clock cycles to complete an AXI write transaction and to forward an interrupt request, respectively. Combined with the Aurora 64B/66B core, the Prism Advanced Bridge used 76.23%–92.32% more LUTs and 49.16%–70.43% more flip-flops than the AXI Chip2Chip core, and it needed 69.15%–73.91% more LUTs and 33.62%–36.19% more flip-flops than the Prism Standard Bridge. The Prism Kyokko core (for two transceivers) utilized slightly more LUTs but less than half of the flip-flops two Aurora 64B/66B cores required.

Based on the AXI throughput measurement results of the Prism Standard Bridge and the AXI Chip2Chip core, we suppose that the latter uses a similar simple packetization mechanism. We have shown that it is possible to substantially increase AXI write and read throughput over a high-speed serial link by leveraging our novel advanced packetization mechanism instead of a simple one. To verify the causation between the advanced packetization mechanism and the increased AXI throughput, we have compared the Prism Advanced Bridge with the Prism Standard Bridge, which uses a simple packetization mechanism but is otherwise identical to the former. In times of ever-increasing logic resources and line rates of serial links, the AXI throughput increase of the Prism Advanced Bridge likely warrants the necessary additional logic resources for most designs. In the uncommon case that a design is restricted to a comparably low line rate of $\leq$6.6 Gbit/s and latency is of paramount importance, the Aurora 8B/10B may be preferable to the Aurora 64B/66B core.

Evaluations of the Prism Bridge without a link-layer core and in combination with a link-layer core configured to provide a multi-lane channel (i.e., multiple bidirectional physical links appearing as one bidirectional logical channel) would be intriguing. While the former enables precise latency measurements, the latter provides more insight into the viability of various architectural and clock-domain–related design choices. We intend to use the architecture of the bridge presented in this paper to implement an extended bridge that supports the *AXI Coherency Extensions* (ACE) family of protocols. This bridge will facilitate the further exploration of hardware-supported coherency of memory shared across systems. Furthermore, the existing Prism Bridge Generator could be reworked into a more general application that produces protocol serializers based on input specifications. The advanced packetizer and depacketizer could be redesigned to be general (instead of individually generated) modules with access to a configuration memory block. This would lay the groundwork for an ASIC implementation configurable for specific AXI signal widths. Additionally, we are currently developing the Prism Interrupt Controller intended to add conveniences, such as selectable level- and edge-sensitivity and interrupt request routing capabilities based on the Prism Bridge's basic interrupt request forwarding. Finally, porting the Prism Bridge to Intel hardware, like the recently released Agilex SoC-FPGA series, would be a leap towards flexibly bridging AMD Xilinx and Intel (MP)SoC-FPGA platforms on a system level.

ACKNOWLEDGMENT

The authors acknowledge support by the German Research Foundation and the Open Access Publication Fund of TU Berlin. The AMD Xilinx University Program (XUP) donated a Vivado ML Enterprise Edition license.

REFERENCES

[1] *Form 10-K*, Xilinx, San Jose, CA, USA, 2002.

[2] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet DS083 (V5.0)*, Xilinx, San Jose, CA, USA, Jun. 2011.

[3] *Form 10-K*, Xilinx, San Jose, CA, USA, 2020.

[4] *Versal Architecture and Product Data Sheet: Overview DS950 (V1.17)*, AMD Xilinx, Santa Clara, CA, USA, Nov. 2022.

[5] J. Lant, C. Concatto, A. Attwood, J. A. Pascual, M. Ashworth, J. Navaridas, M. Luján, and J. Goodacre, "Enabling shared memory communication in networks of MPSoCs," *Concurrency Comput., Pract. Exper.*, vol. 31, no. 21, p. e4774, Nov. 2019, doi: 10.1002/cpe.4774.

[6] M. Bielski, C. Pinto, D. Raho, and R. Pacalet, "Survey on memory and devices disaggregation solutions for HPC systems," in *Proc. IEEE Int. Conf. Comput. Sci. Eng. (CSE), IEEE Int. Conf. Embedded Ubiquitous Comput. (EUC), 15th Int. Symp. Distrib. Comput. Appl. Bus. Eng. (DCABES)*, Aug. 2016, pp. 197–204, doi: 10.1109/CSE-EUC-DCABES.2016.185.

[7] R. Giorgi, M. Procaccini, and F. Khalili, "AXIOM: A scalable, efficient and reconfigurable embedded platform," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, May 2019, pp. 480–485, doi: 10.23919/DATE.2019.8715168.

