Optimization of a real-time capable Checkpoint/Restore Mechanism by Hardware Assistance

Johannes Fischer

2019-07-08

This is the final report for an Interdisciplinary Project (IDP) of the master study program computer science at the Technical University of Munich. The topic of this IDP is an optimization of a checkpoint/restore component with a hardware-assisted implementation in order to achieve real-time capability. In course of this IDP, a time analysis of the checkpoint/restore mechanism is carried out, software-based program code partly re-implemented on the Zynq-7000 Programmable SoC and evaluated regarding time consumption.

1. Introduction

The IDP bases on previous theses and work of the Cooperative Integration Architecture for Future Smart Mobility Solutions (KIA4SM) research project at the chair of Operating Systems. This project unifies the communication of Smart Mobility devices and vehicles under a common run-time environment. This environment includes universally applicable Electronic Control Units (ECUs), which are simulated by single-board computers using the Genode OS framework with the underlying Fiasco.OC microkernel. The Fiasco.OC kernel and Genode OS framework were chosen for the project, due to the increased security and reliability in comparison to widely used monolithic kernels, e.g. the Linux kernel.

A focus is the dynamical distribution of tasks on ECUs which requires a migration of processes from one ECU to another running ECU. To provide fault tolerance a Checkpoint/Restore (C/R) design, lent from the desktop and server domain, was implemented. C/R is an activity which allows to save a process and restore it at another time and on another location. Thereby the memory, thread data, and kernel state is stored and restored with the goal of running the process as if nothing happened.

The C/R implementation of KIA4SM supports the approaches of shared memory and redundant memory. The redundant memory attempt duplicates the process memory by applying each memory write to a second redundant memory. The shared memory approach is about pausing the observed process, creating an image of the memory and finally resuming the execution of the process. This approach replaces the continuous load by a punctually time-out, as there are no duplicate write executions, but the process is frozen during the checkpoint creation. Both approaches are software-based and do not yet fulfill the real-time capability. Therefore the optimization with help of hardware-assistance is part of the current research. While another IDP deals with the optimization of the first approach, this IDP focuses on the shared memory approach.

Due to the fact that there is no available co-processor which fulfills the task of memory copying in such a specific domain, inefficient parts of the C/R mechanism are identified and are replaced by a hardware-based implementation on the Zynq-7000 Programmable System on Chip (SoC).

2. Related Work

Data Transfer Optimization in FPGA based Embedded Linux System

In his master thesis, Lipponen evaluates solutions to optimize data transfer from an FPGA to the Linux user space using the Xilinx Zynq-7000 SoCs. A shared DMA buffer based architecture was implemented succesfully. The Linux syscall *mmap* is used to remap a Direct Memory Access (DMA) buffer, which is allocated by a kernel module, to user space. The maximum throughput of the system increased from initial ∼100MB/s to ∼125Mb/s. [14]

Hardware-assisted Memory Tracing on New SoCs Embedding FPGA Fabrics

Li, Duc and Pacelet present a Memory Access Monitor built on the Zynq-based ZedBoard running Linux with all memory accesses routed through a Programmable Logic (PL). Their concept allows to capture the sequence of memory loads/stores of any running application. Not only triggering the record through a characteristic sequence of memory loads/stores, but also dynamically modifying executions of a running application through code injection are possible. Benchmarks showed that the Linux boot sequence and applications performs 5 times slower. [13]

Hardware Acceleration in Genode OS Using Dynamic Partial Reconfiguration

At the time of writing this report, the paper of Dörflinger, Albers and Fiethe are the only one which focuses on Dynamic Partial Reconfiguration (DPR) in combination with the Genode OS platform. Outsourcing multiple tasks to a FPGA will sooner or later be restricted by the available resources of the FPGA. Due to this limitation, only few tasks can be migrated from software to hardware. This paper describes a solution which use DPR to time-share resources of the FPGA by swapping hardware tasks in reconfigurable regions. Therefore it combines the performance gain of hardware acceleration with the flexibility of software tasks. The reconfiguration of a task with a bitstream size of ∼2MiB takes ∼20ms. Due to this overhead, reconfigurations should be scheduled by smart algorithm which minimize the number of reconfigurations. [2]

3. Profiling Techniques

In order to measure the performance improvements of the hardware-accelerated implementations, an adequate solution for profiling and time measurements is required. Beside a hardware-only solution based on DSTREAM from ARM, there are many software-based tools for the Linux platform. But also Genode provides tools for application analysis. These tools are evaluated regarding following requirements:

Call Tree The call tree of functions shall be extracted.

Time Consumption The elapsed time of a function shall be extracted.

Support for Fiasco.OC Syscalls, e.g. provided by Linux, are not available under Fiasco.OC. A porting to the Fiasco.OC microkernel might be time-consuming.

Support for Genode Duo to the hierarchical application layer concept and the absent of a terminal by default, software-based tools might require bigger adjustments.

libc Even if the library *libc* is available in Genode, a dependency should be avoided.

Remote Target The profiling take place on a separate hardware platform, like the Zybo Board. Many measurements will take place and the board will be reset between every run. This asks for a transfer of the gathered results from the target to a host computer before the next run.

3.1. DSTREAM - A Hardware-based Solution

The hardware-based tracing is built around the development board **ZedBoard** (photo 7). It is equipped with a Zynq©-7000 programmable SoC and provides a FMC connection. [23] In combination with the **FMC XM105** break-out card, it can be connected to a DSTREAM. [6] The **DSTREAM** is shipped with the Software **DS-5 Development Studio** and provides a high-performance interface for real-time tracing of not only bare-metal applications, but also applications running in context of an operating system. [3] Especially the total amount of trace buffer can be extended up to 4 GiB by an extra FPGA core which is loaded after powering on the Zedboard. [20] This powerful tooling promises an exact tracing with a minimal performance overhead.

3.1.1. Setup

Figure 3.1 shows the types of data exchanged between the host and the Zedboard. The blue components belong to the tracing, where the FMC XM105 is connected to the FMC

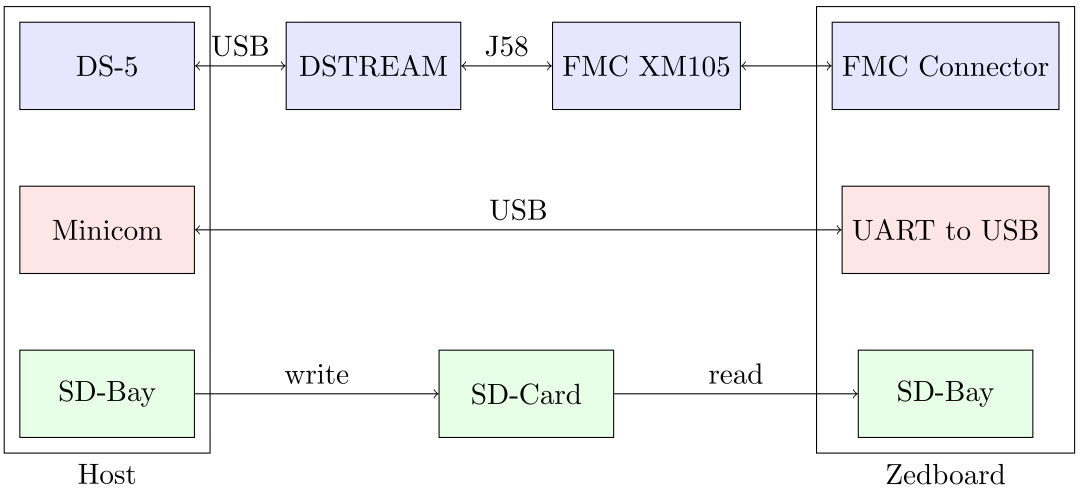


Figure 3.1.: Hardware-based Tracing Setup

Connector of the Zedboard and pass through a JTAG link to the DSTREAM unit. The red connection is an UART connection which is provided by an USB link. Finally, the board reads the boot image from a SD-Card which was previously prepared.

3.1.2. Configuration

Before an application can be traced by DS-5, a new DS-5 Debugger project need to be created and configured. The configuration of a project is divided in several tabs (photo 7).

Debug Symbols In order to recognize the execution of an application at runtime and resolving its debugging symbols, the fixed loading address and a file paths of the unstripped binary are set.

```
add-symbol-file hello_client 0x0115a000
add-symbol-file hello_server 0x0119f000
add-symbol-file init 0x011010000
```

Luckily, Genode always loads an application at a fixed address during boot, which can be extracted from log files generated during boot.

OS Awareness The OS Awareness tab in the configuration menu provides a drop-down menu with several predefined operating-systems which can be selected to improve the tracing experience. If an OS was recognized, OS-related information like currently running processes would be shown during a tracing session. The fact that Fiasco.OC and Genode are not in the list of supported OS is not a problem, as DS-5 is also capable of handling bare-metal applications. But if a support is necessary in future, DS-5 can easily be extended by a custom *OS-Awareness* backend written in Python.

Connection This tab allows to select the used manufacturer, board, project type and debug operation. In this case, the `Xilinx/Zync-7000 SoC/Bare Metal Debug/ Cortex-A9x2 SMP` is selected.

3.1.3. Procedure

1. Start DS-5.
2. Connect to UART with `minicom` in order to receive the boot log of Genode. The alternative application `screen` often lead to freezes in combination with this USB-UART interface. You can also extend DS-5 by a local terminal window and run minicom there.
3. Power on the Zedboard. Due to unknown reasons several characters are printed to the terminal. Press a key in order to open the u-boot console.
4. Flash the tracing core to the FPGA by running following commands in U-boot

   ```
   fatload mmc 0:1 ${scriptaddr} tracing.bit
   fpga loadb 0 ${scriptaddr} 9000000
   ```

5. Back in DS-5, enable the connection in the Debug Control Window.
6. DS-5 is ready to trace. The trace window shows, that the `Buffer Used` is zero. During tracing, the used buffer will increase. In order to start tracing, press the button Start Capture in the trace view.
7. Boot Genode by executing the command `boot` in u-boot.
8. In order to stop tracing, press the Pause Button in the Debug View. Both CPU cores stop. It is not possible to restart the CPU cores. Power off the Zedboard and continue with step 3 again.

3.1.4. Evaluation

DS-5 successfully parse the unstripped binaries and shows all debugging symbols. It is possible to set breakpoints for a selected function before starting a tracing session. After tracing, the Register View is filled and Trace View shows details about the Opcodes at the halted position. Furthermore, several problems occurred during the tracing sessions.

DS-5 is not able to trace executed functions in the Stack View. This view normally provides information about the current function tree on stack of an application. It might be caused by the missing OS-Awareness as Genode comes with a specialized concept of memory management. DS-5 also supports a GNU Debugger (GDB) instance as backend. The missing connection to such a instance could be another reason for the empty Stack View. Finally the DS-5 documentation is not clear about that.

The tracing is often interrupted by the error message *Unable to Connect* in DS-5 and the status LED of DSTREAM turns red. According to the message, it failed to get the list of active threads from the device and the connection is closed. The cause of this error could not be found, but a solution is to power off the target board and set up a new connection in DS-5.
Due to unknown reasons, the Trace View lists many messages regarding *unknown instruction addresses* and *cannot access target memory.* Furthermore the view does not provide the expected information of function calls regarding to timing. But the knowledge about when a function is called and duration of the function call are necessary for the evaluation of this project.

