DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimization of a real-time capable Checkpoint/Restore mechanism for L4 Fiasco.OC/Genode by hardware-assisted memory tracing and copying

Sebastian Bachmaier

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimization of a real-time capable Checkpoint/Restore mechanism for L4 Fiasco.OC/Genode by hardware-assisted memory tracing and copying

Optimierung eines echtzeitfähigen Checkpoint/Restore Mechanismus für L4 Fiasco.OC/Genode durch hardwaregestütztes Memory Tracing und Copying

Author:	Sebastian Bachmaier
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl, M.Sc.
Submission Date:	15.01.2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.01.2019 Sebastian Bachmaier

Acknowledgments

Sincere thanks are due to Prof. Dr. Baumgarten for the possibility to write this thesis at the Chair of Operating Systems and Sebastian Eckl for his continuous support and motivation.

I thank further David Werner and Alexander Weidinger for their professional advice and explanations regarding mechanisms in the Genode operating system and Matthias Adams for his great expertise in Genode driver development which was highly appreciated.

Abstract

Checkpoint/Restore mechanisms for real-time capable operating systems are applicable for a wide variety of uses cases. Some of these systems run performance critical applications. Recent integration of a checkpoint/restore mechanism developed by Huber and Stark utilize specific elements of the operating system (such as the Page Fault mechanism of the Memory Management Unit) to keep track of memory changes. The increased software overhead of this approach by the mechanism can be countered by developing a FPGA based component. Following the ideas of W. Li et al. and developing an AXI based peripheral which is utilized as wrapper around the main memory, memory writes can be traced and processed. With the help of a FIFO buffered write mechanism memory writes can be copied to specific backup locations. Copying can be paused so checkpoints can be stored in a consistent state without interrupting other components. An integration into L4 Fiasco.OC / Genode as a driver component allows the use in Genode components. Even tough having significant drawbacks such as performance issues due to AXI bus write speed and disabled caching, it can be shown, that an integration into the existing checkpoint/restore component is possible such that the resulting component finds application in a wide variety of domains such as cloud computing or connected cars.

Contents

1. Introduction

Real-time based operating systems have a wide variety of use cases. [PS01][Lee08][SSK92] Its purpose is, in contrast to ordinary operating systems, to process requests within a certain limit of execution time which has been determined before the execution[Fur+12]. Such a real-time system can be the combination of the L4 Fiasco.OC microkernel and the Genode operating systems framework.[Fes15]

To guarantee the integrity of critical processes it is furthermore desired to checkpoint processes during the execution. System errors, hardware failures or a simple loss of power can lead to failures in execution of programs. Such programs have to be restored afterwards from a checkpoint stored previously.[MME04][LNP90] One vital part of the checkpoint/restore process is the tracing of memory accesses and their replica in form of a copy to store specific memory snapshots.[Wer16]

Multiple approaches have been done implementing checkpoint/restore mechanisms for *L4 Fiasco.OC/Genode*. These however were all based on software approaches that exploit and extend mechanisms of the operating system.[Sch17][Hub16][Jos18]

New applications for hardware extensions like Hardware transactional memory (HTM)[LKN14], Floating-point-gate-arrays (FPGAs)[Geh+16], and peripherals with direct memory access [Xil15] allow a combination of software and newly developed hardware features. [Dic+09] Additionally other work suggests the use of FPGA for the acceleration of the migration in cloud architectures [FVS15] showing already early approaches for hardware accelerated memory checkpointing. [Dar+15].

This thesis investigates whether hardware-assisted memory management adds advantageous additions like improved parallelism, fixed real-time constrains and reduced software complexity to checkpoint/restore techniques and whether it is actually capable to extend a real-time based checkpoint/restore mechanisms on *L4 Fiasco.OC/Genode* via implementing memory tracing and copying. Conclusively is it supposed to show that hardware assisted memory tracing and copying can cope with bottlenecks and constraints (like blocking concurrency and slow memory tracing) which was imposed by software restrictions. Additionally it shall be shown how a developed FPGA based hardware assisted memory concept compares to other approaches (like multi-core supported tracing).

The thesis is structured the following way:

The second chapter presents the theoretical background to a variety of concepts and technologies which are necessary to understand the following implementations. Then, recent developments in the domain of real-time checkpoint/restore mechanisms and hardware assisted memory tracing are explained, this thesis is based on. Subsequently the requirements of real-time based checkpoint restore mechanisms are stated and are put into contrast with multiple approaches and concepts. The fifth chapter shows the motivation for the development of such a component and develops one concept. Ensuing it is shown how an actual implementation does work. It is furthermore analyzed by a working example and resulting real-time metrics, such as timing constraints and performance measurements as well as measurements in performance to the original system. Chapter 7 discusses whether the concepts suffices all necessary requirements or which further steps will be needed to achieve a complete goal completed by limitations which arose. Afterwards future improvements are suggested complementing the developed research followed by lastly summarizing the key concepts and results which have been developed and an indication of its importance in a context.

2. Theoretical Background

This chapter serves as a summary to all necessary theoretical concepts and technological backgrounds the thesis is based on.

2.1. L4 Fiasco.OC / Genode

Genode OS[Fes15] is an operating systems framework, which can be combined with Fiasco.OC[1] as a kernel fulfill real-time requirements. Genode supports to be used on a wide variety of kernel such as Linux or the Fiasco.OC kernel (as used in this thesis). One specialty of Genode is its so called capability system. Capabilities are rights granted by a parent component running on the Genode system. They represent physical limitations like RAM and CPU time. and can be accesses by creating capability sessions which can then be passed onto a child component (child process instantiated by a parent process). [Rei18][Fes15].

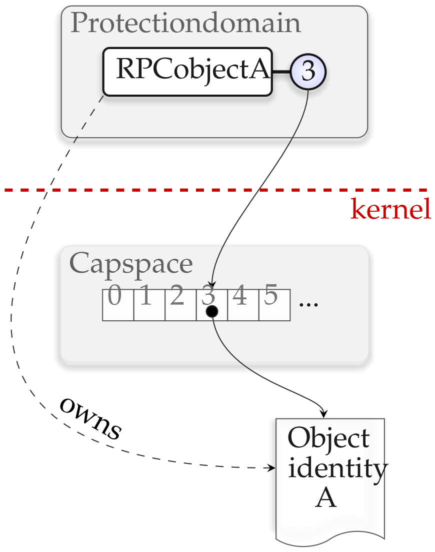


Figure 2.1.: RPC entrypoint of a Genode component[Fes15, P.37]

[1]see `https://os.inf.tu-dresden.de/fiasco/`

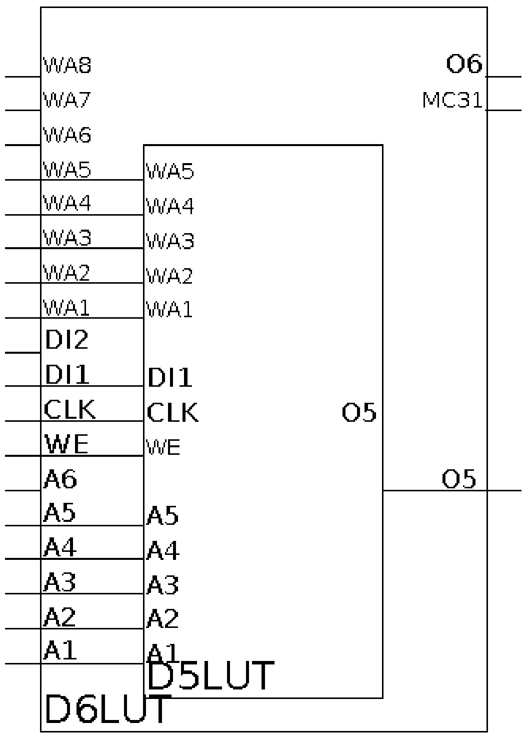
To provide service to other components a Genode component provides a so called RPC entrypoint (see Figure 2.1). This entrypoint can be accesses via connecting over a capability located in the kernel space. Therefore all communication with the component will be executed via a context switch[Tan09] to the kernel and again to the driver. Therefore the kernel can verify the integrity of allocated resources so no restricted resources can be obtained without proper rights management. [Fes15]

2.2. FPGAs

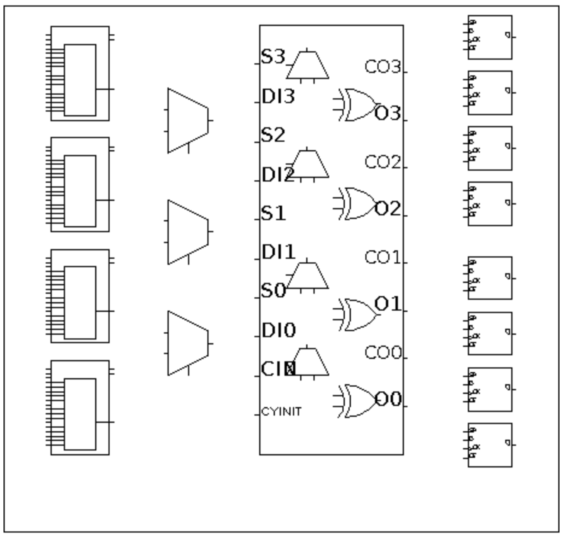
A Field Programmable Gate Array (FPGA) is a re-programmable integrated circuit. In comparison to standard, application-specific circuits (ASICs), a FPGA is not bound to a predefined digital logic function but can be re-programmed after it has been built.[Geh+16]

Due to being more flexible than ASICs and in addition being re-programmable, FPGAs are mostly used for prototyping and demonstration purposes, but can also be applied in domains where quick adaptation but also fast processing is necessary [HBB04].

Their main disadvantage is their cost, due to being not as densely packed in the sense of implementable logic compared to ASICs[Ham+09]. This leads to a small reduction in switching speed as well [KR07].



(a) Xilinx D6LUT cell



(b) Xilinx Zynq-7000 function block

Figure 2.2.: A Xilinx D6LUT cell (in 2.2a) and a function block (in 2.2b) as taken from Xilinx Vivado 2018.2. The Xilinx D6LUT cell shows the input A1 To A5 as well as the clock (CLK). Additionally it consists of programming input WA1 to WA8. Additionally it can be seen that a Xilinx D6LUT cell is internally reduced to a Xilinx D5LUT cell. The function block shows four Xilinx D6LUT (on the left) connected with multiplexers to interconnect their output and redirect it properly (second from left) following interconnection logic to combine the Xilinx D6LUT's output (second from right) as well as registers and their outputs (on the right).[Xil16] [2]

[2]Screenshot from the *Device* panel in Xilinx Viado 2018.2 [Xil19]

To be re-programmable and implement a component that manifests a logical function FPGAs make use of several parts:

1. Function Blocks consisting of logic cells
2. Interconnects
3. IO and specialized hardware

Function Blocks
Function Blocks (as seen in Figure 2.2b) are separated logic consisting of several logic cells. They are in most FPGA architectures implemented as a combination of binary lookup tables and Flip-Flops (as seen in Figure 2.2a). The look-up tables are from different degrees (depending on the FPGAs architecture) and implement N:1 functions (called LUT-N cells, with N being a positive but fixed integer). Modern FPGA designs make use from LUT-2 up to LUT-6 cells. [Geh+16, P. 273] [Xil16]

The actual programming of the FPGA happens with the instantiation of the lookup table's function. A digital logic function (implemented in a hardware description language like VHDL or Verilog), is transformed from its behavioral structure to a synthesized logic function. This logic function is then implemented via a placing algorithm which sets the correct definitions for the lookup tables. Registers like signals in VHDL can be mapped to Flip-Flips positioned in the Function Blocks. Due to a maximum amount of Function Blocks not every digital logic function can be implemented on every FPGA. [Geh+16, P. 273]

Interconnects
Function blocks are interconnected using programmable switches in a gate matrix. Using a programmable multiplexer, outgoing signals of the function block can be rerouted to the next block. Therefore a digital logic function can span over multiple function blocks. [Geh+16, P. 275]

IO
Additionally, the FPGA has to be connected to other parts of the logic board it is used on. Those can either be indicating LEDs, sensors, switches, screens or other peripherals, but also ASICs containing integrated processing units themselves. [Geh+16, P. 274]

2.3. Xilinx Zynq 7000 & Xilinx Development Platform

Zynq-7000
Zynq-7000 is a development platform created by Xilinx. It combines a Cortex-A9

processing platform with a programmable FPGA from the Xilinx Kintex®-7 or Artix-7 platform. It allows to run standard ARM programs on the processors, while communicating with user defined interfaces with the FPGA components.[Xil18][Inc19e]

Available boards that support the Zynq-7000 platform and are interesting for this thesis are the following:

1. Zybo Zynq-7000 [Inc19b]
2. ZedBoard Zynq-7000 [Inc19a]

Distinguishable from other Xilinx FPGA products does the Zynq-7000 family focus mostly on the interplay between software and the FPGA in comparison to FPGA only products.[Inc19e]

Xilinx Development Platform

The Xilinx development platform is mostly used in combination with the Viado Design Suite. The main application of the Vivado Design Suite, is Vivado itself, which is used for the implementation of the FPGA design.[Xil19][Inc19e] Different modules imported from vendor libraries can be combined. Those modules can be connected and are called IP (Intellectual Property). Each IP is one well defined hardware module written by a specific organization or company. The company can then specify the parameters as VHDL generics that can be modified by the IP user. The IPs are mainly written in VHDL or Verilog.[Inc19d] After a design has been produced, Vivado can then be used to produce a so called Bitstream file which contains all necessary data to be programmed to the FPGA.[Inc19c] The conversion of a design to a Bitstream file follows the following steps (as seen in [Cor09][Inc17]):

1. Hardware synthesizing: converts the VHDL code into a digital logical function
2. Implementation: converts the digital logical function into the FPGA design. This contains placement of the LUT cells and Function Blocks as well as the programming of the interconnects (compare to chapter 2.2).
3. Bitstream generation: The implemented design gets serialized and converted into a Bitstream file which can be used then on the Zynq-7000 platform.

2.4. AMBA 4 AXI Protocol

The *Advanced Microcontroller Bus Architecture 4* (AMBA 4) is a memory access bus design specification. The related protocol called *Advanced eXtensible Interface* (AXI) is the digital signal protocol used for communication on the AMBA 4 bus design, therefore abbreviated AXI-4, or subsequently just AXI.[ARM11]

Each peripheral of a system is connected to all other peripheral over so called AXI-Interconnects.[Xil17a] Every peripheral is responsible for a specific range in the main memory space. The AXI-Interconnects route intelligently memory write and read accesses to the correct peripheral which has to respond accordingly. The AXI protocol handles the way reads and writes are sent and confirmed. The communication is thereby always done between a AXI-master which initiates the memory writes and reads and a connected AXI-slave which responds. Peripherals which consists of only AXI-slaves therefore never initiate memory transactions themselves, and in the same way AXI-master-only devices never respond to memory transactions and are therefore not memory mapped. [ARM11]

2.4.1. AXI4 Types

AXI comes in three forms[Xil11]:

1. AXI-Lite
2. AXI-Full
3. AXI-Stream

All 3 variants are perfectly capable of sending information between peripherals. Their difference is mainly in the grouping (and therefore the speed) of memory transactions.

AXI-Lite doesn't allow grouping. Every transaction is initialized and confirmed exclusively.[ARM11, P. 91][Xil11]

AXI-Full introduces an additional burst mode. This burst mode allows to initialize one starting memory address and a burst size. The AXI-master writes after a handshake on every clock cycle new data. That data gets then written to the addresses starting with the address specified in the handshake. It gets followed by iteratively increasing addresses for its whole specified size, until the burst stops. Therefore AXI-Full allows to write one memory bus width per clock cycles once the transaction handshake has been finished.[ARM11, P. 20][Xil11]

AXI-Stream is limited in its capabilities in the sense that it's unidirectional. Supporting therefore writes only. Its advantages are very long burst lengths leading to increased performance [AMB4][Xil11]

2.4.2. AXI-Channels

AXI-communication uses, apart of the global clock and reset signal and optional low power mode signal, 5 different signal types which can be grouped as the following[ARM11, P. 27]:

1. Write address channel: Used to transfer the write address of a transaction from master to slave
2. Write data channel: Used to transfer data for a transaction from master to slave
3. Write response channel: Used to confirm whether and when writes have been accomplished successfully
4. Read address channel: Used to transfer the address for a read transaction
5. Read data channel signals: Used to transfer the data as a response to a read demand

These channels are logical channels. They themselves consist of multiple signals. The type of signals used, depend on the type of AXI-protocol used (AXI-Lite, AXI-Stream, AXI-Full). Additionally, they are always dependent on the width of the address bus as well as the width of the data bus due to AXI not being bound to a specific architecture and therefore bus width.

For AXI-Lite, the signals used are the following:

Table 2.1.: List of all *Write address channel* signals for AXI-Lite [ARM11, P. 29]

Signal	Source	Description
AWADDR	Master	Address of memory to write to
AWVALID	Master	Write address has been set by Master
AWREADY	Slave	Slave confirms that address has been set by Master

Table 2.2.: List of all *Write data channel signals* for AXI-Lite [ARM11, P. 30]

Signal	Source	Description
WDATA	Master	Data to write
WSTRB	Master	Byte masking of valid bytes to write
WVALID	Master	Data has been set by Master
WREADY	Slave	Slave confirms that data has been set by Master

Table 2.3.: List of all *Write response channel* signals for AXI-Lite [ARM11, P. 31]

Signal	Source	Description
BRESP	Slave	The transactions write status. "00" indicates a successful write
BVALID	Slave	Slave confirms whether write has been successful
BREADY	Master	Master confirms the Slave's confirmation

Table 2.4.: List of all *Read address channel signals* for AXI-Lite [ARM11, P. 32]

Signal	Source	Description
ARADDR	Master	Address the master wants to read
ARPROT	Master	Master sets read flag like (exclusive access). "000" is normal read
ARVALID	Master	Read address has been set by Master
ARREADY	Slave	Slave confirms that data has been set by Master

Table 2.5.: List of all *Read data channel signals* for AXI-Lite [ARM11, P. 33]

Signal	Source	Description
RDATA	Slave	Read response from the Slave
RRESP	Slave	The read transaction's status. "00" indicates a successful read
RVALID	Slave	Read response from Slave has been set
RREADY	Master	Master confirms that response has been set by Slave

Each and every signal has a specific task in the write/read process of the AXI-Lite access, as indicated in their description. The two processes can be understood as state machines for the slave and master (see Figure 2.4 for write process and 2.3 for read process). Every action done by the other communication partner, except the actual write / read itself, pushes the machine forward in execution. This leads to the disadvantage, that a malicious partner or one, that is not correctly working, can be responsible for blocking the bus if no precautions are made. [ARM11, P. 4]

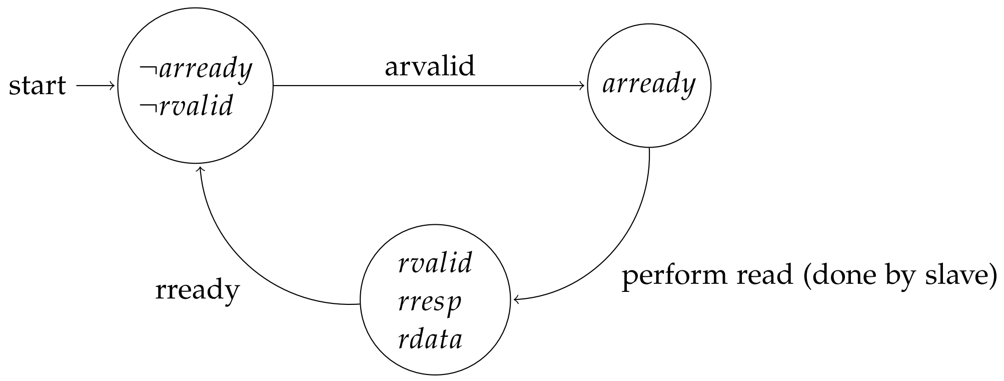


Figure 2.3.: AXI-Lite read process. From the slave's viewpoint. State transitions are initiated from the master setting the corresponding states (if not indicated otherwise). Once the connected master triggers an action the slave goes to the appropriate state and response with the corresponding variable (e.g awready for awvalid).

