Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Robin Heinemann

born in Kassel (Germany)

2023

Implementation of an FPGA-based memory mapped buffer for real-time communication with a neuromorphic ASIC

This Bachelor Thesis has been carried out by Robin Heinemann at the
Kirchhoff Institute for Physics in Heidelberg
under the supervision of
Dr. Johannes Schemmel

Implementation of an FPGA-based memory mapped buffer for real-time communication with a neuromorphic ASIC

BrainScaleS-2 is a mixed signal neuromorphic ASIC combining accelerated physical emulation of brain-inspired neurons and synapses with digital control logic. Experiments using the BrainScaleS-2 ASIC require real-time stimuli, since the analog emulation performed by the ASIC cannot be stopped and continued arbitrarily. This real-time communication is realized using an FPGA, while a conventional host computer controls the experiment. A buffer for the data to be sent to the ASIC and for data received from the ASIC is necessary in the FPGA, due to the mismatch of bandwidth to the host computer and the ASIC. The current buffer implementation has several limitations. It is not able to make use of all of the connected DRAM and the bandwidth of the buffer is smaller than the bandwidth between the FPGA and the ASIC for some cases, leading to nondeterministic timing that can reduce reproducibility. This thesis presents a new design for this buffer that trades a lower maximum rate of experiments for the ability to use the full size of the available DRAM. The bandwidth is increased to always provide deterministic timing. Furthermore, the options for organization of data to be transmitted to and data received from the ASIC are extended to allow repeated use and sparse readout of data. Basic integration with the BSS2 software stack was performed as well, that passes all unit and integration tests and transparently allows higher level software to use the new buffer.

Entwicklung eines FPGA-basierten memory-mapped Zwischenspeicher für Echtzeit-Kommunikation mit einem neuromorphen ASIC

BrainScaleS-2 ist ein Mixed-Signal neuromorpher ASIC. Dabei wird beschleunigte, analoge Emulation von dem Hirn nachempfundenen von Neuronen und Synapsen mit digitaler Logik zur Konfiguration kombiniert. Da der analoge Teil des ASIC nicht pausiert und wieder gestartet werden kann, setzt die Durchführung von Experimenten mit dem BrainScaleS-2 ASIC Echtzeit-Übertragung der Stimuli voraus. Um diese Echtzeit-Übertragung für einen konventionellen Computer, der das Experiment durchführt, zu ermöglichen, wird ein FPGA verwendet. Aufgrund der unterschiedlichen Bandbreiten zwischen Computer und ASIC wird ein Zwischenspeicher auf dem FPGA benötigt. Die aktuelle Umsetzung dieses Zwischenspeicher hat den Nachteil, dass nicht der komplette an den FPGA angeschlossene Speicher ausgenutzt werden kann und in manchen Fällen die vom Zwischenspeicher erreichte Bandbreite kleiner ist als die Bandbreite zwischen ASIC und FPGA. Wenn diese Bandbreite nicht erreicht wird, kann die benötigte Echtzeit-Übertragung nicht garantiert werden und die Reproduzierbarkeit von Experimenten sinkt. In dieser Arbeit wird eine neue Architektur für diesen Zwischenspeicher entwickelt und umgesetzt, die die Ausnutzung des ganzen an den FPGA angebundenen DDR3 Speichers ermöglicht. Dabei ist die Bandbreite des Zwischenspeichers immer groß genug, um die Echtzeit-Übertragung zu garantieren. Zusätzlich wird wiederholtes Senden und partielles Auslesen der Daten ermöglicht. Der einzige Nachteil der neuen Umsetzung ist eine Reduktion der maximal erreichten Anzahl an Experimenten die in einer bestimmten Zeit durchgeführt werden können. Vorschläge zur Verbesserung dieser Rate werden ebenfalls vorgestellt. Die Programmbibliotheken die zur Durchführung von Experimenten verwendet werden wurden angepasst, sodass transparent für die Experimente der neue Zwischenspeicher verwendet wird.

Contents

1. Introduction **2**

2. Background **3**

2.1. AXI and stream interfaces . . . 4

2.1.1. Stream-Interfaces . . . 4

2.1.2. AXI . . . 5

2.2. Playback Executor . . . 6

2.3. Playback and trace buffer . . . 7

3. Implementation **11**

3.1. FAXI . . . 12

3.2. AXIDMA . . . 12

3.3. New playback and trace buffer . . . 13

3.4. Theory of operation . . . 15

3.5. Software integration . . . 16

3.5.1. Allocator . . . 20

3.6. Verification and comparison . . . 21

3.6.1. Playback and trace buffer . . . 21

3.6.2. flange-dram . . . 22

3.6.3. Bandwidth verification . . . 25

4. Results **28**

4.1. Playback and trace bandwidth . . . 28

4.2. FAXI based memory mapped communication . . . 31

4.3. Experiment rate . . . 32

4.4. flange-dram performance . . . 34

4.5. FPGA resources usage . . . 34

5. Summary and discussion **36**

6. Outlook **37**

6.1. Latency reduction . . . 37

6.2. Higher level software integration . . . 37

6.3. Unified memory with PPU . . . 37

6.4. Support of new host interfaces . . . 37

Bibliography **39**

A. Code environment **41**

1. Introduction

The field of neuromorphic computing draws inspiration from the structure and operations of the human brain to develop new approaches to computation. Neuromorphic computation devices can take many forms, such as digital simulation of neural systems on specialized digital computation devices (Mayr, Hoeppner, and Furber, 2019). A different approach is the analog emulation of a neuron model, which promises efficiency and performance improvements over purely digital devices. This approach is taken by the BrainScaleS-2 (BSS2) system (Pehle et al., 2022) developed by the Electronic Vision(s) Group of the Heidelberg University is the second generation of the BrainScaleS neuromorphic computing platform. At its core it performs time-continuous analog emulation of the AdEx (Brette and Gerstner, 2005) model for neurons. This is combined with sophisticated digital processing provided by microprocessors (PPU) extended with specialized SIMD units. The emulation is accelerated compared to the biological time by a factor of 1000. The current silicon implementation of this architecture contains two of these microprocessors as well as 512 neurons and $512 \cdot 256$ synapses. This silicon realization will be called the BrainScaleS-2 ASIC hereafter. The BrainScaleS-2 ASIC is used in several modes of operation, like operation as a network attached accelerator (Müller, Mauch, et al., 2020) as well as usage as an edge computation platform (Stradmann et al., 2022).

Communication with the BrainScaleS-2 ASIC has real-time requirements, needs high bandwidth and requires precise timing, as the time continuous analog emulation performed by the ASIC cannot be paused and resumed.

Current systems using the BrainScaleS-2 ASIC use an FPGA as a bridge between a conventional host computer and the ASIC to realize the real-time communication with the ASIC. For this purpose the FPGA includes a buffer. This buffer is used to store data prior to transmission to the ASIC and store data received from the ASIC until it is sent back to the host computer.

In this thesis a replacement design of this buffer is developed and implemented. It improves the reliability and increases the usable size while extending it with more functionality.

2. Background

Several steps are necessary to perform experiments with the BrainScaleS-2 ASIC:

1. The components of the ASIC (such as the parameters controlling the behaviour of its neurons and synapses) have to be configured.
2. Input data has to be transmitted to the ASIC.
3. Data is processed by the ASIC.
4. Output data generated by the ASIC is transmitted to the FPGA.

These steps may happen partly in parallel, especially the transmission of input data, processing and generation of output data. Communication with the ASIC has real-time requirements as the time continuous analog emulation performed by the ASIC cannot be paused and resumed. The transmission of input data, in the form of neuron events, requires precise timing. Accordingly, neuron events received back from the ASIC have to be timestamped with similar precision. The scheduling of event transmission and timestamping of the received events on the FPGA is realized by the so-called playback executor block.

To perform an experiment the host computer generates a sequence of instructions. This sequence of instructions contains the operations necessary to configure the ASIC and contains instructions defining the timing and content of events to be transmitted to the ASIC. This sequence of instructions, called playback program, is then processed by the playback executor with a timing resolution of $8\,\mathrm{ns}$. Events received from the ASIC are timestamped with the same resolution.

The high speed interface used for the communication between the ASIC and the FPGA is made up out of sixteen LVDS lanes, eight for each direction. These lanes are able to operate at up to $2\,\mathrm{Gbit\,s^{-1}}$ for a total of $16\,\mathrm{Gbit\,s^{-1}}$ of full duplex bandwidth.

This thesis will focus on the network attached accelerator deployment, where the FPGA is connected to a host computer over a network connection.

The current network attached accelerator deployment used by the Electronic Vision(s) group called "BSS2-Cube" (Müller, Mauch, et al., 2020) uses a Xilinx Kintex7 FPGA. The LVDS lanes of the ASIC are operated at $1\,\mathrm{Gbit\,s^{-1}}$. The host and the FPGA communicate using the UDP protocol over Gigabit Ethernet. A custom automatic repeat request (ARQ) protocol (Karasenko, 2014) developed by the Electronic Vision(s) group is used to ensure reliable and ordered transmission of a stream of 64 bit words over the unreliable and unordered UDP protocol. In this thesis this protocol will be called Host-ARQ hereafter.

As the bandwidth between the host computer and the FPGA is at least 8 times smaller than the bandwidth between the FPGA and the ASIC, the playback program as well as the received neuron events need to be buffered by the FPGA. For this, the FPGA is connected to, an external $512\,\mathrm{MiB}$ DDR3 memory. Figure 1 gives an overview over the interaction between the host computer, the FPGA and its communication with the BrainScaleS-2 ASIC.

Following, subsection 2.1 and subsection 2.2 give a more detailed overview of the FPGA design and especially the playback executor. Afterwards subsection 2.3 provides a detailed description of the current buffer implementation for the playback program and the neuron events. It furthermore outlines the general requirements of this buffer such as the required data rate and highlights the problems of the current implementation. In section 3, the implementation of a new design for the buffer is described. Finally, section 4 will compare the old and new buffer implementation in detail.

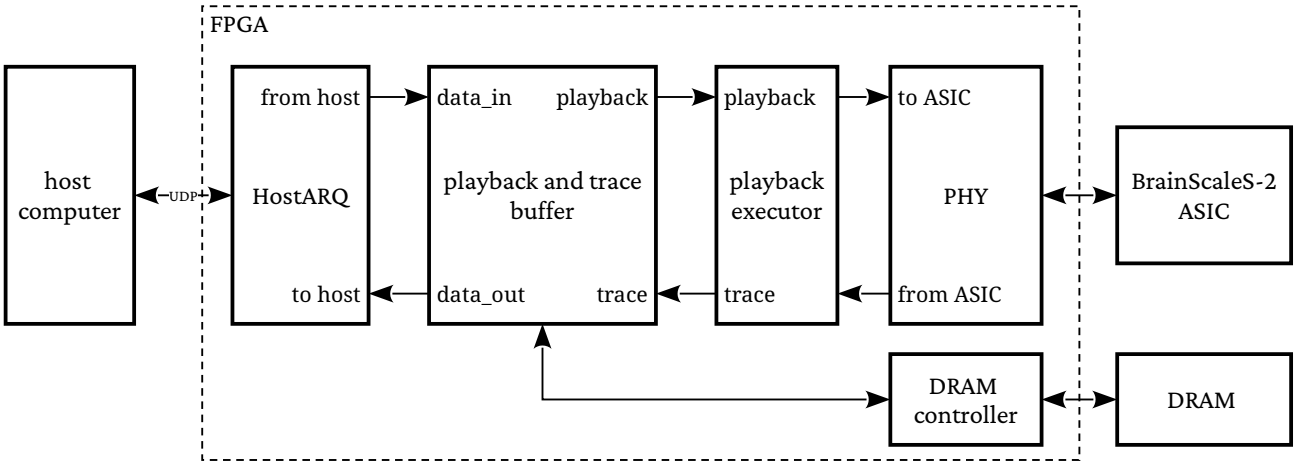

Figure 1: Schematic overview of the FPGA design facilitating the communication between a network attached host computer and a BrainScaleS-2 ASIC. The host computer communicates with the FPGA using the UDP based Host-ARQ protocol. The playback programs received by the FPGA are stored in the playback and trace buffer and are executed using the playback executor. Execution by the playback executor generates events and control messages that are transmitted to the ASIC. Events received from the ASIC are timestamped by the playback executor and stored in the trace buffer until they are sent to the host.

2.1. AXI and stream interfaces

2.1.1. Stream-Interfaces

Many components of the old and the new FPGA design are connected using stream interfaces. Stream interfaces provide an unidirectional transfer of data from one component to another. Throughout this thesis two different stream interfaces, AXI-Stream and the "ValidNext stream" will be encountered. The AXI-Stream interface is a standard interface used by many of the components provided by Xilinx used by the old and the new buffer design. The ValidNext stream interface is used by many of the components developed by the Electronic Vision(s) group, such as the playback executor.

The AXI-Stream interface connects a single Transmitter to a single Receiver to transport a unidirectional data stream from the Transmitter to the Receiver. An AXI-Stream consists of at least five signals:

- *ACLK* is the clock signal used by the stream. All other signals will be sampled on the rising edge of this clock.
- *ARESETn* is the reset signal used by the stream. It is active low.
- *TDATA* is the signal carrying the data word. It is a multiple of eight bits wide and driven by the Transmitter.
- *TVALID* is a single bit signal driven by the Transmitter indicating that valid data is present on the *TDATA* signal.
- *TREADY* is a single bit signal driven by the Receiver indicating that it can accept data.

The relation of *TDATA*, *TVALID* and *TREADY* is governed by a set of rules. Data is transferred from the Transmitter to the Receiver when *TREADY* and *TVALID* are driven high simultaneously. Furthermore, a Transmitter is not allowed to wait for the Receiver to drive *TREADY* high before asserting *TVALID*. The Receiver, on the contrary, is allowed to wait for the Transmitter to drive *TVALID* high before asserting *TREADY*.

When both the Transmitter and the Receiver can process the data fast enough, data can be transferred on every clock cycle. When this is the case *TREADY* and *TVALID* will be driven high continuously. AXI-Stream defines a set of further signals that can extend the functionality of this stream interface. In this thesis two optional signals will be relevant. The first is *TKEEP*. *TKEEP* has one bit for every eight bits contained in the *TDATA* signal and is driven by the Transmitter to indicate which bytes of the *TDATA* signal contain valid data. If bit n of *TKEEP* is driven high, bits $8n$ to $8(n+1)-1$ of *TDATA* will contain valid data, if it is driven low the data contained in these bits is to be ignored by the Receiver. The second optional signal that will be encountered is *TLAST*. This is a single bit that is driven by the Transmitter which indicates that the current transfer is the last transfer of a packet.

The second stream type encountered in this thesis is used by many of the components of the FPGA design that were developed by the Electronic Vision(s) group. It is for example used by the playback executor. In this thesis it will be referred to as ValidNext stream. It is closely related to AXI-Stream, but replaces the *TREADY* signal with a *NEXT* signal. This is a single bit signal driven by the Reciever to indicate that the current data was processed, and the Transmitter can present the next data word on *TDATA*. Furthermoren it has different rules regarding *TVALID* and *NEXT* compared to the rules of regarding *TVALID* and *TREADY*. For a ValidNext stream the Transmitter is allowed to wait until the Receiver drives *NEXT* high before asserting *TVALID*.

This different rule set means that in general an AXI-Stream and a ValidNext stream cannot be connected together by simply connecting the *TREADY* and the *NEXT* signals as they can deadlock. For example when connecting a ValidNext stream Transmitter to an AXI-Stream Receiver, the ValidNext stream Transmitter is allowed to wait until the AXI-Stream Receiver drives *TREADY* (connected to *NEXT*) until it drives *TVALID*. However, the AXI-Stream Receiver is allowed to wait until *TVALID* is driven high before asserting *TREADY*. In this case both the Receiver and the Transmitter will wait forever, and no progress will be made. This means when connecting a ValidNext stream Transmitter to an AXI-Stream Receiver, one has to use an AXI-Stream Receiver that will assert *TREADY* without waiting until *TVALID* is asserted by the Transmitter. Connecting incompatible Receivers and Transmitters can be achieved by inserting a suitable adapter, like a skid buffer (a FIFO that can hold two words).

2.1.2. AXI

AXI (AMBA, n.d.) is a standard protocol used for communication between some components of the FPGA design. It allows for communication between a single Manager and a Subordinate. Compared to a stream interface it provides bidirectional data transport and additional signals controlling source and target of the data that is transmitted. While the old playback and trace buffer design. It is only used for communication between the DDR3 controller and the playback and trace buffer, it will be used for more components in the new playback and trace buffer design. The basic operations of the AXI protocol are memory mapped read and write transactions. The AXI protocol uses an *ACLK* and an *ARESETn* signal that play the same role as they do in an AXI-Stream as well as five independent channels:

- The *AW* channel transmits information about a write from the Manager to the Subordinate. This information contains the address and the number of words that will be written (the burst size).
- The *AR* channel transmits information about a read from the Manager to the Subordinate. This information contains the address and the number of words that should be read (the burst size).
- The *W* channel transmits the data that is written from the Manager to the Subordinate.