Due to the fact that the hardware-based solution does not deliver the desired results, software-based tools are evaluated in the next chapter.

3.2. Software-based Solutions

In contrast to previously introduced hardware-based tracing method, software solutions implies drawbacks in performance as tracing and the target application share the same resources like CPU and memory. Following list focuses on the most famous and used tools.

Streamline is part of the DS-5 Development Studio and samples the program counter (PC) address at regular intervals to generate a profile of where the processor spends most of the time. Streamline requires an agent to be installed and running on the target. It consists of the kernel module *gator.ko* and relies on the Linux API *perf* and the *proc* directory. [10]

DTrace is short for Dynamic Tracing. The tool targets user-level software, including applications, databases and webservers, but also the operating system kernel and device drivers. A live running target is patched with instrumentation code and information such as a log of the arguments with which a specific function is being called are extracted. DTrace is not supported by Genode. [4]

dynamoRIO is a dynamic binary instrumentation framework. It inserts trampolines into the target application that invoke actions at specific program points. Unfortently, the supported operating systems is limited to Windows, Linux, and Android. [9]

gprof collects and arranges statistics of a target application. Every function of the target is patched at the head and tail to collect timing information. As the tool uses Linux kernel syscalls like `profil(2)` or `setitimer(2)`, it only targets the Linux platform. [11]

Systemtap focuses on kernel profiling, but can also attach to DTrace markers when they are compiled into an application. The scripting language of Systemtap defines subroutines (*handler*) which are executed by the Linux kernel once an event occurs.

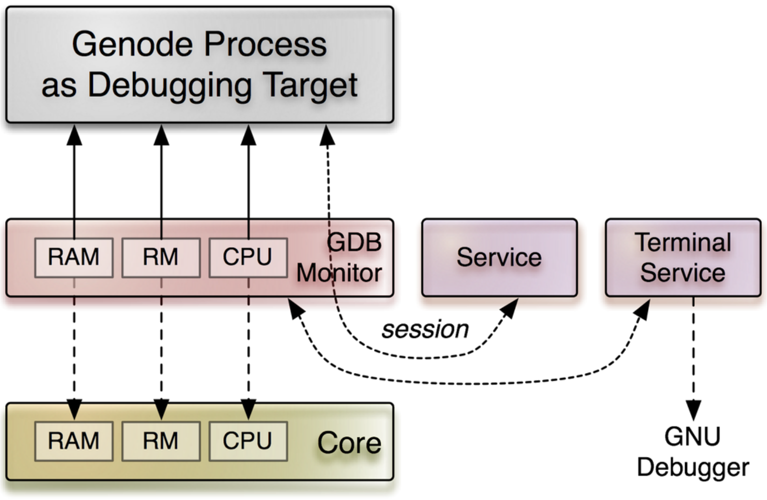


Figure 3.2.: Session Intercept by GDB Monitor

There are several kind of events like entering/exiting a function, timer expiration or session termination. Due to the strong focus on the Linux kernel, there is no support for the Genode platform. [18]

valgrind is an instrumentation framework for building dynamic analysis tools. The target application is temporary translated into a simpler form called Intermediate Representation, manipulated and finally translated back into machine code. The result runs in a virtual machine using just-in-time compilation techniques. Valgrind supports a few profiling techniques but focuses on detecting memory leaks, threading bugs and branch-prediction. [21]

OProfile provides a statistical profiler for Linux systems. It leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics. Not only a single process profiling, but also system-wide profiling is possible. The examples demonstrates a powerful tooling. [16]
Unfortently, the porting to Genode and Fiasco.OC is not a trivial task and requires several adaptations in order to support the requirements.

In summary, all presented tools do not target the Genode platform and mostly depend on the API of the Linux kernel. But as Fiasco.OC does not implement the commonly used Linux API *perf* and does not provide a *proc* directory, no tool is suitable. Therefore following chapter evaluates tracing techniques by Fiasco.OC and Genode.

3.3. Fiasco.OC & Genode-based Solutions

3.3.1. GDB Monitor

As shown in figure 3.2, the GDB Monitor is a Genode process that sits in-between the target and its normal parent. In particular, GDB monitor provides local implementations of the CPU and RM (and ROM) services and redirects session requests by the child component to the real core services. However, by having the target to interact with GDB monitor instead of core directly, GDB monitor gains full control over all threads and memory objects and the address space of the target. [19]
Even if GDB does not directly provide a performance analysis, it can be instrumentalized by the *poor man's profiler* method to achieve the goal. The target is periodically halted and a call stack is produced each time. If all call stacks are combined, a global overview of called functions and their consumed time can be produced. [17]
However, GDB Monitor for Fiasco.OC on ARM did not received the same amount of testing as the x86_32 version and is known to consume a lot of memory. Finally the method is not suitable for measuring the elapsed time of FPGA operations. While a software-based implementation is paused by every halt, the FPGA implementation continues. This would result in a bias if both implementations are compared.

3.3.2. Fiasco.OC Kernerl Debugger

Fiasco.OC is packed with the Fiasco Kernel Debugger (JDB) which supports CPU performance counters. These can also be enabled and controlled by the userland command. [5]

```
#include <l4/sys/kdebug.h>
enter_kdebug("*#");
```

3.3.3. Genode Report Session

A Report Session allows a component to propagate its internal state to the outside. Information are exchanged as XML-formed data and stored in a shared memory dataspace. If data of traced functions are propagated with a report session, another component will be required to transfer these information to the host. Due to this overhead, a Report Session is not efficient for a tracing mechanism. [8]

4. Concept

4.1. Profiler

As no previously introduced tool fulfills all requirements, the library *profiler* is implemented for this IDP.

Like the tool *gprof*, every function is patched at the head and the tail to measure the current time. The newer timestamp is subtracted by the older timestamp which results in the elapsed time of the function. As shown in figure 4.1, the calculated period is passed to the Log Session and printed to a UART device by a UART Session. If the target is emulated in Qemu, the Genode Toolchain will automatically capture the output and writes it to a log file. In case of real hardware, the log is captured with tools like *minicom*, *screen* or *tio*. Due to the fact, that tracing data and logging messages are mixed sent, the tracing information must be extracted from the log file again. The tool *profiler-filter* automatically takes over this process after a successful Genode Run-Script execution and produces a JSON formed *.profile* file. Finally this profile can be parsed by the Python library *profilerlib* which provides a simple API for handling and analyzing the data.

Not only the elapsed time of a traced function is recorded, but also the name of the function and a corresponding color which is taken in account when plotting a graph from the *.profile* file. The Python script *profiler-plot* generates plots with help of the libraries *profilerlib* and *matplotlib*. These can interactive be viewed in a GUI and exported as graphics *(.png, .jpg)*, as vector graphics *(.svg)* and as portable document format *(.pdf)*. The automation of this last process step is also supported by the Genode Run-Script integration of the Profiler.

As previously mentioned, the head and tail of a function need to be patched in order to measure the elapsed time. This can be achieved through following options:

Binary Patching Every function of a binary is patched after compiling. As the injected code depends on a connection to a Timer Session, this option seems to be error-prone and difficult to implement.

GCC Function Instrumentation If the flag *-finstrument-functions* is enabled for the gcc compiler, the functions *__cyg_profile_func_enter* and *__cyg_profile_func_exit* are called whenever a function is entered and exit. Unfortunately, this option results in application freezes when applied on Genode applications. This might be caused

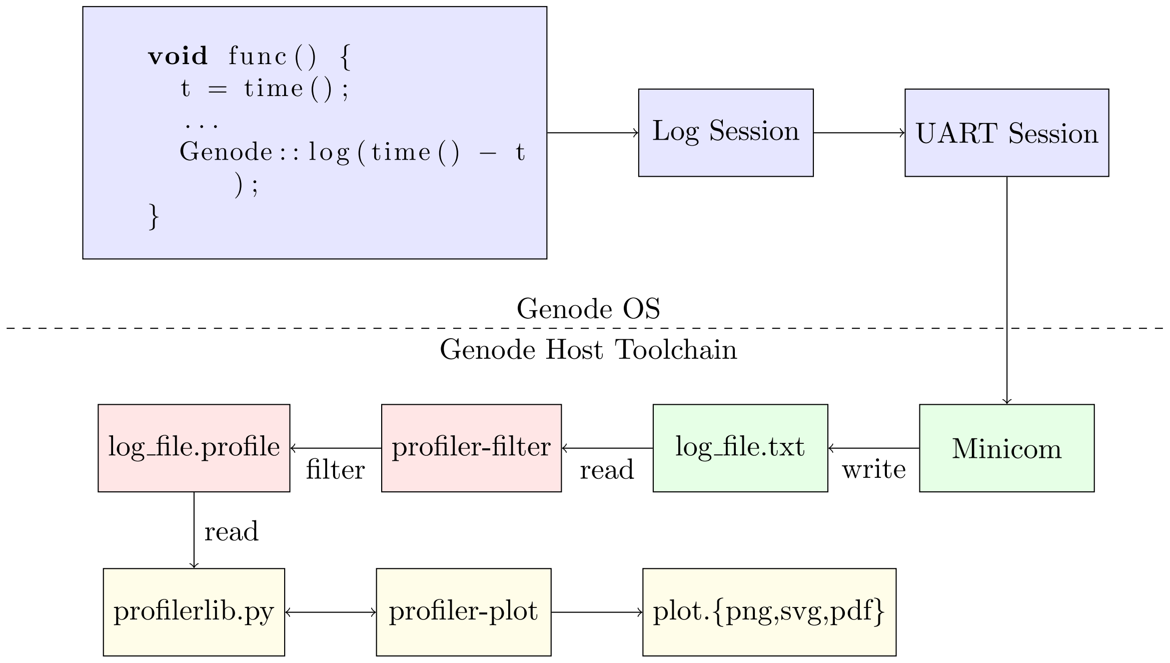


Figure 4.1.: Profiler Data Flow

by recursive calls during the interaction with the Timer Session. Excluding possible recursive functions from the instrumentation with the flag *-fprofile-exclude-files* did not work. [7]

Manual Patching Every function is manually patched in the source code.

The *Profiler* relies on the **Manual Patching** by the developer, but simplifies this procedure. The fact about C++, that objects are destructed when the scope is left, gives a method to automatically do the second time measurement. As shown below, the class *Profile* does the first time measurement in the constructor. The second timestamp is measured in the destructor.

```
void func() {
    new Profile(); // t = time()
    /* function body */
    ~Profile(); // Genode::log(time()-t);
}
```

Finally, the object creation is wrapped in the macro `PROFILE_FUNCTION(color)` which just need to be inserted at the head of a function:

```
void func() {
    PROFILE_FUNCTION("blue")
    /* function body */
}
```

This concept does not only allow to trace functions, but also code fragments by wrapping the code in a scope.

```
{
    PROFILE_SCOPE("blue", "code_fragment")
    /* code fragment */
}
```

4.2. Copying with Direct Memory Access

The Evaluation of the C/R mechanism according to performance revealed that the memory copying is a significant bottleneck. Outsourcing the copy processes to hardware-based implementation with DMA promises an enormous performance gain. Genode implements the software-only function *Genode::memcpy* for copying data. This function is replaced by hardware-based functionality implemented by the Central Direct Memory Access(CDMA) IP Core. This core can be driven in a simple mode and in the faster scather-mode. [1]

4.2.1. Simple Mode

The simple mode of the CDMA core only supports a single memory copying up to a maximum size of 1 MiB. After the copy, the CDMA core must be programmed again for the next copying. Due to this restriction, a dataspace with a bigger size is split into multiple CDMA programming sequences by the CDMA driver (figure 4.2).