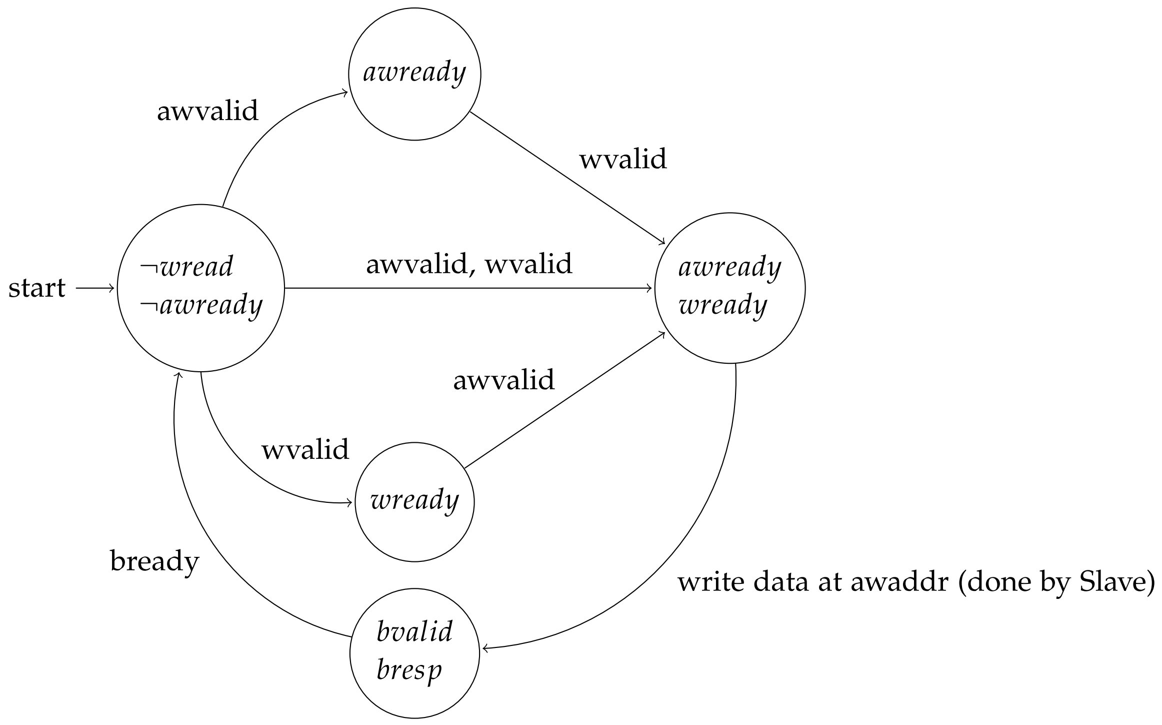


Figure 2.4.: AXI-Lite writing process. A memory write from the slave's viewpoint. State transitions are initiated from the master setting the corresponding states (if not indicated otherwise). Once the connected master triggers an action the slave goes to the appropriate state and responds with the corresponding variable (e.g awready for awvalid).

3. Related Work

This chapter shows previous research this work is based on as well as other approaches. It focuses on implementations for *L4 Fiasco.OC/Genode* but also tangles recent general improvements for the use of hardware assisted memory tracing and copying.

3.1. General Checkpoint/Restore Mechanisms

Early approaches for checkpoint/restore mechanisms where mostly developed to support process migration.[LNP90]

Nevertheless have further applications (like program debugging) been found which served as a use case for checkpoint/restore implementations[Wit88].[Sch17] This resulted ultimately in the development of stand-alone Linux components and programming libraries such as CRIU[CRI18][Wer16] and libckpt [Pla+94] Further use cases have been found in the field of cloud computing[Ros14] as well as embedded systems [Kra+09].

3.2. Checkpoint/Restore Mechanism in L4 Fiasco.OC/Genode

The integration of this work is placed in context with previous implementations that have been done[Wer16][Hub16][Jos18] for L4 Fasico.OC/Genode as part of the *ArgOs Research* project[1] . They analyzed multiple approaches and achieved several types of software implementations for a checkpoint/restore mechanism as part of the ongoing project.

3.2.1. Checkpoint/Restore via Shared Memory

In a first draft Werner[Wer16] showed that a proposed migration of an existing checkpoint/restore mechanism[CRI18] is capable of coping with required real time limits. Therefore they proposed to implement a shared memory approach. The program responsible for checkpointing shares a Genode dataspace with the component which is supposed to be checkpointed (subsequently called target). As soon as a checkpoint

[1]see `https://argos-research.github.io/`

is supposed to take place the target gets halted, the checkpoint mechanism copies the main memory of the target and then resumes the target's normal execution. Huber [Hub16] showed that an actual implementation can be realized (called *RTCR*) but leads to disadvantages concerning the real-time requirements due to the paused execution of the target during the checkpoint's creation.

3.2.2. Checkpoint via Incremental Multi-Core Support

Additionally to the shared memory approach which checkpoints whole memory areas Schweigert[Sch17] developed an incremental memory checkpointing mechanism with multicore support. Their ideas are based on previous studies comparing different ideas of checkpointing.[Vas+11][Gio+05][Vog+15] They discuss the use of indicating bits which are set by the Memory Management System for memory pages. Such flags/bits are the following (see [Sch17, P.12]):

1. Write Bit: Which can be set to trigger a page fault as soon as data is written to a memory page
2. Dirty Bit: An indication set by the Memory Management Unit to indicate that a memory page has been modified

Schweigert indicated that those techniques are used in the Linux operating system's *fork* mechanism[Sch17, P.13] such that a *Copy-on-write* mechanism can be implemented using the *Dirty bit*.

They then further implemented in the existing *L4 Fiasco.OC/Genode* mechanism the possibility to trigger events every time a dataspace gets newly accesses since the last checkpoint. This can then be traced and stored (similar to the *Dirty Bit*) as part of an incremental checkpoint. Therefore the RTCR component can keep track of all dataspaces accessed by the component which is supposed to be checkpointed.[Sch17, P.27]

3.2.3. Hijacking System Functionalities

In his thesis "Development of a Redundant Memory Based Checkpoint/Restore Mechanism for *L4 Fiasco.OC/Genode*" Stark [Jos18] showed, that several functionalities have the capability of aiding in a checkpoint/restore mechanism.

Tracing via Hardware and Software Watchpoints

First of all they discussed whether hardware or software watchpoints can be used to trace memory modifications. The idea was to launch a system interrupt when write accesses are performed with the help of setting a hardware watchpoint at specific locations. It could be shown that the amount of hardware watchpoints available does not suffice the amount a parallel points needed. Additionally the use of software watchpoints was discussed but the idea was quickly dropped due to them not being supported by *L4 Fiasco.OC/Genode* [Jos18, P.27].

Checkpoint via Redundant Memory

Ultimately implemented was the approach of a redundant memory based checkpoint/restore mechanism. It does not halt the component but copies the memory directly into a second backup area, which happens on each write access.

The backup process can copy the backup area periodically. New memory accesses can happen at the same time. The targets accesses will be relocated to another memory backup space. Different memory backup spaces can be differentially combined to a new complete memory snapshot. The main hurdle was the *On Memory Copy* which has been implemented by triggering page faults on each memory access and modifying the page fault mechanism. [Jos18, P.25 - 72]. Stark shows that such an implementation leads to the desired outcome but strong penalties in performance where assumed [Jos18, P. 80].

3.3. Hardware Based Memory Tracing

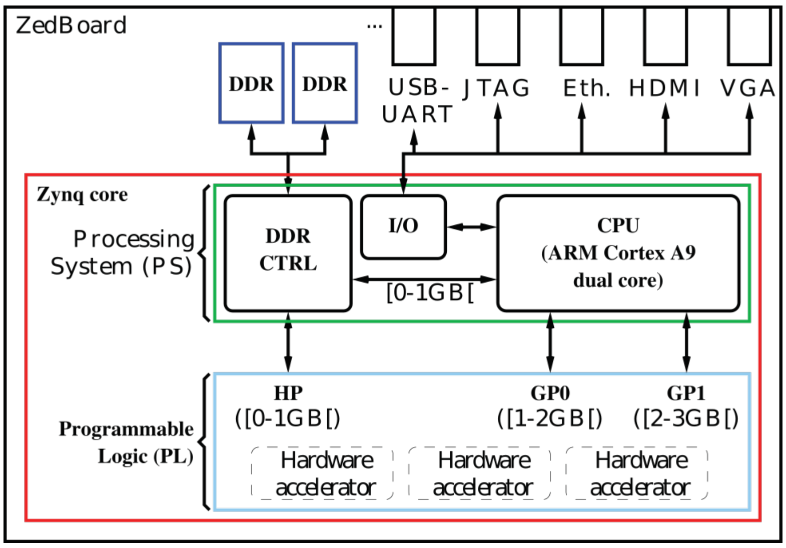


Figure 3.1.: The memory redirection done by W. Li et al.[LDP15]. Implementing a memory bridge that redirects memory accesses over the Programmable Logic and back to the Zynq processing system. They were able to implement memory tracing via a FPGA based Zedboard.[LDP15, P.463]

Substantial work in achieving hardware supported memory tracing has been done by W. Li et al.[LDP15] Using a Zynq-7000 based Xilinx Zedboard, a FPGA peripheral based memory bridge (see Figure3.1) was used to indicate and store read and write accesses by a memory monitor storing accesses in a FIFO queue.

They were able to show, that Linux can be booted using the synthesized hardware. Also they were able to implement, that memory accesses can be traced. W. Li et al. showed, even though having certain drawback (disabled L1, L2 cache, FIFO queue capacity limitations) that memory accesses can be achieved this way.

This thesis is mainly based on this idea, transferring and extending this concept.

3.4. Hardware Based Memory Copying and Multi-Versioning

Concerning copying and multiversion of memory via hardware, recent new approaches implemented by Intel and its Hardware Transactional Memory (HTM) have been made. Especially for the use of atomic transaction and databases it could be shown, that parallel programs can synchronize the memory state while accessing data. Memory can be copied and accessed in parallel in different memory ranges without having inconsistent memory ranges.[LKN14]

3.5. Xilinx Direct Memory Access Component

Xilinx developed an integral component available to be used for the Zynq-7000 platform called the *Xilinx DMA IP*[Xil15]. On a Zynq-7000 based system, it allows to bypass the CPU and other memory management components to directly access the the main memory from FPGA components. Therefore it can also be used to quickly copy memory spaces in the main memory. Being independent on the CPU and processing system however, tracing of memory accesses seem to not be able with the *Xilinx DMA IP*.

4. Conception & Motivation

4.1. Motivation

As seen in Chapter 3.2.3 current checkpoint/restore mechanisms in *L4 Fiasco.OC/Genode* have serious bottlenecks regarding parallelism during copying and tracing overhead (compare the page fault mechanism of Stark[Jos18]).

It is desired to replace features like the page fault on write mechanism with a hardware solution that traces memory writes without leading to further performance degradation.

Additionally the work of Schweigert[Sch17] can be enhanced and supported significantly if a hardware assisted solution is capable of detecting memory changes that can be incrementally combined, if tracing of memory changes can be detected on a more granular level, that allows to cope with the checkpointing of writes with less overhead.

Furthermore can the mechanism of Huber [Hub16] be enhanced via having a true parallel checkpointing mechanism instead of a concurrent one.

4.2. Requirements

A component is supposed to be developed improving on those ideas. It is required to trace memory changes as well as to prepare the memory for a serialization into a checkpoint.

Therefore several hurdles arise for a real time based checkpoint/restore mechanism:

1. Memory must be copied while being in a consistent state without blocking access to the dataspace or locking them while being used by the program to be checkpointed. Therefore the checkpointing shall be non interrupting.
2. Memory accesses shall be traced, monitored and accessed by an external component.
3. A checkpointing component needs to be able to control the flow of which memory areas are traced and where a potential copying is processed.
4. A reliable real-time metric needs to be definable, meaning that the duration of actions performed by the tracing and copying needs to be deterministic and finite.

4.3. Approaches

Based on those previous requirements the checkpointing component is based on the ideas in the following chapters.

4.3.1. Principal Concept

Imitating mainly the bridge mechanism of W. Li et al.[LDP15] a AXI-bridge shall be used to redirect all memory accesses done by the operating system to a FPGA based peripheral. The peripheral will equally redirect the memory accesses back to the Zynq processing system so they can be forwarded to the main memory.

Furthermore shall the AXI peripheral copy the memory access to another memory area which is supposed to be configurable while the system is running. Also must the AXI peripheral be integrated seaming-less into *L4 Fiasco.OC/Genode* . It is advantageous if Genode dataspaces can be directly mapped onto each other.

Additionally to the already existing work, tracing is not supposed to be done on the memory access level but on arbitrary level, therefore indicating which pages have been modified and which have not.

4.3.2. Memory Tracing

A checkpoint/restore mechanism, that is not supposed to checkpoint all memory spaces every time a checkpoint is created, needs to trace which memory changes have been done.

A distinction can be done between three types of granularities whose changes can be detected:

1. Transaction Wise (for every memory transaction)
2. Page Wise (for every virtual / physical page)
3. Compile time fix but arbitrary

Due to not knowing which type of application the checkpoint / restore mechanism is used in general, it is advantageous to implement arbitrary (but fix) granularity if possible.

Furthermore the memory accesses need to be stored. There are two possible options regarding the type of data structure which can be implemented

1. Binary Flags: for every type of memory area for which it is supposed to be indicated whether it has been modified, a bit flag could be set.

2. List of modified memory spaces: A list could be implemented (as a FIFO queue, map etc.) that stores all memory areas that have been modified

For this concept the second option seems to be the better solution. The data structure in option one needs one bit for every element that the system needs to survey. Due to space on an FPGA being limited, this does not seem to be feasible to implement. The second solution can be implemented using a map (similar to a hashmap) that stores keys and bits inside the data structure. As soon as it needs to be checked whether a memory page has been modified, the key can be probed and verified whether the key has been inserted and the bit is set to 1. In the case, that memory spaces which have been modified need to be returned, the content of the map can be used.

4.3.3. Memory Copying

Additionally to tracing memory changes, it is important to pre-process the memory for accesses via the checkpointing process as well. When the checkpoint process has been launched, the responsible program needs a specific amount of time to copy memory pages into the backup space. Forcefully, the content of a memory page has to be consistent.

It is not possible, that a page is modified by its program while being saved. Either memory accesses are treated exclusively, using locks or semaphores[TW87], or they are available in multiple versions. Locks (as implemented by [Hub16]) are disadvantageous to a real-time system as programs need to hold specific constraints regarding process limits and can not be stopped during their execution, which would necessarily happen. Therefore multi-versioning is a viable solution.

Both accessing programs (the page's owner as well as the checkpointer) could theoretically access Read (R) as well as Write (W) operations on the page. The checkpointer however can refrain from processing writing accesses to those pages because they simply need to be stored. Therefore Write-Write conflicts[WV01] do not need to be modeled.

Therefore the process can be achieved by having two versions of memory pages available. One for the page's owner to which it can write to; and a second for the checkpointer (subsequently abbreviated *Page V1* and *Page V2*). Memory accesses to the page's owner's page are transparent, meaning that they do not get modified in any way by the checkpointer. Memory accesses to the checkpointer's page however are targeted either by synchronizing with the owner's page or by read accesses from the checkpointer. To not having read write conflicts the latter ones have to be mutually exclusive.

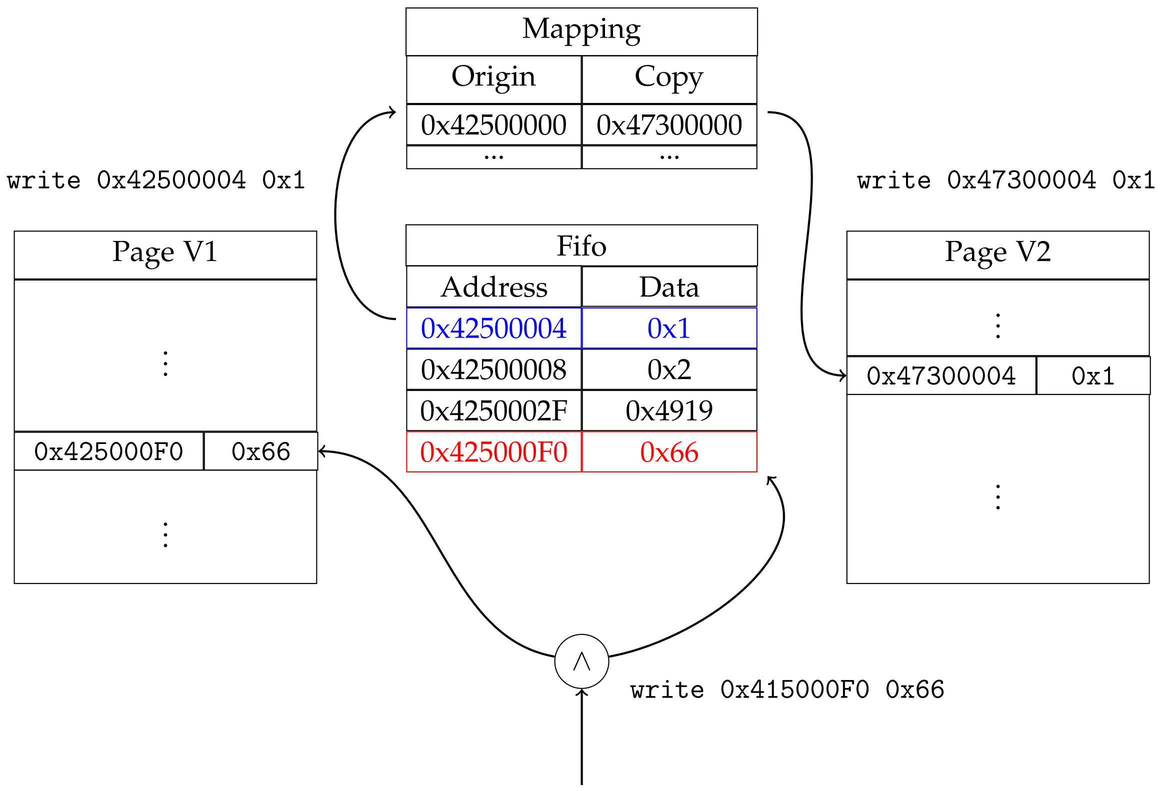


Figure 4.1.: Buffered write concept. 0x66 gets written to address *0x4250_00F0*. Therefore it is also stored in the FIFO copy queue. If write back is enabled, the memory gets written back (like in this case memory write 0x1 to the address *0x4250_0004*) to the second version of the page. The memory address needs to be transformed before being copied.