- The *R* channel transmits the data that is read from the Subordinate to the Manager.
- The *B* channel transmits a response that contains the result of a write from the Subordinate to the Manager

Each of these channels uses the same handshaking signals *READY* and *VALID* as well as rules of an AXI-Stream. Furthermore, the *W* and *R* channels use a *LAST* signal to indicate the last word of a burst.

Using these five channels read and write transactions are performed. A read transaction is initiated by the Manager by transmitting the address and the number of words that should be read on the *AR* channel. The subordinate then responds with the corresponding data on the *R* channel. A write transaction transmits the address that should be written to and the number of words that should be written on the *AW* channel and the data that should be written on the *W* channel. The Subordinate responds with the result of a write transaction one the *B* channel once it is completed.

Every channel operates separately from each other. This means that for example the data to be written can be transmitted by the Manager on the *W* channel before the address information is transmitted on the *AW* channel. A Manager is also allowed to transmit a second read on the *AR* channel before having received the answer to the first. From this it follows that the maximum data rate supported by the bus specification is a single data word each clock cycle on both the read and write channels. The actual achievable data rate depends on the specific implementation of the Manager and the Subordinate.

2.2. Playback Executor

The playback executor is responsible for processing the instruction stream that is received from the playback buffer as well as receiving, time stamping and transmission to the trace buffer of events from the ASIC. Figure 2 shows a schematic overview of this module. This section will only give a brief overview of this module, a more detailed description can be found in (Rettig, 2019).

The instructions that are processed by the executor can broadly be categorized into the three different categories:

- Instructions in the read category perform read operations on the several buses connected to the playback executor and result in response data that is sent to the trace stream.
- Instructions of the write category perform writes to the connected buses.
- Instructions of the wait category are used to pause the processing of the instruction stream until a specific event takes place. This can for example be the elapsing of a specific duration or the completion of a read.

A special instruction, the `halt()` instruction is used to delineate separate experiments from each other. The `halt()` instruction marks the end of a playback program and is looped back to the trace data where it can be used to differentiate trace data belonging to different playback programs.

There are numerous sources like the result data of instructions in the read category and the events received from the ASIC for trace data that are transmitted by the playback executor on the trace stream. The data from these different sources is combined into the single trace stream using an arbiter, the trace arbiter, which uses a combination of round-robin and priority arbitration.

The playback executor is operating at a $125\,\mathrm{MHz}$ clock rate. The number of clock cycles that are required to process an instruction depends on the specific instruction that is processed, however for every instruction atleast one clock cycle is required. The highest theoretically possible rate of instructions that the playback

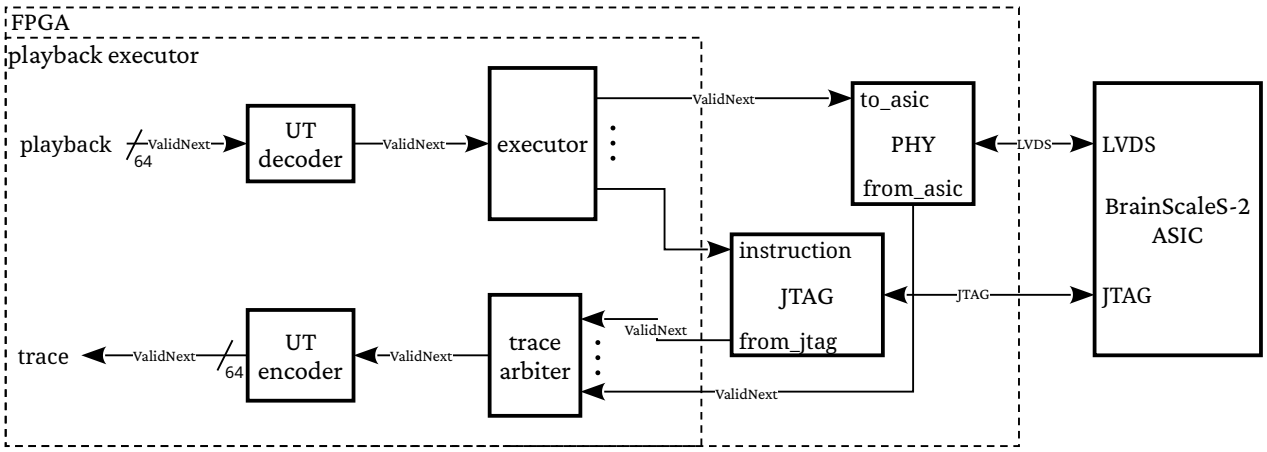

Figure 2: Overview of the playback executor. The playback executor receives a UT encoded stream of instructions that is decoded by the UT decoder. Each instruction is executed by the executor. Depending on the instruction this can for example cause an event to be sent to the ASIC or a JTAG operation to be performed. Data received from the ASIC as well as data that is for example generated by instructions that are read from one of the other FPGA buses is combined into a single variable word width stream by the trace arbiter and encoded into a fixed word width stream by the UT encoder. This overview is simplified and does not include all interfaces that the executor has access to. Furthermore, it does not include all sources for trace data.

executor can process is one instruction every clock cycle. Similar to the instruction stream the playback executor can also emit at most a single trace word every clock cycle.

Both the playback instruction stream and the trace data are fundamentally streams of variable size words. For transmission and reception over the fixed width Host-ARQ streams they are encoded using the UT encoding scheme (Karasenko, 2020).

For the playback instruction stream this encoding is performed on the host. On the FPGA the playback executor decodes the instruction stream before processing it. Likewise, the trace data generated by the playback executor is encoded by the playback executor before being sent to the trace stream.

The encoding and decoding also operates at a $125\,\mathrm{MHz}$ clock rate and can produce/consume at most one 64 bit sized word per clock cycle.

This means the maximum data rate at which the playback stream can be processed and the maximum data rate that is sent on the trace stream is

$$64\,\mathrm{bit} \cdot 125\,\mathrm{MHz} = 8\,\mathrm{Gbit\,s^{-1}}$$

2.3. Playback and trace buffer

The bandwidth between FPGA and the ASIC at $8\,\mathrm{Gbit\,s^{-1}}$ far exceeds the bandwidth between the host and the FPGA of 1 Gbit $\mathrm{s^{-1}}$. To allow transmission and reception of the full data rate supported between the FPGA and the ASIC the FPGA is connected to $512\,\mathrm{MiB}$ of DDR3 memory that is used as buffer for the playback and trace data. This section will describe the current implementation of this playback and trace buffer and highlight its shortcomings. A schematic overview of the current design is shown in Figure 3. The FPGA design uses the XilinxMIG to allow access to this DDR3 memory using the AXI protocol.

The playback and trace buffer has two responsibilities:

- It stores the playback instruction stream received from the host into the DDR3 memory and once a

complete playback program was received, it reads the playback program from the memory and transmits it to the playback executor

- It receives the trace data from the playback executor and stores it to the DDR3 memory until it is transmitted back to the host.

In the current FPGA design it operates as a pair of FIFOs. One for the playback stream and one for the trace stream. The input side of the playback buffer is a ValidNext stream that appends to the already stored data. On the output side the data is transmitted to the playback executor in the same order it was received. The trace buffer operates analogously with the input and the outside switched.

This FIFO is implemented using the Xilinx VFIFO core, which implements a multichannel FIFO backed by an AXI accessible memory. It is connected to the AXI interface of the DDR3 controller. The XilinxMIG is used as this DDR3 controller. The VFIFO core is used in a configuration using two channels. The first channel is used for the playback data and the second channel is used for the trace data.

The playback control block is responsible for scheduling the transmission of the playback instruction stream to the playback executor. It allows data to be transmitted from the VFIFO to the playback executor only when two conditions are fulfilled:

1. The first condition is that the FIFO between the VFIFO and the playback executor is full or the VFIFO channel for the playback data is empty.

2. The second condition is that the VFIFO channel for the playback data is full or a `halt()` instruction was written to the VFIFO but not transmitted to the playback executor yet.

When a complete playback program fits into the VFIFO playback channel, these two conditions enforce that playback of the instructions that make up the playback program is only started once it was completely transmitted to the VFIFO, as each playback program ends with a `halt()` instruction. This ensures that the rate of instructions that can be transmitted to the playback executor is not limited by the slow Host-ARQ interface but instead by the VFIFO and indirectly by the DDR3 memory bandwidth. When a playback program does not fit completely in the VFIFO playback channel, playback of it is started whenever the VFIFO playback channel is full. This means that depending on the rate the playback executor is processing playback instructions it is possible that the Host-ARQ interface can be the limiting factor for the playback rate. The VFIFO is configured with a burst size of $2048\,\mathrm{B}$ and using 8192 pages of $4\,\mathrm{KiB}$ allocated to each the playback and the trace channel, which means it can store at most $8192 \cdot 4\,\mathrm{KiB} = 32\,\mathrm{MiB}$ per channel.

This implementation of the playback and trace buffer has several shortcomings:

1. It can only use $64\,\mathrm{MiB}$ ($32\,\mathrm{MiB}$ for the playback and $32\,\mathrm{MiB}$ for the trace data) of the available $512\,\mathrm{MiB}$ of memory

2. The FIFO interface prohibits reading a non-contiguous block of the received trace data

3. The FIFO interface precludes already transmitted playback program or parts of them to be reused.

4. Allocation of the total memory to either playback memory or trace memory is fixed, changing the size of memory allocated to each of them requires generating a new FPGA bitstream.

5. It does not manage to achieve the maximum possible data rate achievable by the playback executor for all playback program sizes, which can cause the timing of events transmitted to the ASIC to be different

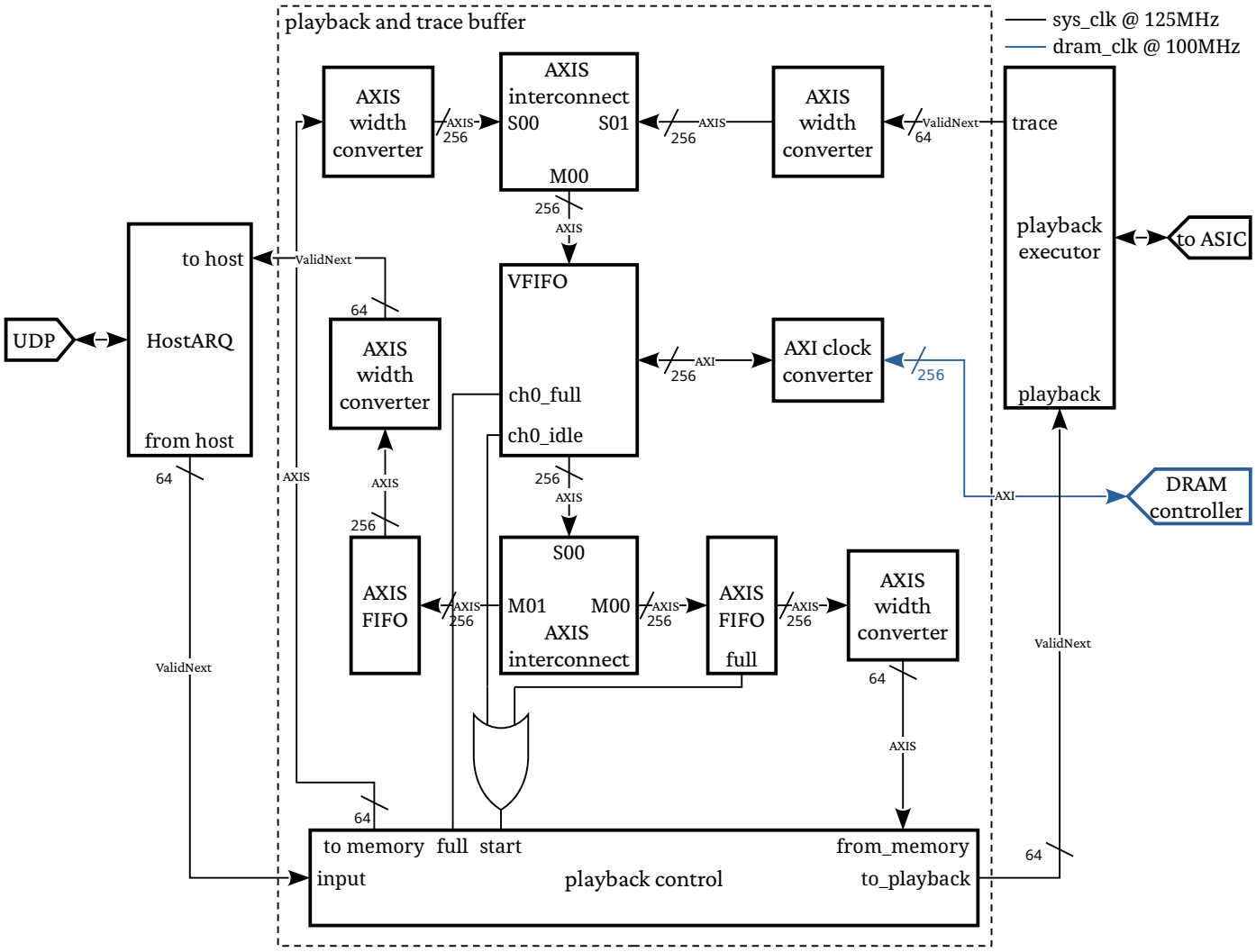

Figure 3: Schematic overview of the old, FIFO-based, module handling the playback and trace data. Data received from the host is buffered in the DDR3 memory and gated by the playback control block to be transmitted to the playback executor once at least one complete playback program was received from the host. In the opposite direction the trace data is stored in the DDR3 memory until it is transmitted to the host.

from the intended timing. This in turn could lead to wrong interpretation of the output of the ASIC when this is not accounted for or to reduced repeatability of experiments.

The maximum size of the two VFIFO channels can be increased by increasing the burst size and allocated pages up to a maximum of $256\,\mathrm{MiB}$ (See table 4-1 in (*AXI Virtual FIFO Controller v2.0 Product Guide (PG038)* 2015)), which is still only half of the total size of the memory.

A replacement for the playback and trace buffer should fulfill the following requirements

- Use the complete $512\,\mathrm{MiB}$ of available memory.
- Support the maximum possible data rate of $8\,\mathrm{Gbit\,s^{-1}}$ supported by the playback executor when transferring the trace data from the playback executor to the memory.
- Allow reuse of already transmitted playback instructions. Because the bandwidth between the host and the FPGA is a lot smaller than the bandwidth between the FPGA and the ASIC this can increase the rate experiments can be performed.
- Allow out of order access to the received trace data. This can reduce the time required to transfer the relevant trace data from the FPGA to the host, when not all trace data is relevant. It furthermore allows for an increased level of introspection of the state of the FPGA, especially in case of unexpected behavior.

3. Implementation

To alleviate the shortcoming of the old playback and trace buffer design it is fundamentally redesigned. There are four operations that need to be performed by the playback and trace buffer:

1. Write playback instructions sent by the host to the DDR3 memory.
2. Transfer previously received playback instructions from the DDR3 memory to the playback executor.
3. Write the trace data generated by the playback executor to the DDR3 memory.
4. Send the trace data from the DDR3 memory back to the host.

These operations can be grouped into two categories: First host-side operations, which includes the first and the last of the four listed operations. The operations in this category allow the host to read and write the DDR3 memory. The second category is FPGA-side operations, which includes the second and third operation listed. The operations in this category are responsible for reading and writing to the DDR3 memory to generate the playback instructions stream for the playback executor and to store the trace data generated by the playback executor. The first category of operations will be implemented using a bridge between the Host-ARQ streams and the AXI protocol (FAXI block), to allow memory mapped read and write access to the whole DDR3 memory by the host. To implement the second category of operations a scatter gather DMA engine is used. It, on the one hand, assembles the playback stream for the playback executor from data that was written by the host to the DDR3 memory and, on the other hand, stores the trace data transmitted by the playback executor to the DDR3 memory.

Figure 4 shows a schematic overview of the replacement block developed in this thesis.