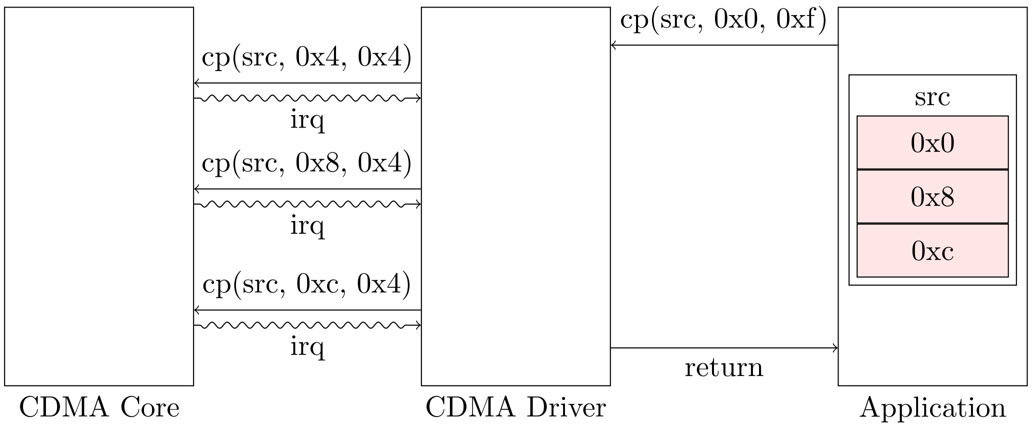


Figure 4.2.: Simple Mode of CDMA Core

The programming sequence for the simple mode consists of following steps:

1. Verify that CDMA core is idle
2. Enable interrupt for a completed transfer
3. Enable interrupt for transfer errors
4. Write desired transfer source address
5. Write desired transfer destination address
6. Write the numbers of bytes to transfer (BTT)

Writing the BTT register also starts the transfer. A interrupted is sent when a transfer is completed or failed. Finally the interrupt bit is cleared in order to announce a successful receiving of the interrupt. The copying is done and the CDMA core is ready for another transfer.

4.2.2. Scather Gather Mode

The multiple reprogramming of the CDMA core, as it is done by the CDMA driver for the simple mode, is shifted into the CDMA core when using the Scather Gather Mode. The driver pre-programs an instruction list of transfer descriptors into a memory-resident data structure and informs the CDMA core about the physical address of the list in memory. A descriptor contains the source address, the destination address and the bytes to transfer. As every transfer descriptor links to the next transfer descriptor in the list, the CDMA core schedules through the list and process the descriptors one by one. As shown in figure 4.3, the benefit of such a mode is the driver-independent transfer of memory areas which also results in a slightly performance gain.

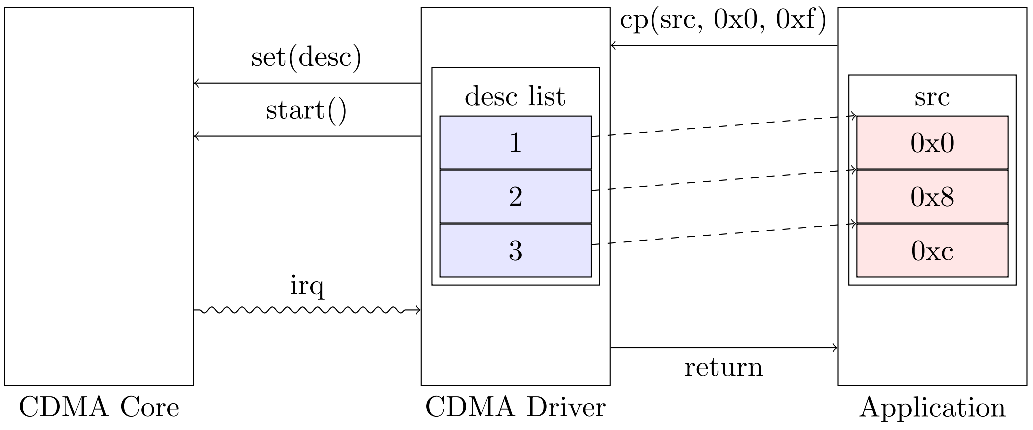


Figure 4.3.: Scather Gather Mode of CDMA

The programming of the CDMA Core in Scather Gather Mode includes following steps:

1. Create a transfer descriptor list
2. Verify that CDMA core is idle
3. Write physical address of the head of the list
4. Enable interrupt for a completed transfer
5. Enable interrupt for transfer errors
6. Write physical address of the tail of the list

With the last step, the channel fetching and processing of descriptors starts. The CDMA scather gather operations continue until the descriptor at the defined tail is processed. Finally the interrupts are set and the engine idles.

4.3. Capability Filter

Beside memory copying, the C/R mechanism only consists of copying small data portions like registers, capabilities and states of sessions. As there are also no CPU-heavy algorithms which can be outsourced to a FPGA, the following *cap_filter* driver was implemented in order to further investigate the possibilities of extracting data from other OS components. The *cap_filter* reads the badge/capability array of the PD Session, filters valid badges and provides an aggregated view to a client components. A similar functionality is implemented in the C/R mechanism which means that this code segment can be replaced by a call to the *cap_filter* driver.

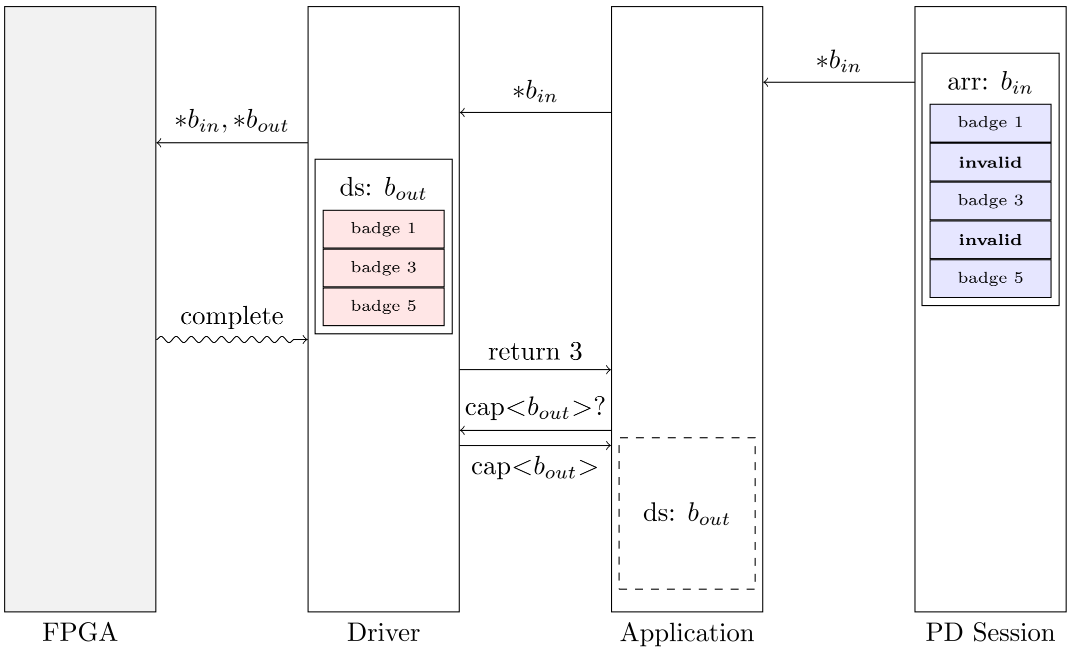


Figure 4.4.: Sequence Diagram of Driver Communication

4.3.1. Driver

The communication of all involved parts is illustrated in figure 4.4. The array of badges b_{in} is located in the **PD Session**. In this case, the **Application** is the C/R mechanism, which communicates with the **PD Session** and the **Driver**. The **Driver** allocates a dataspace b_{out} which acts as container for valid badges. In this chapter, the FPGA component is threaded as black-box, but the next section pays more attention to it. As the previously introduced CDMA core, it provides a memory mapped IO interface with registers for the address of the source array b_{in} and the address of the destination array b_{out}.

1. At first, the **Application** requests the physical address of b_{in} at the **PD Session**.
2. The **Application** calls the **Driver** and passes the address.
3. The dataspace b_{out} is allocated as destination for valid badges
4. The **FPGA** is programmed with the physical addresses of b_{in} and b_{out}
5. The **FPGA** filters valid badges from b_{in} and writes these to b_{out}.
6. An interrupt signals the **Driver** that all badges are processed.
7. The count of valid badges in b_{out} is returned to the calling **Application**.
8. To access b_{out},the **Application** requests the capability of b_{out}.

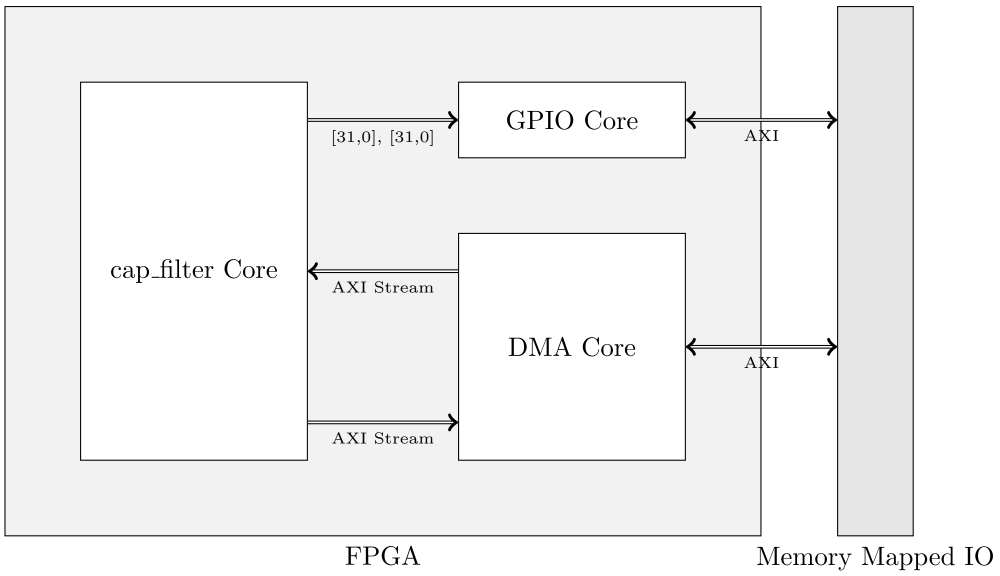


Figure 4.5.: Communication between FPGA Cores

9. The **Driver** shares the access to b_{out}.
10. Finally, the **Application** attaches b_{out} to its own region map and has access to the filtered badges.