[8] F. Chaix, A. Ioannou, N. Kossifidis, N. Dimou, G. Ieronymakis, M. Marazakis, V. Papaefstathiou, V. Flouris, M. Ligerakis, G. Ailamakis, T. Vavouris, A. Damianakis, M. Katevenis, and I. Mavroidis, "Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping," in *Proc. IEEE/ACM Int. Workshop Heterogeneous High-Perform. Reconfigurable Comput. (HRC)*, Nov. 2019, pp. 34–41, doi: 10.1109/H2RC49586.2019.00010.

[9] *AMBA AXI and ACE, Protocol Specification, Issue H.c, ARM IHI 0022H.c (ID012621)*, Arm, Cambridge, U.K., Jan. 2021.

[10] *AXI Chip2Chip V5.0, LogiCORE IP Product Guide, PG067*, AMD Xilinx, Santa Clara, CA, USA, May 2022.

[11] *AMBA AXI4 Interface Protocol*, AMD Xilinx, Santa Clara, CA, USA, 2023. Accessed: Mar. 23, 2023. [Online]. Available: https://www.xilinx.com/products/ intellectual—property/axi.html

[12] *Aurora 8B/10B V11.1, LogiCORE IP Product Guide, PG046*, AMD Xilinx, San Jose, CA, USA, May 2022.

[13] *Aurora 8B/10B Protocol Specification, SP002 (V2.3)*, Xilinx, San Jose, CA, USA, Oct. 2014.

[14] A. X. Widmer and P. A. Franaszek, "A DC-balanced, partitioned-block, 8B/10B transmission code," *IBM J. Res. Develop.*, vol. 27, no. 5, pp. 440–451, Sep. 1983.

[15] *Aurora 64B/66B V12.0, LogiCORE IP Product Guide, PG074*, AMD Xilinx, Santa Clara, CA, USA, Oct. 2022.

[16] *Aurora 64B/66B Protocol Specification, SP011 (V1.3)*, Xilinx, San Jose, CA, USA, Oct. 2014.

[17] A. Tomori and Y. Osana, "Kyokko: A vendor-independent high-speed serial communication controller," in *Proc. 11th Int. Symp. Highly Efficient Accel. Reconfigurable Technol.*, Jun. 2021, Art. no. 7, doi: 10.1145/3468044.3468051.

[18] A. Tomori and Y. Osana, "FPL demo: Kyokko—An aurora 64b66b compatible 100 Gbps communication controller," in *Proc. 32nd Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2022, p. 472.

[19] C. Brewer, N. Franchoni, R. Ripley, A. Geist, T. Wise, S. Sabogal, G. Crum, S. Heyward, and C. Wilson, "NASA SpaceCube intelligent multi-purpose system for enabling remote sensing, communication, and navigation in mission architectures," in *Proc. Small Satell. Conf.*, 2020. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2020/all2020/136/

[20] J. D. Chimeh, "A new architecture for massive MIMO testbed laboratory in 5G," in *Proc. 10th Int. Symp. onTelecommunications (IST)*, Dec. 2020, pp. 15–19, doi: 10.1109/IST50524.2020.9345868.

[21] J. Flich, R. Tornero, D. Rodriguez, D. Russo, J. M. Martínez, and C. Hernández, "From a FPGA prototyping platform to a computing platform: The MANGO experience," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 7–12, doi: 10.23919/DATE51398.2021.9474051.

[22] D. Lettnin and M. Winterholer, *Embedded Software Verification and Debugging* (Embedded Systems), 1st ed. New York, NY, USA: Springer, 2017, doi: 10.1007/978-1-4614-2266-2.

[23] J. Wildman, S. Moore, L. Veytser, T. Capuano, J. Shapiro, J. Hillger, and B. Cheng, "Towards rapid waveform design and deployment via modular signal processing frameworks," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Jan. 2019, pp. 1031–1036, doi: 10.1109/MILCOM.2018.8599731.

[24] R. Murphy, B. Barnett, L. Wagner, J. Wildman, L. Veytser, D. Wiggins, S. Buscemi, T. Arganbright, S. Clark, and B. Cheng, "CHIL: Common-modem hardware integrated library," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Jan. 2019, pp. 1–6, doi: 10.1109/MILCOM.2018.8599859.