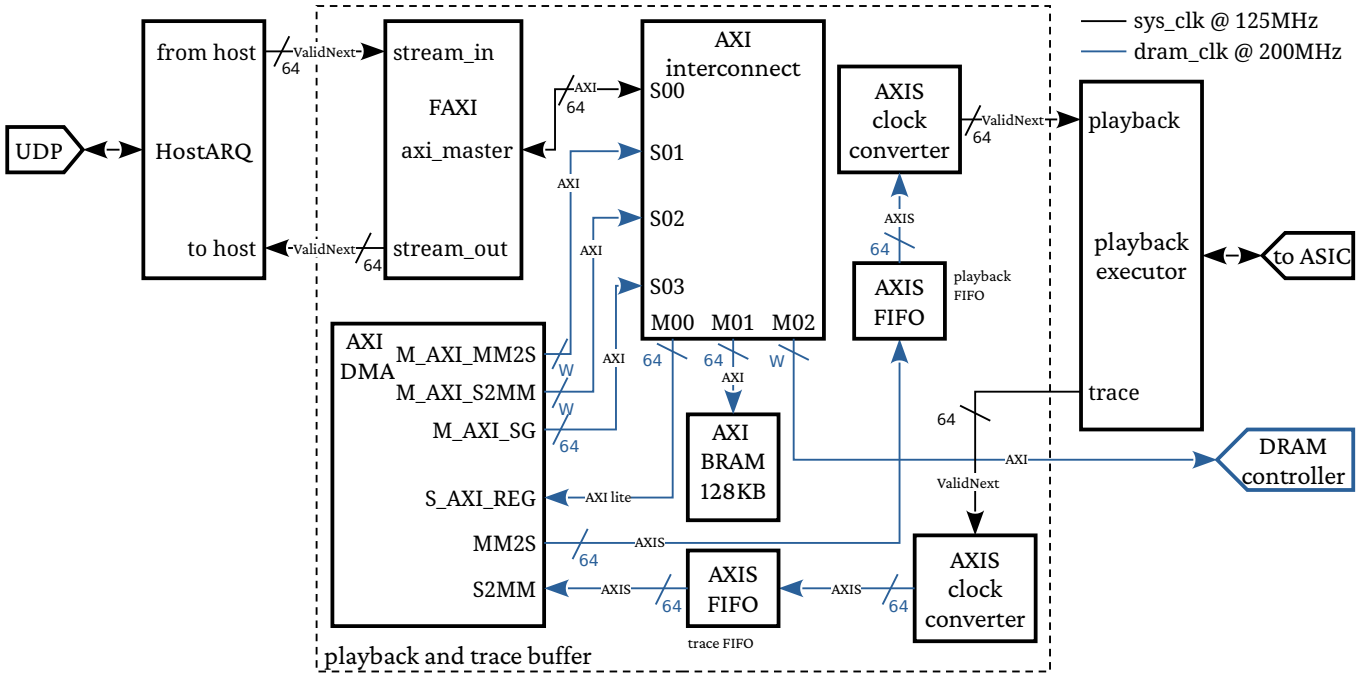

Figure 4: Schematic overview of the new playback and trace buffer. Instead of implementing two FIFOs it allows direct read and write access to the DDR3 memory by the host. A scatter gather DMA engine is used to read from the DDR3 memory and transmit the read data as the playback stream to the playback executor as well as writing the trace data from the playback executor to the DDR3 memory.

feature	status
data width	64 bit
address width	64 bit
transaction ID	not supported
AxLOCK	not supported
AxCACHE	not supported
AxPROT	not supported
AxQOS	not supported
AxREGION	not supported
user signals	not supported
narrow transfers	not supported
write strobe	partially, see description

Table 1: Summary of the AXI features supported by FAXI. AXI usually allows a different write strobe for each data word of a write transaction. FAXI only allows a fixed write strobe for a complete write transaction.

3.1. FAXI

To allow the host to read and write to the DDR3 memory the FAXI unit is used. This unit was developed by the Electronic Vision(s) group for a different FPGA design. It implements a bridge between the Host-ARQ FPGA interface of two 64 bit wide streams and an AXI Subordinate with data width of 64 bit and an address size of 64 bit. Read and write requests are encoded in the Host-ARQ stream using a 64 bit header controlling the kind of operation (read or write) and the burst size. This header is followed by a 64 bit address and for a write requests by the specified number of 64 bit words. Read data and write responses are sent back to the host following a similar scheme of a 64 bit header followed by the read data words. Finally, FAXI supports a globalfence operation, that blocks the processing of the further data until every outstanding read or write transaction is completed. The subset of AXI that is supported by flange-dram is summarized in Table 1.

Host-ARQ is configured to use a maximum packet size of 180 words with 64 bit per word. Each packet has an overhead of 78 B from Gigabit Ethernet, IPv4, UDP and Host-ARQ. The maximum burst length of an AXI write is 256 words and to encode a write-burst FAXI has an overhead of two 64 bit words (the header and the address). From the maximum data rate $1\,\mathrm{Gbit\,s^{-1}}$ of Gigabit Ethernet one obtains for the maximum write bandwidth possible

$$B_{\mathrm{FAXI,w}} = 180 \cdot 64\,\mathrm{bit}\frac{1\,\mathrm{Gbit\,s^{-1}}}{78\,\mathrm{B} + 180 \cdot 64\,\mathrm{bit}} \cdot \frac{256 \cdot 64\,\mathrm{bit}}{(256+2)64\,\mathrm{bit}} \approx 941\,\mathrm{Mbit\,s^{-1}}$$

For a read the overhead due to the FAXI encoding is only one 64 bit word and one obtains

$$B_{\mathrm{FAXI,w}} = 180 \cdot 64\,\mathrm{bit}\frac{1\,\mathrm{Gbit\,s^{-1}}}{78\,\mathrm{B} + 180 \cdot 64\,\mathrm{bit}} \cdot \frac{256 \cdot 64\,\mathrm{bit}}{(256+1)64\,\mathrm{bit}} \approx 945\,\mathrm{Mbit\,s^{-1}}$$

3.2. AXIDMA

For the scatter gather DMA engine the AXIDMA IP core by Xilinx (*AXI DMA LogiCORE IP Product Guide (PG021)* 2022) is used. This DMA engine is split into two separate channels, the MM2S channel and the S2MM channel. The MM2S channel is used to read data from an AXI Subordinate and transmit the data using an AXI-Stream. It is used to read the data written to the DDR3 memory by the host using FAXI and transmit it to the playback executor. The S2MM channel performs the opposite operation and writes data from an

AXI Subordinate	address	size
DDR3 memory	$\texttt{0000}\,0000_{16}$	512 MiB
scatter gather descriptor memory	$\texttt{A000}\,0000_{16}$	128 KiB
AXIDMA registers	$\texttt{B000}\,0000_{16}$	8 KiB

Table 2: AXI memory map.

AXI-Stream to an AXI Subordinate. It is used to write the trace data received from the playback executor to the DDR3 memory. Moreover, the AXIDMA core has a separate set of AXI-Lite accessible registers that are used to control its operation.

The AXIDMA DMA engine can be used to perform scatter and gather operations. In the scatter gather mode the operation of these channels is controlled using a chain of descriptors. Each descriptor contains an address, a buffer length, the address of the next descriptor, as well as a status and a control field. For the MM2S channel, the address specified in the descriptor is the address of the first byte that is read by the channel. The total number of consecutive bytes read from this address is specified by the buffer length. When every byte specified by the descriptor was read, the AXIDMA sets the completed flag in the status field and, the next descriptor as specified by the next descriptor address is used to continue the operation. The control field contains a start and an end of frame flag. The latter is used to generate the *TLAST* signal of the AXI-Stream driven by the MM2S channel. Operation of the MM2S channel is started by writing the address of the first descriptor and the address of the last descriptor to the `curdesc` and `taildesc` registers of the MM2S channel.

The S2MM channel operates similarly. For each descriptor it writes up to the specified buffer length consecutive bytes from the S2MM AXI-Stream Receiver to the address specified in the descriptor. Whenever the last word of a packet as specified by the *TLAST* signal or the number of bytes specified by the buffer length was written, the completed flag as well as the number of transferred bytes is updated in the status field of the descriptor and the next descriptor is read from the specified address for the next descriptor.

The AXIDMA core uses separate AXI Managers for the S2MM and the MM2S channel as well as the scatter gather descriptors.

3.3. New playback and trace buffer

To store the scatter gather descriptors there are two options. One could use the main DDR3 memory or a separate memory. Using a separate memory has several advantages. It ensures that interaction such as reading and writing the descriptor cannot have any effect on the reads and writes to the DDR3 memory performed by the S2MM and MM2S channels. Additionally, reads and writes to it have a lower latency than reads and writes to the DDR3 memory. Each descriptor has a size of 64 B, accordingly the 128 KiB memory used for the descriptors allows for up to 2048 descriptors.

Using an AXI interconnect allows operations from multiple AXI Managers to be multiplexed to multiple AXI Subordinates based on the address. For the playback and trace buffer design it is used to allow access from FAXI and both AXIDMA channels to the AXI interface of the DDR3 controller. Table 2 contains an overview of the memory map that was chosen. In this case the Xilinx AXI Smartconnect (*SmartConnect v1.0 LogiCORE IP Product Guide* 2022) is used to allow FAXI to access the DDR3 memory, the AXIDMA registers and the scatter gather descriptor memory while also allowing both AXIDMA channels to access the DDR3 memory and AXIDMA to access the scatter gather descriptor memory.

As every address that is accessible according to the address map given in Table 2 can be represented with

$32\,\mathrm{bit}$ an address width of $32\,\mathrm{bit}$ is used every AXI bus. The AXIDMA, XilinxMIG, AXI Smartconnect and the AXIBRAMController do not have a fixed AXI data width but instead allow a variety of different configurations. Their configuration was selected to use the minimal width that satisfy the requirement of $8\,\mathrm{Gbit\,s^{-1}}$ bandwidth for the playback and the trace stream generated by the MM2S and S2MM channels. Minimizing the width directly reduces the required amount of FPGA resources like LUTs, FFs and routing resources. The FPGA design is limited by these routing resources (Ilmberger, 2023). As described in subsubsection 2.1.2 the theoretical maximum data rate of an AXI bus is determined by the clock frequency and the data width.

There are three choices for the clock

1. The $125\,\mathrm{MHz}$ clock shared by the Host-ARQ and the playback executor.
2. The same clock as the XilinxMIG memory interface.
3. A clock not shared with any of the ports

The second option was selected with the XilinxMIG operating in 2:1 mode resulting in a clock frequency of $200\,\mathrm{MHz}$ in contrast to the 4:1 mode used by the old buffer design. Using the 2:1 mode instead of the 4:1 mode with a clock frequency of $100\,\mathrm{MHz}$ halves the required data width as the clock frequency is doubled and therefore reduces the number of FPGA resources needed. For the same reason using the $200\,\mathrm{MHz}$ was preferred over the $125\,\mathrm{MHz}$ clock of the Host-ARQ and playback executor interfaces. Furthermore, using a clock that is shared with some ports of the module reduces the required clock domain crossing logic.

The S2MM an MM2S AXI-Streams are configured to use a data width of 64 bit. At clock frequency of $200\,\mathrm{MHz}$ this results in a bandwidth of $12.8\,\mathrm{Gbit\,s^{-1}}$, satisfying the minimum requirement of $8\,\mathrm{Gbit\,s^{-1}}$. By choosing the same data width as the playback and trace streams of the playback executor no width conversion is necessary. Only limited information about the bandwidth that can be achived by the AXIDMA is provided by Xilinx. Xilinx specifies that an AXIDMA operating at a clock frequency of $100\,\mathrm{MHz}$ is able to achieve $99.76\,\%$ of the theoretical throughput on the MM2S channel and $74.64\,\%$ of the theoretical throughput on the S2MM channel when transferring $10\,000\,\mathrm{B}$ (*AXI DMA LogiCORE IP Product Guide (PG021)* 2022). Assuming the relative throughput is independent of the clock frequency operation at $200\,\mathrm{MHz}$ should be able to achieve a bandwidth greater than the requirement of $8\,\mathrm{Gbit\,s^{-1}}$.

For the data width W of the remaining XilinxMIG AXI Subordinate interface and the S2MM and MM2S AXI Manager a choice of 64 bit and $128\,\mathrm{bit}$ is evaluated.

Xilinx specifies that the bandwidth achievable by the XilinxMIG will vary depending on the access pattern and other system parameters. It is of course limited by the maximum bandwidth that is achievable using the DDR3 interface of the memory. An upper bound of this bandwidth B_{DDR3} can be determined from the clock frequency the memory is operated at $f_{\mathrm{DDR3}} = 400\,\mathrm{MHz}$ and the number of data lanes $n_{\mathrm{dq}} = 32$

$$B_{\mathrm{DDR3}} = 2 f_{\mathrm{DDR3}} n_{\mathrm{dq}} \cdot 1\,\mathrm{bit} = 25.6\,\mathrm{Gbit\,s^{-1}}$$

This upper bound is not strict, it can not be reached continuously due to the operations required by the DDR3 protocol like refresh pauses and row pre-charging time (JEDEC, 2012). As DDR3 operates in a half-duplex fashion, this bandwidth is shared by both reads and writes to the memory. This yields an efficiency necessary to satisfy the full bandwidth on both the trace and the playback channel of

$$\frac{2 \cdot 8\,\mathrm{Gbit\,s^{-1}}}{B_{\mathrm{DDR3}}} = 62.5\,\%$$

Finally, three different choices for small BRAM-based FIFO added to the playback instruction and trace data streams between the playback executor and the AXIDMA (labeled `playback FIFO` and `trace FIFO` in Figure 4) are evaluated:

- No FIFOs.
- A packet mode 256 word FIFO for the playback instruction stream. A packet mode FIFO will only transmit data on its Transmitter interface if it is full, or it contains at least one whole packet, as signaled by the *TLAST* signal.
- A packet mode 256 word FIFO for the playback instruction stream and a (non packet mode) 256 word FIFO for the trace data stream.

Guided by measurements of the actually achieved bandwidth on the trace and the playback streams as described in subsubsection 3.6.3 and subsection 4.1 the data width W is chosen to be $128\,\mathrm{bit}$ and both, the `playback FIFO` and the `trace FIFO` are included.

Lastly note that in this case the ValidNext stream Transmitter of the playback executor used for the trace stream can be directly connected to the S2MM AXI-Stream Receiver, as the S2MM AXI-Stream Receiver does not wait for a *TVALID* signal until it asserts *TREADY*.

3.4. Theory of operation

The old playback and trace buffer design was used by the host for sending the playback programs that it wants to execute to the FPGA and then receiving the resulting trace data. The new design requires more steps to execute a set of playback programs and receive the generated trace data. First, the host writes the playback instructions corresponding to a playback program that should be executed to the DDR3 memory using the FAXI block. It does not have to place the instructions into one contiguous region of the memory but instead can split the instructions into multiple regions. To execute the playback instructions the host writes a descriptor chain to the scatter gather descriptor chain memory that instructs the AXIDMA to read the (potentially multiple) regions belonging to each playback program in the correct sequence. Furthermore, the host creates a chain of descriptors that is used by the S2MM channel to store the received trace data in unused regions of the DDR3 memory.

The old playback and trace buffer design sends back the trace data to the host as soon as trace data is generated. To separate which trace data was received for which playback program the host then uses the `halt()` instruction at the end of each playback program which is looped back to the trace data once it is executed by the playback executor. In this new design readout of the trace data has to be initiated by the host. The host can determine the number of received bytes from the status fields of the descriptor chain used for the trace data. However, the completed flag and the number of received bytes stored in the status field of a descriptor used by the S2MM channel is only updated under two circumstances. It is updated when the number of bytes written matches the configured buffer length or when the S2MM channel receives a whole packet (as signaled by the *TLAST* signal). In general the trace descriptor chain cannot be configured to contain exactly the number of bytes used by the trace data, as the amount of trace data that is generated cannot be known ahead of time in general so the *TLAST* signal has to be used to control the S2MM channel to switch to the next descriptor and update the status field. The UTEncoder used by the playback executor to generate the fixed word width trace data stream has therefore been extended to support the *TLAST* signal so that the *TLAST* signal can be generated by the playback executor whenever a `halt()` instruction is added to the trace stream.

After the host has written the descriptor chain for the playback and the trace data, it starts the execution of the playback program configuring the S2MM and the MM2S channel with the correct descriptors. By polling the status field of the trace data descriptors the host waits for every playback program to be executed and finally reads the generated trace data.

This section outlines that executing playback programs and receiving their results requires substantially more steps with the new design. The most significant difference is that the transmission of the trace data from the FPGA and to the host is not started as soon as it is generated, but instead only when a playback program is finished. This can potentially significantly increase the experiment runtime for certain experiments. This disadvantage will be further analyzed at end of subsection 3.5 and in subsection 4.3. Finally, subsection 6.1 will outline possible future extensions of this new design to alleviate this disadvantage.

3.5. Software integration

The FPGA design is only one part of the tools required to perform experiments using the BrainScaleS-2 ASIC. On the host computer side, a set of software components are used for the description and execution of experiments. These software components follow a layered approach with layers exposing an increasingly more abstract interface to their users. A detailed description of this software stack can be found in (Müller, Arnold, et al., 2022).