4.3.2. FPGA Cores

In comparison to the CDMA core, the task of filtering capabilities is more complex and requires multiple cores (**DMA** & **GPIO**) as well as the custom core **cap_filter**. As shown in figure 4.5, the **DMA** core take over the task of reading memory into an AXI Stream and writing an AXI Stream back to memory. At first, the **DMA** core is programmed by the driver to read the array b_{in} into the outgoing stream. This is connected to the **cap_filter**, which processes the array and writes valid badges to the incoming AXI stream of the DMA core. Finally the DMA core is programmed by the driver to write the data back to memory b_{out}. The current number of valid and totally parsed badges is exported as two 32-width registers. These are provided as memory mapped IO to the driver by the **GPIO** core.

4.3.3. Core cap_filter

The **cap_filter** core is designed in a way, that it supports asynchronous clocks for reading from and writing to the streams. This is mainly due to the fact that the reading operations takes 4 ticks, while the writing operations only last one tick.

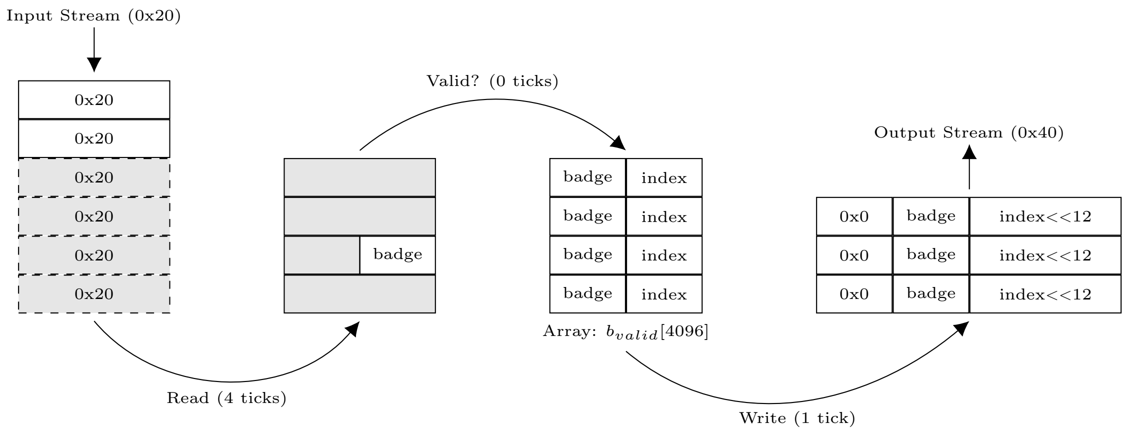


Figure 4.6.: Filtering valid capabilities from AXI Stream

An array b_{valid} of length 4096 is implemented to buffer the reading of valid capabilities. This is necessary, as the number of bytes which are transferred to b_{out} depends on the count of valid capabilities. But the total count is only known, when b_{in} is entirely processed. Worth mentioning is also the fact that the array b_{in} is read with a stream width of 4 bytes, but valid capabilities are written out by a stream with width of 8 bytes. As shown in figure 4.6, the process can be illustrated in three steps.

1. The first step only identifies an element of the array b_{in} and reads the badge. The elements of the array has a length of 16 bytes. The badge is a 16-bit integer and starts at the 6th byte of the element. As reading 4 bytes takes one tick, the whole parsing of an element takes 4 ticks.

2. If a badge is not unused (0x0000) or marked as invalid (0xffff), it is valid and will be stored in the array b_{valid}. This also includes the position of the corresponding element in the array b_{in}. No further ticks are required, as these operations are parallized with the reading of the badge.

3. An element of b_{valid} is written as well formed C-struct to the output stream. This way, b_{out} can be interpreted as an array of C-structs by the driver. The capability of a badge is calculated by shifting the index value by 12 bits. This operation is also parallized and does not require extra ticks.

If a default bandwidth (4 bytes) for the output stream would be chosen, a logic for transferring a C-struct with multiple ticks was necessary. A bandwidth of 8 bytes allows to write a C-struct at once to the stream and bypass the additional logic. The figure also shows a padding of 16 bits at the beginning of every struct. This is necessary, as the C compiler also adds a padding to ensure a correct alignment in memory.

5. Implementation

5.1. Driver

The programming of the **DMA** and **CDMA** cores differ only slightly. Therefore the programming of the **CDMA** core in simple mode is exemplary introduced for the `cap_drv` and `cdma_drv` drivers. Both consist of a definition and a programming of the Memory Mapped I/O (MMIO) interface.

5.1.1. Definition

Genode provides the classes `Attached_io_mem_dataspace` and `Mmio` to simplify a definition of memory-mapped registers. Listing 5.1 shows the necessary registers in order to program the simple mode of the CDMA core. The memory offset and length of a register is documented in the CDMA IP Core manual. The control register `CDMACR` provides several configuration flags for the core. The source and destination addresses are set by the `SA` and `DA` registers. In order to address 64-bit memory, the higher 32 bits of an address can be written to the registers `SA_MSB` and `DA_MSB`. The register `BTT` defines the number of bytes to transfer. [1]

Listing 5.1: Memory-Mapped IO of CDMA Driver

```
struct Cdma::Mmio_cdma : Attached_io_mem_dataspace, Mmio
{
    Mmio_cdma(Genode::Env &env, Genode::addr_t const mmio_address)
    :
    Genode::Attached_io_mem_dataspace(env, mmio_address, 0x32),
    Genode::Mmio((Genode::addr_t)local_addr<void>())
    {}

    struct CDMACR : Register<0x00, 32>
    {
        struct Undefined : Bitfield<0,1> {};
        struct TailPntrEn : Bitfield<1,1> {};
        struct Reset : Bitfield<2,1> {};
        struct SGMode : Bitfield<3,1> {};
        struct Key_Hole_Read : Bitfield<4,1> {};
        struct Key_Hole_Write : Bitfield<5,1> {};
        struct Cyclic_BD_Enable : Bitfield<6,1> {};
        struct IOC_IrqEn : Bitfield<12,1> {};
        struct Dly_IrqEn : Bitfield<13,1> {};
        struct Err_IrqEn : Bitfield<14,1> {};
        struct IRQThreshold : Bitfield<16,8> {};
        struct IRQDelay : Bitfield<24, 8> {};
    };

    struct SA : Register<0x18, 32> {};
    struct SA_MSB : Register<0x1c, 32> {};

    struct DA : Register<0x20, 32> {};
    struct DA_MSB : Register<0x24, 32> {};

    struct BTT : Register<0x28, 32>{};
};
```

5.1.2. Programming

The programming follows the steps introduced in section Simple Mode.

1. In order to program the MMIO, an instance of the previously introduced class `Cdma::Mmio_cdma` is initialized. The location in memory of the MMIO is defined by the variable `cdma_address`. This address is also configured in Vivado and links the AXI Bus of the CDMA core to that location.

```
Mmio_cdma _mmio_cdma = Mmio_cdma(env, cdma_address);
```

2. A signal receiver for the interrupt must be specified. Like the `cdma_address`, the interrupt number `irq_number` depends on the configuration of the FPGA.

```
Genode::Irq_connection _irq = (env, irq_number);
Genode::Signal_receiver sig_rec;
Genode::Signal_context  sig_ctx;
_irq.sigh(sig_rec.manage(&sig_ctx));
_irq.ack_irq();
```

3. The interrupts for complete and failed transfers are enabled

```
_mmio_cdma.write<Mmio_cdma::CDMACR::IOC_IrqEn>(1);
_mmio_cdma.write<Mmio_cdma::CDMACR::Err_IrqEn>(1);
```

4. The source address `src` is written to `SA` and `SA_MSB`

```
_mmio_cdma.write<Mmio_cdma::SA>((uint32_t) src);
_mmio_cdma.write<Mmio_cdma::SA_MSB>((uint32_t) (src<<32));
```

5. The destination address `dst` is written to `DA` and `DA_MSB`

```
_mmio_cdma.write<Mmio_cdma::DA>((uint32_t) dst);
_mmio_cdma.write<Mmio_cdma::DA_MSB>((uint32_t) (dst<<32));
```

6. The count of bytes to transfer `btt` is written to `BTT`. Writing to this register starts the copying process.

```
_mmio_cdma.write<Mmio_cdma::BTT>(btt);
```

7. Wait until an interrupt for a complete transfer is fired by the CDMA core.

```
sig_rec.wait_for_signal();
```

8. If the transfer is successfully, the register `IOC_Irq` will be set. In this case, writing a `1` to this register will notify the hardware about the received interrupt. In order to get triggered by a new interrupt again, the interrupt object `_irq` must be reset.

```
if(_mmio_cdma.read<Mmio_cdma::CDMASR::IOC_Irq>()) {
    _mmio_cdma.write<Mmio_cdma::CDMASR::IOC_Irq>(1);
    _irq.ack_irq();
}
```

5.1.3. Error Handling

If an error occurred during the memory transfer, the register `Err_Irq` is set. In such a case, the driver prints a specific error message to the log and passes an Inter-process Communication (IPC) exception to the calling client. The type of error is specified in detail by following registers:

SGIntErr A internal error has been encountered by the DataMover on the data transport channel.

SGSlvErr AXI slave error response has been received by the AXI DataMover during an AXI transfer.

SGDecErr An AXI decode error has been received by the AXI DataMover. This error occurs if the DataMover issues an address request to an invalid location.

5.2. Block Design

The block design 7.3 shows all involved IP cores, AXI buses and signals of the FPGA implementation.

processing_system7_0 The Processing System 7 core is is the software interface around the Zynq-7000 platform and acts as a logic connection between the Programmable System (PS) and the PL. It provides extended multiplexed I/O, programmable logic I/O and AXI I/O groups. As shown in the block design 7.3, three registers of the interrupt interface *IRQ_F2P* are in use. The high-performance AXI bus *S_AXI_HP0* is used for the memory transfers. The GPIO AXI *M_AXI_GP0* is part of the **cap_filter** logic and introduced later.

rst_ps7_0_100M The System Reset Module core provides customized resets for an entire processor system, including the processor, the interconnect and peripherals. This core is mandatory and linked with the reset signals of all other cores.

ps7_axi_periph The AXI Interconnect core connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices.

axi_smc The AXI SmartConnect core is the successor to the AXI Interconncet core. It is more tightly integrated into the Vivado design environment to automatically configure and adapt to connected AXI master and slave IP with minimal user intervention.

lxconcat_1 The Concat IP core is used for concatenating bus signals of varying widths. In this case, it concats the interrupt signals from **axi_dma_0** and **axi_cdma_0** and passes these to the **processing_system7_0**.

axi_cdma_0 The Central DMA core provides high-bandwidth Direct Memory Access (DMA) between a memory-mapped source address and a memory-mapped destination address. The block design 7.6 shows the directly involved cores for a simple memory copying in orange. In comparation, the block design 7.5 shows that a second AXI connection will be necessary, if the scather-gather mode of the CDMA core is in use.

cap_filter_0 The Capability Filter core filters valid capabilities which are read in from the *S00_AXIS* AXI stream and written out to the *M00_AXIS* stream. As shown in the block design 7.4, both AXI streams are connected to the **axi_dma_0** core.

axi_dma_0 The Direct Memory Access core reads memory and passes it to an AXI4-Stream-type target peripherals. It also reads from an AXI4-Stream and write data back to memory.

axi_gpio_0 The General Purpose Input/Output (GPIO) core provides input/output signals to the AXI interface. This is used in order to map the *total_kcap_count* and *total_kcap_count* signals of the **cap_filter** core as 32-bit integer to memory.