To achieve this there are two states for a page :

1. Page not accessed by checkpointer
2. Page accessed by checkpointer via read accesses

In state number one, memory accesses can easily be continuously synchronized. That means, that write accesses of the page owner's program are written to both versions of the page (in correct order).

In state number two, memory writes of the page's owner can only be written to *Page V1* (therefore keeping *Page V2*). There are two possibilities that could be done to re-synchronize *Page V1* and *Page V2* in the event of switching from state 2 to state

1. The first one is to copy the whole *Page V1* to *Page V2*. This however would lead to the same implications that posed the problem in the first place if *Page V1* is accessed via write accesses during the copy. The second solution is to buffer all incoming write accesses to a FIFO queue which is then emptied once the exclusive access to *Page V2* is uplifted when the checkpointer is done reading.

The process can be simplified by buffering a write accesses to *Page V2* into the FIFO queue and emptying it only (via write back to *Page V2*) when the machine is not in state 2 (as seen in Figure 4.1)

4.4. Other Approaches

Continuous Memory Externalization Instead of keeping a memory copy on the device itself, the bridged logic could externalize the memory accesses and writes to a second device over a peripheral interface (like Ethernet, USB, etc.). A second device connected to the target on the other end of the peripheral could write a differential checkpoint file. As soon as the restore happens, the checkpoint file gets transferred and the component can be restored by the operating system.

The main advantage of this approach is, that copying is continuous and current memory states can be saved exactly to the point the system crashes. Nevertheless, a second device is needed which raises cost for the overall system. Additionally could the external connection impose a significant drawback.

Direct DDR controller bridge The Xilinx Zynq-7000 processing system itself is connected over an IO-Port via the DDR controller. A bridge could be introduces which modifies the communication of the DDR controller itself. This could lead to higher speeds in processing, due to not having to reroute memory accesses over AXI. However, it may have certain drawbacks regarding the compatibility of different DDR technologies.

FPGA based memory management unit Supposedly the FPGA infrastructure allows for modification of the MMU a rewrite of the memory translation could lead to the desired outcome.

As soon as an address is translated from its virtual to the physical address the MMU could mark the written page as dirty. The advantages would be a seamless integration into to the memory write process. More difficult might be the integration of memory copying. This could be resolved with pages being copied when accessed during the creation of the checkpoint, similar to the Linux fork mechanism [Lov05, P.64] (as mentioned in Chapter 3.2.2). Nevertheless does it not seem to be possible to modify the MMU on current Xilinx boards.

Full FPGA based processing system Instead of developing a memory bridge between an ASIC based memory system and an FPGA, a complete FPGA based processing solution can be implemented. The processor could completely rewrite the memory management and restructure how memory is traced and copied. Supplementary can processor instructions extend known available architectures, such as RISC-V[Wat+14], by an API to modify the tracing/copying mechanism.

Virtualization based hardware modifications Similar to the approach which is supposed to be implemented, Xilinx developed a special version of the Qemu vrtualization software that can simulate the Zynq-7000 processing system on an arbitrary target computer [1]. This Qemu configuration could be changed to allow memory tracing over a modified memory management internas, also with modifying the MMU or built in memory translations. Afterwards the *L4 Fiasco.OC/Genode* system could be booted in the virtualized environment.

Using hardware transactional memory As mentioned in Chapter 3.4 is Hardware Transactional Memory (HTT) already used for multiversioning of memory transactions (as seen in [LKN14]). Such a process could be used as well to copy memory areas atomically without having to lock memory pages and therefore pause memory accessing components.

4.5. Comparison of Approaches

All approaches can be compared and their disadvantages and advantages can be shown

Memory Bridging via AXI-bridge
Advantages: A memory bridging is able to trace all memory accesses if they are rerouted. They can be modified arbitrarily in VHDL code.
Disadvantages: Every memory access needs forcefully to be redirected. Additionally, caching needs to be disabled (see [LDP15]).

[1]see `https://www.xilinx.com/video/soc/introduction-to-qemu.html`

Continuous Memory Externalisation
Advantages: It allows checkpoint to be done continuously.
Disadvantages: It increases the cost due to additional hardware.

Direct DDR controller bridge
Advantages: A direct DDR controller bridge allows the direct memory access without having to reroute to the processing system.
Disadvantages: The DDR protocol can have various types of protocols.

FPGA based memory management unit
Advantages: A FPGA based MMU can trace all memory accesses
Disadvantages: It may not be feasible due to the MMU being integrated into the CPU.

Full FPGA based approach
Advantages: A full FPGA based approach can be implemented to rewrite every aspect how memory is managed inside the CPU.
Disadvantages: A complete CPU including a processing system needs to be developed or altered.

Vitalization based hardware modifications
Advantages: Virtualization allows the modification of underlying mechanisms in the memory management internals.
Disadvantages: The Xilinx QEMU hardware simulations are not hardware accelerated. Therefore performance problems might be occurring.

Hardware transactional memory
Advantages; HTT already achieves hardware multi-versioning in some CPUs.
Disadvantages: Just a view architectures support HTT.

As seen *Memory Bridging* is the most straight forward method to be implemented. It is simpler then a DDR controller or a reprogrammed MMU. But has cost advantages over continuous memory externalization and HTT methods.

5. Implementation

This Chapter treats the general implementation [1] of the hardware and software components as well as their interplay.

5.1. General Concept

To set up the successful memory tracing and copying via hardware in *L4 Fiasco.OC/Genode* three components are being used:

1. FPGA based memory component
2. *L4 Fiasco.OC/Genode* driver
3. Driver instantitation in an example component or in the RTCR

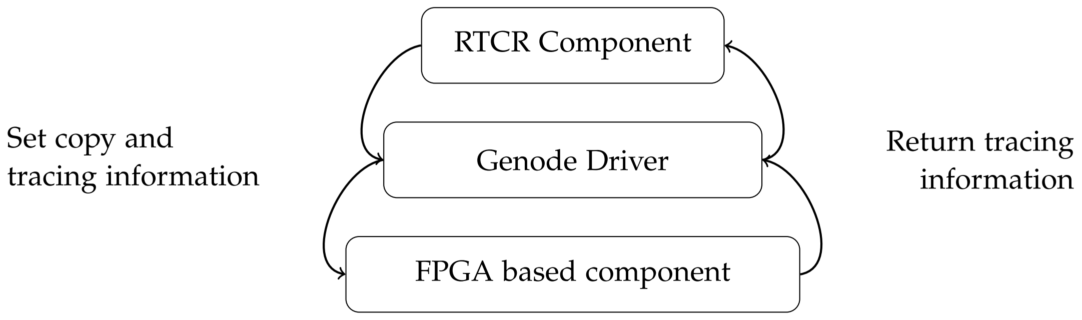


Figure 5.1.: Overview over the three system components running on the operating systems and their interplay

Every component is part of a multi-layered access hierarchy and shows a specific interplay (see Figure 5.1). The FPGA based component (furthermore called AXI-peripheral) is supposed to trace and copy the memory accesses and to expose them to the operating system. A *L4 Fiasco.OC/Genode* driver utilizes the exposed functionalities

[1]As found in `https://github.com/sebastianbachmaier/genode-HW-Memtrace`

and exposes a driver interface. A Genode component can use the driver to set up a memory translation to copy memory as well as tracing memory changes. Therefore it is necessary that the communication between the three components is well specified, which allows the exact communication and behavior.

5.2. Hardware Implementation

This section describes the implementation of the FPGA based memory component for the Digilent Zybo board[Inc19b] and the Xilinx Toolchain[Xil19].

5.2.1. Xilinx Toolchain

Xilinx Zynq-7000 systems are programmed using a specific tool chain, notably Xilinx Vivado and Xilinx SDK[Xil19].

The central component in a Zynq-7000 FPGA environment is the Zynq processing system[Xil17c]. The Zynq processing system module connects and specifies which ports are used in the communication between the integrated System-on-a-chip of the Zynq-7000 board and its communication with other peripherals (like DDR memory controller, FPGA and etc.). Voltages and a wide variety of clock frequencies can be adjusted as well.

5.2.2. AXI-Buses

The AXI-peripheral is supposed to be implemented as one VHDL component in a central IP module. To connect to the Zynq processing system, the AXI peripheral has the following four AXI-Bus ports:

Table 5.1.: All AXI-4 memory buses connected to the component

Name	Type
s_ram_axi	AXI-Lite Slave
m_ram_axi	AXI-Lite Master
m_copy_axi	AXI-Lite Master
s_contr_axi	AXI-Lite Slave

The four AXI-Buses have the following roles:

s_ram_axi : The *s_ram_axi* bus virtualises the memory address space. All memory accesses from the operating system that were destined for the main memory enter this bus.

m_ram_axi The *m_ram_axi* bus forwards memory accesses of the *s_ram_axi bus* to the main memory. Memory addresses that are inserted into this bus get transformed so they are destined for the main memory located at the Zynq processing system. It forwards and transforms reads and writes.

m_copy_axi The *m_copy_axi* bus copies the data received from *s_ram_axi* to the memory. Therefore it transforms write request from *s_ram_axi* to different memory locations and sends them back to the main memory. *m_copy_axi* never launches a read request due to it being only responsible for copying writes.

s_contr_axi The *s_contr_axi* bus is mapped to the memory space of the API connecting the FPGA component to the Genode driver. It is supposed to handle reads and writes from the API variables and its launch on specific actions.

5.2.3. VHDL Conventions

The VHDL file is developed in one single file (called *memtrace.vhd*) written in sequential VHDL code. It has one process consisting of four state machines as well as the static remapping from *s_ram_axi* to *m_ram_axi*.

Variables and signals are named after the following conventions:

Table 5.2.: Naming convention of vhdl variables and signals

Prefix	Meaning
s_ram_axi_	Part of the s_ram_axi bus
m_ram_axi_	Part of the s_ram_axi bus
m_copy_axi_	Part of the m_copy_axi bus
s_contr_axi_	Part of the s_contr_axi bus
b_	Variable of type boolean
v_	Temporary variable
t_	Composed types in the form of records or arrays
counter_clk_	Variable counting clock cycles
counter_transactions_	Variable counting transactions

5.2.4. Memory Mapping

All peripherals that implement the AXI-Bus are memory mapped. This means that a specific part of the range in the physical address range is reserved for every device. Additionally are all memory accesses that target that specific address range send to the device itself. The device is then able to respond to memory writes and reads similar to the main memory but with implementing its own read and write function.[Xil18]

Due to the physical address range being part of a 32 bit architecture the memory addresses are always in the form of *0xXXXX_XXXX*. The mapping itself is just implemented for AXI slave buses because they are the only ones which are targeted with memory accesses. AXI master buses launch memory accesses themselves. AXI periherals always start in the address range. after *0x4000_0000* on the Zynq processing system[Xil18]. The main memory is mapped in the beginning of the address range (*0x0000_0000*). Due to the main memory being 512 MiB, and *s_ram_axi* being a static remap of the main memory, the address range of *s_ram_axi* results in *0x4000_0000* to *0x5FFF_FFFF*. Therefore *s_contr_axi* can be mapped afterwards with an arbitrary but not too small size of 64*KiB*. This results in a complete memory mapping as following:

Table 5.3.: Memory mapping

Device	Memory start	Memory end
Main Memory	*0x0000_0000*	*0x1FFFF_FFFF*
s_ram_axi_	*0x4000_0000*	*0x5FFFF_FFFF*
s_contr_axi_	*0x6000_0000*	*0x6000_FFFF*

5.2.5. AXI Bridging Mechanism

W. Li et al[LDP15] proposed an AXI bridging mechanism. It bridges all memory accesses done by one AXI bus to the next one. Therefore it can be used to redirect memory requests to the main memory, using a memory transformation. The only two data ports in the AXI-Lite bus containing memory addresses are *AWADDR* and *RADDR*.

To redirect memory accesses from the *s_ram_axi* space to the main memory the *0x4000_0000* to *0x5FFFF_FFFF* address range needs to be remapped to the *0x0000_0000* to *0x1FFFF_FFFF* address range.

This means that a memory address, like for example *0x4150_0000*, needs to be mapped to the same offset in the main memory accordingly, *0x0150_0000* in this example.

Therefore, a binary transformation is needed, that transforms *0x4XXX_XXXX* to *0x0XXX_XXXX* and *0x1XXX_XXXX* to *0x5XXX_XXXX*.
This can be achieved by applying a logical *AND* with *0x3FFF_FFFF* to the original address (as seen in the example of Figure 5.2).

```
0x4150 0023 ∧
0x3FFF FFFF
-------------------------------------------------
0b0100 0001 0101 0000 0000 0000 0010 0011 ∧
0b0011 1111 1111 1111 1111 1111 1111 1111
-------------------------------------------------
0b0000 0001 0101 0000 0000 0000 0010 0011
-------------------------------------------------
0x0150 0023
```

(a) Mapping of *0x4150_0023* to *0x0150_0023*

```
0x5150 0023 ∧
0x3FFF FFFF
-------------------------------------------------
0b0101 0001 0101 0000 0000 0000 0010 0011 ∧
0b0011 1111 1111 1111 1111 1111 1111 1111
-------------------------------------------------
0b0001 0001 0101 0000 0000 0000 0010 0011
-------------------------------------------------
0x1150 0023
```

(b) Mapping of *0x5150_0023* to *0x1150_0023*

Figure 5.2.: Memory addresses that are send to the *s_ram_axi_* port gets transferred to target the main memory via logical transformation with *0x3FFF_FFFF*. After the transformation they are in the address range of the main memory but with the same offset as the original address.

Therefore *AWADDR* and *RADDR* will be transformed accordingly. All other signals of the two AXI-Lite Buses (see Chapter 2.4) will be redirected directly resulting in the following (similar to [LDP15]):

Listing 5.1: memtrace.vhd

```
--s_ram to m_ram
m_ram_axi_awaddr <= s_ram_axi_awaddr and x"3FFFFFFF";
m_ram_axi_awprot <= s_ram_axi_awprot  ;
m_ram_axi_awvalid <= s_ram_axi_awvalid  ;
m_ram_axi_wdata <= s_ram_axi_wdata  ;
m_ram_axi_wstrb <= s_ram_axi_wstrb  ;
m_ram_axi_wvalid <= s_ram_axi_wvalid  ;
m_ram_axi_bready <= s_ram_axi_bready  ;
m_ram_axi_araddr <= s_ram_axi_araddr and x"3FFFFFFF";
m_ram_axi_arprot <= s_ram_axi_arprot  ;
m_ram_axi_arvalid <= s_ram_axi_arvalid  ;
m_ram_axi_rready <= s_ram_axi_rready  ;
```

```
--m_ram to s_ram
s_ram_axi_awready <= m_ram_axi_awready  ;
s_ram_axi_wready <= m_ram_axi_wready  ;
s_ram_axi_bresp <= m_ram_axi_bresp  ;
s_ram_axi_bvalid <= m_ram_axi_bvalid  ;
s_ram_axi_arready <= m_ram_axi_arready  ;
s_ram_axi_rdata <= m_ram_axi_rdata  ;
s_ram_axi_rresp <= m_ram_axi_rresp  ;
s_ram_axi_rvalid <= m_ram_axi_rvalid  ;
```

This allows to transparently rewrite main memory accesses to the main memory.

5.2.6. Intercepting Memory Writes

To process the memory tracing and copying, it is necessary to intercept writes launched by the *m_ram_axi* bus. As seen in Figure 2.4 a memory access gets launched exactly as soon as the master has set *awvalid* and *wvalid*. This indicates that the data *wdata*, the memory address *awaddr* as well as the write strobe *wstrb* have been set. Accordingly a state machine can be implemented to follow the memory write process., therefore detecting the memory write and launching the appropriate action.

This can be seen in Listing 5.2. The state machine in the VHDL code is following the proposed state machine of an AXI write transaction. It is mimicking the transaction from *m_ram_axi* to *s_ram_axi*. Therefore it is triggered by the memory write access of *m_ram_axi*. *s_copy_rcv_State* is the signal which represents the state of the state machine. As soon as a state prerequisite, like *m_ram_axi_bvalid* = '1', (Line 613) are fulfilled, the state machine changes to the next state. Nevertheless, compared to Figure 2.4 there are only two states:

1. *s_IDLE*: Machine waits for a transaction to start
2. *s_RCV*: Data and address have been set on the bus.

In comparison to the state machine in Figure 2.4, a state where *s_ram_axi_awready* = *′1′* or *m_ram_axi_wready* = *′1′* does not need to be modeled because of the write information already being available on the bus as soon as the *valid* flags are set (only the master bus′ write action needs to be followed, not the slaves).

Listing 5.2: memtrace.vhd - With the state variable *s_copy_rcv_State* and its two states. The actual intercept of memory address and data follows in between Line 617 and 635 (will be shown in Chapter 5.2.7).

```
613     CASE s_copy_rcv_State IS
614       WHEN s_IDLE =>
615         -- CONNECTED SLAVE AND CONNECTED MASTER AGREE on TRANSACTION SO
                 MAKE COPY TO FIFO
616         if m_ram_axi_bvalid='0' and s_ram_axi_wvalid = '1' and
               s_ram_axi_awvalid = '1' then
617           counter_clk_ram_write := counter_clk_ram_write + 1;
```

```
635           s_copy_rcv_State <= s_RCV;
636         end if;
637
638       WHEN s_RCV =>
639         counter_clk_ram_write := counter_clk_ram_write + 1;
640         -- WIT FOR CONNECTED SLAVE TO CONFIRM ORIGINAL WRITE
641         if m_ram_axi_bvalid = '1' then
642                         counter_transactions_ram_write :=
                                counter_transactions_ram_write + 1;
643           s_copy_rcv_State <= s_IDLE;
644         end if;
```

```
644       WHEN others =>
645         s_copy_rcv_State <= s_IDLE;
646     END CASE;
```

5.2.7. Memory Copying

For memory copying, memory writes needs to be extracted and its content need to be stored. Additionally to writing the memory accesses back to the main memory it is necessary to process a buffered copy to the second version (as described in Chapter 4.3.3).