To use the new playback and trace buffer described in this thesis, changes to this software stack are required. These changes can be categorized into to two different categories

1. Changes that are required because the operations that need to be performed by the host to execute a playback program on an FPGA and to receive the resulting trace data have changed. For example the host has to program the AXIDMA block correctly and needs to use the FAXI block to read and write to the playback and trace memory.

2. Changes that allow the software to make use of the new features made possible by this new playback and trace buffer. This includes the ability to reuse (parts) of an already transmitted playback program and partial readout of the received trace data.

In this thesis only the changes falling into the first category are performed. Changes that fall into the second category are prepared by making the lower level API more flexible, however these additional features are not exposed to the higher levels. The changes belonging to the first category are implemented by extending the BSS2 software architecture with an additional layer in the communication category, ayo (Axi memorY Orchestrator) that is responsible for communication using the FAXI interface and the configuration of the AXIDMA block. Figure 5 shows an overview of the software stack and the location f this new ayo layer. The ayo layer is used by the hxcomm library. hxcomm is used by the higher level software to run a single playback program and retrieve the trace data that is generated for them. It is responsible for the UT encoding of the playback instructions and UT decoding of the trace data as well as the abstraction of the different backends for communication with either the actual hardware or a simulation of the hardware.

Listing 1 gives an overview over the main API of ayo. The alloc function is used to reserve a region of memory that is big enough to hold the specified number of bytes. This function returns an opaque Handle that is used by the free function, which marks the region as unused again as well as the read and write functions that are used to read and write from the memory region corresponding to the Handle. Finally, the run is used to schedule the execution of one or more playback programs. The list of playback_regions defines the order and the location of the parts of the playback programs, that will be sent to the playback executor, while the list

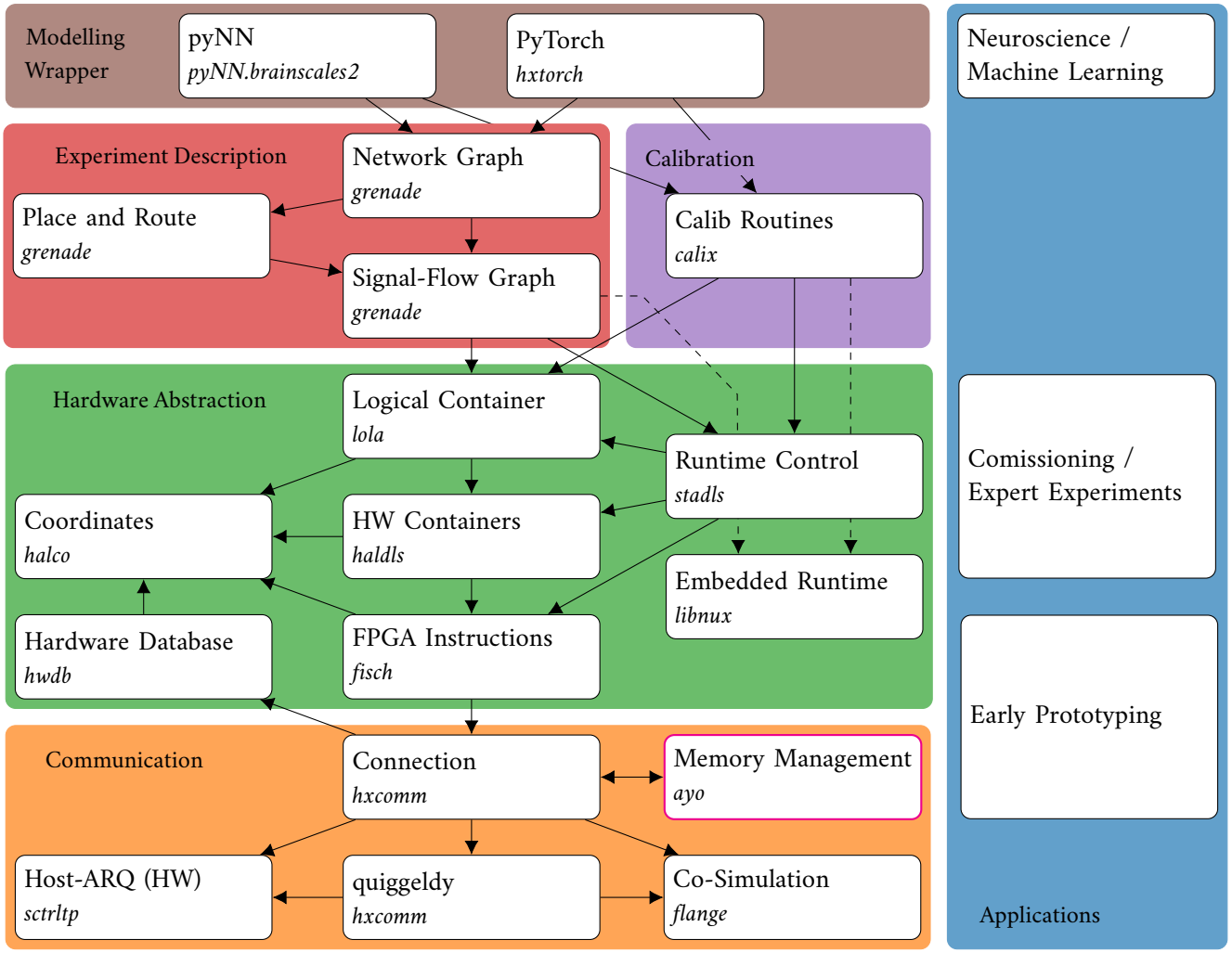

Figure 5: Schematic overview of the BSS2 software architecture. In this thesis it was extended by the ayo component marked in magenta. Figure modified from (Müller, Arnold, et al., 2022)

```
using axi_word_t = uint64_t;
class AYO
{
    Handle alloc(ddr3_size_t bytes);

    void free(Handle handle);

    std::future<WriteResult> write(Handle target, std::vector<axi_word_t>
    ↪ words);

    std::future<std::vector<axi_word_t>> read(Handle target);

    RunResult run(std::vector<WriteResult> playback_regions, std::vector<Handle>
    ↪ trace_regions);
};

class RunResult {
    void wait();
    std::future<std::vector<axi_word_t>> read();
    std::future<Status> status();
    std::vector<Result> traces();
    std::vector<Result> playback_programs();
}

class Result {
    void wait();
    std::future<std::vector<axi_word_t>> read();
    std::future<SingleStatus> status();
}
```

Listing 1: Overview of the interface presented by ayo. It was simplified for brevity.

of trace_regions contains a list of memory regions that is used for the trace data. The RunResult returned by this function is used to query the status of the execution as well as read back of the trace data.

With the modifications performed in this thesis hxcomm uses only a subset of the functionality provided by the ayo layer and the interface of hxcomm is not modified. Accordingly, all higher software levels can use this new playback and trace buffer design without any changes. The interface of hxcomm allows exactly one playback program to be executed and its trace data to be received. With ayo this is performed by simply allocating two memory regions, one that contains the whole playback program and a second one that covers the rest of the available memory for the trace data. The playback program is written to the playback and trace memory using write. Afterwards the playback program is executed by using the run function with exactly one playback and one trace region. Finally, the received trace data is read and transmitted to the higher level.

Internally the run functions converts the list of playback and trace memory regions into a playback and a trace descriptor chain. Consecutive playback regions are merged and each resulting playback and trace memory region is described using at least one descriptor. Regions that are longer than the maximum buffer length that can be specified in a descriptor (67 108 863 Byte) are converted into multiple descriptors. The descriptor chains for the playback region are linked together in the same order as they were given to the run function, the same applies to the descriptor chains for the trace regions. Consecutive trace regions are not merged together,

as every playback program needs at least one trace descriptor, due to the `halt()` instruction ends a playback program causing the AXIDMA to switch to the next trace descriptor as described previously. Figure 6 shows a schematic overview of how the descriptor chains are created by the run function.

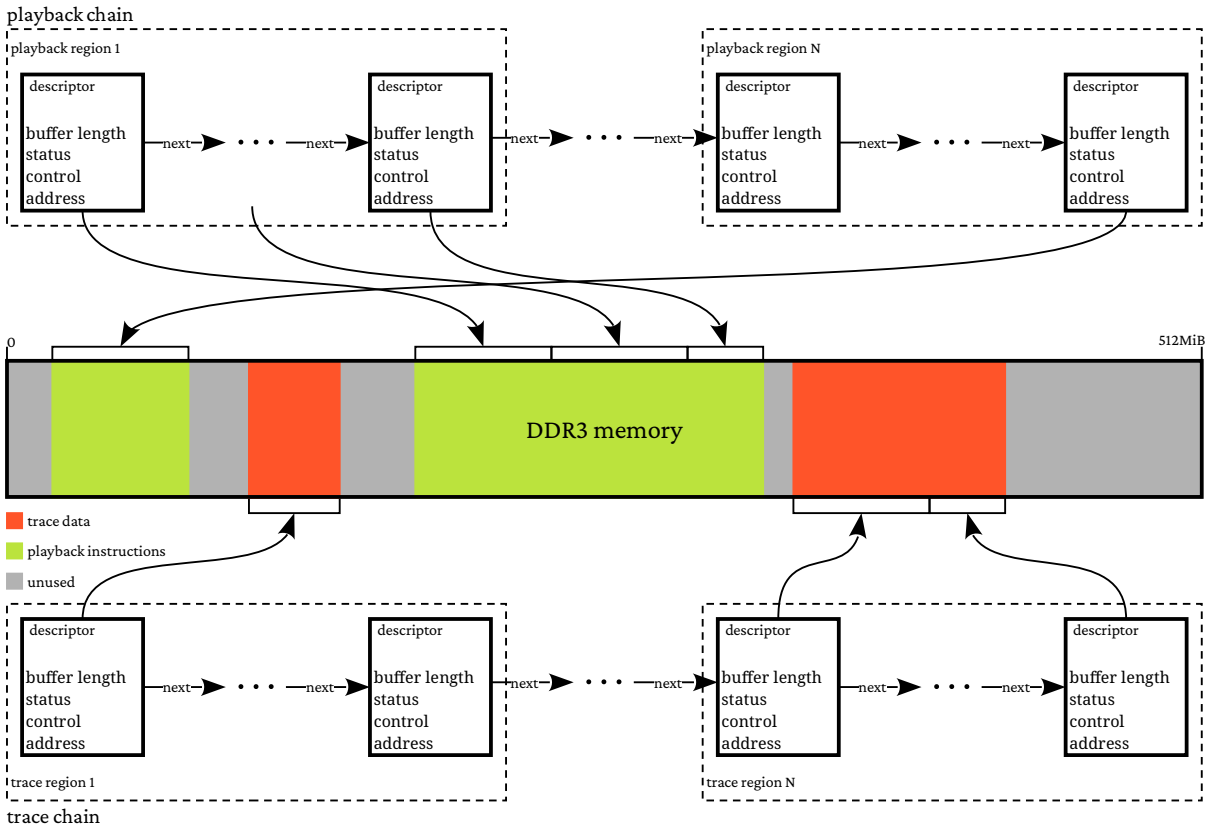

Figure 6: Schematic overview of an example for the playback and trace descriptor chains created for a set of playback and trace regions. In this example two playback regions were specified. The first playback region contains more than twice the maximum permissible buffer length for a single descriptor and is therefore converted into a chain of three descriptors. The second playback region is small enough for one descriptor to be sufficient. Note that the first playback region is stored after the second in the memory, but due to the order of the playback descriptors, the first playback region will be transmitted to the playback executor before the second one. This example also has two trace regions, the first being small enough to be described by a single descriptor and the second needing two descriptors. The order of the playback and trace regions in the DDR3 memory is independent of the order they are actually used in.

The rate of experiments that can be performed in a hardware in-the-loop style operation, where each playback program depends on the trace data of the previous playback program is expected to be limited mainly by the round trip time between the host and the FPGA. The old playback and trace buffer design requires the minimum possible number of round trips — one — from the host to the FPGA. It sends the playback program and then receives the trace data as it is generated. To perform a single experiment with the new playback and trace buffer design, five steps need to be performed:

1. The playback program is written to the DDR3 memory
2. A descriptor chain for the playback program and the memory region that captures the trace data is written to the descriptor memory
3. Operation of the S2MM and MM2S channels of the AXIDMA is started by writing the proper descriptor

chain addresses to the `curdesc` and `taildesc` descriptors.

4. The status field of the descriptor(s) for the trace data is polled to determine when the playback program has finished execution as well as the size of the generated trace data.

5. The trace data is read from the DDR3 memory.

The first two steps need to be completed before the third step is performed to guarantee both the playback and the descriptor data was written before it is read by the AXIDMA unit. A naive implementation would wait for the reception of the write response for the first two steps until it continues with the third step, but using the special globalfence operation of the FAXI block ayo does not have to wait for response to the write operations in steps item 1 to item 3 and can instead ensure the write operations for the first two steps was completed before the AXIDMA unit will be configured and in turn the data written in the first two steps will be read by the AXIDMA by inserting a globalfence operation before the write operations that configure the AXIDMA. It is however unavoidable to wait for the response to the read operations that poll the status field of the trace descriptors, as the response data is used to determine when the readout of the trace data can be started as well as determining the size of the generated trace data. For an experiment with a single trace descriptor the lowest number of round trips that are necessary is therefore the number of times the status field has to be read until the playback program is completed plus one round trip for the readout of the trace data yielding a minimum of two round trips. Accordingly, in the case of hardware in-the-loop style operation with small playback programs it is expected that the rate of experiments is at most half of the rate that can be achieved with the old playback and trace buffer design. Future extensions of the playback and trace buffer design could reduce the number of roundtrips required again, by for example introducing a separate channel for the trace data, that bypasses the trace buffer and directly sends the trace data to the host.

3.5.1. Allocator

With the old FIFO based system, the VFIFO module in the FPGA is responsible for placing the playback data in the DDR3 memory, reading it again from the address it was placed at, and vice versa for the trace data. Changing the host interface to be memory mapped instead of FIFO based shifts the responsibility deciding the placement of the playback and trace data from the FPGA to the host. Furthermore, the playback data is no longer accessed in a strictly FIFO way, but instead can be built up from multiple blocks, with some blocks potentially being used by multiple experiments. Management of the memory is abstracted by ayo which simply presents the higher layers with the alloc and free functions that are used to reserve regions of memory and mark them as unused again. In the ayo layer tracking which regions of memory are already used for playback or trace data as well as allocating new regions or marking previously used regions as unused is done by an allocator, which in turn provides the address for each of the memory regions. Its interface is described in Listing 2.

On creation, the allocator is given the size of the memory used for the playback and trace. Its alloc function finds an unused region in the memory that has at least a size of size bytes and returns the address of the first byte in this region. When a memory region is no longer needed it can be marked as free again by calling the free function with the address of the first byte of the region. Furthermore, the sum of the size of all unused regions can be queried using the available_space function.

Layers using ayo can provide a custom implementation of the allocator. A default implementation is provided by the ayo layer. In the default implementation the allocator keeps a single list of free regions and allocates regions from this list using a best-fit strategy. When freeing a region creates consecutive free regions they are

```cpp
template <typename T>
concept Allocator = requires(T allocator, size_t maximum_size, size_t size,
    size_t pointer)
{
    { T(maximum_size) } -> std::same_as<T>;
    { allocator.alloc(size) } -> std::same_as<size_t>;
    { allocator.free(pointer) };
    { allocator.available_space() } -> std::same_as<size_t>;
};
```

Listing 2: Interface of the allocator. An allocator is give the size of the memory region it manages on creation. The alloc function is used to find a region of memory that is not yet marked as used by previous calls to alloc that fits at least *size* bytes. The offset of the first byte of this region from the start of the complete memory is returned. This offset will also be called the pointer to this region. Using the free function a region of memory is marked as unused again. It is called with the pointer to a memory region. Finally the available_space function returns the number of bytes that are not part of memory regions marked as used.

combined to form a larger free region. This implementation was chosen for its simplicity while being sufficient for all tests performed in this thesis.

3.6. Verification and comparison

The correct operation of the new playback and trace buffer as well as the ayo layer integrating it into the BSS2 software architecture is verified at different layers using unit and integration tests. Moreover, various performance aspects of the old and the new playback and trace buffer design are compared.

3.6.1. Playback and trace buffer

At the lowest level the correct operation of the playback and trace buffer that is part of the FPGA design is verified on its own using simulation with the xcelium simulator. Writing simulation test benches for FPGA cores can be done in a variety of ways, which can be broadly classified into two categories

- Writing test benches in a HDL for example using UVM ("IEEE Standard for Universal Verification Methodology Language Reference Manual" 2020).
- Writing test benches in a programming language and using a Co-simulation interface like VPI ("IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language" 2018) or DPI ("IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language" 2018) to interact with the simulated FPGA core.