5.3. Core cap_filter

This section focus on the implementation of the **cap_filter** core which is written in Verilog. For the introduced Verilog Code applies, that the code segments are presented without a reset-logic. The full code also contains an input wire for reseting the module to an initial state.

Module Definition

As shown in listing 5.2, the implementation starts with the definition of parameters, input and output wires. The parameters `C_M00_AXIS_TDATA_WIDTH` and `C_S00_AXIS_TDATA_WIDTH` configure the bandwidth of the AXI streams. The names of variables follow the naming guide of Xilinx. The word `AXIS` stands for an AXI Stream. Variables including `M00` are related to the outgoing AXI Stream (Master). Variables related to the incoming AXI Stream contains a `S00` (Slave).
Both streams are defined by a set of signals. The signals *tlast* and *tstrb* are optional and used for burst transfers. As a burst transfer is not supported by this core, the signals are just not defined. This is totally valid and the Vivado Suite is capable of recognizing and handling the missing singals.
Beside the streams, there are two 32-bit registers `total_kcap_count` and `valid_kcap_count`, which exports the number of totally parsed badges and valid badges.

Registers

The internal module state is stored in registers which hold one or multiple bits. The `uint32_counter` counts every fourth byte which is read. The array `kcap_array` stores

Listing 5.2: Module Definition

```
module cap_filter_v1_0
(
    parameter integer C_M00_AXIS_TDATA_WIDTH = 64,
    parameter integer C_S00_AXIS_TDATA_WIDTH = 32,
    parameter integer KCAP_ARRAY_SIZE = 4096,
)
(
    // Ports of Axi Master Bus Interface M00_AXIS
    input wire  m00_axis_aclk ,
    output wire  m00_axis_tvalid ,
    output wire [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata ,
    input wire  m00_axis_tready ,

    // Ports of Axi Slave Bus Interface S00_AXIS
    input wire  s00_axis_aclk ,
    output wire  s00_axis_tready ,
    input wire [C_S00_AXIS_TDATA_WIDTH-1 : 0] s00_axis_tdata ,
    input wire  s00_axis_tvalid ,

    output wire [31 : 0] total_kcap_count ,
    output wire [31 : 0] valid_kcap_count ,
);
```

valid capabilities and corresponding indices. If an element is read from the `kcap_array`, it will be stored to `m00_axis_tdata_reg`. The register `read_kcap_total` stores the total count of read badges. The registers `m00_kcap` and `m00_badge` store the data which are currently written to the outgoing stream. The last two registers represents the current read and write position in the `kcap_array`.

```
reg [31:0] uint32_counter = 0;
reg [31:0] kcap_array[KCAP_ARRAY_SIZE:0];
reg [31:0] m00_axis_tdata_reg;
reg [15:0] read_kcap_total;
wire [15:0] m00_kcap;
wire [15:0] m00_badge;
reg [ADDR_WIDTH:0] read_kcap_index;
reg [ADDR_WIDTH:0] write_kcap_index;
```

Step 1: Reading from Input Stream

Listing 5.3 shows step 1 and 2 of the parsing process. The `assign` clause is instantly evaluated, while the `always` clause from line 3 to 21 is only evaluated on the rise of the clock signal. The `assign` in line 1 makes sure that data are only read, if the `kcap_array` is not exhausted. The code block from line 6 to 20 is executed whenever new data are available in register `s00_axis_tdata`. The seeking for the position in the byte stream

Listing 5.3: Reading & Validating from Input Stream

```
1  assign s00_axis_tready = (read_kcap_index < KCAP_ARRAY_SIZE);
2
3  always @(posedge s00_axis_aclk) begin
4      if(s00_axis_tvalid && s00_axis_tready) begin
5          // every read struct has size of four uint32.
6          if(uint32_counter == 4) begin
7              uint32_counter = 1;
8          end else begin
9              uint32_counter = uint32_counter + 1;
10         end
11
12         if(uint32_counter == 2) begin
13             s00_badge = s00_axis_tdata[31:16];
14             if (s00_badge != UNUSED && s00_badge != INVALID_ID) begin
15                 kcap_array[read_kcap_index[ADDR_WIDTH-1:0]] = {
                       s00_badge, read_kcap_total};
16                 read_kcap_index = read_kcap_index + 1;
17             end
18             read_kcap_total = read_kcap_total + 1;
19         end
20     end
21 end
```

is implemented in lines 6-12. If the correct 4 bytes are identified, the badge will be extracted from `s00_axis_tdata` and will be written to `s00_badge`.

Step 2: Validating Badge

The validation of the badge take place in line 14. If `s00_badge` is valid, it will be stored to `kcap_array`. The `read_kcap_index` is increased to point to the next free array cell. Finally the total count of read badges is incresed in line 18.

Step 3: Writing to Output Stream

Starting with line 1 and 2 of listing 5.4, the total count of read and valid badges are assigned to the outgoing wires. Like in step 1, the writing is syncronized with a clock signal. But instead of the `s00_axis_aclk`, the `m00_axis_aclk` signal is used. This signal belongs to the outgoing stream. Using two separate clock signals allows to asynchronous drive the reading and writing with different clock speeds. Depending on the block design, this might be required sometimes. A successful write operation depends on multiple conditions:

Line 6 The receiver of the outgoing stream notifies, that data can be received. Therefore `m00_axis_tready` must be `1`.

Line 9 The sender of the outgoing stream notifies, that the current data on the stream are valid: `m00_axis_tvalid` (resp. `write`) is set to `1`.

Line 6 New data should be available in `kcap_array`: `write_kcap_index < read_kcap_index`.

Line 14 Data is assigned to `m00_axis_tdata`

If these conditions match, the next unwritten element of `kcap_array` is stored to `m00_axis_tdata_reg`. This register is split into a badge (`m00_badge`) and a capability (`m00_kcap`) part by the assignment in line 15. The C-struct is constructed from these two parts in line 14. The shift of `m00_badge` is implemented by appending twelve `0x0`.

The write process lasts one tick. Due to line 5, the data are invalidated during the next clock rise.

Listing 5.4: Writing to Output Stream

```
 1  assign total_kcap_count = read_kcap_total;
 2  assign valid_kcap_count = read_kcap_index;
 3
 4  always @(posedge m00_axis_aclk) begin
 5      write = 0;
 6      if((write_kcap_index < read_kcap_index) && m00_axis_tready ) begin
 7          m00_axis_tdata_reg = kcap_array[write_kcap_index[ADDR_WIDTH
                -1:0]];
 8          write_kcap_index = write_kcap_index + 1;
 9          write = 1;
10      end
11  end
12
13  assign m00_axis_tvalid = write;
14  assign m00_axis_tdata = { 16'haaaa, m00_badge, 4'h0, m00_kcap, 12'h000
        };
15  assign {m00_badge, m00_kcap} = m00_axis_tdata_reg;
```

6. Evaluation

6.1. Overhead of Profiler

Using software-based profiling techniques always implies an overhead, as the CPU does not only execute the profiled code, but also the code necessary for profiling. Knowing the bias is essential to evaluate the measured results. Especially this is even more important for the *Profiler*, where IPC calls to the Timer, Log Session and UART Session, as well as C-Object construction and destruction are part of the ongoing profiling. In order to measure the overhead, the Genode Repository of the *Profiler* comes with the target **elapsed_time**. In the source code of this application, a profiler instance is nested in a scope which is measured by another profiler instance. The Timer Session provides timestamps with a resolution of 1ms. Due to this fact, short-term measurements should be repeated several times in order to achieve an exact result. Running **elapsed_time** for 500000 times results in an average overhead of 0.9ms on the Zybo Board.

6.2. Profiling Checkpoint/Restore

Figure 7.7 shows the duration of each function which is called during C/R. In this example the test application *sheep_counter* was checkpointed/restored on the Zybo board. Checkpointing takes 5 seconds and restoring is finished in 3 seconds. It appears that duplicating the ram dataspaces `_prepare_ram_dataspaces` and attaching region maps `_prepare_attached_regions` last 50% (2.3 seconds) of the full checkpoint process. It can be assumed that creating and attaching dataspaces comes with a big performance loss. The duplication time of each dataspace during checkpointing could be improved by allocating and attaching the second backup dataspace when the child queries the first one. So the second dataspace already exists, when the child is going to be checkpointed. As long as the child does not allocate additional dataspaces, the time for copying memory takes around 350 ms.
Regarding to the restoring process, the methods `_restore_cap_space` and `_restore_dataspace_content` take the longest with 20%.

6.3. Software- and Hardware-based Memory Copying

As shown in figure 6.1, the hardware-based memory copying approach comes with an performance gain of 176%. Comparing the simple and scather/gather mode of the CDMA driver shows that the performance gain is almost negligible with 2ms. This performance test was also executed on the Zybo Board. The major disadvantage of the

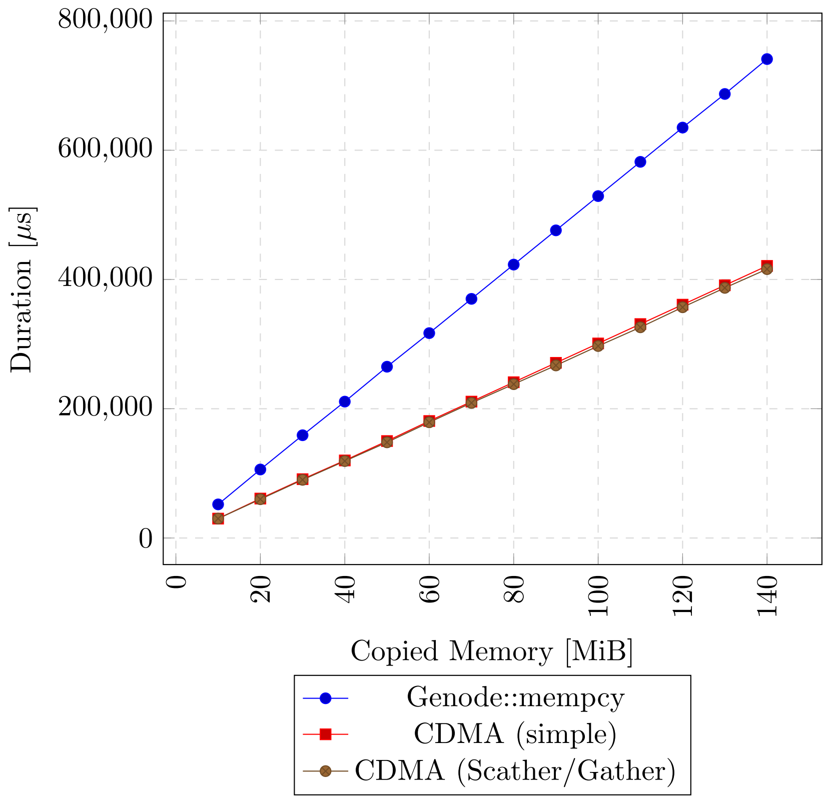


Figure 6.1.: Performance of Software- and Hardware-based Memory Copying

cdma_drv driver is the fact that the source dataspace need to be allocated with parameter `Cache_attribute::UNCACHED`. Without this parameter, cached data in the CPU own cache might not be directly written back to memory. This would lead to incomplete data when copied with *cdma_drv*. Luckily, the `Ram_session` is intercepted by the C/R mechanism and the child application can be supplied with uncached allocations. The performance loss of an uncached allocation in comparison with a cached allocation was not examined.
A dataspace can only be used as destination for CDMA copying, if something was written to it. It seems that Genode keeps track of allocated, but empty dataspaces and does not allocate physical memory until an arbitrary byte was written. Due to that fact, the *cdma_drv* is not a drop-in replacement for the default `Genode::memcpy` command.