Three types of information are needed to successfully process the copy.

1. What's the source of the copy?
2. When is a copy about take place?
3. Where to write the copy to?

Therefore it is vital to allow user control over the mapping of pages. This can be achieved by implementing a VHDL based memory mapping. It allows to map one memory range to another equally sized memory range.

The map stores three types of information:

1. The physical source address where a copy has to take place when memory writes target that space.
2. The physical destination address to which the memory will be copied to.
3. The memory range size in byte, which indicates the offset in bytes from the physical source start address, which will still trigger a copy. The memory range of the source address space and the range of the destination address space are equally large in size.

Therefore a composed VHDL datatype *t_map_entry* is introduced (see Listing 5.3, Line 192). This can be grouped to a VHDL array (Line 199), representing a map in the signal *DMAP*.

Listing 5.3: memtrace.vhd - Data structure of DMAP

```
191   -- MAP
192  type t_map_entry is record
193    key   : unsigned(KEY_WIDTH-1 downto 0);
194    value   : unsigned(VALUE_WIDTH-1 downto 0);
195    size   : unsigned(32-1 downto 0);
196    is_set : STD_LOGIC;
197  end record t_map_entry;
198
```

```
type t_map is array (0 to MAX_SIZE-1) of t_map_entry;

signal DMAP : t_map :=
(
  others =>(
    (others =>'0'),
    (others=>'0'),
    (others=>'0'),
    '0'
  )
);
```

There are two algorithms used to insert and find elements in the map structure. These algorithms use the ability of VHDL-93 to repeat specific instructions during the synthesizes, using a *for loop*. Therefore, all keys stores into the map get verified during the execution at the same time, resulting in a one clock cycle lookup. Disadvantageous is the limit of 2^{16} loop executions[2] which limits the maximum amount of watchable pages to 2^{16}. The insertion algorithm is described in Listing 5.4. The algorithm to find a the memory range is implemented accordingly (see Listing 5.5).

Listing 5.4: memtrace.vhd - Inserting records to DMAP. The algorithm is composed of two steps. First of all it searches whether the new key (the memory page start) fits into an already traced memory range (Line 348) which is therefore updated. In the case it has not already been found, a new available place in the array will be searched to insert the record (Line 359 to 367). If no place is found no record is inserted. A VHDL procedure can be used here because the returned result can be stored in a already predefined variable *v_found* which is of type *t_map_entry* itself.

```
338    procedure map_at (
339      key: in unsigned(KEY_WIDTH-1 downto 0);
340      value: in unsigned(VALUE_WIDTH-1 downto 0);
341      size: in unsigned(32-1 downto 0)
342    ) is
343    begin
344      -- TEST IF KEY ALREADY AVAILABLE
345      for I in 0 to MAX_SIZE-1 loop
346        -- CHECK IF KEY IS IN RANGE AND SET
```

[2]see https://www.xilinx.com/support/answers/58823.html

```
   if (key >= DMAP(I).key and key < (DMAP(I).key + DMAP(I).size) and DMAP(I).is_set = '1') then
    -- ENTRY FOUND, UPDATE VALUE
    DMAP(I).value <= value-(key-DMAP(I).key);
    DMAP(I).size <= value-(key-DMAP(I).size);
    v_found := DMAP(I);
    exit;
   end if;
  end loop;
  -- KEY NOT AVAILABLE
  if v_found.is_set = '0' then
   for I in 0 to MAX_SIZE-1 loop
    if (DMAP(I).is_set='0') then
     -- FREE SPACE FOUND, INSERT INTO MAP
     DMAP(I).key <= key;
     DMAP(I).value <= value;
     DMAP(I).size <= size;
     DMAP(I).is_set <= '1';
     v_found := DMAP(I);
     exit;
    end if;
   end loop;
  end if;
 end;
```

Listing 5.5: memtrace.vhd - Finding records in DMAP. Similar to the algorithm for inserting records, this algorithm also searches in parallel for key conformity, meaning key being in watched address range. Key conformity is reached if the key to probe is in the memory range of the stored key, meaning that key to probe is bigger or equal to the stored key and less then the stored key plus the memory range (as seen in Line 222). The value that is returned is the offset the key represents to the the origin's start address but added to the copy address (Line 224).

```
212 function map_find (
213   DMAPIn : in t_map;
214   key: in unsigned(KEY_WIDTH-1 downto 0)
215 )
216   return t_map_entry is
```

```
217    -- default
218    variable v_found : t_map_entry := ((others =>'0'), (others=>'0'), (
           others=>'0'), '0');
219  begin
220    --search for element
221    for I in 0 to MAX_SIZE-1 loop
222      if (key >= DMAPIn(I).key and key < (DMAPIn(I).key + DMAPIn(I).size)
             and DMAPIn(I).is_set = '1') then
223        v_found.key := key;
224        v_found.value := (key-DMAPIn(I).key)+DMAPIn(I).value;
225        v_found.is_set := '1';
226        exit;
227      end if;
228    end loop;
229    return v_found;
230  end;
```

To further implementing the copy of a key, write information will be redirected via a FIFO queue (as mentioned in Chapter 4.3.3) to the *m_copy_axi*.

As previously mentioned, a write transaction transports three types of information

1. Write address
2. Write data
3. Write strobe (which bytes of the memory bus to write)

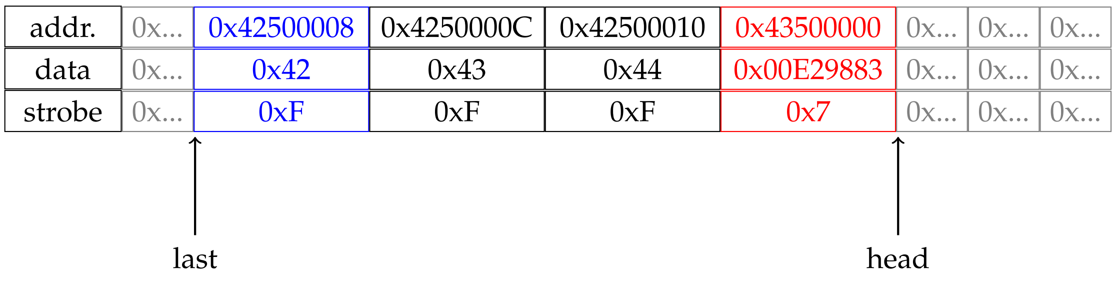
Those can be stored in a composite VHDL type *t_write* (see Listing 5.6, Line 236).

Listing 5.6: memtrace.vhd

```
235  -- record of a write consist of addr, data, and valid byte masking (
         strobe)
236  type t_write is record
237    addr  : STD_LOGIC_VECTOR(C_s_ram_axi_DATA_WIDTH-1 downto 0);
238    data  : STD_LOGIC_VECTOR(C_s_ram_axi_ADDR_WIDTH-1 downto 0);
239    strb : STD_LOGIC_VECTOR((C_s_ram_axi_DATA_WIDTH/8)-1 downto 0);
240  end record t_write;
241
242  type t_FIFO is array (FIFO_CAPACITY-1 downto 0) of t_write;
243  signal FIFO : t_FIFO := (others => ((others => '0'),(others => '0'),(
         others => '0')));
```

```
244 signal head : integer := 0;
245 signal last : integer := 0;
```

A simple FIFO queue can be implemented by creating another new data-type which is an array of *t_write* (Line 242 and 243). Additionally two markers are introduced, *head* and *last* (Line 244 and 245) whereby *head* implements the most recently inserted element and *last* indicates the oldest element which has been inserted, therefore the first element which will be removed. An example configuration of a VHDL FIFO queue can be seen in in Figure 5.3.



addr.	0x...	0x42500008	0x4250000C	0x42500010	0x43500000	0x...	0x...	0x...
data	0x...	0x42	0x43	0x44	0x00E29883	0x...	0x...	0x...
strobe	0x...	0xF	0xF	0xF	0x7	0x...	0x...	0x...

Figure 5.3.: Example of the VHDL FIFO mechanism with a FIFO queue of maximum capacity of 8. It contains 4 elements. *head* indicates the first free position. *last* indicates the position of the element which gets removed first when the next *POP* operation is performed. An insert operation checks firstly whether *head* is one position below *last*. If no, then the element gets written to the position of *head* and *head* gets incremented. if *head* is then bigger then the FIFO queue's capacity it restarts at position 0 (so after the increment a modulo operation with the maximum FIFO capacity is performed on head).

Insertion can be implemented in VHDL with a FIFO *Pushback* operation at position *head*.

The insert mechanism (Listing 5.7) verifies at the beginning whether the FIFO queue is not full (Line 394) and then the data is inserted at position *head* (Line 396). Afterwards *head* moved to the next position.

Listing 5.7: memtrace.vhd - Memory is inserted into the FIFO queue.

```
386  procedure fifo_insert (
387    DataIn : in t_write;
388    headIn: in integer;
389    lastIn: in integer
390  ) is
391  begin
392    --CHECK IF FIFO NOT FULL
393    if( not (headIn = ((lastIn -1) mod FIFO_CAPACITY ) )) then
394      -- INSERT AND UPDATE HEAD
395      FIFO(headIn) <= DataIn;
396      head <= (headIn + 1) mod FIFO_CAPACITY;
397    end if;
398  end;
```

The FIFO get mechanism (Listing 5.8) works similarly. It verifies whether the FIFO queue is not empty (Line 409). If not, an element is returned (Line 411) and the *last* position gets incremented so the element is removed.

Listing 5.8: memtrace.vhd - Returning an element from the FIFO queue.

```
401      fifoIn : in t_FIFO;
402      headIn: in integer;
403      lastIn: in integer
404    ) return t_write is
405      variable element : t_write := ((others => '0'), (others => '0'), (
           others => '0'));
406    begin
407      --CHECK IF FIFO NOT EMPTY
408      if ( not (last = headIn )) then
409        --REMOVE AND UPDATE LAST
410        element := fifoIn(lastIn);
411        last <= (lastIn + 1) mod FIFO_CAPACITY;
412      end if;
413      return element;
414    end;
```

As seen in Chapter 5.2.6 and Listing 5.2 inserting is done in the *s_IDLE* step of the intercepting state machine. As seen in Listing 5.9 does *b_copying* (Line 623) represent the variable which has stored whether copying is globally enabled. If a translation via *map_find* (Line 621) has been found the memory is inserted into the FIFO queue via the mechanism in Line 625 to 628.

Listing 5.9: memtrace.vhd - Copy memory to FIFO

```
619              -- INSERT INTO FIFO IF ENABLED
620              v_found_trace :=((others =>'0'), (others=>'0'), (others=>'0')
                     , '0');
621                          v_found_trace := map_find(DMAP, unsigned(
                                 s_ram_axi_awaddr));
622                          --HAS ADRESS BEEN FOUND?
623              IF b_copying = true and fifoFull = '0' and v_found_trace.
                     is_set = '1' THEN
624                -- SET FIFO DATA
625                v_fifoSet.addr := std_logic_vector(v_found_trace.value);
626                v_fifoSet.data := s_ram_axi_wdata;
627                v_fifoSet.strb := s_ram_axi_wstrb;
```

```
              fifo_insert(v_fifoSet, head, last);
            END IF;
```

Additionally a write back to the main memory is implemented via a write back state machine to the *m_copy_axi* bus. This removes elements from the FIFO queue (see Listing 5.10).

This time three states are implemented:

1. *s_IDLE* (Line 570): Does only change if copying is not paused and the FIFO queue is not empty and no other state machine is working at the moment. An element is retrieved from the FIFO queue (Line 576), a memory transformation is done again with a logical *AND* operation (LINE 580) and the write data is put on the memory bus (LINE 580, 581, 582). The transformation needs to be done at this point due to not being done before the insertion into the FIFO queue.

2. *s_CLONE* (Line 591): After setting both valid states (*m_copy_axi_awvalid* (Line 592) and *m_copy_axi_wvalid* (Line 586)) the master bus waits for both valid states to be set (Line 595).

3. *s_CLONE_DONE* (Line 598): Once the connected Slaves confirms the write via setting *m_copy_axi_bvalid* (Line 600) the master can confirm the confirmation (Line 603) and reset the transfer.

Listing 5.10: memtrace.vhd -

```
569        CASE s_copy_clone_State IS
570          WHEN s_IDLE =>
571            -- FIFO NOT EMPTY AND WRITE IS ENABLED
572            IF b_pause_copy = false AND fifoEmpty = '0' AND
                   m_copy_axi_bvalid = '0' and s_copy_rcv_State = s_IDLE and
                   s_copy_clone_State = s_IDLE and s_contr_read_State = s_IDLE
                     and s_contr_State = s_IDLE THEN --and switch1In = '1'
573              counter := counter + 1;
574              counter_clk_copy := counter_clk_copy + 1;
575              --GET VALUE FROM FIFO QUEUE
576              v_fifoGet := fifo_get(FIFO, head, last);
577              --HAS ADRESS BEEN FOUND?
578
579              -- TRANSFORM ADRESS PUT WRITE INFORMATION ON MASTER COPY AXI
                   BUS
580              m_copy_axi_awaddr <= v_fifoGet.addr and x"3FFFFFFF";
```

```
              m_copy_axi_wdata <= v_fifoGet.data;
              m_copy_axi_wstrb <= v_fifoGet.strb;
              -- PROTOCOL IS ALWAYS "000" (no exclusive access etc.)
              m_copy_axi_awprot <= "000";
              -- ENABLE WRITE
              m_copy_axi_wvalid <= '1';

              s_copy_clone_State <= s_CLONE;

            END IF;
          WHEN s_CLONE =>
                        m_copy_axi_awvalid <= '1';
            counter_clk_copy := counter_clk_copy + 1;
                        -- WAIT FOR CONNECTED SLAVE TO CONFIRM WRITE
            if m_copy_axi_awready = '1' and m_copy_axi_wready = '1' then
              s_copy_clone_State <= s_CLONE_DONE;
            end if;
          WHEN s_CLONE_DONE =>
            -- WAIT FOR CONNECTED SLAVE TO FINISH WRITING
            if m_copy_axi_bvalid = '1' then
              counter_clk_copy := counter_clk_copy + 1;
              counter_transactions_copy := counter_transactions_copy + 1;
              m_copy_axi_bready <= '1';
              m_copy_axi_awvalid <= '0';
              m_copy_axi_wvalid <= '0';
              s_copy_clone_State <= s_IDLE;
            end if;
          WHEN others =>
            s_copy_clone_State <= s_IDLE;
        END CASE;
```

5.2.8. Memory Tracing

Memory tracing, as described in Chapter 4.3.2, can be implemented as well using the memory intercepting algorithm described in Chapter 5.2.6 Therefore an algorithm can intercept at this point a memory write to store whether the set of address space has been modified. Once a memory access has been detected and the memory is supposed to be traced it is updated inside a map to flag the according memory as dirty.

To store a memory page a new data type is introduced *t_dirty_entry*. It contains the identifier of a page and whether the page has been flagged as dirty. Nevertheless it shall be mentioned that it is not necessarily the same page as treated in the operating system or the MMU nor the whole dataspace as added to *DMAP* but junks of the dataspaces that are being watched, so the grantularity can be choosen at compile time. This can be specified with the the generic *PAGE_OFFSET* setting, which indicates all bits of a memory address that are not part of the page.

Listing 5.11: memtrace.vhd - DIRTY is used to save pages that have been modified.

```
251 type t_dirty_entry is record
252   page : UNSIGNED(ADDR_WIDTH-PAGE_OFFSET-1 downto 0);
253   is_dirty : STD_LOGIC;
254 end record t_dirty_entry ;
255
256 type t_dirty is array (0 to MAX_SIZE_DIRTY-1) of t_dirty_entry;
257 signal DIRTY : t_dirty := ( others => ((others=>'0'),'0') );
```

As seen in Listing 5.11 multiple instances of *t_dirty_entry* (specified by the generic *MAX_SIZE_DIRTY*) representing a map (called *DIRTY*) (Line 251, 256 and 257). This map stores all pages that have been modified. Furthermore *DIRTY* does not contain all possible pages but just only the pages that are flagged as modified. If they get unflagged as modified they can still be contained in *DIRTY* but they are flagged as not modified and can be overwritten.
There are two algorithms represented for *DIRTY* :

1. Removing and returning one modified page (as seen in Listing 5.12): Uses again a for loop (Line 377) to find the first modified page (Line 378). The first modified page is the page which has its *is_dirty* flag set. *v_page_found* is a variable of type *t_dirty_entry* a well, which gets then returned.

2. Adding a modified page (as seen in Listing 5.13). Works similar in the sense, that is searches the page with a for loop (Line 422 to 429) and if it has not been inserted it searches for a free place to insert (Line 431 to 438).

Listing 5.12: memtrace.vhd - Returning and removing a modified page

```
371   procedure get_dirty_page is
372   begin
373     v_page_found.is_dirty := '0';
374
375     -- FIND PAGE
```

```
    for I in 0 to MAX_SIZE_DIRTY-1 loop
      if DIRTY(i).is_dirty = '1' then
        DIRTY(i).is_dirty <= '0';
        v_page_found.page := DIRTY(i).page;
        v_page_found.is_dirty := '1';
        exit;
      end if;
    end loop;
  end;
```

Listing 5.13: memtrace.vhd - Flagging a Page as modified

```
  procedure dirty_at (
    page: in unsigned(ADDR_WIDTH-PAGE_OFFSET-1 downto 0)
  ) is
  begin
    v_page_found.is_dirty := '0';
    for I in 0 to MAX_SIZE_DIRTY-1 loop
      -- CHECK IF PAGE ALREADY SET
      if DIRTY(I).page = page then
        -- FREE SPACE FOUND, SET PAGE
        v_page_found.is_dirty := '1';
        exit;
      end if;
    end loop;
    if v_page_found.is_dirty = '0' then
      for I in 0 to MAX_SIZE_DIRTY-1 loop
        if (DIRTY(I).is_dirty = '0') then
          -- FREE SPACE FOUND, SET PAGE
          DIRTY(I).page <= page;
          DIRTY(I).is_dirty <= '1';
          exit;
        end if;
      end loop;
    end if;
  end;
```

Additionally it is necessary to limit the memory tracing to user defined memory spaces, otherwise for example the checkpointer itself would trigger memory traces. This can be achieved as well utilizing the memory map (see Chapter 5.2.6) and trace only memory the areas that have been mapped (see Listing 5.14). If the variable *b_pause_trace* is enabled (can be disabled if tracing is disabled), then every time the element has been found in the translation (*v_found_trace.is_set* is set) the address gets inserted via the algorithm. It gets truncated via a rigthshift and the *PAGE_OFFSET* to achieve the desired granularity.