For verification of the standalone playback and trace buffer the cocotb framework was used. This is a Co-simulation framework that uses the VPI and VHPI ("IEEE Standard VHDL Language Reference Manual - Procedural Language Application Interface" 2007) interface to allow writing test benches using the Python programming language. Using cocotb has multiple advantages. The cocotb-axi module provides abstractions for using AXI, like AXI and AXI-Stream Transmitter and Receiver implementations, as well as the AXIRAM module, an AXI Subordinate that implements a RAM. Using these abstractions, the Host-ARQ read and write, as well as the playback and trace streams are replaced with AXI-Stream Transmitters and Receivers from the

cocotb-axi module and the AXI DRAM controller is replaced by the AXIRAM module. Furthermore, using Python makes it possible to use the rich ecosystem of Python modules to perform various tasks. In this test bench the construct (Arkadiusz Bulski and Simpson, 2022) module was used to perform serialization and deserialization of the bit fields that have to be created to interact with the FAXI or the AXIDMA module.

3.6.2. flange-dram

The communication layer of the BSS2 OS presents higher level software with an unified interface for different communication backends (see Figure 5). This allows all software using hxcomm to transparently switch between interacting actual hardware and simulated hardware. In the simulated case, communication between the hxcomm and the simulator simulating a combination of the FPGA design and the ASIC is implemented using the flange library (Spilger et al., 2018).

flange consists of two parts. The first part is a library that is loaded by the simulator. This library interacts with the simulator using the DPI interface and exposes functions acting as stream Receiver and a stream Transmitter, as well as functions to schedule special actions in the simulation like stopping or resetting the simulation. The stream Transmitter and Receiver are used to replace the Host-ARQ block and are connected to the input and output stream ports of the playback and trace buffer. An RCF (Delta V Software, 2020) based network server exposes the special actions as well as reading or writing to the streams to the second part of flange, which uses these to remotely control the simulation.

For simulations of the FPGA design modified in this thesis the Electronic Vision(s) group used either of two different simulation models for the AXI accessible DRAM. The first uses the same XilinxMIG IP core as used for synthesis and connects it to a DDR3 simulation model provided by micron (Micron, 2015). The second option replaces both the MIG and the DDR3 model with an AXIBRAMController connected to a behavioral model of a chain of Block-RAMs. While the first option offers a more accurate simulation it reduces the simulation performance. The second option allows for a faster simulation, but in addition to being less accurate also only models a size of $32\,\mathrm{MiB}$ instead of the actual $512\,\mathrm{MiB}$ present on hardware.

To test that ayo correctly interacts with the FAXI and AXIDMA blocks, it is useful to be able to test interaction with the FAXI and the AXIDMA block separately. This is only possible if the DRAM that is accessed by both can be accessed by the test suite without using FAXI. In a simulation environment there are several options to accomplish that:

- Expose an interface to the test suite to interact with the design hierarchy. This could for example use VPI to allow enumeration, reading and writing of all verilog signals. Using this interface, the test suite could read or write to the signals corresponding to the memory of the DRAM simulation model. This interface has several advantages. It does not need any modification of the simulated design and is general enough to be used with both the AXIBRAMController based model and the MIG based model. Furthermore, this interface could also be used for verification of other components, that requires access to signals in their design hierarchy. The main disadvantage is that this interface couples the test suite and the FPGA design more tightly, because the test suite no longer only accesses the top level ports of the design, but can access any signal in the FPGA design.

- Expose an interface to the test suite that allows the test suite to act as an AXI Manager and interact with the simulation. Using an AXI interconnect the rest of the FPGA design and this AXI Manager could be connected to the AXI DRAM simultaneously. This option also does not need any modification to the AXI DRAM simulation model. In addition to that it could also be extended to be usable outside of simulation

feature	status
data width	$8 \cdot 2^n, n \in \mathbb{N}_0$ bits
address width	<= 64 bits
transaction ID	not supported
AxLOCK	not supported
AxCACHE	not supported
AxPROT	not supported
AxQOS	not supported
AxREGION	not supported
user signals	not supported
narrow transfers	not supported

Table 3: Summary of the AXI features supported by flange-dram

and in turn making the test suite usable with simulation and in hardware by using a synthesizable AXI Manager that is accessible over a side band communication method like JTAG in hardware.

- Add a third simulation model for the AXI accessible DRAM, that exposes its content directly to the test suite via flange. This has the advantage of being able to offer fast simulations, because the DDR3 simulation model is not used. Further, it could have the ability to simulate the whole $512\,\mathrm{MiB}$ of memory. However, it of course could not be used together with one of the other options for the AXI DRAM simulation model, specifically the more accurate MIG and DDR3 simulation model and therefore, offers a less accurate simulation.

In this thesis the last option was implemented to enable reads and writes to the memory from the test suite, fast simulations and the ability to simulate a memory with the same size as the actual memory present on the hardware. The flange-dram extension of flange was developed. It exposes an AXI Subordinate interface to the simulated FPGA design and an interface to the test suite that allows for reading from and writing to the underlying memory was developed. It is implemented using a SystemVerilog and a C++ part work together via DPI. Figure 7 shows an overview of its implementation. The SystemVerilog part is realized in the dpi_axi_ram module. This module creates a new memory using the create_axi_ram function of the flange-dram library, that is exposed to the module via DPI, and receives a unique id identifying this memory. It collects transactions on the *AW* and *AR* AXI channels and complete bursts on the *W* AXI channel. These transactions and bursts are transferred to the C++ part using DPI. Furthermore, it receives *B* transactions and *R* bursts from the C++ part and writes these to the respective AXI channels. Each instantiation of the dpi_axi_ram module creates a separate memory with a configurable size and address as well as data width. The C++ part manages the backing memory. The backing memory is allocated using mmap, which, if memory overcommitment is enabled, allows simulation of very large memories, while only using physical memory for the pages that are actually written to. It receives *AW* transactions and *W* bursts, matches them together, writes the data to the backing memory and generates a corresponding *B* response, as well as receives *AR* transaction, reads the corresponding data from the backing memory and generates an *R* burst. Table 3 summarizes the supported AXI features.

Finally, reads and writes to the backing memory are exposed to the test suite using the RCF framework already used by flange. The correct operation of flange-dram itself was verified with a test suite using the AXI Manager provided by cocotb

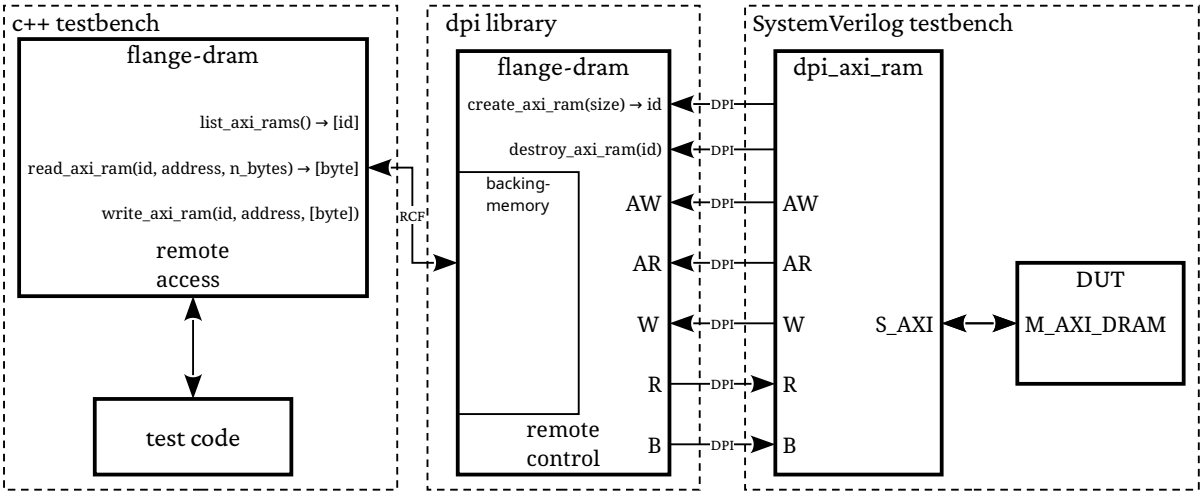

Figure 7: Schematic overview of flange-dram and its interaction with the C++ test suite as well as the SystemVerilog DUT. flange-dram is split into two halves that communicate via the RCF remote procedure call framework. The first half is a DPI library that exports several functions to be used by the SystemVerilog code. dpi_axi_ram is a SystemVerilog module that has an AXI Subordinate interface and uses the functions exported by the flange-dram DPI library to translate the AXI read and write transactions to read and write operations on memory allocated by the flange-dram library. This module uses the create_axi_ram function to create a new memory and receives an id that identifies this memory. Transactions received on the *AW*, *AR* and *W* AXI channels are communicated to the DPI library, using the id to identify the memory that they are targeting. Read and write response data is received using the R and B functions and converted to transfers on the *R* and *B* channels. Every instantiation of the dpi_axi_ram module creates a separate memory that can have different total sizes of the memory and different AXI data widths. The second half of flange-dram is a library used by the C++ testbench to read and write the contents of the memories created by the DPI library. Using the list_axi_rams function a list of the ids corresponding to the memories created by the DPI library can be obtained and the read_axi_ram and write_axi_ram are used to read and write from a memory identified by the id.

3.6.3. Bandwidth verification

The new playback and trace generator was designed to be able to sustain the maximum possible bandwidth of the trace and playback streams. However, without verification of this in hardware it is impossible to determine if it is actually able to achieve this bandwidth, due to the limited information on the performance of the AXIDMA and the XilinxMIG given.

For the AXIDMA IP Core it is expected, that it can process the descriptors at a fixed rate R that is slower than 1 descriptor per clock cycle. This means that if a sufficiently low buffer length with each descriptor is used, the bandwidth of the playback and trace streams will not be limited by the XilinxMIG or DDR3 memory but instead by this rate R. The minimal buffer length L for which the full playback and trace bandwidth could be reached is then

$$L = \frac{B_{\text{pb\_trace}}}{R}$$

Where $B_{\text{pb\_trace}}$ is the number of bytes that can be transmitted on the playback and trace streams per clock cycle, here $B_{\text{pb\_trace}} = 8\,\text{B}$.

To verify the actually achieved bandwidth the playback executor was extended by a dummy data generator. This dummy data generator can be programmed to emit the programmed number of words on the trace stream using the newly added `emit_dummy(N)` instruction. The payload for this dummy data is an FPGA internal counter that increments with every clock cycle of the playback executor called systime. Dummy words are given the highest priority in the trace arbitration. The dummy data generation is mainly used to verify the bandwidth of the trace stream. To verify the bandwidth of the playback stream an instruction that can be processed by the playback executor on every cycle and has no unwanted side effect is chosen. In this case the `reset_sleep()` instruction was used. It resets a counter internal to the playback executor that is not used by the experiments performed here. Finally, every instruction and trace data word used in these experiments has a UT encoding of exactly one 64 bit word giving a one to one correspondence between the encoded and the unencoded playback instruction and trace data streams.

The first scenario that is investigated is the maximum bandwidth that can be achieved by the playback stream while minimal trace data is generated. Different buffer lengths L of the descriptors are evaluated. The descriptor memory has space for 2048 descriptors. One descriptor is needed for the trace data that will contain the value of the systime counter when the first and the last playback instruction was executed, and an additional descriptor is necessary to hold the `halt()` instruction used to mark the end of a playback program. This leaves 2046 descriptors that are filled with playback instructions. The first and last playback instruction are `emit_dummy(N)` instruction that each cause the dummy data generator to emit a single dummy word. A single clock cycle is required to execute these instructions. The dummy words emitted by these two instructions will contain the systime counter when they were generated. From this the value the systime counter had when the first and the last playback instruction were executed can be inferred. All other playback instructions are filled with the `reset_sleep()` instruction. The buffer length of each descriptor is varied by varying the number of words N contained in the memory region used by each descriptor. Figure 8 shows a diagram of the descriptor chains that are generated for this scenario.

As described in subsection 3.2 the access pattern of a DDR3 memory can have an influence on the read and write bandwidth that can be achieved. To investigate this effect three different placements of the playback instructions were used:

1. With the *linear* placement all playback instructions are located in the DDR3 memory in the same order they are executed and therefore read from the memory.

2. With the *random* placement the location of each region of memory used for each descriptor is randomized.

3. With the *random dense* placement the location of each region of memory used for each descriptor is randomized, but no gaps are allowed.

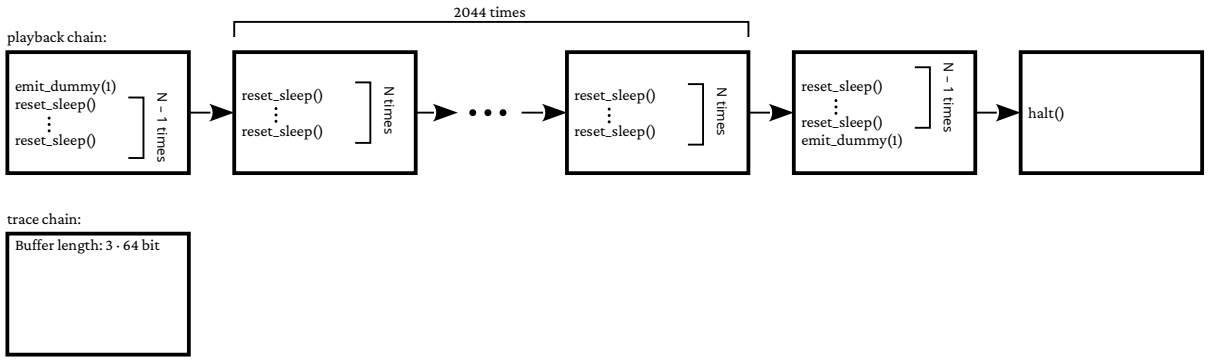

Figure 8: Schematic overview of the playback and trace buffer chains used to measure the playback bandwidth.

The second scenario investigates the maximum bandwidth that can be achieved by the trace stream while minimal words are transmitted on the playback stream. A single descriptor is used for the playback stream, that first configures a long timeout to avoid interruption of the generation of the dummy trace data by a timeout notification. It configures the dummy data generator to generate dummy trace data and after waiting for the dummy data generator to be idle terminates the playback program using the `halt()` instruction. The looped back `halt()` instruction is received by its own trace descriptor. The other 2046 trace descriptors are configured in one single descriptor chain. The number of words N that is received by each descriptor is varied. Finally, for the placement of the trace memory region used by each of the descriptors the same three placement strategies as for the first scenario are used. In this case the number of clock cycles that were needed to receive the whole trace data can be determined by comparing the systime value of the first and the last dummy word that is written to the trace memory. Figure 9 shows an overview of the descriptor chains used in this scenario.

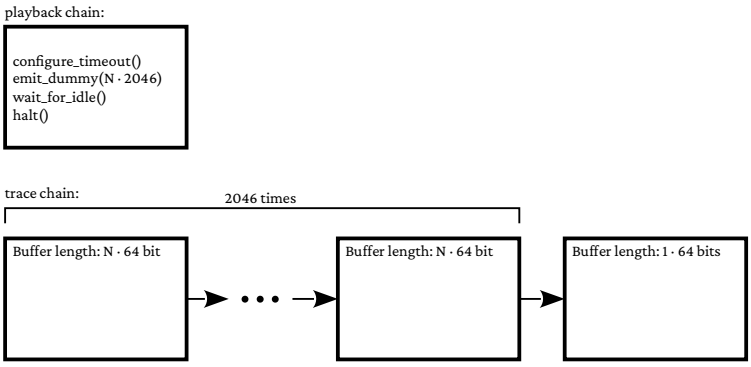

Figure 9: Schematic overview of the playback and trace buffer chains used to measure the trace bandwidth

The third scenario measures the maximum bandwidth achieved on the playback and trace streams when both are used at the same time. Using the dummy data generator, the playback executor is configured to generate the same number of trace words as it will receive playback instructions. One playback descriptor is used in the beginning to configure the timeout already mentioned in the second scenario. Furthermore, an additional descriptor is used for the playback chain that contains the `halt()` instruction used to indicate the end of a playback program. The `halt()` instruction is looped back to a third descriptor used in the trace chain. The remaining 2045 descriptors are split evenly between the playback and the trace chain. One descriptor stays

unused. The first instruction in the playback stream after the timeout configuration configures the dummy data generator to emit the number of dummy words necessary to fill the trace descriptor chain, while the last instruction before the `halt()` instruction causes systime counter to be read from an FPGA internal bus. This causes the value, that the systime counter had one clock cycle after it was executed, to be placed into a FIFO internal to the playback executor where it will remain until it can be written to the trace stream. As the dummy data generator has the highest priority this can only happen after all dummy data was generated and therefore does not influence the trace stream. Figure 10 gives an overview of the playback and trace chains that are generated for this case.