[25] X. Yang, Y. Hou, and H. He, "A processing-in-memory architecture programming paradigm for wireless Internet-of-Things applications," *Sensors*, vol. 19, no. 1, p. 140, Jan. 2019, doi: 10.3390/s19010140.

[26] Z. Zhu, G. Di Guglielmo, Q. Cheng, M. Glick, J. Kwon, H. Guan, L. P. Carloni, and K. Bergman, "Photonic switched optically connected memory: An approach to address memory challenges in deep learning," *J. Lightw. Technol.*, vol. 38, no. 10, pp. 2815–2825, Feb. 2020, doi: 10.1109/JLT.2020.2975976.

[27] T. Mehner, L. E. Ardila-Perez, M. N. Balzer, O. Sander, D. Tcherniakhovski, M. Schleicher, M. Fuchs, G. Fedi, G. Gimas, G. M. Iles, M. Pesaresi, A. W. Rose, and T. Schuh, "ZynqMP-based board-management mezzanines for serenity ATCA-blades," *J. Instrum.*, vol. 17, no. 3, Mar. 2022, Art. no. C03009, doi: 10.1088/1748-0221/17/03/c03009.

[28] C. G. Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea, and T. Williams, "IPbus: A flexible Ethernet-based control system for xTCA hardware," *J. Instrum.*, vol. 10, Feb. 2015, Art. no. C02019, doi: 10.1088/1748-0221/10/02/C02019.

[29] A. Byszuk, K. Pozniak, W. M. Zabolotny, K. Bunkowski, M. Bluj, K. Doroba, P. Drabik, M. Górski, A. Kalinowski, K. Kierzkowski, M. Konecki, J. Królikowski, W. Oklinski, M. Olszewski, A. Skala, and K. Zawistowski, "OMTF firmware overview," *Proc. SPIE*, vol. 9662, Sep. 2015, Art. no. 966241, doi: 10.1117/12.2207432.

[30] D. Acosta, G. Brown, A. Carnes, M. Carver, D. Curry, G. P. D. Giovanni, I. Furic, A. Kropivnitskaya, A. Madorsky, M. Matveev, P. Padley, D. Rank, C. Reeves, B. Scurlock, and S. Wang, "The CMS modular track finder boards, MTF6 and MTF7," *J. Instrum.*, vol. 8, no. 12, Dec. 2013, Art. no. C12034, doi: 10.1088/1748-0221/8/12/C12034.

[31] A. Gabrielli, G. Gebbia, F. Alfonsi, G. D'Amen, N. Giangiacomi, D. Soverini, G. Balbi, D. Falchieri, and R. Travaglini, "A PCI express board proposed for the upgrade of the ATLAS TDAQ read-out system," in *Proc. 6th Annu. Conf. Large Hadron Collider Phys.-PoS(LHCP)*, Oct. 2018, p. 76, doi: 10.22323/1.321.0076.

[32] N. Giangiacomi, F. Alfonsi, G. d'Amen, G. Balbi, D. Falchieri, A. Gabrielli, G. Gebbia, G. Pellegrini, and D. Soverini, "General purpose readout board π LUP: Overview and results," *IEEE Trans. Nucl. Sci.*, vol. 66, no. 7, pp. 1021–1027, Jul. 2019, doi: 10.1109/TNS.2019.2914332.

[33] N. Loukas, "The barrel calorimeter processor demonstrator board for the phase II upgrade of the CMS ECAL barrel," CERN, Meyrin, Switzerland, Tech. Rep. CMS-CR-2018-301, Oct. 2018. [Online]. Available: https://cds.cern.ch/record/2644903

[34] W. Smith, M. Vicente, T. Gorski, A. Svetek, J. Tikalsky, R. Fobes, and S. Dasu, "Next generation ATCA control infrastructure for the CMS phase-2 upgrades," in *Proc. Top. Workshop Electron. Part. Phys.-PoS(TWEPP-17)*, Mar. 2018, p. 102, doi: 10.22323/1.313.0102.

[35] R. Spiwoks, A. Armbruster, G. Carrillo-Montoya, M. Chelstowska, P. Czodrowski, P.-O. Deviveiros, T. Eifert, N. Ellis, G. Galster, S. Haas, L. Helary, O. L. Nikolos, A. Marzin, T. Pauly, V. Ryjov, K. Schmieden, M. S. Oliveira, J. Stelzer, P. Vichoudis, and T. Wengler, "The ATLAS muon-to-central-trigger-processor-interface (MUCTPI) upgrade," in *Proc. Int. Conf. Technol. Instrum. Part. Phys.*, in Springer Proceedings in Physics, vol. 212. Singapore: Springer, Aug. 2018, pp. 360–365, doi: 10.1007/978-981-13-1313-4_69.