Listing 5.14: memtrace.vhd -

```
            --SET PAGE AS DIRTY
            IF NOT b_pause_trace AND v_found_trace.is_set = '1' THEN
                dirty_at(shift_right(unsigned(s_ram_axi_awaddr),
                    PAGE_OFFSET));
            END IF;
```

Integration using Vivado

To connect the developed IP design, one general purpose slave (slave GP0) and one general purpose master port (master GP0) need to be added to the Zynq procesing system (as seen in Figure 5.4). An upgrade to high performance slave ports could be implemented as well. However this only works if AXI-Full is implemented to support high performance burst modes. Between the Zynq processing system and the developed IP Block there are two Xilinx AXI Interconnects[Xil17a]. AXI Interconnect 1 on the left connects the master port of the Zynq processing system with *s_ram_axi* and *s_contr_axi*. The second AXI Interconnect bundles the two master ports of the IP Block (*m_ram_axi* and *m_copy_axi*) with the slave port of the Zynq processing system. An AXI interconnect is able to channel the access exclusively so just one transaction is done at a time. Therefore it buffers accesses in the case two master ports are connected and both launch a transaction simultaneously. Adding a second slave port to the Zynq processing system, results in an error where the two master buses block each other during a parallel execution, resulting in erroneous data access commands.

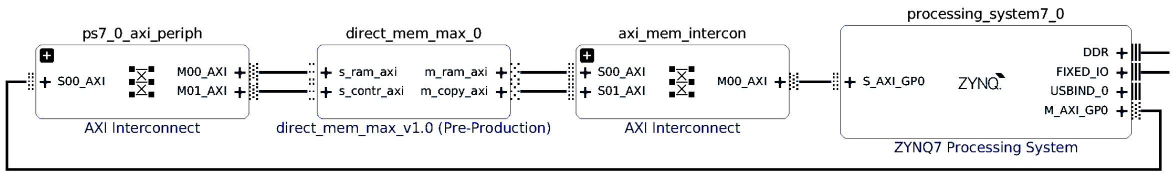


Figure 5.4.: The simplified view on the connection of the IP-Block and the Zynq processing system.

5.2.9. Example Memory accesses

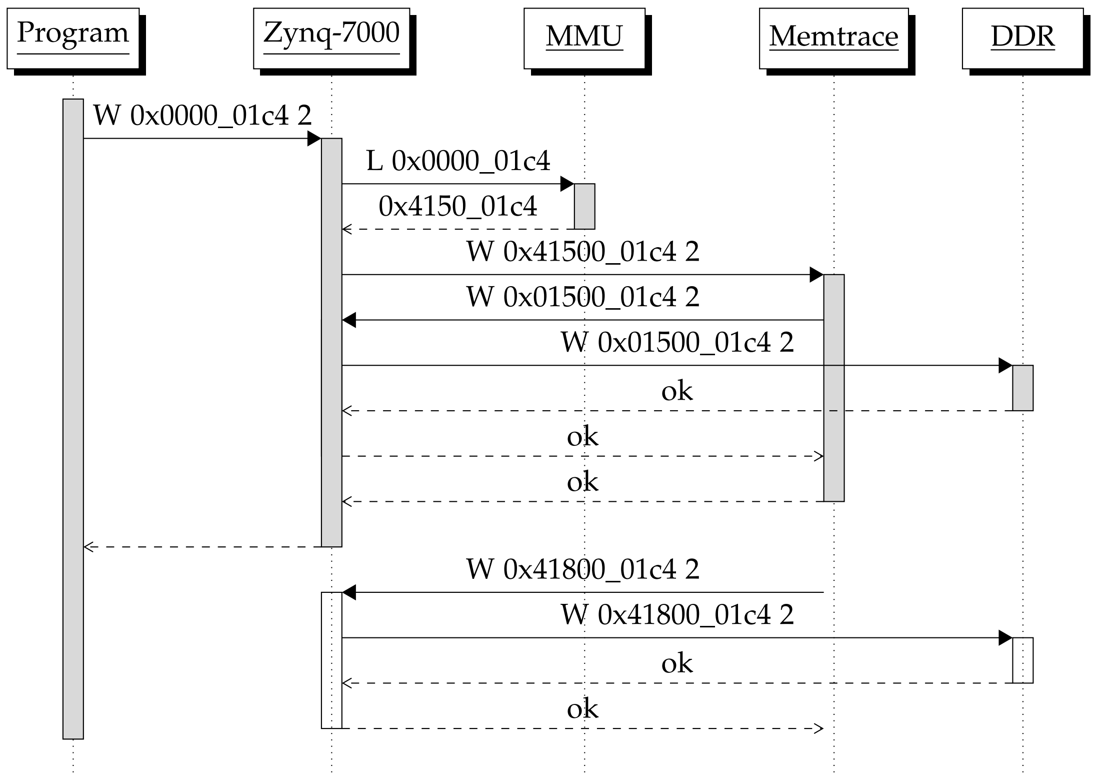


Figure 5.5.: Example execution of memory write

In summary an example memory write access can be seen in Figure 5.5. In this example a bare metal program, which is a program without operating system, but in this case with activated virtual memory management, writes the value 0x2 to the memory address *0x0000_01c4* (abbreviated *W 0x0000_01c4 2*). After a memory translation from virtual memory to the physical memory (lookup via *L 0x0000_01c4*) has been performed, the Zynq processing system redirect the memory write with the physical address to the AXI peripheral (abbreviated *W 0x41500_01c4 2* to *Memtrace*). The AXI peripheral (*Memtrace*) does the memory translation to target the address in the main memory (AXI bridging mechanism) via *W 0x01500_01c4 2*. Therefore the Zynq processing system redirects the access to the main memory.

After a successful write to the main memory the AXI peripheral performs the copy (see *W 0x41800_0000 2*, in case a memory translation exists from *0x41500_01c4* to

0x41800_0000).

5.2.10. API between FPGA-component and Software

Mainly responsible for accessing the FPGA component is the API exposed by the component itself. The API section of the component is memory mapped at *0x6000_0000* in the physical memory. It can be accessed by writing and reading specific memory addresses, accessed over the *s_contr_axi* AXI-Lite bus.

The following main functions are exposed by the API to comply with the requirements stated in Chapter 4.2 allowing to control the the translation of memory described in Chapter 5.2.4. They are subsequently abbreviated with with their number in this list.

1. Loading a memory translation
2. Setting a memory translation
3. Loading a memory page which has been modified
4. Enabling copying globally (set as default)
5. Disabling copying globally (this is implemented so a "tracing only implementation" is possible)
6. Pause copying
7. Unpause copying
8. Pause tracing, in case untraced action is supposed to be performed
9. Unpause tracing

These operations are controlled by the exposed API (see Figure 5.6). It consists of eight equally sized, 32 bit fields, which are either read write enabled or read only *(RO)*. To launch an action a value has to be written to the operation field. For example in case of Action 3, a "3" has to be written to the operation field. Even if a "3" has already been the content of the operation field the action is performed.

	0	32	64	96 128
API	Operation	Status *(RO)*	Address Copy	Address Origin
	Translation Size	Modified Page *(RO)*	FIFO Length *(RO)*	Enabled Flags *(RO)*
DEBUG	Count Clk Copy	Count Copy	Count Clk Bridge	Count Bridge

Figure 5.6.: Memory layout of the API exposed by the hardware component at *0x6000_0000*. Every element is exactly 32 bit. Additionally to the API itself in the first 256 byte there 128 of debug counters which count clock cycles and transactions of the copy state machine as well as the bridge state machine.

The following shows how each action is supposed to be operated:

Action 1: Loading a memory translation
Action 1 is supposed to load a already set memory translation of the *DMAP* data structure. As seen in Figure 5.7 a memory address has to be set into the origin field. Then the operation has to be launched by writing "1" into the operation field. If the memory address was in a translation range it will be returned. In the translation range the memory address is located with the following formula : $ORIGIN\_BASE - ORIGIN\_COPY + TRANSLATION\_BASE$. Also the status field will be set to *0x202*. If no address has been found the status code will be set to *0x404*, which are the only two possible status values.

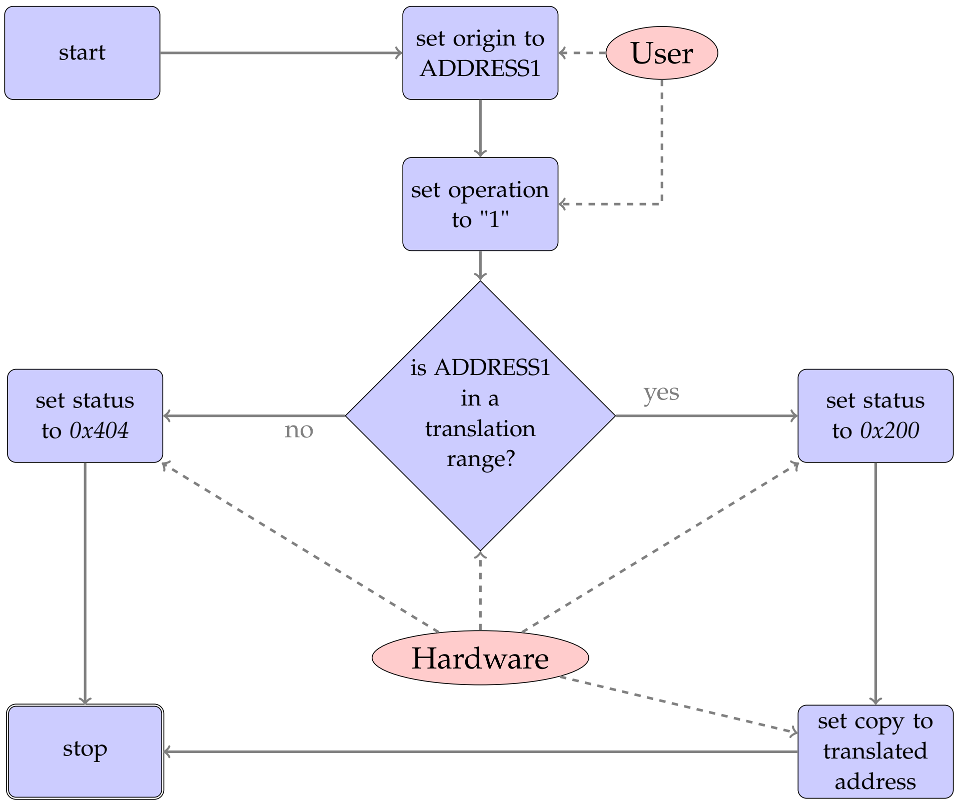


Figure 5.7.: An example get memory operation. The translated address has the same offset to the copy address as ADDRESS 1 is to the original base translation. For example if the memory translation *0x4150_0000* to *0x4170_0000* exists, *0x4150_0014* will be translated to *0x4170_0014*.

Action 2: Setting a memory translation
This works similar to Action 1, but with the difference that the copy field has to be set as well and the origin and copy field are the base addresses of the translations which is supposed to occur. After writing "2" to the operation field there can happen two possibilities. If the origin field is already in a translated memory range the translation will be updated, or a new translation will be set due to every memory address only being able to be bound to one translation at a time.

Action 3: Loading a memory page which has been modified
This action is launched by writing "3" to the operation field. If a modification to a translated page has been stored (a page was flagged as dirty), the address of a modified page inside a translation will be written to the modified page field and the status field

will be set to *0x202*. If no modified page has been found the status code will be set to *0x404*. This action also removes the dirty flag from the page. Therefore it won't be returned again if the action is performed another time. The returned pages occur in the order, that they are stores in the data structure, which can be assumed as random.

Action 4/5: En-/Disabling copying
Copying can be enabled globally via writing "4" to the operation field and disabled via writing "5" to the operations field. Global deactivation means, that even though translations can be set (which is necessary to trace information), the write accesses are not stored in the FIFO queue and not written to the translated dataspaces. Copying is enabled by default.

Action 6/7: Pause -/Unpause copying
Works similar to Action 4 and 5. Writing "6" to the operation field pauses the copying until it is unpaused via writing "7" to the operation field. Pausing copying means, that write accesses are still stored into the FIFO queue, they are just not written to the translated page. As soon as copying is unpaused, the FIFO queue will be emptied in order and the content will be written to the translated page. This is useful if the target of a translation space wants to be copied without interrupting the program using the origin space of the translation. Copying is unpaused by default.

Action 8/9: Pause -/Unpause tracing
Works similar to Action 4 and 5. Writing "8" to the operation field disables the tracing until it is reenabled via writing "9" to the operation field. This can be used to set up memory translations and not store the set up memory accesses to the translated pages. Tracing is unpaused by default.

5.3. Software Implementation and Modifications

To achieve the goal of making the hardware API developed in Chapter 5.2.10 as well as the copying mechanism usable for a checkpoint restore mechanism, they need to be integrated into *L4 Fiasco.OC/Genode* .

5.3.1. Software Integration

The most well suited point of integration is a remapping of the main memory base address of the kernel to the main memory of the Fiasco.OC kernel such that it is located in the memory range of the developed AXI peripheral, as also suggested by

[LDP15]. This can be achieved by changing the operating system's base ram address to the AXI-peripheral in *L4 Fiasco.OC/Genode* . Therefore all memory accesses will be relocated and directed to the AXI peripheral. Additionally a Genode hardware driver can be developed to make use of the API developed in 5.2.10. Because the API does enforce any convention concerning the type of physical memory addresses which will be mapped, any software object whose physical address can be calculated, is able to be copied. Ultimately those physical addresses will be those of dataspaces attached to a Genode component, because they represent the logical equivalent to a processes' data block which exists always coherently (see [Fes15]). Therefore it needs to be proven that dataspaces can be used to be copied and its pages can be traced.

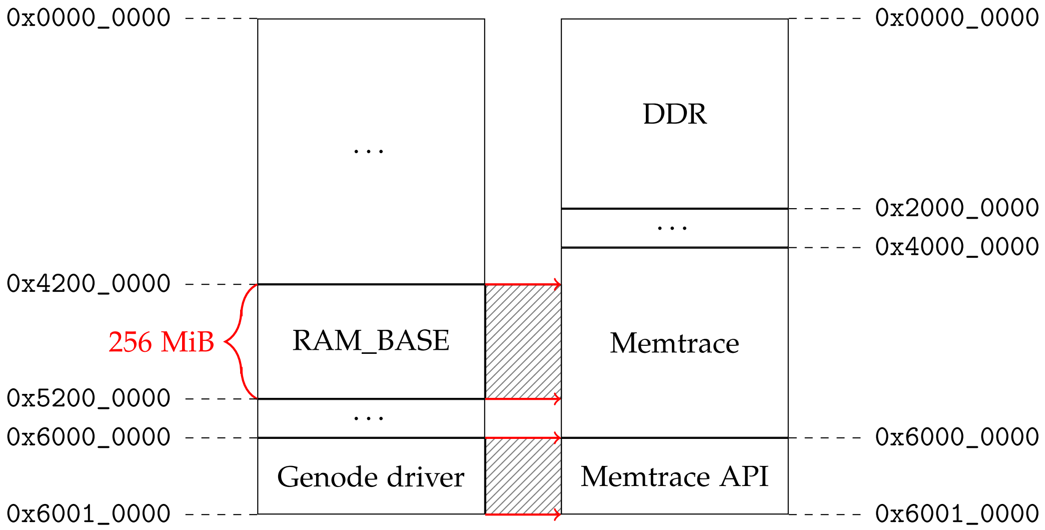


Figure 5.8.: The memory mapping established between L4/Fiasco.OC Genode and the developed AXI peripheral.

From the operating system's point of view does the physical memory mapping therefore look like in Figure 5.8. The operating system assumes the main memory to be located between *0x4200_0000* and *0x51FF_FFFF* (the reason it does not start at *0x4000_0000* will discussed in Chapter 6.4.2). The Genode driver is connected to the memory space at *0x6000_0000* to *0x6000_FFFF* (where the API is located). Noticeable is the fact, that the operating system is not connected to the main memory any more, but it is completely redirected via the AXI peripheral.

5.3.2. Configuration Modifications & Boot Process

Genode and L4 Fiasco.OC Modifications

To achieve the mentioned redirect of the main memory to the AXI peripheral it is necessary to change the main memory base address for the *L4 Fiasco.OC / Genode* system. There are multiple files which indicate to be able to achieve the desired goal.

First of all the directory *genode/repos/base/include/spec* seems to indicate to be a location of board configuration files. Especially the file *genode/repos/base/include/spec/zybo/-drivers/board_base.h* contains memory mapping variables such as *RAM_0_BASE* and *RAM_0_SIZE*. Nevertheless changing those addresses does not lead to the desired effect. Changing those variables and starting the system leads to now effect. Following the variables utilization in *L4 Fiasco.OC / Genode* shows, that they are not used in the already existing configuration.

Secondly another approach is the modification of the file *zedboard.conf* located in the Fiasco.OC repository in the directory *src/kernel/foc/l4/mk/platforms/*. It specifies the variables named *PLATFORM_RAM_BASE* as well as *PLATFORM_RAM_SIZE_MB*. Those can be changed to the desired values (see Listing 5.15).

Listing 5.15: zedboard.conf

```
PLATFORM_NAME = "Xilinx␣Zynq␣Zedboard"
PLATFORM_ARCH = arm
PLATFORM_RAM_BASE = 0x42000000
PLATFORM_RAM_SIZE_MB = 256
PLATFORM_UART_NR = 1
```

Additionally, the file named *Modules* in *src/kernel/foc/kernel/fiasco/src/kern/arm/bsp/zynq/* which also contains a variable named *RAM_PHYS_BASE* must be changed to *0x42000000*. Compiling and booting the operation system displays an error, that the Genode Core component is located not in a memory address that is in the physical memory space. This error can be solved by changing the variable *LD_TEXT_ADDR* in the file *genode/base/src/core/spec/zybo/target.mk* to a memory address which is located in the newly defined address space (e.g. *0x4300_0000*).
Booting the system again reveals a successful run.

Bootloader

```
Zybo-SDCARD
├── fpga.bin ........................................ FPGA bitstream file
├── image.elf ................................ L4/Fiasco.OC Genode Kernel
├── modules.list .............................. List of Genode modules
├── genode ........................ All Genode modules and configuration
│   ├── core
│   ├── config
│   └── ...
├── boot.bin ..................... First file to boot, loads u-boot.img
└── u-boot.img ................................... U-Boot bootloader
```

Figure 5.9.: Files listed on the SD-Card used to boot the Genode system on the Zybo board

The Zybo board includes three forms of booting options:

1. JTAG Boot: The operating system including the FPGA file will be transferred via a USB connection from a connected laptop (preferably using Xilinx SDK[3]).
2. OSIF: Booting from the built in flash memory normally occupied by a Linux system).
3. SD-Card: Booting via a Micro-SD card inserted into the board.

To boot the Genode system with the FPGA, a specific version of *Uboot* is used, which has been configured by Xilinx [4]. A jumper which can be set manually on the board allows to set the specific boot option. Putting the *U-Boot* bootloader including the operating system and the FPGA configuration file on an SD-Card (see Figure 5.9) allows the Zybo system to boot. In SD-Card boot mode, the Zybo board searches a file called *boot.bin* which is configured to load the initial bootloader into the main memory and start it.