In this case again the number of words N used in each descriptor and the placement of the playback instructions and the trace data is varied. Here five different configurations for the placement were investigated:

1. *linear* placement, that places all playback instructions is the same order as they are executed, and also places the trace data for each descriptor consecutively.

2. *random* placement that randomizes the location of each region of playback instructions and trace data

3. *random dense* placement that randomizes the location of each region of playback instructions and trace data while not leaving any gaps

4. *interleaved* placement that places the playback instructions and the trace data into alternating regions leaving gaps between each of them

5. *interleaved dense* placement that places the playback instructions and the trace data into alternating consecutive regions

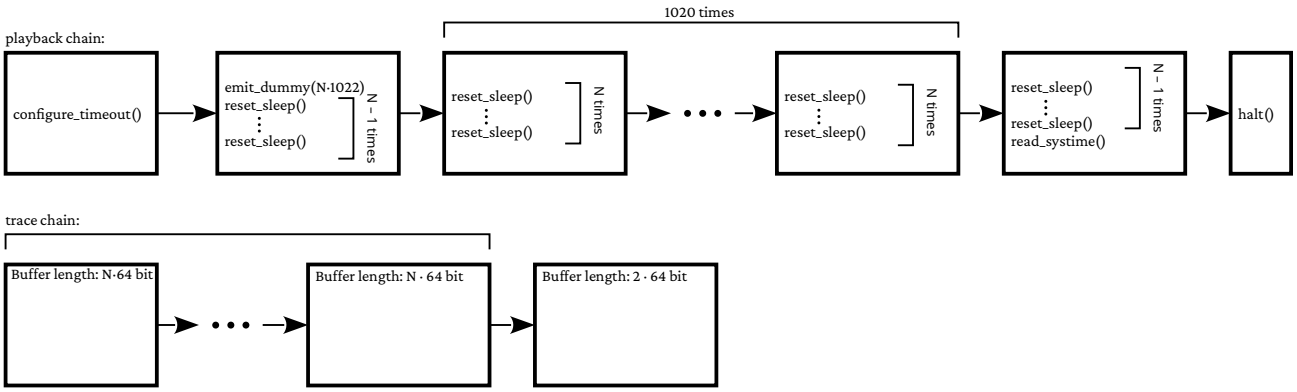

Figure 10: Schematic overview of the playback and trace buffer chains used to measure simultaneous playback and trace bandwidth

4. Results

4.1. Playback and trace bandwidth

The playback and trace bandwidth is measured for the three scenarios (maximum playback bandwidth, maximal trace bandwidth and maximal simultaneous playback and trace bandwidth) that were described in subsubsection 3.6.3. The following results highlight the need to use a data width $W = 128\,\text{bit}$ instead of $W = 64\,\text{bit}$ for the AXI bus between the DDR3 memory and the AXIDMA and the advantage of adding the `playback` as well as the `trace FIFO`.

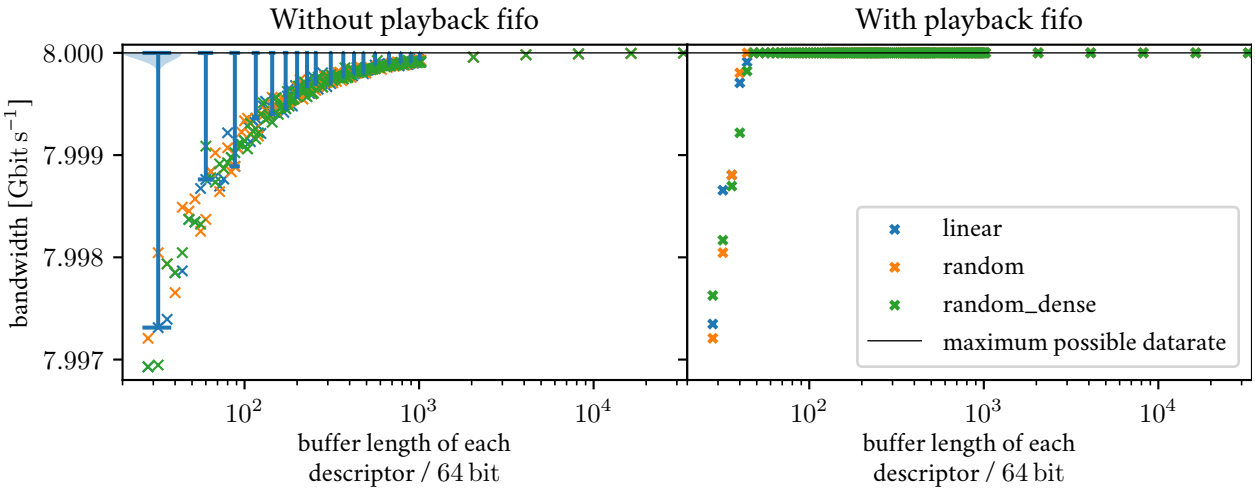

Figure 11: Comparison of the measured playback bandwidth, when only generating minimal trace data. The buffer buffer length for each descriptor was varied. Each of the three described placements for the memory containig the playback instructions was tested. For each buffer length and memory placement the measurement was repeated 1000 times and the crosses show the minimum bandwidth that was measured. The left panel additionally shows the distribution of all measurements for some of the buffer lengths tested with the random memory placement. Both panels used a data width for the AXI bus between the AXIDMA and the DDR3 memory controller of 128 bit, and the left panel did not use the additional `playback FIFO`, while the right panel used the additional `playback FIFO`

Figure 12 compares the playback bandwidth with and without `playback FIFO` when $W = 128\,\text{bit}$. The y-axis is scaled to only show a small region around the maximum possible bandwidth. We can observe, that for the maximum bandwidth cannot be achieved for small buffer lengths. This is in line with our expectations of a fixed minimum number of clock cycles required to process a descriptor. In addition to that, the maximum bandwidth is not always reached when using no `playback FIFO`. There is no buffer length where the achieved bandwidth was always maximal. In contrast, the bandwidth was always maximal for any buffer length greater than or equal to $48 \cdot 64\,\text{bit}$ when using the `playback FIFO`. This is true for any of the memory placement patterns tested.

For the trace bandwidth tests, the maximum bandwidth was achieved regardless of the chosen value for W and with or without the `trace FIFO`. In the case of $W = 128\,\text{bit}$ and when using the `trace FIFO`, for any buffer length greater than or equal to $52 \cdot 64\,\text{bit}$, the bandwidth was always maximal.

Finally, the simultaneous playback and trace bandwidth is investigated. Figure 12 shows the measured simultaneous trace and playback bandwidths for three different cases ($W = 64\,\text{bit}$ without `trace FIFO`, $W = 128\,\text{bit}$ without `trace FIFO` and $W = 128\,\text{bit}$ with `trace FIFO`). While the playback stream is again able to reach the maximum possible bandwidth for every buffer length greater than or equal to $68 \cdot 64\,\text{bit}$ for

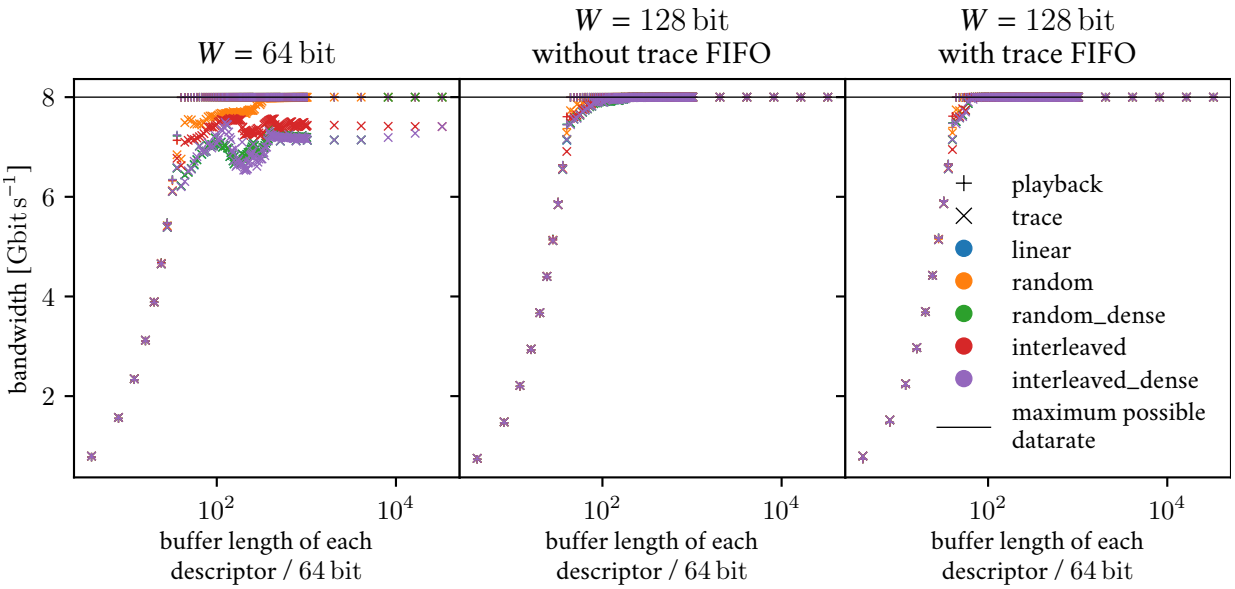

Figure 12: Comparison of the measured playback and trace bandwidth, for simultaneous playback and generation of trace data. The buffer length for each descriptor was varied. Each of the five described placements for the memory containing the playback instructions were tested. For each buffer length and memory placement, the measurement was repeated 1000 times. Both kinds of crosses show the minimum bandwidth that was measured. Playback bandwidth uses + symbols, while trace bandwidth uses × symbols. The parameter W describes the data width of the AXI bus between the AXIDMA and the DDR3 controller.

each of the three different cases, the trace stream cannot reach the maximum bandwidth for every memory placement, when using $W = 64\,\text{bit}$. Only the *linear*, *random* and the *random dense* memory placement are able to achieve the maximum possible bandwidth, however they only reach it for a buffer length greater than or equal to $16\,384 \cdot 64\,\text{bit}$, $2048 \cdot 64\,\text{bit}$ and $16\,384 \cdot 64\,\text{bit}$ respectively. Using 128 bit and no `trace FIFO` is able to reach the maximum possible bandwidth for all tested memory placements. The minimum buffer length for the trace descriptors that is required to always reach the maximum bandwidth is $720 \cdot 64\,\text{bit}$. Similarly using $W = 128\,\text{bit}$ and the `trace FIFO` is able to reach the maximum possible trace bandwidth, but in contrast to the design not using the `trace FIFO`, the minimum buffer length required to reach the maximum bandwidth on the trace stream is reduced to $80 \cdot 64\,\text{bit}$.

For all following experiments only the version with $W = 128\,\text{bit}$ and both the `playback` and the `trace FIFO` will be used.

Figure 13 and Figure 14 show comparisons between the playback, trace and simultaneous playback and trace bandwidth depending on the size of the playback program and or generated trace data. These bandwidths are measured using a modification of the three tests described in subsubsection 3.6.3. Instead of using the maximum number of descriptors possible the minimum number of descriptors that are needed to hold the complete playback program trace data is used.

The old playback and trace buffer design is not able to achieve the maximum bandwidth for many of the tested playback program and or trace data sizes. This is expected for very large trace data and playback programs, as it is only able to store 32 MiB of trace data and playback program instructions. playback programs greater than 32 MiB are limited by the bandwidth between the host and the FPGA as the instructions for them will be transmitted during their execution. The same applies to trace data bigger than 32 MiB, which is only accepted from the playback executor at the rate it can be sent to the host. In addition to that these measurements also reveal that the old playback and trace buffer design is not able to achieve the full playback and trace bandwidth

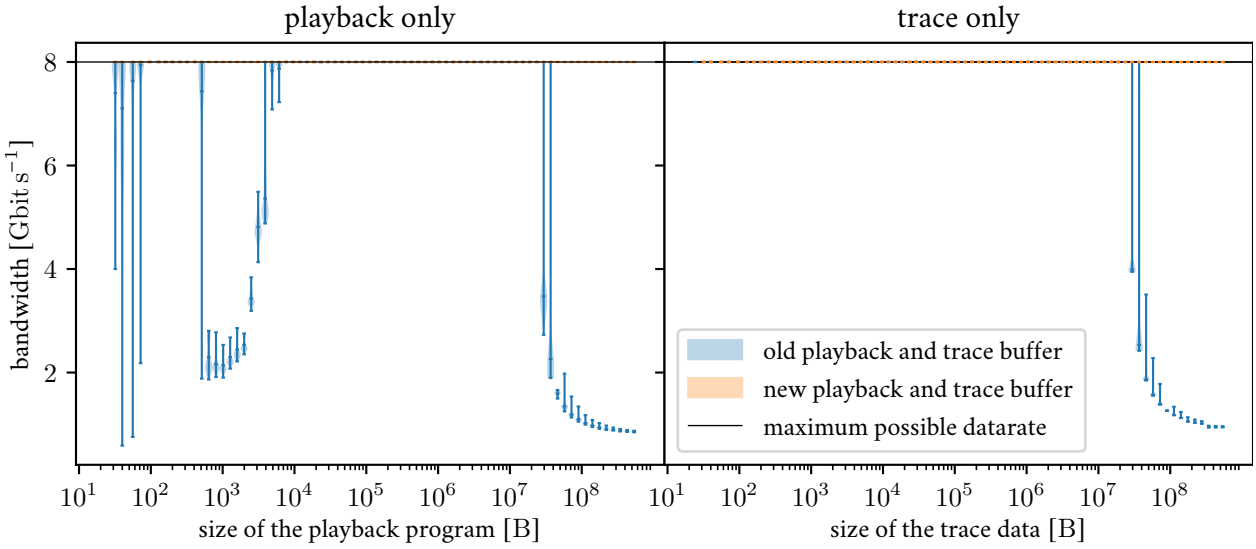

Figure 13: Comparison of the measured playback and trace bandwidth between the old and the new buffer design for different total sizes of the playback instructions or generated trace data. For the new buffer design linear memory placement and the maximum possible buffer length was used. The left panel shows the playback bandwidth when only minimal trace data was generated, while the right panel shows the trace bandwidth when only minimal playback instructions were generated. Each measurement was repeated 1000 times and the distribution of the bandwidths is visualized using a violin plot. The three vertical bars show the maximum, minimum and average bandwidth.

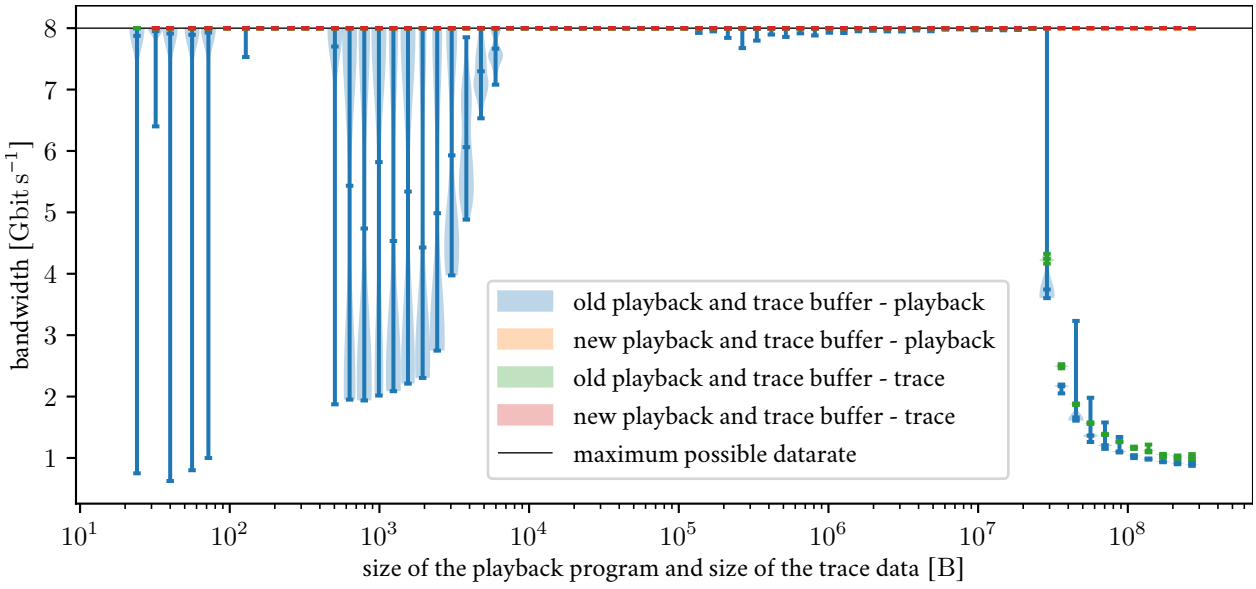

Figure 14: Comparison of the measured simultaneous playback and trace bandwidth between the old and the new buffer design for different total sizes of the playback instructions or generated trace data. For the new buffer design linear memory placement and the maximum buffer length possible was used. Each measurement was repeated 1000 times. The three vertical bars show the maximum, minimum and average bandwidth.

for playback programs and or trace data smaller than 32 MiB. In cases where the playback bandwidth is not maximal, the timing of the instructions can deviate from the intended timing and therefore repeatability suffers. Also, interpretation of results could be wrong. In contrast, the new playback and trace buffer design achieves the maximum bandwidth for any size of the playback program and or generated trace data that fit into the memory and therefore is there able to execute playback programs with strictly deterministic timing.