[36] A. D. Ioannou, K. Georgopoulos, P. Malakonakis, D. N. Pnevmatikatos, V. D. Papaefstathiou, I. Papaefstathiou, and I. Mavroidis, "UNILOGIC: A novel architecture for highly parallel reconfigurable systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, pp. 1–32, 2020, doi: 10.1145/3409115.

[37] *Vivado Design Suite User Guide, Synthesis, UG901 (V2022.2)*, AMD Xilinx, Santa Clara, CA, USA, Nov. 2022.

[38] *Vivado Design Suite User Guide, Creating and Packaging Custom IP, UG1118 (V2022.2)*, AMD Xilinx, Santa Clara, CA, USA, Nov. 2022.

[39] *Vivado Design Suite User Guide, Designing IP Subsystems Using IP Integrator, UG994 (V2022.2)*, AMD Xilinx, Santa Clara, CA, USA, Oct. 2022.

[40] *UltraScale Architecture GTH Transceivers, User Guide, UG576 (V1.7.1)*, Xilinx, San Jose, CA, USA, Aug. 2021.

[41] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, Mar. 1984, doi: 10.1147/rd.282.0124.

[42] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[43] W. J. Dally and J. W. Poulton, *Digital Systems Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 1998, doi: 10.1017/CBO9781139166980.

[44] *UltraScale Architecture Libraries Guide, UG974 (V2022.2)*, AMD Xilinx, Santa Clara, CA, USA, Oct. 2022.

[45] W. Reisig, *Understanding Petri Nets*, 1st ed. Berlin, Germany: Springer, 2013, doi: 10.1007/978-3-642-33278-4.

[46] *ZCU102 Evaluation Board, User Guide, UG1182 (V1.7)*, AMD Xilinx, Santa Clara, CA, USA, Feb. 2023.

[47] *AXI Traffic Generator V3.0, LogiCORE IP Product Guide, PG125*, Xilinx, San Jose, CA, USA, Feb. 2019.

[48] *AXI Block RAM (BRAM) Controller V4.1, LogiCORE IP Product Guide, PG078*, Xilinx, San Jose, CA, USA, May 2019.
[49] *Block Memory Generator v8.4, LogiCORE IP Product Guide, PG058*, Xilinx, San Jose, CA, USA, Aug. 2021.
[50] *AXI Performance Monitor V5.0, LogiCORE IP Product Guide, PG037*, Xilinx, San Jose, CA, USA, Oct. 2017.
[51] *IBERT for UltraScale GTH Transceivers v1.4, Logi-CORE IP Product Guide, PG173*, Xilinx, San Jose, CA, USA, Feb. 2021.
[52] *UltraScale FPGAs Transceivers Wizard V1.7, Logi-CORE IP Product Guide, PG182 (V1.7)*, Xilinx, San Jose, CA, USA, Dec. 2020.
[53] *Zynq UltraScale+MPSoC Data Sheet: Overview, DS891 (V1.10)*, AMD Xilinx, Santa Clara, CA, USA, Nov. 2022.

ROBERT DREHMEL (Member, IEEE) received the master's degree (Hons.) in computer science from the Trier University of Applied Sciences, Germany, in 2016. He is currently pursuing the Ph.D. degree in computer science with Technische Universität Berlin, Germany. He has been with the software and hardware development industries, since 2001, and has been a FreeBSD developer. His research interests include embedded systems, field-programmable gate arrays (FPGAs), and operating systems. He received the Ph.D. Scholarship from the German Academic Scholarship Foundation (Studienstiftung des deutschen Volkes).

HANS-ULRICH HEISS received the Diploma, Ph.D., and Habilitation degrees in computer science from the Karlsruhe Institute of Technology, Germany. He had research and teaching positions at the IBM Watson Research Center, Yorktown Heights, NY, USA; the University of Helsinki, the Ilmenau University of Technology; and Paderborn University. From 2001 to 2021, he was a Full Professor of computer science with Technische Universität Berlin (TU Berlin). His research interests include operating systems, distributed and parallel systems, performance evaluation, resource management, and self-organization.