It can be configured further to include the necessary loading of files into the memory. This can be done by modifying the file *include/configs/zynq-common.h* in the Xilinx *U-Boot* folder like the following (Listing 5.16):

[3]https://www.xilinx.com/products/design-tools/embedded-software/sdk.html
[4]https://github.com/Xilinx/u-boot-xlnx

Listing 5.16: zynq-common.h

```
229  "kernel_image=image.elf\0" \
```

```
231  "imgloadaddr=0x55000000\0" \
```

```
236  "bitstream_image=fpga.bin\0" \
237  "boot_image=BOOT.bin\0" \
238  "loadbit_addr=0x100000\0" \
```

```
282  "sdboot=if mmcinfo; then " \
283     "echo Load FPGA... && " \
284     "load mmc 0 ${loadbit_addr} ${bitstream_image} && "\
285     "fpga load 0 ${loadbit_addr} ${filesize} && "\
286     "echo Load Kernel ... && "\
287     "load mmc 0 ${imgloadaddr} ${kernel_image} && " \
288     "bootelf ${imgloadaddr} && "\
289    "fi\0" \
```

As shown, Line 229 to Line 238 prepare variables of memory addresses and file names. Line 284 loads the FPGA configuration file into the memory. The FPGA file needs to be available in the *.bin* format. In the Xilinx Vivado produces a *.bit* file. Therefore, the header information (first 224 byte) needs to be removed [5]. Line 285 programs the FPGA with the loaded file. Line 287 loads the kernel file *image.elf* into the memory. Line 288 starts the operating system's kernel.

5.3.3. Genode Driver

Genode drivers are the only components allowed to access physical dataspaces themselves without having a dataspace capability. Therefore, to implement a connection to the AXI-peripheral API mapped at *0x6000_0000*, a Genode driver needs to be implemented.

A Genode driver can be created following the example set by the GPIO driver rhe Raspberry Pi, located in *genode/repos/os/src/drivers/gpio/spec/rpi*, *genode/repos/os/include/gpio_session* and *genode/repos/os/include/gpi*.

A Genode driver consists in general of three parts[Fes15]:

1. The include driver files in the *include/...* folder

[5] `https://github.com/topic-embedded-products/meta-topic/blob/master/recipes-bsp/fpga/fpga-bit-to-bin/fpga-bit-to-bin.py`

2. The driver's Genode component located in the source folder
3. The include session files in the *include/..._session* folder

Driver include files The common interface from a component using the driver are the driver include files located in the *include/...* folder. It exposes the *Driver* class which contains all abstract functions that the component can access, either by the client (the program using the driver) by implementing the RPC connection to the driver or by the driver by implementing the actual algorithm. Additionally there are two classes in the *components.h* file: *Session_component* and *Root*.

Driver component A driver component is a standalone module which runs in privileged mode in the Genode system so it can access physical memory or kernel specific resources.
As described in Chapter 2.1, a Genode component creates an RPC entrypoint to provide its service to other components which can be accessible via a capability.
The component consists of a main file (*main.cc*) which instantiates an object of the type *Zybo_driver*. The *Zybo_driver* implements the actual set algorithms exposed to cover the RPC-entrypoint. It is associated to a *FPGA_CONTR_REG* object which inherits from a *Attached_io_mem_dataspace* and *Mmio* class located in the Genode module. The latter two classes allow the *FPGA_CONTR_REG* to access the physical memory directly. As it can be seen in Listing 5.18, *FPGA_CONTR_REG* contains *C++ structs* that inherit from a *Register* class, which is specified by the number of bits that are supposed to be accessed over this memory interface (see Line 33 to 38 in Listing 5.18). The Zybo driver can write and read from those registers to implement the algorithm (for example the *get_translation* algorithm in Listing 5.17). Therefore it can implement the conventions established by the hardware API in Chapter 5.2.10. Additionally it instantiates an object of the included *Root* class which creates itself a *Session_component* if a connection from a client is established. A full class diagram can be seen in Figure 5.10.

Listing 5.17: driver.h

```
  Genode::addr_t get_translation(
    Genode::addr_t origin
  ) {
    _fpga_contr.write<FPGA_CONTR_REG::Op>(NOP);
    _fpga_contr.write<FPGA_CONTR_REG::Origin>(origin);
    _fpga_contr.write<FPGA_CONTR_REG::Op>(GET);
    _fpga_contr.write<FPGA_CONTR_REG::Op>(NOP);
    return (Genode::addr_t) _fpga_contr.read<FPGA_CONTR_REG::Copy>();
    }
```

Listing 5.18: memtrace.h

```
struct FPGA_CONTR_REG : Genode::Attached_io_mem_dataspace, Genode::Mmio
{
  FPGA_CONTR_REG(Genode::addr_t const mmio_base,
          Genode::size_t const mmio_size)
  : Genode::Attached_io_mem_dataspace(mmio_base, mmio_size),
    Genode::Mmio((Genode::addr_t)local_addr<void>()) { }

  struct Op   : Register<0x00, 32> {};
  struct Status  : Register<0x04, 32> {};
  struct Origin  : Register<0x08, 32> {};
  struct Copy  : Register<0x0C, 32> {};
  struct Size  : Register<0x10, 32> {};
  struct Dirty   : Register<0x14, 32> {};
```

Driver session A driver session is established by the the component using the developed Driver. Therefore a *Connection* object is created which instantiates a *Session_client* which represents a session to the drive, therefore being an *RPC_client* itself. It overrides the virtual function from the include files with the actual RPC calls (as seen in Listing 5.19).

Listing 5.19: client.h

```
  Genode::addr_t get_translation(
    Genode::addr_t origin
  ) override { return call<Rpc_get_translation>(origin); }
```

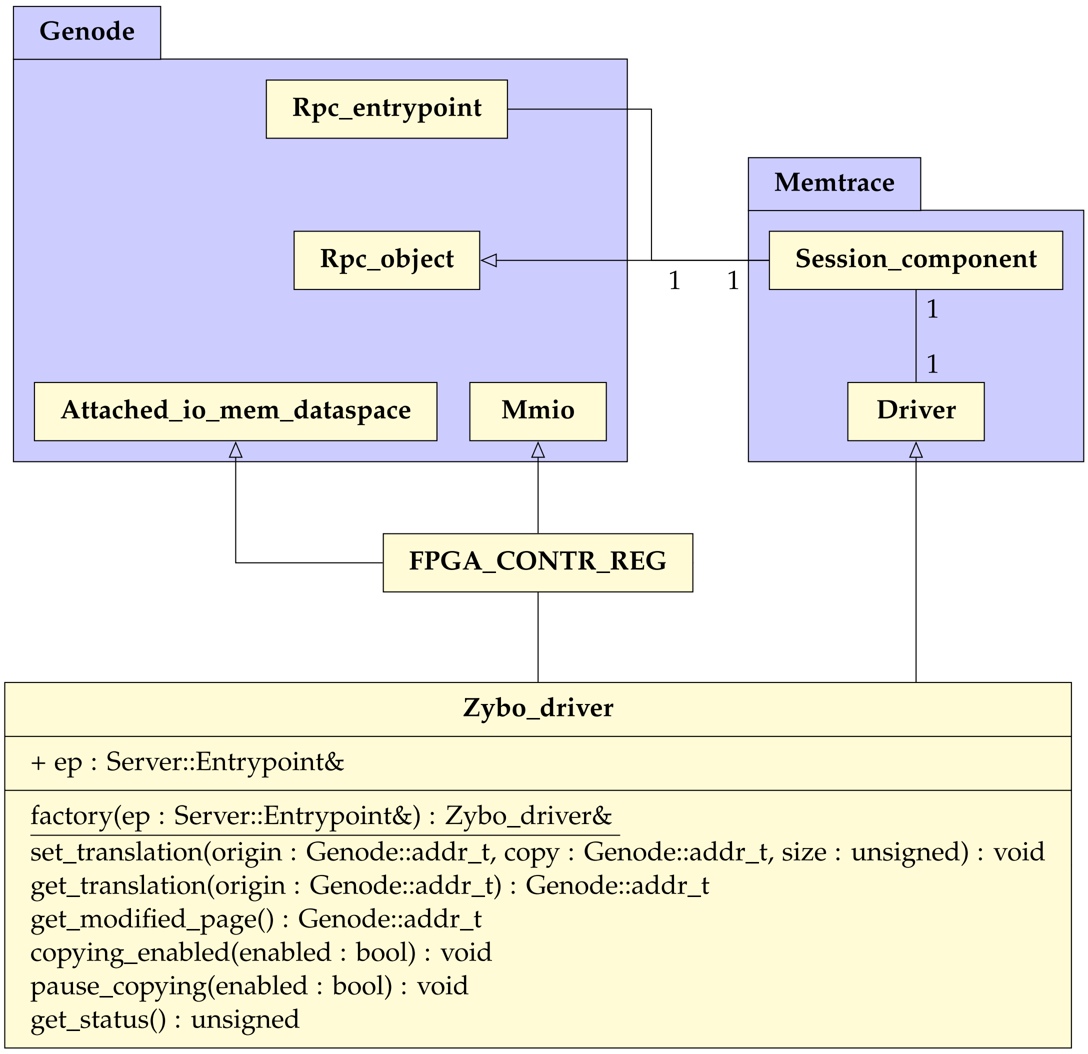

Genode
Rpc_entrypoint
Rpc_object
Attached_io_mem_dataspace
Mmio
Memtrace
Session_component
Driver
1
1
1
1
FPGA_CONTR_REG
Zybo_driver
+ ep : Server::Entrypoint&
factory(ep : Server::Entrypoint&) : Zybo_driver&
set_translation(origin : Genode::addr_t, copy : Genode::addr_t, size : unsigned) : void
get_translation(origin : Genode::addr_t) : Genode::addr_t
get_modified_page() : Genode::addr_t
copying_enabled(enabled : bool) : void
pause_copying(enabled : bool) : void
get_status() : unsigned

Figure 5.10.: Class diagram of the implemented Genode driver and partially from the driver session

6. Results

This chapter shows results based on the implementation. it will show working examples, clock cycle count timing experiments as well as performance measurements.

6.1. Functional Testing via a Genode Component

6.1.1. Working Example

This section shows, that the developed driver and hardware component is functional by implementing an example usage in a Genode component.

Simple Tracing and Copying To use the developed AXI peripheral a components needs to connect to the driver. This can be done (as seen in described in Chapter 5.3.3).

Listing 6.1: main.cc

```
  addr_t addr[4];
  addr_t phys[4];
  Dataspace_capability ds_caps[4] = {
    env.ram().alloc(4*4096, UNCACHED),
    env.ram().alloc(4*4096, UNCACHED),
    env.ram().alloc(4*4096, UNCACHED),
    env.ram().alloc(4*4096, UNCACHED),
  };

  for(int i = 0; i < 4; i++)
  {
    addr[i] = env.rm().attach(ds_caps[i]);
    phys[i] = Genode::Dataspace_client(ds_caps[i]).phys_addr();
    printf("physical address of ds_cap[%i] : %08lX\n", i, phys[i]);
    printf("stuff %08lX\n", Genode::Dataspace_client(env.rm().dataspace
        ()).phys_addr());
  }

```

```
152     Memtrace::Connection mem(env);
153     /** map ds_cap[0] to ds_cap[1] and ds_cap[2] to ds_cap[3] **/
154     mem.set_translation(phys[0], phys[1], 4*4096);
155     mem.set_translation(phys[3], phys[2], 4*4096);
```

As seen in Listing 6.1 in Line 135 a Genode dataspace can be created and its physical address can be calculated using a *Dataspace_client* (Line 147). Afterwards a connection to the driver can be established creating a *Connection* object (Line 152). Using the API a translation can be done by mapping the physical addresses (origin and copy) to each other (Line 154 and 155)-

In this example four dataspaces are created. *Dataspace 0* is mapped to *Dataspace 1*, and *Dataspace 3* is mapped to *Dataspace 2*. Copying is enabled by default, therefore, if writing to *Dataspace 3* all memory accesses should be copied to *Dataspace 2* as well.

Listing 6.2: serial_output.txt

```
 1 ...
 2 [init -> fpga] physical address of ds_cap[0] : 423b8000
 3 [init -> fpga] physical address of ds_cap[1] : 423bc000
 4 [init -> fpga] physical address of ds_cap[2] : 423c0000
 5 [init -> fpga] physical address of ds_cap[3] : 423c4000
 6 ...
 7 [init -> fpga] Show memory at 10000
 8 [init -> fpga] 00010000: 00000000 00000001 00000002 00000003
 9 [init -> fpga] 00010010: 00000004 00000005 00000006 00000007
10 [init -> fpga] 00010020: 00000008 00000009 0000000a 0000000b
11 [init -> fpga] 00010030: 0000000c 0000000d 0000000e 0000000f
12 [init -> fpga]
13 [init -> fpga] Show memory at 14000
14 [init -> fpga] 00014000: 00000000 00000001 00000002 00000003
15 [init -> fpga] 00014010: 00000004 00000005 00000006 00000007
16 [init -> fpga] 00014020: 00000008 00000009 0000000a 0000000b
17 [init -> fpga] 00014030: 0000000c 0000000d 0000000e 0000000f
18 ...
19 [init -> fpga] (Page|Status): 423c4000 : 00000200
20 ...
```

As seen in Listing 6.2, which represents parts of a serial output of the Zybo board, writing of values *0x1* to *0xF* to *Dataspace 3* (which has physical address *0x423c_4000* and virtual address *0x0001_4000*, see Line 5 and Line 13) get copied to *Dataspace 2*, which has physical address *423c_0000* and virtual address *0x0001_0000*, see line 4 and line 7.

Additionally calling the *get_modified_page* and *get_status* functions of the Genode driver reveals that the modification of this page has been detected successfully (Line 19).

Copying with Pausing With the same setup as before (Listing 6.1, assume creation of 2 Dataspaces, with their virtual addresses stored in *addr_t*) a second working example can verify the pause during copy mechanism (see Listing 6.3). Thereby 8 values will be written to *Dataspace 0* which is mapped to *Dataspace 1*. During the write of the 8 values (Line 118 and 123) the copying has been paused (Line 120). Even though the writing to *Dataspace 0* continues (Line 123) before *Dataspace 1* has been processed (Line 126) it can be seen that the content of Dataspace has been changed (see content of *Dataspace 0* of the output in Listing 6.4 in Line 2-3 and for *Dataspace 0* in Line 6-7). After the copying has been resumed (Listing 6.3, Line 128) the output shows that the content has been transferred (Listing 6.4 in Line 14-15 for *Dataspace 1*), therefore the internal FIFO queue has been emptied.

Listing 6.3: main.cc

```
112   unsigned* array[2] = {
113     (unsigned*) addr[0],
114     (unsigned*) addr[1]
115   };
116
117   for (unsigned i = 0; i < 4; ++i)
118     array[0][i] = 0x42 + i;
119
120   mem.pause_copying(true);
121
122   for (unsigned i = 4; i < 8; ++i)
123     array[0][i] = 0x42 + i;
124
125   dump_mem(addr[0], 32);
126   dump_mem(addr[1], 32);
127
128   mem.pause_copying(false);
129
130   dump_mem(addr[0], 32);
131   dump_mem(addr[1], 32);
```

Listing 6.4: serial_output2.txt

```
init -> fpga] Show memory at 8000
[init -> fpga] 00008000: 00000042 00000043 00000044 00000045
[init -> fpga] 00008010: 00000046 00000047 00000048 00000049
[init -> fpga]
[init -> fpga] Show memory at c000
[init -> fpga] 0000c000: 00000042 00000043 00000044 00000045
[init -> fpga] 0000c010: 00000000 00000000 00000000 00000000
[init -> fpga]
[init -> fpga] Show memory at 8000
[init -> fpga] 00008000: 00000042 00000043 00000044 00000045
[init -> fpga] 00008010: 00000046 00000047 00000048 00000049
[init -> fpga]
[init -> fpga] Show memory at c000
[init -> fpga] 0000c000: 00000042 00000043 00000044 00000045
[init -> fpga] 0000c010: 00000046 00000047 00000048 00000049
```

6.1.2. Caching

The AXI-peripheral is being dependent on memory writes being directed directly to it. Therefore, the caching of memory accesses by the processor can be harmful to the functionality. As already remarked by W. Li et al [LDP15] it is necessary to disable the caching of data.

This can successfully be done by creating dataspaces with the flag *UNCACHED* (see Listing 6.1 Line 138-141). Not using this flag results in the copying not working or data being loaded incorrectly, even though they might be copied correctly, because they might be read from the cache.

6.2. Timing Tests via Clock-Cycle Counter

One complication that might occur, is that the AXI bridge writes memory faster to the *m_ram_axi* than the *m_copy_axi* can write back the memory.

For verification, the number of transactions and the number of clock cycles per transaction can be counted. This can be implemented via inserted "counting integer variables" (see Table 5.2) in the right position in the VHDL code and exposing the content via the memory mapped API. Via *Uboot*'s memory management commands (*mm* to set memory and *md* to read memory) can this be set and verified. With this the following experiment can be executed with the following steps:

1. set a translation from *0x4150_0000* to *0x4170_0000*
2. repeat 10 times {
3. write 1 to *0x4150_0000*
4. store transaction and clock cycle counters
5. }

This results in the data seen in Figure 6.1. It can be seen, that number of clock cycles is nearly constant for both, the bridge mechanism and the copying. For the bridge mechanism the number of clock cycles per transaction is 19 or 20, for the copying mechanism the number of clock cycles is 6. Therefore it can be concluded that the copying mechanism can be executes quicker then the bridge mechanism. Therefore it is not possible, that the FIFO queue overflows because elements are inserted more quickly than they are removed, if copying is not paused.

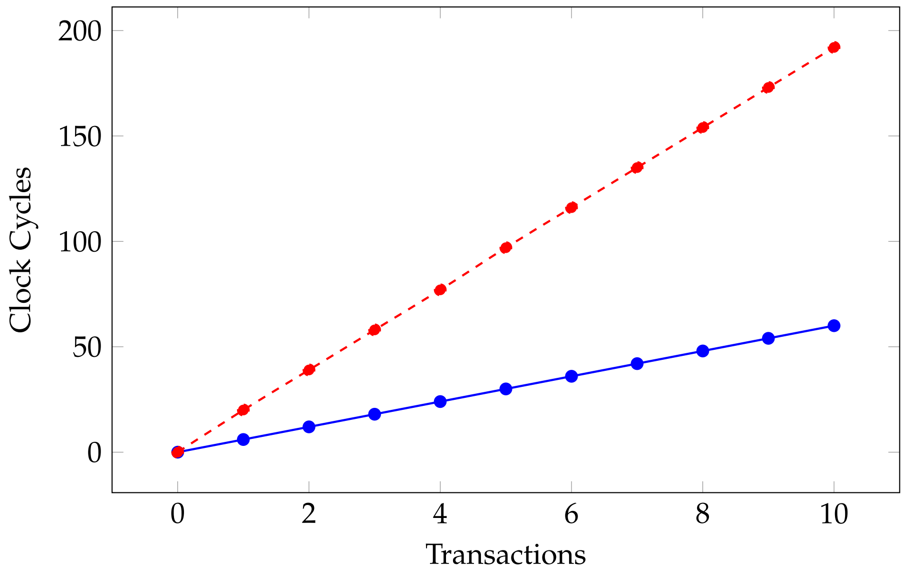


Figure 6.1.: Accumulated number of clocks spent per transactions writing data to the *m_copy_axi* bus (blue line) and *m_ram_axi* bus (red dashed line). (see also Appendix A for the full protocol)

Ultimately the constant *cycle write back time* results in the fulfillment of the hard real time[But11] requirements imposed by the nature of the *L4 Fiasco.OC/Genode* operating system. Therefore the maximum amount of time needed to process an amount of transactions can be calculated via the following equation:

$$T = \frac{C \cdot N}{Frequency} \tag{6.1}$$

This calculates the amount of time T needed to process N transactions in the AXI peripheral with C as the number of clock cycles per transaction which has been measured. *Frequency* is the AXI-Peripheral bus frequency.