6.4. Verification of cap_filter Core

For the verification of the *cap_filter* Core, a so called Vivado Testbench Project was created. As shown in figure 6.2, the *cap_filter* core is threaten as black box module and the input and output signals are connected to a testbench. The test-cases are written in

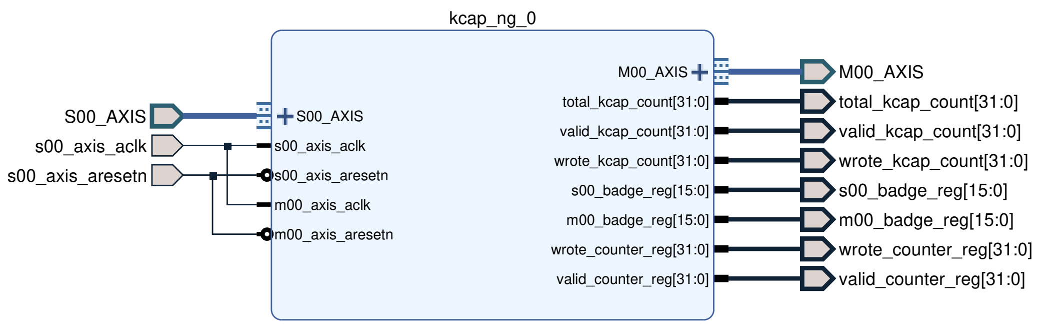


Figure 6.2.: Testbench of cap_filter Core

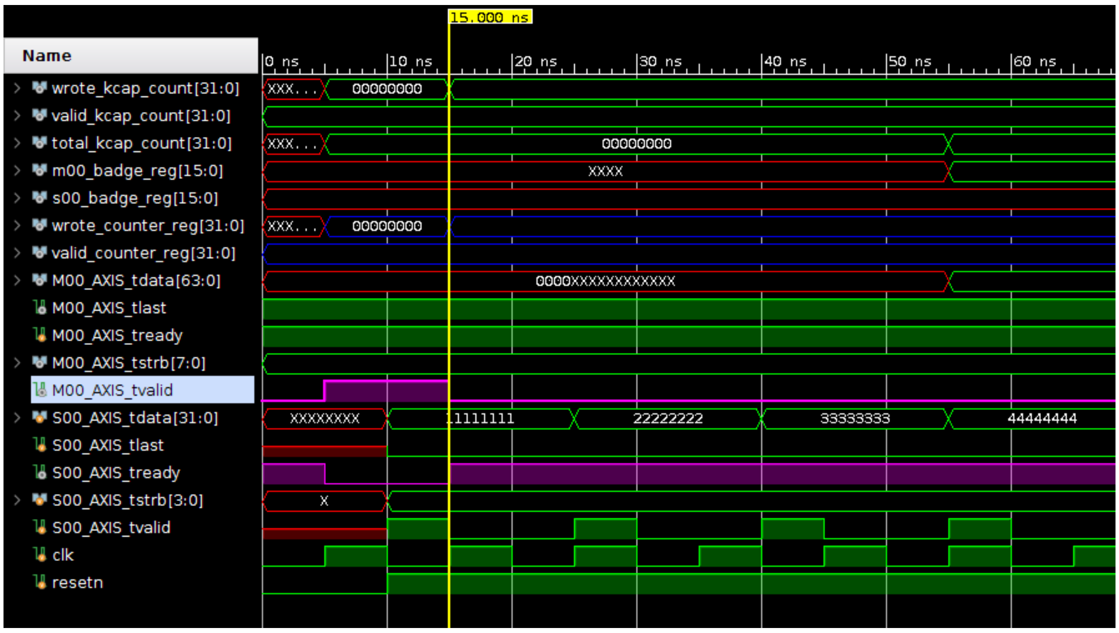


Figure 6.3.: Integrated Logic Analyzer Instance for cap_filter Core

Verilog and simulate the input signals of the black box. The output signals are verified with an Integrated Logic Analyzer (ILA) core. This core can be connected to any AXI bus or wire to monitor internal signals of a design. The GUI of such an ILA instance is shown by figure 6.3.
Beside this black box testing, the driver was also integrated in the C/R mechanism, where it leads to the same results as the software-based implementation.

7. Conclusion

In the beginning of this report, the chapter Profiling Techniques evaluated hardware-based and software-based techniques for profiling applications regarding performance. Not all requirements necessary to profile the C/R could be fulfilled by the introduced tools. Therefore the *Profiler*, a simple solution based on the Timer and Log Session of Genode, is implemented. The overhead is 1ms for each measured scope of program code.
With help of the *Profiler*, a fine-grained timing diagram of all function executions during a C/R could be drawn for the first time. This allows to identify bottlenecks and long-taking operations, like the allocation and attachment of dataspaces. In general, it showed that only starting the required threads of an application already takes four times longer than the targeted real-time capability, which raises the question whether the Genode platform will ever fulfill the real-time requirement at all.

The memory copying is one of the bigger bottlenecks, which is the reason for a hardware-based driver, introduced in chapter Concept. With this driver, a performance gain of 170% is achieved. Nevertheless, this approach can not replace all software-based memory copying as restrictions like the need of uncached dataspaces exist.

In order to evaluate concepts of hardware-accelerated memory reading and manipulation, the *cap_filter* IP core is developed. That the corresponding driver successfully work, does not only show the power of DMA, but also that the security concepts of Genode can be bypassed.

Acronyms

C/R Checkpoint/Restore. 3, 13, 15, 16, 27, 28

DMA Direct Memory Access. 4, 13

DPR Dynamic Partial Reconfiguration. 4

ECU Electronic Control Unit. 3

GDB GNU Debugger. 7, 10

IDP Interdisciplinary Project. 2, 3, 11

IPC Inter-process Communication. 22, 27

KIA4SM Cooperative Integration Architecture for Future Smart Mobility Solutions. 3

MMIO Memory Mapped I/O. 19, 21

PL Programmable Logic. 4, 22

PS Programmable System. 22

SoC System on Chip. 3–5

Bibliography

[1] *AXI Central Direct Memory Access v4.1. LogiCORE IP Product Guide.* URL: `https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf` (visited on 06/23/2019).

[2] Alexander Dörflinger et al. "Hardware Acceleration in Genode OS Using Dynamic Partial Reconfiguration". In: *Architecture of Computing Systems – ARCS 2018.* Ed. by Mladen Berekovic et al. Cham: Springer International Publishing, 2018, pp. 283–293. ISBN: 978-3-319-77610-1.

[3] *DSTREAM - Arm Developer.* URL: `https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream` (visited on 05/09/2018).

[4] *dtrace.org.* URL: `http://dtrace.org/blogs/` (visited on 06/19/2019).

[5] *Fiasco Kernel Debugger Manual.* URL: `http://fiasco.inf.tu-dresden.de/doc/jdb.pdf` (visited on 06/19/2019).

[6] *FMC XM105 Debug Card.* URL: `https://www.xilinx.com/products/boards-and-kits/hw-fmc-xm105-g.html` (visited on 05/09/2018).

[7] *GCC - Manuel:Instrumentation-Options.* URL: `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html` (visited on 06/23/2019).

[8] *Genode OS Framework 16.05 - Book.* URL: `https://genode.org/documentation/genode-foundations-16-05.pdf` (visited on 06/19/2019).

[9] *Github - DynamoRIO.* URL: `https://github.com/DynamoRIO/dynamorio` (visited on 06/19/2019).

[10] *Github - gator.* URL: `https://github.com/ARM-software/gator` (visited on 06/19/2019).

[11] *GNU gprof.* URL: `https://sourceware.org/binutils/docs/gprof/` (visited on 06/19/2019).

[12] Karol Gugala, Aleksandra Swietlicka, and Kolanowski. "Neural controller implementation in embedded system with use of FPGA coprocessor". In: *ISTET 2013: International Symposiumon Theoretical Electrical Engineering.* 2013, pp. 53–54.

[13] Letitia W. Li, Guillaume Duc, and Renaud Pacalet. "Hardware-assisted Memory Tracing on New SoCs Embedding FPGA Fabrics". In: *Proceedings of the 31st Annual Computer Security Applications Conference.* ACSAC 2015. Los Angeles, CA, USA: ACM, 2015, pp. 461–470. ISBN: 978-1-4503-3682-6. DOI: `10.1145/2818000.2818030`. URL: `http://doi.acm.org/10.1145/2818000.2818030`.

[14] Jan Lipponen. “Data transfer optimization in FPGA based embedded Linux system”. MA thesis. Tampere University of Technology, May 2018. URL: `http://dspace.cc.tut.fi/dpub/handle/123456789/26132`.

[15] W. Liu, H. Chen, and L. Ma. “Moving object detection and tracking based on ZYNQ FPGA and ARM SOC”. In: *IET International Radar Conference 2015*. Oct. 2015, pp. 1–4. DOI: `10.1049/cp.2015.1356`.

[16] *OProfile - Examples.* URL: `https://oprofile.sourceforge.io/examples/` (visited on 06/22/2019).

[17] *poo man’s profiler.* URL: `https://poormansprofiler.org/` (visited on 06/19/2019).

[18] *sourceware.org - Systemtap.* URL: `https://sourceware.org/systemtap/` (visited on 06/22/2019).

[19] *User-level debugging on Genode via GDB.* URL: `https://genode.org/documentation/developer-resources/gdb` (visited on 06/19/2019).