4.2. FAXI based memory mapped communication

The round trip time and the bandwidth of the FAXI based AXI Manager over Host-ARQ are measured. All measurements were performed from the same host computer (`EpycHost1`), reserved exclusively for these test to avoid other running experiments influencing the results. To measure the round trip time, minimum size reads and writes of 8 Bytes from multiple AXI Subordinates are performed. For reads, the time elapsed between the transmission of header and address and the reception of the read data is measured, while for writes the time elapsed between the transmission of the write transaction and the reception of the write response is measured. The measured round trip time is visualized in Figure 15 and table Table 4 summarizes the average latency that was measured.

action	location	round trip time
read	DDR3 memory	(61 542 ± 71) ns
write	DDR3 memory	(61 278 ± 74) ns
read	AXIDMA register	(61 607 ± 85) ns
write	AXIDMA register	(61 551 ± 75) ns
read	descriptor memory	(61 630 ± 310) ns
write	descriptor memory	(61 245 ± 58) ns

Table 4: Summary of the round trip time measured for read and write operations to different AXI Subordinates

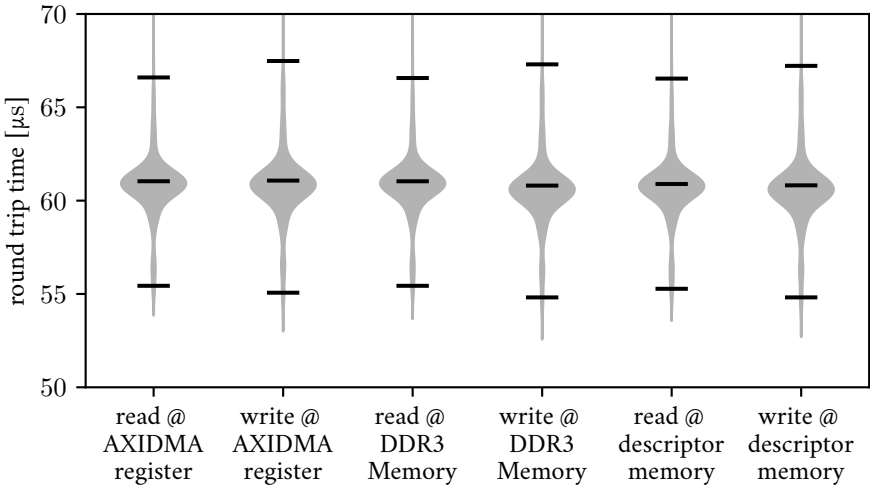

Figure 15: Measured round trip time of reads and writes of 8 Bytes from multiple different AXI Subordinates. Each operation was performed 10 000 times. The three bars show the mean as well as the first and 99th percentile. The order of the single measurements was randomized minimize the impact network performance fluctuations.

The bandwidth for reads and write to the DDR3 memory was measured for different read and write sizes, by measuring the time that elapses between sending of the read and write transactions and the reception of

their response. A baseline for the maximum bandwidth of the Host-ARQ protocol for sending and receiving data from the FPGA. The maximum bandwidth for sending was measured using a test sink built into the FPGA implementation of the Host-ARQ protocol. A dummy data generator built into the design was used to measure the maximum bandwidth for data received from the FPGA. For sending data to the FPGA a baseline bandwidth of $BW_{\mathrm{tx}} = (110.6 \pm 1.1)\,\mathrm{MiB\,s^{-1}}$ and for reception of data from the FPGA a baseline bandwidth of $BW_{\mathrm{rx}} = (118.6 \pm 1.3)\,\mathrm{MiB\,s^{-1}}$ was measured. Figure 16 shows the read and write bandwidth that was measured. For small sizes, the read and write bandwidth is limited by the round trip time and with increasing size of the reads and writes the fraction of time spent due to the round trip time decreases. For both reads and writes, a bandwidth close to the maximum possible bandwidth measured using Host-ARQ directly is reached.

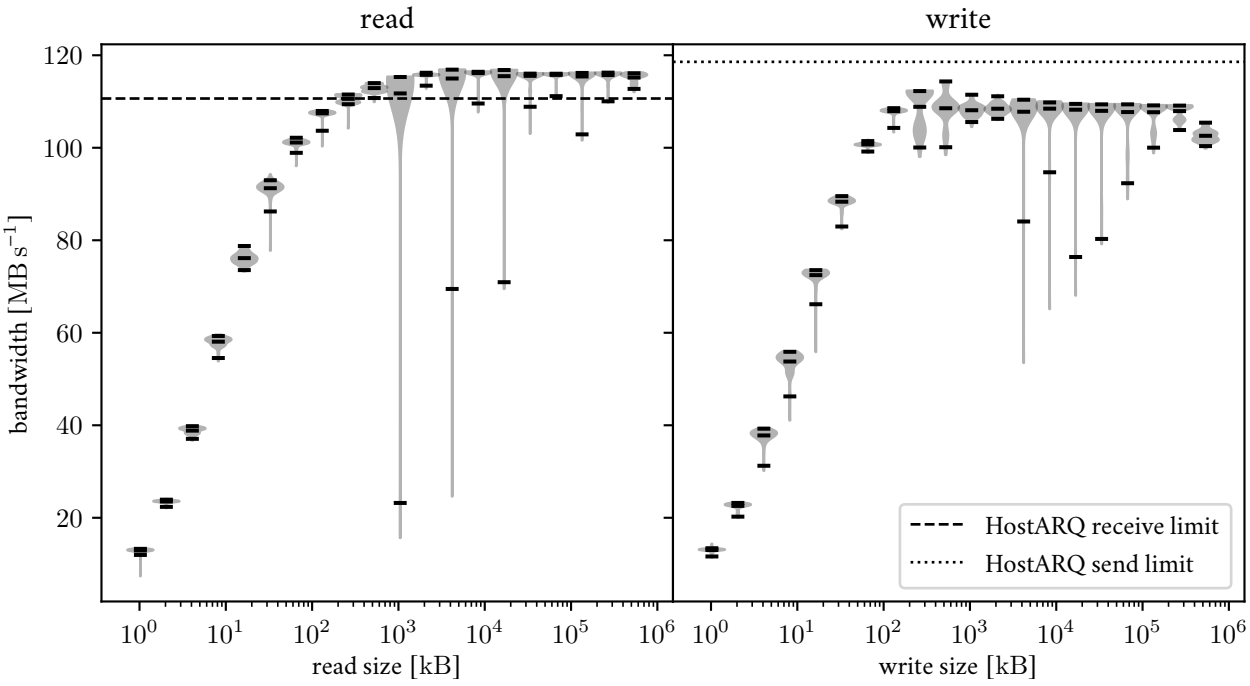

Figure 16: Measured read and write bandwidth for reads from the DDR3 memory of different sizes using FAXI. The left panel shows the read bandwidth and the right panel the write bandwidth. For each size the bandwidth was measured 100 times and is summarized using a violin plot. The three bars show the mean as well as the first and 99th percentile. The horizontal line shows the average transmission and reception speed measured when using Host-ARQ by itself.

4.3. Experiment rate

At last, the rate of experiments that can be performed in a hardware in-the-loop style is measured. The experiment that was used to test this is the same one used to determine the simultaneous playback and trace stream bandwidth. The size of the playback program and the generated trace data is varied. Figure 17 shows a comparison between the experiment rate that is achieved using the playback and trace buffer design presented in this thesis and the old playback and trace buffer design. To determine the experiment rate, the duration of the execution of a single experiment including the creating of the playback program and the reception of the trace data is measured. An upper bound for the rate of experiments R_{max} can be calculated from the round trip time, the time required to transmit the complete playback program and the time required to read back all the trace data:

$$R_{\mathrm{max}}(S) = (RTT + S/BW_{\mathrm{tx}} + S/BW_{\mathrm{rx}})^{-1}$$

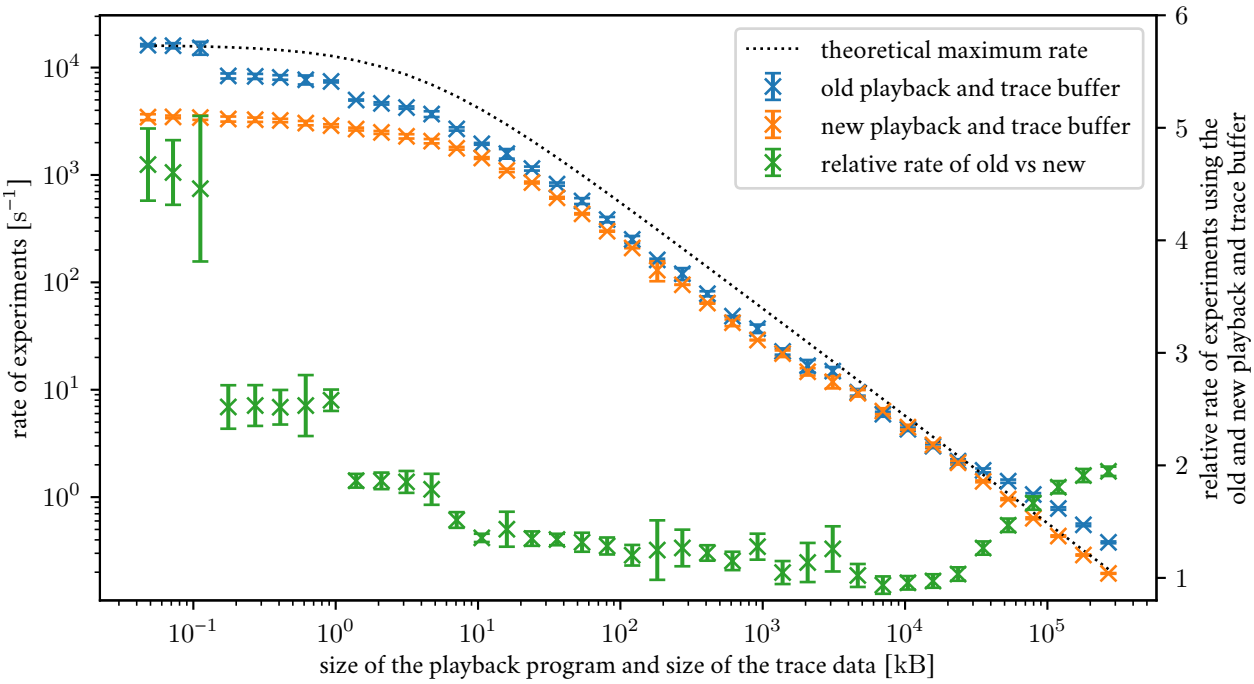

Figure 17: Comparison of the rate of experiments between the old and the new playback and trace buffer. For each size the time required to perform a single experiment was measured 100 times. In green the ration between the experiment rate of the old and the new playback and trace buffer design is shown. The vertical line indicates the location of 32 MiB on the x-axis. Finally the maximum achievable rate considering the round trip time and Host-ARQ bandwidth is shown in black.

Here S is the size of the playback program and the generated trace data, RTT is the round trip time, in this case the round trip time measured for the FAXI writes to the DDR3 memory was used. BW_{tx} an BW_{rx} are the maximum bandwidth that was measured for transmission and reception to and from the FPGA respectively. As can be seen in Figure 17 for very small playback programs the round trip time dominates the time required to perform an experiment as expected. The new playback and trace buffer design is at least two times slower than old playback and trace buffer design (because it needs at least two round trips to perform a single experiment). Furthermore, the experiment rate using the new trace and playback buffer design is always lower than the old one as the trace data is only read out once an experiment is completed. Before completion, it is not known how much trace data was actually generated and written to the DDR3 memory by the AXIDMA. The old trace and playback buffer design is able to send the trace data to the host while it is being generated instead. As the experiment rate is mainly limited by the bandwidth of the connection between the host and the FPGA, the effect of these differences deceases with increasing playback and trace size. The measurements show that the ratio between the rate of experiments with the old and the new playback buffer design increases again for playback programs that are larger than 32 MiB. This is caused by the playback memory in the old playback and trace buffer design being filled up for playback programs that are larger than 32 MiB which causes the execution of the playback program to be started before it is received completely. In this case the transmission of the playback program from the host to the FPGA and its execution overlap. In contrast to that, the new playback and trace buffer can use the whole size of the DDR3 memory and the execution of the playback program is not started until it is received completely. While this reduces the rate of experiments that can be performed compared to the old design, it allows for the timing of the execution of the playback instructions to be deterministic, which is not the case for the old buffer design. This nondeterminism of the old buffer design reduces repeatability and could lead to wrong interpretation of results.

4.4. flange-dram performance

Figure 18 shows a comparison between performance of the three different AXI accessible DRAM simulation models. The time required to execute a test case, that reads the JTAGID of the ASIC, when using the hxcomm simulation backend, is measured. This test case is part of the hxcomm test suite. For this test xcelium with version 21.03-s009 was used. One can see that the XilinxMIG and DDR3 based simulation is a lot slower than the other two options. Furthermore, the first repetition of the test case requires almost double the time of all following repetitions, when using the XilinxMIG based simulation. This is caused by the link training that is performed before the XilinxMIG can operate. The XilinxMIG offers a special mode for simulation, that speeds up the link training by skipping some steps of it. This mode was enabled for these tests. The other two simulation models do not require this link training phase. On average one execution of the test case requires (147 ± 14) s for the XilinxMIG and DDR3 simulation model option, (7.23 ± 0.12) s when using flange-dram and (6.86 ± 0.33) s when using the simulation model using Block-RAM. Using flange-dram provides a speedup of 20.33 ± 0.47 compared to using the XilinxMIG and the DDR3 simulation model.

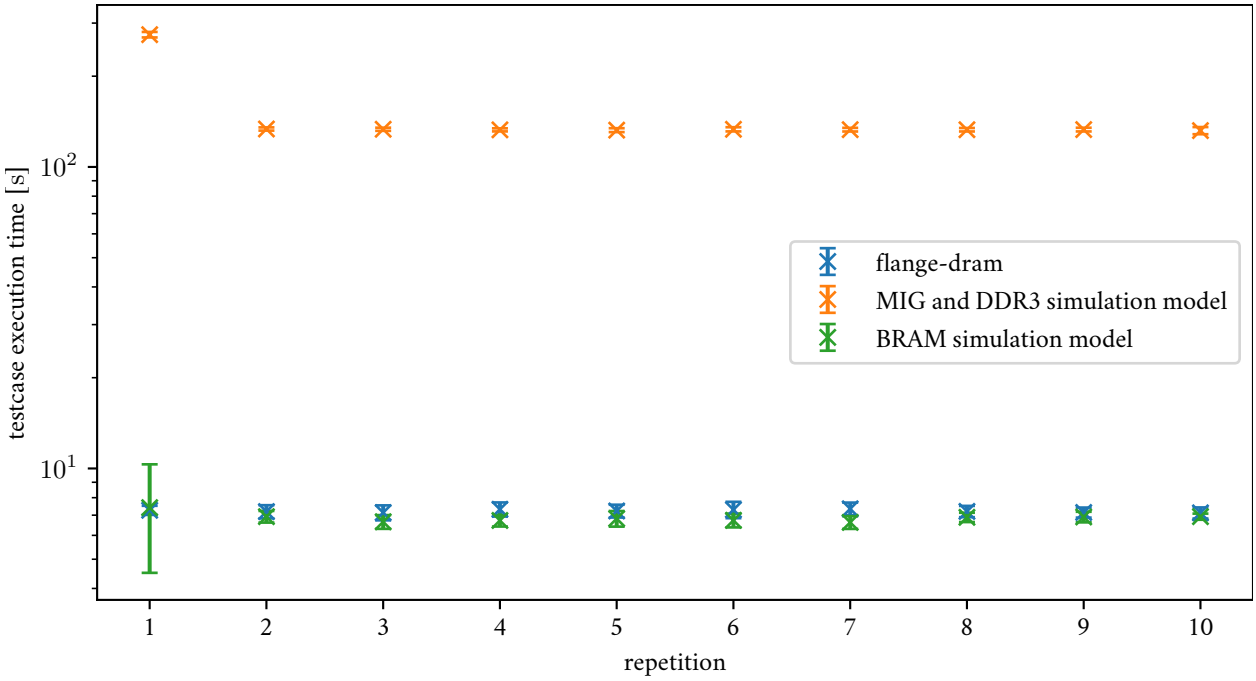

Figure 18: Time required by the hxcomm test case that reads the JTAGID of the ASIC using a FPGA and ASIC simulation. The three different choices for the AXI DDR3 memory simulation model are evaluated. The hxcomm test case is repeated 10 times in series and the time required for each shown separately. Each point is measured 10 times. Also note the logarithmic scaling of the y-axis.