Therefore for example, emptying a FIFO queue storing 50 elements (therefore processing $N = 50$ transactions) and assuming a AXI peripheral frequency of $100MHz$ (AXI peripheral frequency is not CPU clock and $100MHz$ is usually the default) with a measured clock cycles per transaction count of $C = 6$ (see Figure 6.1) results in the following time needed :

$$T = \frac{6 \cdot transactions^{-1} \cdot 50 \cdot transactions}{100 \cdot 10^6 \cdot s^{-1}} = 3\mu s \tag{6.2}$$

6.3. Performance Measurement of Cached and Uncached Dataspaces with AXI-Peripheral

As described in chapter 6.1.2 Dataspaces that are supposed to be copied and traced via the AXI peripheral need to be created with at *UNCACHED* flag so they are not cached by the processing system's built in cache.

Disabling a processor cache can lead to performance penalties regarding memory writes and reads. To indicate a quantitative approach for the orders of magnitude where the performance penalty is located, the difference in performance between cached and uncached Dataspaces can be profiled[1]..

Furthermore, the developed AXI peripheral can have an effect on the write speed of memory. Therefore an investigation on its influence on performance is tested as well.

6.3.1. Write Access Speed Tests

Processor cache speeds up access to variables that are often used. As mentioned, Dataspaces can be allocated in Genode with the flag *CACHED* and *UNCACHED*. To test the influence of disabling the cache for dataspaces in a experiment, two types of access patterns will be distinguished:

1. Writing often to the same memory addresses

[1]using `https://github.com/pecheur/genode-Profiler`

2. Writing to a lot of different memory spaces

Those access patterns can be simulated with the following tests:

1. Test 1: Writing to every integer of a Dataspace with the size of 16KiB 1000 times
2. Test 2: Writing to every integer of a Dataspace with the size of 16MiB once

Both test write in total 16*MiB*. These tests can be conducted for cached and uncached dataspaces. Additionally it will be run with the AXI bridge enabled and with *L4 Fiasco.OC/Genode* running on plain memory. Measuring the time needed to perform those test leads to the following results:

Type	Test 1	Test 2
Cached	12*ms*	14*ms*
Uncached	12*ms*	13*ms*

(a) Main memory enabled

Type	Test 1	Test 2
Cached	15*ms*	841*ms*
Uncached	917*ms*	972*ms*

(b) AXi peripheral enabled

Table 6.1.: Test 1 and Test 2 executed on the Zybo board with cached and uncached dataspaces

As seen in Table 6.1 cached and uncached memory space of 16MiB can be written to in approximately 10*ms*. As soon as the AXI bridge is enabled the duration rises to around 900*ms*,except for Test 1 in the cached experiment, leading to a decrease in performance with a factor of 90.

6.3.2. Copy Memory Tests

An additional test which compares the performance of copying Dataspaces shows similar results. For this test two dataspaces are created with a size of 16 *MiB*. All 16 *MiB* of *Dataspace 1* will be written with data. Afterwards is the data copied to *Dataspace 2*. The duration of both will be measured. This simulates the behavior of checkpointing dataspaces developed by Huber [Hub16].

This test will be launched again for cached and uncached dataspaces as well as *L4 Fiasco.OC/Genode* utilizing the main memory and the AXI peripheral. For the latter, an additional test will be launched where a translation is set between two additional uncached Dataspaces (*Dataspace 5* and *6*). *Dataspace 5* will be written with memory. The copying to *Dataspace 6* happens implicitly within the AXI peripheral. This leads to the following results:

Type	Write	Copy
Cached	14ms	88ms
Uncached	13ms	128ms

(a) Main memory enabled

Type	Write	Copy
Cached	838ms	2103ms
Uncached	921ms	1938ms
AXI-Perihperal	1750ms	

(b) AXi peripheral enabled

Table 6.2.: Writing and copying memory with enabled and disabled AXI peripheral as well as a memory translation

It can be shown again, that the decrease of performance is not caused by disabling the cache, but rather the use of the AXI peripheral instead of the main memory with an increase in write time of a factor of 90 (comparison of left and write table). However, the included "copy on write" of the AXI peripheral is faster than copying after writing if this is performed as well with an enabled AXI bridge,approximately 1700ms compared to approximately 2900ms.

6.4. Technical Restrictions

6.4.1. Unaligned memory access

Basically all modern processor architectures support aligned memory access. There are just some Arm-Instruction-Set architectures that do not.[Sea01]
Unaligned memory access happens when a process accesses data over the memory bus with a starting address, which is not align with respect to the offset regarding the length of the width of the bus architecture (noticeably 32 or 64 bit).[2]
Unaligned memory accesses lead to so called "bus errors". This means, that a memory read or write accesses stated by a program using the necessary machine instruction could not be fulfilled. This either leads to a processor exception which can be handled by the program or the operating system itself or the program executing the call simple halts.[3][Cor05]
Whether a processor is capable of accessing unaligned memory accesses can either be identified by the processors documentation or tested by an example program (see listing 6.2).

[2]see `https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt`

```c
 1 #include <stdio.h>
 2 #include "platform.h"
 3 #include "xil_printf.h"
 4
 5 int main()
 6 {
 7     init_platform();
 8     printf("Start␣write\n\r");
 9     volatile int *i = 0x51500000;
10     *i = 42;
11     printf("Write␣done\n\r");
12     cleanup_platform();
13     return 0;
14 }
```

(a) Aligned memory accesses

```c
 1 #include <stdio.h>
 2 #include "platform.h"
 3 #include "xil_printf.h"
 4
 5 int main()
 6 {
 7     init_platform();
 8     printf("Start␣write\n\r");
 9     volatile int *i = 0x51500001;
10     *i = 42;
11     printf("Write␣done\n\r");
12     cleanup_platform();
13     return 0;
14 }
```

(b) Unaligned memory accesses

Figure 6.2.: Aligned memory access (on the left) starts at memory 0x5150_0000, therefore aligned to the 32bit (4 byte) bus. Unaligned memory accesses (on the right) starts at memory address 0x5150_0001. The program on the left executes successfully while the program on the right halts during the write in Line 9.

The Zynq-7000 architecture distinguishes between two major types of memory[Xil18, P. 70]:

1. *Device and Strongly Ordered*
2. *Normal Memory*

The developed AXI peripheral is in the category of *Device and Strongly Ordered*. As mentioned in the documentary it is not able to perform unaligned memory accesses [Xil18, P. 70]. The main-memory however is able to perform unaligned accesses.

In general operating systems and other software modules should be able to cope with architectures not supportive of this feature by the use of specific compiler features such as *-mno-unaligned-access* for [4]. The compiler can deal with forcefully packed memory structures by splitting memory accesses. Nevertheless, with explicit unaligned accesses (as in listing 6.2b) can not be dealt with by the compiler.

[4]`https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html`

6.4.2. Restricted AXI Memory Spaces

It can be shown that if an AXI device is mapped to the physical memory starting at 0x4000_0000 a specific number of bytes (until *0x4001_FFFF*) can not be accesses via memory. This has already been mentioned by W. Li et al. [LDP15]. However, they did not state any reason or citation for this behaviour. Nevertheless, it can be verified that writing to memory addresses smaller than *0x4001_FFFF* will lead to a bus error. Memory transaction requested will no longer be answered by the AXI peripheral. Therefore, limiting the operating system to start in another, higher memory area (see 5.3.1).

7. Discussion

7.1. Interpretation of Results

As shown in Chapter 4.2 certain requirements need to be hold to accomplish a successful use of the developed component. All of these requirements can be verified interpreting the results shown in Chapter 6. A working example showed that memory copying can be processed without interrupting the memory access to a secondary component even though the components state can be kept coherently, therefore fulfilling requirement 1.

Additionally the translation can be set from one dataspace to another one. As long as the checkpointing component is aware about the physical address of the components to be checkpointed it is able to control the tracing and copying process (requirement 2).

Furthermore, the developed driver (see Chapter 5.3.3) allows external control over which dataspaces are traced, when exactly a copying is supposed to be paused, and where it is supposed to be taking place (requirement 3).

Lastly, Chapter 6.2 shows, that the amount of overhead can be calculated precisely, which therefore shows, that real-time prerequisites can be stated and trustfully fulfilled (requirement 4).

In summary it can be shown that all requirements can be kept. Therefore the functionality of the components can be agreed upon. Nevertheless do significant performance decrease (see Chapter 6.3) show significant setbacks. Furthermore. those performance problems seem not as firstly assumed to arise from the disabled caching, but from the AXI bridging mechanism itself. Using the formula developed in 6.2 it can be shown, that the AXI-bus is not able to perform faster write speeds with the current settings.

Using $100Mhz$ of AXI bus clock frequency and 20 clock cycles per transactions in the AXI bridge as it was measured (in Chapter 6.2) and assuming the $16\,MiB = 2^2 4B$ written of Test 2 in Chapter 6.3.1 equates to the following:

$$T = \frac{20 \cdot transactions^{-1} \cdot (2^{24}B/4B) \cdot transactions}{100 \cdot 10^6 \cdot s^{-1}} = 838.8ms \tag{7.1}$$

$838.8ms$ is very similar to the results shown in Test 2 with activated AXI peripheral, if the data is routed over the peripheral and not just the cache.

Also do certain drawbacks such as the incapability of dealing with unaligned memory accesses (see Chapter 6.4.1) show possible restrictions in the possible field of application, due to being potential security hazards (malicious programs could forcefully implements unaligned accesses to provoke a bus error). Furthermore, the combination of having a restriction in AXI peripheral access space (see Chapter 6.4.2) as well as the simplified memory translation via logic operation in the AXI memory bridge limits the available space of the full main memory (see Chapter 5.2.5) (which can be seen in Figure 5.8 of Chapter 5.3.1).

At the current level, the drawbacks shown limit therefore the component's capability considerably.

7.2. Possible Integration in an Existing Real-Time Checkpoint Restore Component (RTCR)

.

7.2.1. Checkpointing of Full Genode Components

Even though the component traces and copies memory for a checkpoint and restore process, several steps are still to be made to allow a Genode process to be checkpointed and restored. As developed by Huber [Hub16], there is a mechanism to checkpoint and restore full Genode processes.

Especially the standalone component *RTCR* is responsible for targeting and checkpointing a process. Therefore the *RTCR* component utilizes a class called *Rtcr::Target_child* which can be created to encapsulate another component (as seen in Listing 7.1 in Line 37), setting up the programs' heap and services:

Listing 7.1: rtcr_restore_child/main.cc

```
37    Target_child child { env, heap, parent_services, "sheep_counter", 0 };
38    child.start();
39
40    timer.msleep(3000);
41
42    Target_state ts(env, heap);
43    Checkpointer ckpt(heap, child, ts);
44    ckpt.checkpoint();
45
```

[0]https://github.com/argos-research/genode-CheckpointRestore-SharedMemory

```
Target_child child_restored { env, heap, parent_services, "
    sheep_counter", 0 };
Restorer resto(heap, child_restored, ts);
child_restored.start(resto);
```

Rtcr::Target_child utilizes custom interceptors to log the RPC points (see chapter 5.3.3) of the encapsulated target (see Figure 7.1).

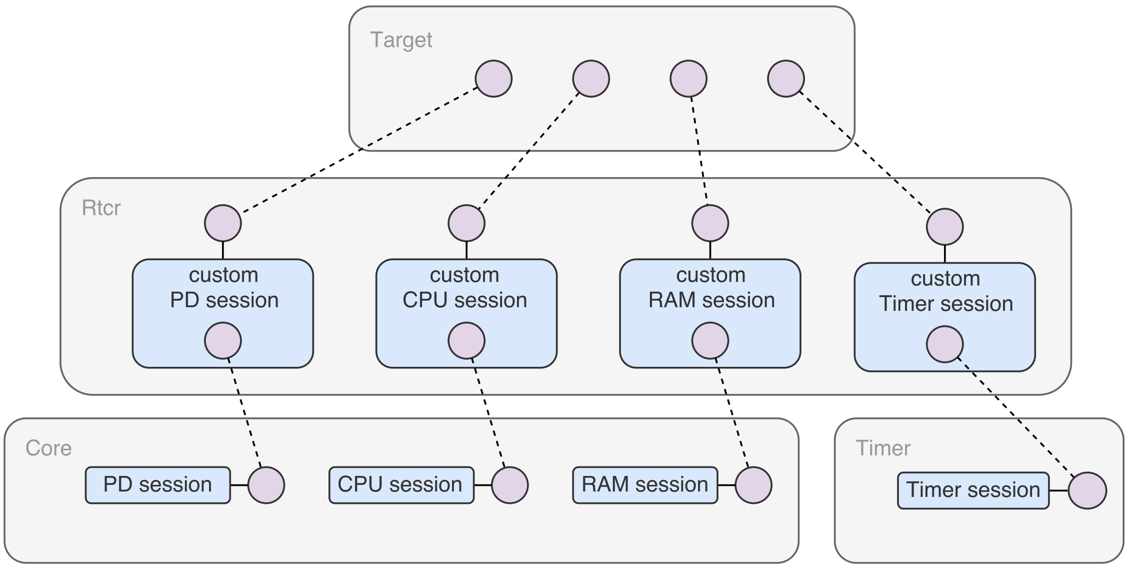


Figure 7.1.: Intercepting of RPC connections and therefore capabilities via *Rtcr::Target_child* and intercepting session components as shown by Huber [Hub16, P. 31]

7.2.2. Current state of Capability Checkpointing

The following capabilties are supposed to be included in the checkpoint/restore process:

Capability Name	Checkpointed via
CPU	*Rtcr::Checkpointer::_prepare_cpu_sessions* [Hub16]
LOG	*Rtcr::Checkpointer::_prepare_log_sessions* [Hub16]
PD	*Rtcr::Checkpointer::_prepare_pd_sessions* [Hub16]
RAM	*Rtcr::Checkpointer::_prepare_ram_sessions* [Hub16]
RM	*Rtcr::Checkpointer::_prepare_rm_sessions* [Hub16]
Timer	*Rtcr::Checkpointer::_prepare_timer_sessions* [Hub16]
Signal Source	*Rtcr::Checkpointer::_prepare_signal_sources* [Hub16]
Signal Context	*Rtcr::Checkpointer::_prepare_signal_context* [Hub16]
Rom	Interceptor available but not checkpointed [Hub16]
Exit	Not yet checkpointed [Joo18, P. 43]
Signal Proxy	Not yet checkpointed [Joo18, P. 44]
Semaphore	Not yet checkpointed [Joo18, P. 45]
Constructor	Not yet checkpointed [Joo18, P. 46]

Table 7.1.: List of Capabilities and where they are checkpointed by the *RTCR* component

To checkpoint further capabilties it is necessary to create an interceptor as well. An interceptor mimics the functionality of the real capability via introducing a wrapper.[Hub16] For the missing Capabilities it would be necessary to replace their *Session_component* classes with such an interceptor (e.g. for the *Signal_Proxy* Capability via replacing the class in *repos/base/include/base/entrypoint.h*). Therefore the new class can then store the connection information to the server component during checkpoint.

7.2.3. Possible Integration Points

The developed AXI-peripheral is not yet integrated in the structure of the *RTCR* component Therefore it does not cope with capabilities. Nevertheless can it be integrated into the *RTCR* component.

One method of integration could be the *_checkpoint_dataspace_content* of the *Checkpointer* class (see Listing 7.2).

As seen there in Line 1422 a source Dataspace is copied to a destination Dataspace. This copying could be done implicitly via the AXI peripheral. So a future checkpointing integration of the AXI peripheral could replace the call of *Genode::memcpy* with two Dataspaces mapped before and translated via the AXI peripheral's API. Therefore the *RTCR* component does not have to halt the component as this point to create a checkpoint, but it can rather use the implicitly copied dataspace connected to it.

Listing 7.2: checkpointer.cc

```
void Checkpointer::_checkpoint_dataspace_content(Genode::
    Dataspace_capability dst_ds_cap,
    Genode::Dataspace_capability src_ds_cap, Genode::addr_t dst_offset,
        Genode::size_t size)
{
  if(verbose_debug) Genode::log("Ckpt::\033[33m", __func__, "\033[0m(dst␣"
      , dst_ds_cap,
      ",␣src␣", src_ds_cap, ",␣dst_offset=", Genode::Hex(dst_offset),
      ",␣copy_size=", Genode::Hex(size), ")");

  char *dst_addr_start = _state._env.rm().attach(dst_ds_cap);
  char *src_addr_start = _state._env.rm().attach(src_ds_cap);

  Genode::memcpy(dst_addr_start + dst_offset, src_addr_start, size);

  _state._env.rm().detach(src_addr_start);
  _state._env.rm().detach(dst_addr_start);
}
```

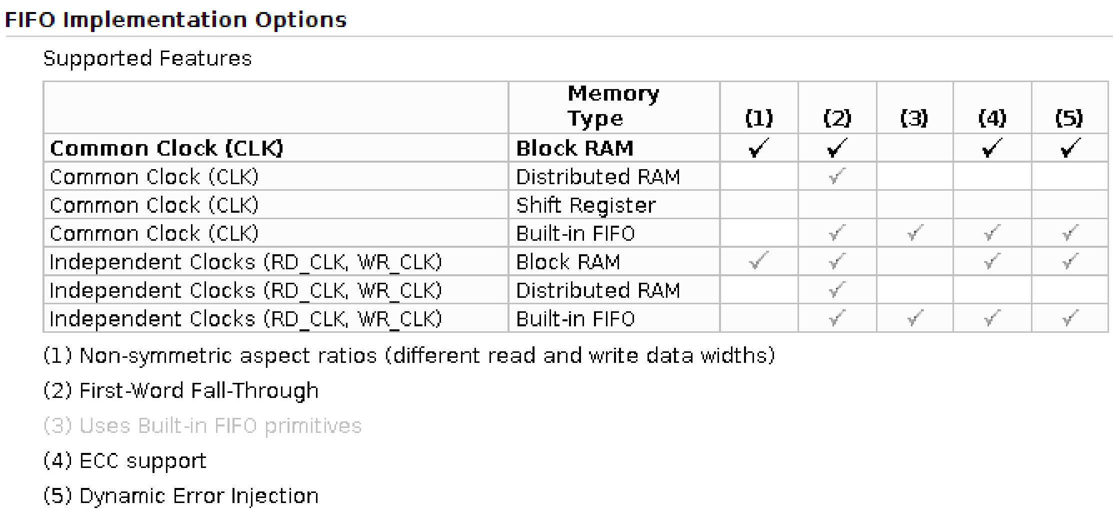
8. Future Work

This chapter shows future areas of improvements that could be done, to achieve higher metrics regarding performance and usability.