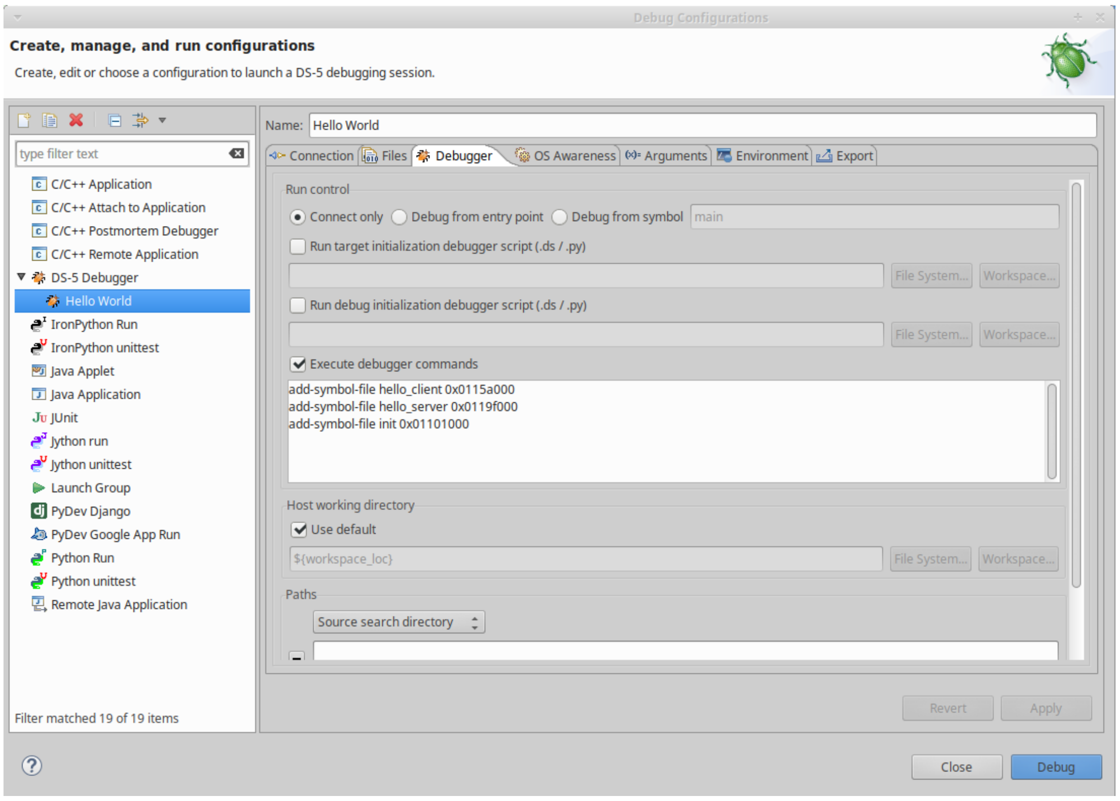
[20] *Using DS-5 with Xilinx Zynq-7000 devices.* URL: `https://developer.arm.com/docs/137812646/latest/using-ds-5-with-xilinx-zynq-7000-devices` (visited on 05/09/2018).

[21] *Valgrind.* URL: `http://valgrind.org/` (visited on 06/22/2019).

[22] Andre Xian Ming Chang and Eugenio Culurciello. “Hardware accelerators for recurrent neural networks on FPGA”. In: May 2017, pp. 1–4. DOI: `10.1109/ISCAS.2017.8050816`.

[23] *Zybo Zynq-7000 ARM/FPGA SoC Trainer Board - Digilent.* URL: `https://store.digilentinc.com/zybo-zynq-7000-arm-fpga-soc-trainer-board/` (visited on 05/09/2018).

Appendix



Create, manage, and run configurations

Create, edit or choose a configuration to launch a DS-5 debugging session.

Filter matched 19 of 19 items

Name: Hello World

Run control

[✓ Connect only] Debug from entry point Debug from symbol main

☐ Run target initialization debugger script (.ds / .py)

☐ Run debug initialization debugger script (.ds / .py)

☑ Execute debugger commands

```
add-symbol-file hello_client 0x0115a000
add-symbol-file hello_server 0x0119f000
add-symbol-file init 0x01101000
```