4.5. FPGA resources usage

Table 5 compares the number of FPGA resources used by the old and new playback buffer design. The "total available resources" entry list the total number of resources available on the FPGA model used. Total LUTs includes LUTs used as logic LUTs, as distributed RAM and as shift register. Note that RAMB36 and RAMB18 are not separate resources, instead one RAMB36 counts as two RAMB18 and vice versa. The "rest of FPGA design" entry gives an estimate of the number of resources used by all parts of the FPGA design that were not modified. Note that this is only an estimate, as the other parts of the design are not completely independent of the modified parts and the changes of the modified parts can influence the synthesis and packing of parts that

are not modified. This comparison includes the XilinxMIG, as its configuration was modified from 4:1 clocking to 2:1 and the data width was halved from 256 bit to 128 bit. As outlined in the table, the new playback buffer design uses more than four times the number of LUTs and FFs as the old design (when not including the MIG in this comparison). However, the number of resources used was not main design goal and the new playback buffer together with the other parts of the FPGA design are well below the total amount of resources provided by the FPGA. The new playback and trace buffer only uses ≈ 18 % of the complete FPGA.

	subcomponent	Total LUTs	FFs	RAMB36	RAMB18	DSP Blocks
total available resources		101 400	202 800	325	650	600
rest of FPGA design		42 878	38 211	76	5	1
old playback and trace buffer		4002	7903	49	3	18
old XilinxMIG		6688	5653	0	0	0
old total		10 690	13 556	49	3	18
new playback and trace buffer		18 170	18 236	51	4	0
	AXIDMA	3232	5071	16	2	0
	AXI interconnect	13 741	11 960	0	0	0
	AXI-Stream clock converter for playback	104	206	0	0	0
	AXI-Stream clock converter for trace	111	238	0	0	0
	AXIBRAMController	278	364	0	0	0
	descriptor memory	30	7	32	0	0
	`playback fifo`	126	92	1	1	0
	`trace fifo`	127	92	1	1	0
	FAXI	429	206	1	0	0
new XilinxMIG		4983	4017	0	0	0
new total		23 153	22 253	51	4	0

Table 5: Comparison between the number of FPGA resources used by the old and the new playback buffer design. The FPGA entry gives the total number of resources available in this FPGA model. Total LUTs counts the number of LUTs used, these can be LUTs used as logic, as distributed RAM or as shift register. Moreover, note that RAMB36 is not a separate resource from RAMB18, instead one RAMB36 counts as two RAMB18 and vice versa. This overview includes the XilinxMIG as it was modified from 4:1 mode to 2:1 mode and from 256 bit to 128 bit data width

5. Summary and discussion

This thesis provides a new design for the playback and trace buffer that improves the old design in several points:

- Usage of the complete $512\,\mathrm{MiB}$ of storage available.
- The ability to reuse (parts of) already transmitted playback programs.
- The ability to read back the trace data in a different order than it was generated.
- Deterministic timing for the execution of any playback programs.

To achieve these goals the operation of the playback and trace buffer was redesigned from the ground. An FPGA module that allows memory mapped access from the host and uses a scatter gather DMA engine to construct the stream of playback instructions from multiple (potentially out of order) blocks was developed. It was integrated with the BSS2 software stack. A new software layer called "ayo" that is responsible for the low level interaction with the playback and trace buffer parts, such as reads and writes to the DDR3 memory holding the playback and trace data as well as the configuration of the DMA engine was implemented. ayo exposes an interface that makes it possible to use the new features, like the reuse of parts of the playback programs and partial readout of the trace data. Basic integration of ayo with the hxcomm layer was performed, which allows all current software to use the new playback and trace buffer design without any modifications, gaining the ability to use the much bigger $512\,\mathrm{MiB}$ of storage available.

The new design for the playback buffer and software integration was tested extensively and compared in detail to the old design. For these tests and comparisons, the simulation environment was extended by a software accessible AXI memory and the FPGA design was extended by a dummy data generator used to generate arbitrary amounts of trace data at maximum data rate.

It was verified that the software layer and memory mapped access to the DDR3 memory is able to achieve a similar bandwidth between the host and the FPGA as the old way of communication with the FPGA.

The only disadvantage of the new playback and trace buffer is the reduced rate of experiments that can be performed in a hardware in-the-loop fashion. For playback programs with a size of at least $1392\,\mathrm{B}$ and a trace data size of at least $1392\,\mathrm{B}$ the rate of experiments that is possible with the new playback and trace buffer design was show to be less than two times lower than the rate for the old design. For any size of playback program and generated trace data, the rate of experiments was always less than 5 times lower. The lower rate of experiments is caused by additional round trip times that are necessary between the FPGA and the host and subsection 6.1 outlines options for further extensions of this playback and trace buffer design to reduce the number of round trip times required and therefore increase the rate of experiments again. It was demonstrated that the complete size of the DDR3 memory is possible to be used instead of only $64\,\mathrm{MiB}$ used by the old buffer design.

Finally, it was verified that for playback program constructed from blocks of at least $68 \cdot 64\,\mathrm{bit}$ and for trace data organized into blocks of at least $80 \cdot 64\,\mathrm{bit}$, the new playback and trace buffer always achieves the maximum possible bandwidth. This was verified to hold even for a variety of possible placements of these blocks in the DDR3 memory space. This constitutes a mayor improvement over the old playback and trace buffer design, which is not able to sustain the maximum possible bandwidth for many playback program and/or trace sizes and improves the reliability.

6. Outlook

The new memory mapped playback and trace buffer and the software integration presented in this thesis lays a foundation for a lot of future improvements of the BrainScaleS stack.

6.1. Latency reduction

The only disadvantage of the new playback and trace buffer is that for small playback programs the rate of experiments that can be performed in a hardware in-the-loop fashion is significantly lower than the rate of experiments that can be performed with the old design. This could be improved using one of several approaches:

- The AXIDMA provides an interrupt signal that is asserted whenever the processing of a descriptor is completed. These interrupts could be used to avoid the need for polling of the trace descriptor status.
- Introduction of a separate low latency path for the trace data, that bypasses the playback and trace buffer and instead is sent directly to the host, similar to the old trace buffer.
- Introduction of memory mapped access from the FPGA to the host, to allow the FPGA to write the host memory directly. This could for example be achieved by implementation of a module similar to FAXI but operating is an opposite direction and providing an AXI Subordinate interface to the FPGA.

6.2. Higher level software integration

The ayo software layer already exposes the new functionality of the new playback and trace buffer such as the ability to reuse block of already transmitted playback programs and partial readout of the generated trace data. However, this functionality is not yet used by the upper layers of the BrainScaleS software stack. As the rate with which experiments that can be performed is limited by the bandwidth between the host and the FPGA for playback programs or trace data greater than 1392 Byte, integration of this functionality with higher level software is expected to allow the rate of some experiments to be improved.

6.3. Unified memory with PPU

The BSS2 architecture includes microprocessors on the ASIC that allow for sophisticated on chip processing. They can for example be used for closed loop operation of the ASIC. The FPGA is connected to a DDR3 memory, independent of the DDR3 memory used for the playback and trace buffer, that provides working memory for these microprocessors, that is used additionally to the on-chip SRAM. In the future it is envisioned to allow memory mapped access to this DDR3 by the host using FAXI. This would allow direct manipulation of the data and the programs running on the microprocessors. Furthermore, the microprocessors are envisioned to use AXI to access the playback and trace memory, descriptor memory and AXIDMA register space. This would allow them to directly create playback instructions and read the trace data and in addition to that control the execution of playback programs.

6.4. Support of new host interfaces

The old playback and trace buffer was tightly coupled to the stream interface of the Host-ARQ protocol. By replacing the FAXI module, the new playback and trace buffer design can be used with any host interface as long as one provides a bridge between the host interface and the AXI bus. Using this, one could for example use

PCIe instead of UDP as the communication interface between the FPGA and the host for a drastically increased bandwidth between the host and the FPGA as well as drastically reduced latency.

References

AMBA, ARM (n.d.). "AMBA AXI and ACE Protocol Specification." In: (). URL: https://developer.arm.com/documentation/ihi0022/hc.

Arkadiusz Bulski, Tomer Filiba and Corbin Simpson (2022). *construct*. URL: https://construct.readthedocs.io/en/latest/.

AXI DMA LogiCORE IP Product Guide (PG021) (2022). Xilinx, Inc. URL: https://docs.xilinx.com/r/en-US/pg021_axi_dma.

AXI Virtual FIFO Controller v2.0 Product Guide (PG038) (2015). Xilinx, Inc. URL: https://docs.xilinx.com/v/u/en-US/pg038_axi_vfifo_ctrl.

Brette, R. and W. Gerstner (2005). "Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity." In: *J. Neurophysiol.* 94, pp. 3637–3642. DOI: 10.1152/jn.00686.2005.

Delta V Software (2020). "Remote Call Framework." In: URL: www.deltavsoft.com.

"IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language" (2018). In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

"IEEE Standard for Universal Verification Methodology Language Reference Manual" (2020). In: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

"IEEE Standard VHDL Language Reference Manual - Procedural Language Application Interface" (2007). In: *IEEE Std 1076c-2007 (Amendment to IEEE Std 1076-2002)*, pp. 1–212. DOI: 10.1109/IEEESTD.2007.4299594.

Ilmberger, Joscha (2023). personal communication.

JEDEC (2012). *DDR3 SDRAM SPECIFICATION*. Tech. rep. URL: https://www.jedec.org/standards-documents/docs/jesd-79-3f.

Karasenko, Vitali (2014). *A communication infrastructure for a neuromorphic system*. Master's thesis (English), University of Heidelberg.

– (2020). "Von Neumann bottlenecks in non-von Neumann computing architectures." PhD thesis. Universität Heidelberg. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/28691/1/KarasenkoPhD.pdf.

Mayr, Christian, Sebastian Hoeppner, and Steve Furber (2019). "Spinnaker 2: A 10 million core processor system for brain simulation and machine learning." In: *arXiv preprint arXiv:1911.02385*.

Micron (2015). *DDR3 SDRAM Verilog Model*. personal communication. URL: https://www.micron.com/products/dram/ddr3-sdram/part-catalog/mt41k128m16jt-107-it.

Müller, Eric, Elias Arnold, et al. (2022). "A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware." In: *Front. Neurosci.* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.884128.

Müller, Eric, Christian Mauch, et al. (Mar. 2020). *Extending BrainScaleS OS for BrainScaleS-2*. Tech. rep. Heidelberg, Germany: Electronic Vision(s), Kirchhoff Institute for Physics, Heidelberg University, Germany. DOI: 2003.13750.

Pehle, Christian et al. (2022). "The BrainScaleS-2 Accelerated Neuromorphic System with Hybrid Plasticity." In: *Frontiers in Neuroscience* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. arXiv: 2201.11063 [cs.NE]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876.

Rettig, Marco (2019). "Characterizing the event interface of the hicann-x." In.

SmartConnect v1.0 LogiCORE IP Product Guide (2022). Xilinx, Inc. URL: https://docs.xilinx.com/r/en-US/pg247-smartconnect.

Spilger, Philipp et al. (2018). *flange*. URL: https://github.com/electronicvisions/flange.

Stradmann, Yannik et al. (2022). "Demonstrating Analog Inference on the BrainScaleS-2 Mobile System." In: *IEEE Open Journal of Circuits and Systems* 3, pp. 252–262. DOI: 10.1109/OJCAS.2022.3208413.

A. Code environment

Table 6 lists the commit hashes that identify the version of the different software and FPGA components used throughout this thesis. For some components additional changes that are not yet merged into the components were used. This includes many of the changes developed in this thesis. Table 7 lists the `Change-ID`s of these changes. The code belonging to these changes can be found on gerrit (`https://gerrit.bioai.eu/`).

repository	commit
ayo	29abb7761e18a22b0aa3b8f06f899ff3d932e4b5
bss-hw-params	8e5e18ebbdece63890eebd4f082084111846e965
fisch	1215ebcdddd123ec4b373dec6e85996c8eedcd94
flange	2bd1631ca1a2608fedb84b5de7586c9b68d2d141
halco	8fcef09d20b45f00ecc9d7cdbbcc6fcb1cf847aa
haldls	8fcef09d20b45f00ecc9d7cdbbcc6fcb1cf847aa
hate	fe1c7f35924b4a96f64c88cfb2fd3c0126ed3e48
hwdb	0143bd6e177cdc2ec1f49e9276884078f3976b22
hxcomm	680bdcfd6678e5a561b0ba884d65dcde571523ff
lib-boost-patches	ed89665b4c066629b69617ede2e8b1fbe65822d9
lib-rcf	8f928103ada425be321f33eb599e93eebabd81f4
libnux	921beaebd23936751b883ba579c1c56fabce5485
logger	87c2b33573dd63141861a6ab96a4a5d0de145b42
pywrap	8eda91fcca8bfccb946a0ee5b40ca82b5b15650e
rant	0d494ce6eedfb74889cf7cee09105258819acb35
sctrltp	42a988e986906f177102813418d5fd22dd646b44
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3
ztl	773660f435e56b1ee7b962e8babfe004ff487cdd
hicann-dls-private	57e29fa5cfb94063e8be55835852625c2a838265
hmf-fpga	a8bf14759ff62d72bef4dacd09f3510228e742c0
hmf-fpga-test	80b8fc93498557722344d1f164d95e84168b9a88
hxfpga	69020247821e557e4c95fc806338d225df317919
lib-extoll-utils	a529b24a84d9776bcc6b39ad65985999ea9eb3bc
lib-vhdl-utils	7e23071e310946398ecc6bf4f1764886ac276e2e
s2pp	84494c3ab040962cf77a4b5e216a73c24fb188e1
verilog-i2c	ad61cd1b90cb60d0776fbc2f4d8fe5f81f28c437
verilog-uart	d7e9f305e1d12bdaebade995997834d2555f5918
visionary-rtl-utils	88d877f9ff8192be5bad89cc4eff7f631dfeeb8b

Table 6: List of commit hashes identifying the version of the software and FPGA repositories.

repository	additional changes
ayo	Ie5012d8b575a63ee97225e3c8dcec0787d6ebc2f
	I179bcc7f516872180c18c4cb805138209f685a84
	Ic641d3bab02a69462b89c37889f91145134aa556
flange	If7780dd27375d4b073a63c3bc34cfd87ac23772f
halco	I129931f8010b9c20dc87df361d393acfb1656785
haldls	I129931f8010b9c20dc87df361d393acfb1656785
hxcomm	Id5251a53698b4bf0a0a89c59971ae42c0b816caf
	I13bf5649f7adaa34058cc2b55c0db1449cdceb66
	I510a72e39edd2a3e05d315ebbe731deded89a763
hxfpga	Ic31ed629ff7d92a59053755ba3c4b6125694b9e0
	If92ea7f519299acf514667da6d244ebaf6f976ce
	I87807ac4d497eba4a2ea0ae80aa63ba9688985cf
	Ia29c666d88eb26dc20a741a14074a15063da4e1f
	I62baf937219012118672801 0dc6c714d4394ac7b
	Ic46ab7a4ad97ff006a5ab9606b78fce3c7aa8dc9
lib-vhdl-utils	Ida2393ab5111a47f1710ce451a54590322c842d4
	I2a8b1a926333e8221a5e86a4f51b96b4443c822d
	I040920155d9b651ea7796111a68fa23f3ce13449
	I472248e52f4a16841e2227626cf8d12dcba0826a
visionary-rtl-utils	Id9f35205596144fa719b223f6bb2dfa37abf60b9
	I68a653809a4632b00f1cd9497a5917f1ec0b979f
	Ica56e53d44b73177580d00cc869fee709c4858c5
	I56ecff5352246ca7026e3b9a3d205dac151ec069

Table 7: List of Change-Ids for the changes not yet merged into main repositories. They can be found on gerrit (`https://gerrit.bioai.eu/`)

Acknowledgements

I would like to thank

- Dr. Johannes Schemmel for the opportunity to write my thesis within the Electronic Vision(s) group.
- Joscha Ilmberger for the great support since my internship.
- Yannik Stradmann, Joscha Ilmberger, Eric Müller and Christian Mauch for great discussions and insights.
- Everybody who helped with proofreading this thesis: Joscha Ilmberger, Yannik Stradmann, Christian Mauch, Jaro Habiger, Anna Klingauf and Dirk Heinemann.
- The Electronic Vision(s) group for providing a great place to work.
- My friends and family for their support during my studies.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 3. März 2023,