8.1. Possible Design Improvements

8.1.1. Replacement of the FIFO Queue

Instead of a built in FIFO queue the AXI peripheral could be implemented using a "Xilinx FIFO Generator IP" [Xil17b][1]. The main disadvantage of the current FIFO queue is, that it takes space on the built in FPGA (see also Chapter 8.1.2).



FIFO Implementation Options

Supported Features

	Memory Type	(1)	(2)	(3)	(4)	(5)
Common Clock (CLK)	**Block RAM**	✓	✓		✓	✓
Common Clock (CLK)	Distributed RAM		✓			
Common Clock (CLK)	Shift Register					
Common Clock (CLK)	Built-in FIFO		✓	✓	✓	✓
Independent Clocks (RD_CLK, WR_CLK)	Block RAM	✓	✓		✓	✓
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM		✓			
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO		✓	✓	✓	✓

(1) Non-symmetric aspect ratios (different read and write data widths)

(2) First-Word Fall-Through

(3) Uses Built-in FIFO primitives

(4) ECC support

(5) Dynamic Error Injection

Figure 8.1.: Modification Panel of the Xilinx Fifo Generator [2]

As seen in Figure 8.1 the Xilinx FIFO generator allows multiple destinations of storage for the FIFO queue's content, such as the external BRAM or the main memory. One disadvantage is, that as external IP communication with the FIFO generator takes at

[1]`https://www.xilinx.com/products/intellectual-property/fifo_generator.html`
[2]Screenshot from Xilinx Vivado

least one clock cycle (setting the enabling read and getting a response takes minimum one clock cycle), therefore slowing the copying. As seen in chapter 6.2 copying is far quicker concerning the clock cycles in comparison to the memory bridge. Therefore additionally clock cycles might not have an impact on the system.

8.1.2. Switching to Zedboard

Due to the Zybo board having only 17600 LUT cells, the increase of the the size for the memory translation map (see *DMAP* in Chapter 5.2.4) or the built in FIFO queue results in needing more resources than available on the board. This results in the following error:[Inc19b]

Listing 8.1: vivado_output.txt

```
[Place 30-640] Place Check : This design requires more Slice LUTs cells
    than are available in the target device. This design requires 41408 of
     such cell types but only 17600 compatible sites are available in the
    target device. Please analyze your synthesis results and constraints
    to ensure the design is mapped to Xilinx primitives as expected. If so
    , please consider targeting a larger device. Please set tcl parameter
    "drc.disableLUTOverUtilError" to 1 to change this error to warning.
```

However, a Zedboard contains 85,000 logic cells therefore being able to also cope with bigger designs [3].

8.1.3. Improving AXI-based Drawbacks

A improved implementation of the AXI peripheral could cope with the mentioned drawbacks. For example mentions W. Li et al. the deactivation of the L1 and L2 cache [LDP15, Chapter. 3.4]. The processors L3 cache however seem to be kept enabled. Porting this particular behavior to the L4 Fiasco.OC / Genode system however could improve the reduced write performance due to current cache deactivation (compare Chapter 6.1.2).

Additionally does the the Zynq processing system distinguish between AXI peripherals and memory devices (as seen in Chapter 6.4.1). Future implementations might investigate, whether it is possible to transform the developed AXI peripheral component to an AXI memory component.

[3]https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentZedBoard.html

8.2. Possible Speed Improvements

As shown earlier has the developed AXI peripheral a performance that is 90 times worse compared to the operating system using normal main memory. To reduce these disadvantage the following approach could be tried (see Chapter 7.1)

As it was shown there are three parameters which can be modified in order to gain higher write performance:

1. Clock cycles per transaction (needs to be reduced)
2. Bus frequency (needs to be raised)
3. Data transferred per transaction (needs to be raised)

All three parameters can be improved. Clock cycles per transaction is mostly based on the transaction overhead. Until now only the AXI-Lite protocol is used for the AXI peripheral. However, this could be changed for AXI-Full. Even though general peripherals (like network driver) show only moderate improvements for some workloads it is especially for block writes possible to achieve substantial improvements [Mak+17]. The bus frequency of the Zybo board can be put up to $250MhZ$ [4] It can not be foreclosed, that other boards can have higher AXI bus frequencies. Nevertheless can higher frequencies possibly influence the transactions per clock cycle, due to AXI interconnects buffering more content and responding later in the case, that for example the Zynq processing system can not act quickly enough. Data per transaction can be achieved by widening the memory bus. The Zynq processing system allows with the use of high performance slave ports (slave HP0 or HP1) 64 bit bus width.

Assuming the, due to overhead in the AXI handshake, unrealistic metrics of a theoretical maximum of 1 clock cycle per transaction, $250MhZ$ AXI bus frequency and 64 bit bus width (therefore 8 Byte per transaction) leads to the following duration for a $16MiB = 2^{24}B$ write:

$$T = \frac{1 \cdot transactions^{-1} \cdot ({}^{2^{24}B}/{}_{8B}) \cdot transactions}{250 \cdot 10^6 \cdot s^{-1}} = 8ms \tag{8.1}$$

$8ms$ is similar (even faster), than write speeds shown with disabled AXI peripheral (see Chapter 6.3.1). However, those speeds are a theoretical maximum which can not be achieved (especially not the amount of clock cycles per transaction).

Nevertheless can the shown attempts be implemented to improve performance.

[4] According to the settings possible in Xilinx Vivado.

9. Conclusion

In summary has the following be shown. Hardware assisted memory tracing and copying provides methods that can support existing checkpoint restore mechanisms (see Chapter 3.2.2) to cope with certain drawbacks (see Chapter 3.2.3). It can be applied in a wide variety of fields (Chapter 1) and supports strong real-time requirements (see Chapter 7.1).

Even though the AXI peripheral based implementation has drawbacks such as reduction memory space as well as problems for safety critical implementations (see Chapter 6.4.2 and 6.4.1) it can be a useful addition for paralleling a memory copying.

However, the performance decreases have a very negative impact on the overall usefulness of the system. A 90 times decrease in performance is not applicable to most systems. Nevertheless. future approaches possibly could be able to resolve those issues resulting in an overall usable system. Further steps (as seen in Chapter 8.2) need to be done to increase the overall write speed. Nevertheless could it be shown that the theoretical maximum in write speeds does lie in an acceptable range.

Afterwards, is a full integration into a the *RTCR* component (see Chapter 7.2) supposed to allow for a full checkpoint restore mechanism of *L4 Fiasco.OC/Genode* components, therefore being applicable to real life use cases.

Furthermore, the then developed application can be applied for a wide variety of systems, including but not solely:

1. Migration of cloud computing processes
2. Backups of safety critical systems (e.g. in connected cars)
3. Migration of distributed high performance applications
4. Distributed database systems

Therefore this wide variety of use cases shows the applicability of hardware assisted checkpoint restore mechanisms. Conclusively could it bee seen, that hardware assisted memory tracing and copying can be transparently applied to a real-time critical checkpoint restore mechanisms if all its drawbacks are coped with.

A. Appendix - Protocol of Timing Experiments

This attachment shows a the protocol of timing experiments done to measure the clock count of the *m_ram_axi* and *m_copy_axi* state machines following the API guidelines of Chapter 5.2.10.

Listing A.1: experiment1_output.txt

```

Zynq> load mmc 0 ${loadbit_addr} ${bitstream_image}
reading fpga.bin
2083740 bytes read in 128 ms (15.5 MiB/s)
Zynq> fpga load 0 ${loadbit_addr} ${filesize}
zynq_align_dma_buffer: Bitstream is not swapped(1) - swap it
Zynq> mm 0x60000008
60000008: 00000000 ? 0x41500000
6000000c: 00000000 ? 0x41700000
60000010: 00000000 ? 4096
60000014: ffffffff ? Zynq> <INTERRUPT>
Zynq> mm 0x60000000
60000000: 00000000 ? 2
60000004: 00000000 ? Zynq> <INTERRUPT>
Zynq>

Zynq> mm 0x41500000
41500000: 00000000 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>

Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000006 00000001 00000014 00000001 ................
```

```
60000030: 0000000c 00000004 0000019d 0000008a ................
60000040: 00000000 00000000 00000000 00000000 ................


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 0000000c 00000002 00000027 00000002 ........'.......
60000030: 0000000c 00000004 000001d9 0000009e ................
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000012 00000003 0000003a 00000003 ........:.......
60000030: 0000000c 00000004 00000215 000000b2 ................
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
```

```
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000018 00000004 0000004d 00000004 ........M.......
60000030: 0000000c 00000004 00000251 000000c6 ........Q.......
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 0000001e 00000005 00000061 00000005 ........a.......
60000030: 0000000c 00000004 0000028d 000000da ................
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000024 00000006 00000074 00000006 $.......t.......
60000030: 0000000c 00000004 000002c9 000000ee ................
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>

Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
```

```
60000020: 0000002a 00000007 00000087 00000007 *...............
60000030: 0000000c 00000004 00000305 00000102 ................
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000030 00000008 0000009a 00000008 0...............
60000030: 0000000c 00000004 00000341 00000116 ........A.......
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>

Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 00000036 00000009 000000ad 00000009 6...............
60000030: 0000000c 00000004 0000037d 0000012a ........}...*...
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>


Zynq> mm 0x41500000
41500000: 00000001 ? 1
41500004: 00000000 ? Zynq> <INTERRUPT>
Zynq> <INTERRUPT>
Zynq> md 0x60000000 14
60000000: 00000002 00000000 41500000 41700000 ..........PA..pA
60000010: 00004096 ffffffff 00000000 00000001 .@..............
60000020: 0000003c 0000000a 000000c0 0000000a <...............
60000030: 0000000c 00000004 000003b9 0000013e ............>...
```

```
60000040: 00000000 00000000 00000000 00000000 ................
Zynq>
```

List of Figures

List of Tables

Bibliography

[AMB4] A. AMBA. “4 AXI4-Stream Protocol.” In: *ARM* 3 (4), p. 2010.

[ARM11] L. ARM. *AMBA® AXI^{TM} and ACE^{TM} Protocol Specificatio, AXI3 ^{TM} , AXI4 ^{TM} , and AXI4-Lite ACE and ACE-Lite ^{TM}*. 2011.

[But11] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[Cor05] I. B. M. Corporation. *Principles of Operation*. 2005.

[Cor09] U. T. M. Corporation. *FPGA synthesis and implementation (Xilinx design flow)*. `http://www.fpgacentral.com/docs/fpga-tutorial/fpga-synthesis-and-implementation-xilinx`. [Online; accessed 12-Jan-2019]. 2009.

[CRI18] CRIU. *CRIU*. `https://criu.org/Main_Page`. [Online; accessed 12-Jan-2019]. 2018.

[Dar+15] D. L. Darrington, M. W. Markland, P. J. Sanders, and R. M. Shok. *Using accelerators in a hybrid architecture for system checkpointing*. US Patent 9,104,617. Aug. 2015.

[Dic+09] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. “Early experience with a commercial hardware transactional memory implementation.” In: (2009).

[Fes15] N. Feske. *Genode Operating System Framework*. 2015.

[Fur+12] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. McRoberts. *Real-Time UNIX® Systems: Design and Application Guide*. Vol. 121. Springer Science & Business Media, 2012.

[FVS15] S. A. Fahmy, K. Vipin, and S. Shreejith. “Virtualized FPGA Accelerators for Efficient Cloud Computing.” In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 430–435. DOI: `10.1109/CloudCom.2015.60`.

[Geh+16] W. Gehrke, M. Winzker, K. Urbanski, and R. Woitowitz. *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. Springer-Verlag, 2016.

[Gio+05] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers." In: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Nov. 2005, pp. 9–9. DOI: `10.1109/SC.2005.76`.

[Ham+09] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji. "A comparative study on ASIC, FPGAs, GPUs and general purpose processors in the O (Nˆ 2) gravitational N-body simulation." In: *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. IEEE. 2009, pp. 447–452.

[HBB04] M. Huebner, T. Becker, and J. Becker. "Real-time LUT-based Network Topologies for Dynamic and Partial FPGA Self-reconfiguration." In: *Proceedings of the 17th Symposium on Integrated Circuits and System Design*. SBCCI '04. Pernambuco, Brazil: ACM, 2004, pp. 28–32. ISBN: 1-58113-947-0. DOI: `10.1145/1016568.1016583`.

[Hub16] D. Huber. *Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode*. 2016.

[Inc17] X. Inc. *Vivado Design Suite - User Guide Implementation*. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug904-vivado-implementation.pdf`. [Online; accessed 12-Jan-2019]. 2017.

[Inc19a] D. Inc. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. `https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/`. [Online; accessed 2-Jan-2019]. 2019.

[Inc19b] D. Inc. *Zybo Zynq-7000 ARM/FPGA SoC Trainer Board*. `https://store.digilentinc.com/zybo-zynq-7000-arm-fpga-soc-trainer-board/`. [Online; accessed 2-Jan-2019]. 2019.

[Inc19c] X. Inc. *FPGA Bitstream*. `https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html`. [Online; accessed 12-Jan-2019]. 2019.

[Inc19d] X. Inc. *Intellectual Property*. `https://www.xilinx.com/products/intellectual-property.html`. [Online; accessed 12-Jan-2019]. 2019.

[Inc19e] X. Inc. *Zynq-7000 SOC*. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. [Online; accessed 12-Jan-2019]. 2019.

[Joo18] L. Joos. *Extending L4 Fiasco.OC/Genode OS by Capability Checkpoint/Restore*. 2018.

[Jos18] S. Josef. *Development of a Redundant Memory Based Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode*. 2018.

[KR07] I. Kuon and J. Rose. “Measuring the Gap Between FPGAs and ASICs.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 203–215. ISSN: 0278-0070. DOI: 10.1109/TCAD.2006.884574.

[Kra+09] S. Kraemer, R. Leupers, D. Petras, and T. Philipp. “A checkpoint/restore framework for SystemC-based virtual platforms.” In: *System-on-Chip, 2009. SOC 2009. International Symposium on*. IEEE. 2009, pp. 161–167.

[LDP15] L. W. Li, G. Duc, and R. Pacalet. “Hardware-assisted memory tracing on new SoCs embedding FPGA fabrics.” In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 461–470.

[Lee08] E. A. Lee. “Cyber physical systems: Design challenges.” In: *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, pp. 363–369.

[LKN14] V. Leis, A. Kemper, and T. Neumann. “Exploiting hardware transactional memory in main-memory databases.” In: *2014 IEEE 30th International Conference on Data Engineering (ICDE)*. IEEE. 2014, pp. 580–591.

[LNP90] K. Li, J. F. Naughton, and J. S. Plank. “Real-time, Concurrent Checkpoint for Parallel Programs.” In: *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*. PPOPP ’90. Seattle, Washington, USA: ACM, 1990, pp. 79–88. ISBN: 0-89791-350-7. DOI: 10.1145/99163.99173.

[Lov05] R. Love. *Linux-Kernel-Handbuch*. 2005.

[Mak+17] M. Makni, M. Baklouti, S. Niar, and M. Abid. “Performance Exploration of AMBA AXI4 Bus Protocols for Wireless Sensor Networks.” In: *Computer Systems and Applications (AICCSA), 2017 IEEE/ACS 14th International Conference on*. IEEE. 2017, pp. 1163–1169.

[MME04] R. Melhem, D. Mossé, and E. Elnozahy. “The interplay of power management and fault recovery in real-time systems.” In: *IEEE Transactions on Computers* 53.2 (2004), pp. 217–231.

[Pla+94] J. S. Plank, M. Beck, G. Kingsley, and K. Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[PS01] P. Pillai and K. G. Shin. “Real-time dynamic voltage scaling for low-power embedded operating systems.” In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 89–102.

[Rei18] A. Reisner. *Finalization of an Existing MigrationExecution Process and CorrespondingPerformance Evaluation with Regard to Real-Time Behaviour*. 2018.

[Ros14] R. Rosen. "Linux containers and the future cloud." In: *Linux J* 240.4 (2014), pp. 86–95.

[Sch17] Q. Schweigert. *Development of a Multicore-Supported Incremental Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode*. 2017.

[Sea01] D. Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[SSK92] D. B. Stewart, D. E. Schmitz, and P. K. Khosla. "The Chimera II real-time operating system for advanced sensor-based control applications." In: *IEEE Transactions on Systems, Man, and Cybernetics* 22.6 (1992), pp. 1282–1295.

[Tan09] A. S. Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[TW87] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[Vas+11] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman. "Comparing different approaches for incremental checkpointing: The showdown." In: *Linux Symposium*. 2011, p. 69.

[Vog+15] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida. "Speculative memory checkpointing." In: *Proceedings of the 16th Annual Middleware Conference*. ACM. 2015, pp. 197–209.

[Wat+14] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCES, 2014.

[Wer16] D. Werner. *Porting an existing linux-based Checkpoint/Restore Mechanism (CRIU) to L4 Fiasco.OC/Genode*. 2016.

[Wit88] L. D. Wittie. *Debugging distributed C programs by real time reply*. Vol. 24. 1. ACM, 1988.

[WV01] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[Xil11] A. Xilinx. "Reference Guide, UG761 (v13. 1)." In: *URL http://www. xilinx. com/support/documentation/ip documentation/ug761 axi reference guide. pdf* (2011).

[Xil15] A. Xilinx. "DMA v7. 1, LogiCORE IP Product Guide, Vivado Design Suite, 2015." In: *URL http://www. xilinx. com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma. pdf* (2015).

[Xil16] Xilinx. *7 Series FPGAs Configurable Logic Block - User Guide*. 2016.

[Xil17a] Xilinx. *AXI Interconnect v2.1 - LogiCORE IP Product Guide*. 2017.

[Xil17b] Xilinx. *FIFO Generator v13.1 - LogiCORE IP Product Guide*. 2017.

[Xil17c] Xilinx. *Processing System 7 v5.5 -LogiCORE IP Product Guide*. 2017.

[Xil18] I. Xilinx. *Zynq-7000 SoC, Technical Reference Manual*. Xilinx, 2018.

[Xil19] Xilinx. *Vivado Design Suite*. `https://www.xilinx.com/products/design-tools/vivado.html`. [Online; accessed 2-Jan-2019]. 2019.