Host working directory

☑ Use default

${workspace_loc}

Paths

Source search directory

Figure 7.1.: Configuration of Project in DS-5

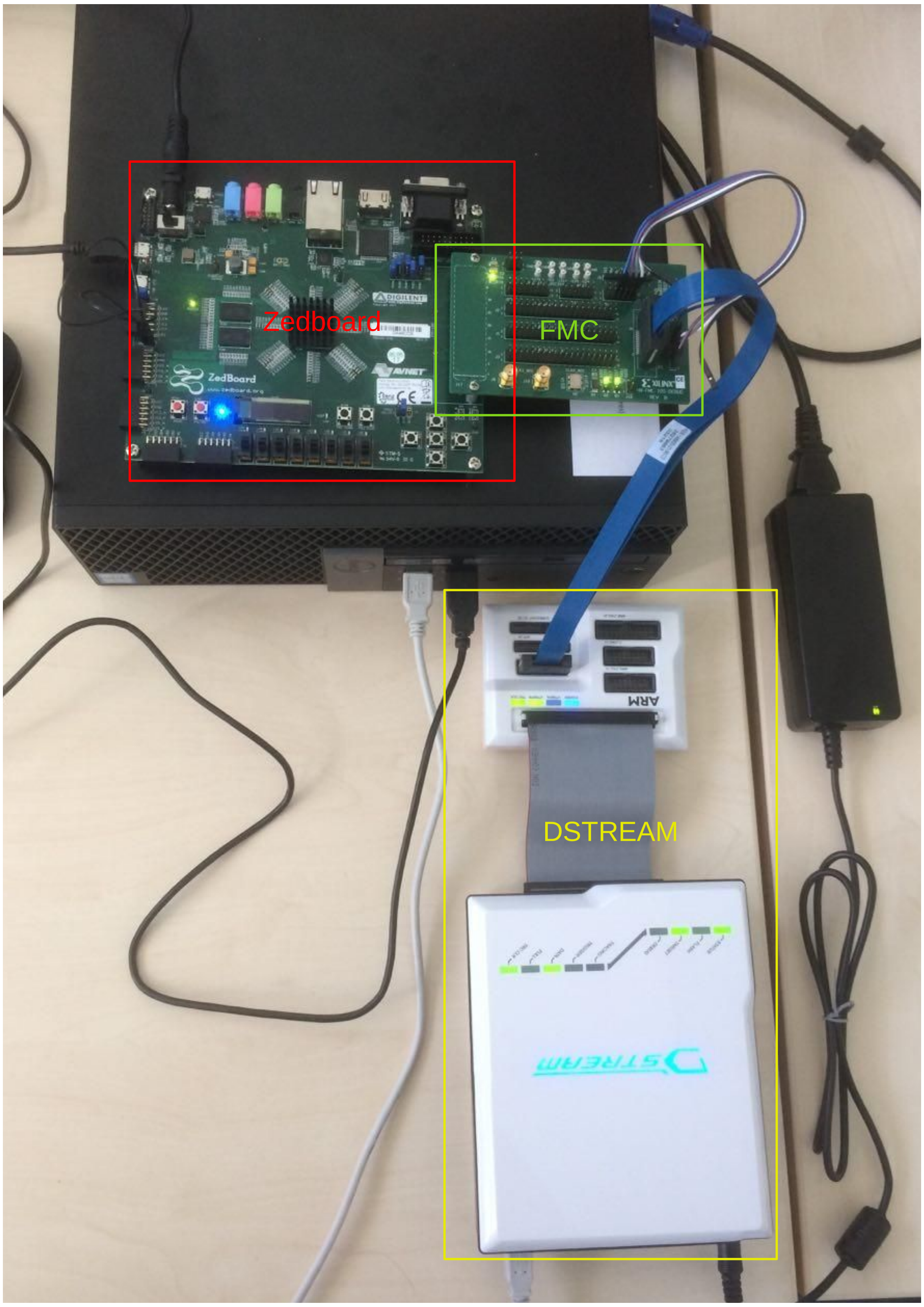


Figure 7.2.: Hardware-based Tracing Setup with DSTREAM

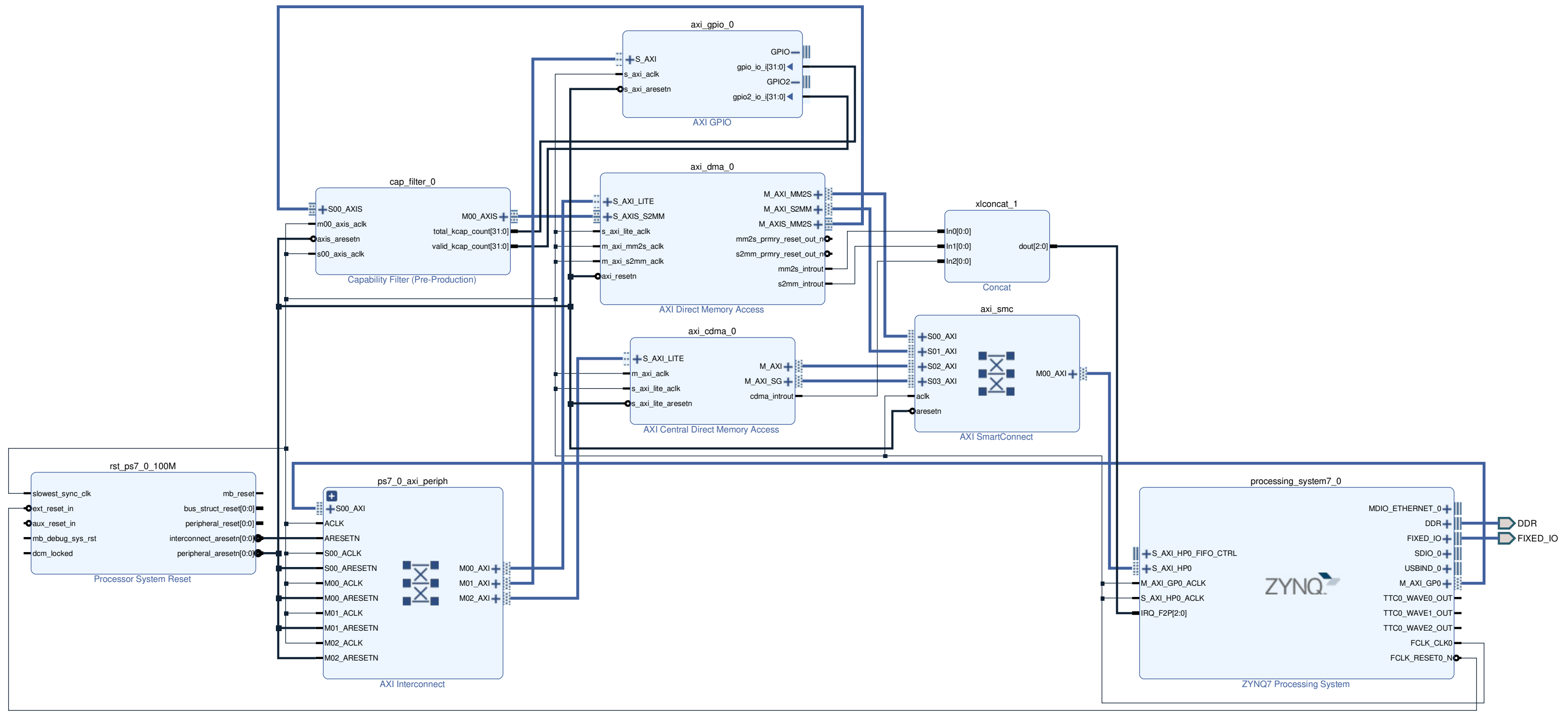


Figure 7.3.: Block design of FPGA component

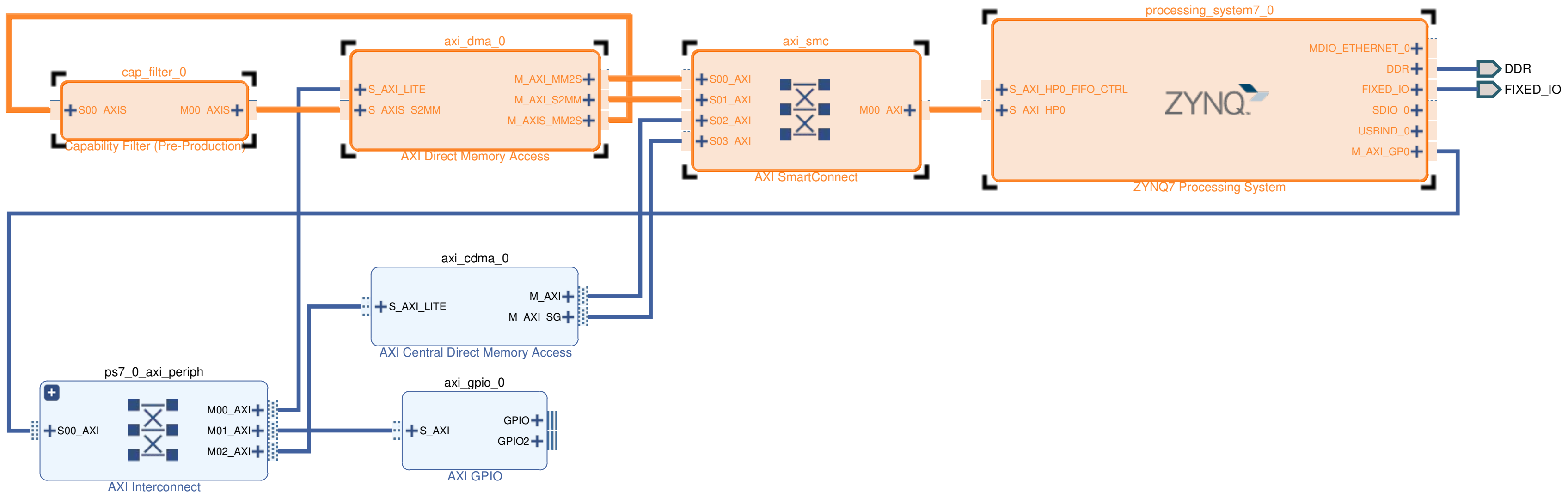


Figure 7.4.: Detail view of block design for capability filtering

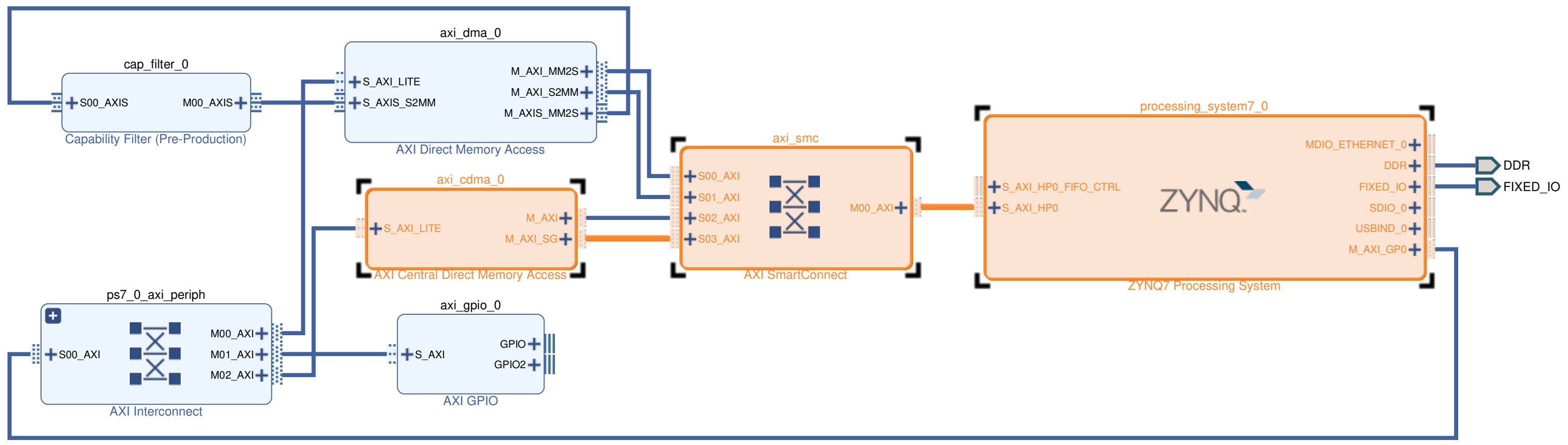


Figure 7.5.: Detail view of block design for DMA copying (scather enabled)

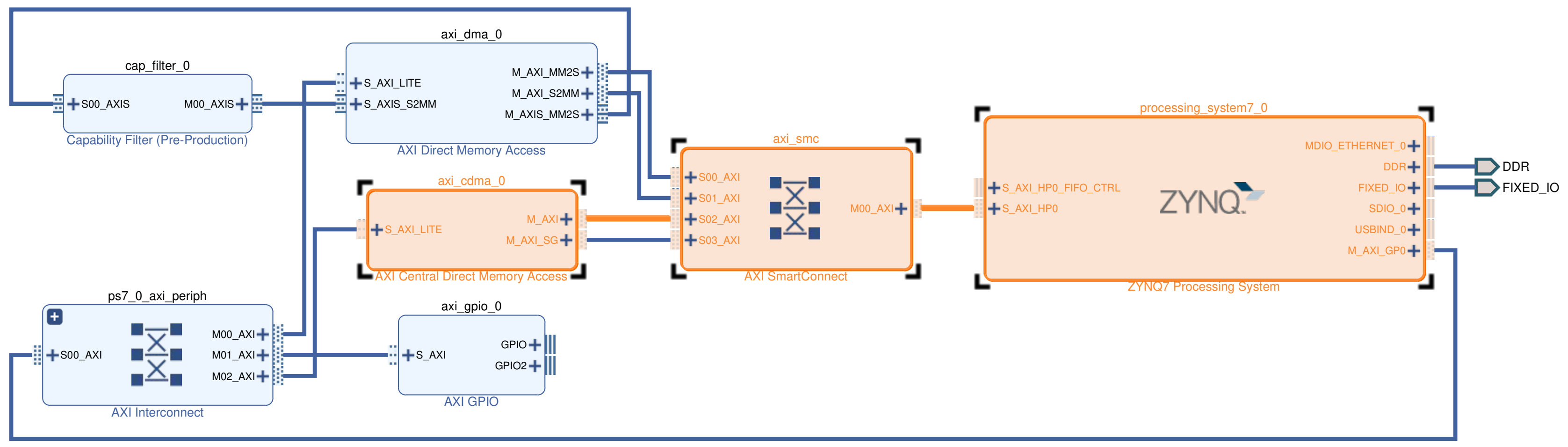


Figure 7.6.: Detail view of block design for DMA copying

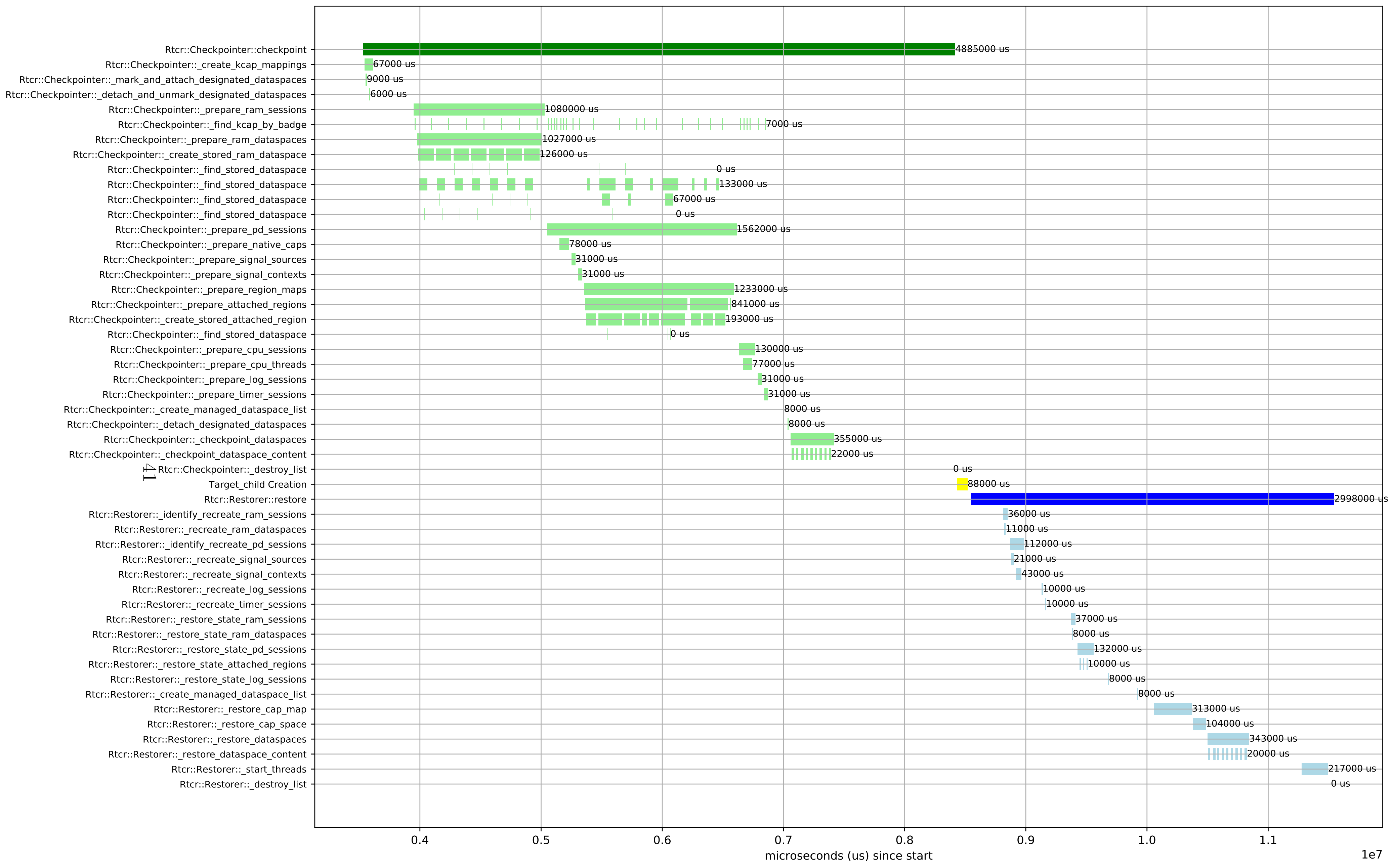


Figure 7.7.: Profiling of Checkpoint/Restore on Zybo Board