Master thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Systems and Control

FPGA-based support for predictable execution model in multi-core CPU

Bc. Maxim Baryshnikov

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Common Cybernetics and Robotics
May 2018

Acknowledgements

I would first like to thank my thesis advisor Ing. Michal Sojka, Ph.D., for his assistance and dedicated involvement in every step throughout the process. Without his great mentorship this work would have never been accomplished. I would also like to thank my family and friends for their great moral support.

Declaration

I hereby declare that I have completed this thesis with the topic “FPGA-based support for predictable execution model in multi-core CPU” independently and that I have included a full list of used references.

Prague, May _, 2018

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, _. května 2018

Abstract

In attempts to make real-time embedded systems less expensive and more powerful, researchers in the field are working on ways to incorporate Commercial-off-the-shelf (COTS) multicore devices into safety-critical designs. The Predictable Execution Model (PREM) is a promising solution to overcome the problems of shared resources interferences on such multicore platforms. One of an existing implementation of PREM employs hypervisor-based memory access monitor. It has overheads, which could be reduced with the use of FPGA-based PREM memory access monitor instead. The aim of this thesis is to implement such solution and prove the efficiency of it comparing to the hypervisor-based one.

The stated PREM watchdog was successfully implemented on Xilinx Zynq Ultrascale+ MPSoC platform using the abilities of ARM's CoreSight Debug & Trace system. The results show that in case of using FPGA-based memory watchdog maintenance takes 2.88 times less than the hypervisor-based solution requires in average (the hypercall time). Hence, the statement that HW-based guard may decrease the overhead of PREM application when compared to the software-based guard is proven.

Keywords: predictable execution, Xilinx Zynq Ultrascale+, MPSoC, FPGA, tracing, memory, PREM

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Z důvodu potřeby snížení nákladů a zvýšení výkonu embedded real-time systémů, pracují vědci po celém světě na způsobech, jak přizpůsobit hotová komerční zařízení bezpečnostně-kritickému designu. Předvídatelný exekuční model je slibné řešení k překonání problémů s interference na sdílených zdrojů na více jádrových platformách. Jedna z již existujících implementací PREM zahrnuje sledování přístupů do paměti založeny na hypervizoru. Problémy, které taková implementace vytváří (overheady v sledovaném softwaru) je možno minimalizovat využitím FPGA založeném na PREM. Cílem této práce je implementace popisovaného řešení a ověřené efektivnosti v porovnání s řešením založeném na hypervisoru.

Uvedeny PREM watchdog byl úspěšně implementován na platformě Xilinx Zynq Ultrascale+ MPSoC využitím moznosti trasovacího frameworku CoreSight. Výsledky ukazuji že v případě použiti uvedeného watchdogu založeného na FPGA, trvají přístupy 2.88 krát menší dobu než přístupy k hypervizoru pomoci hypercallu. Tímto se tvrzeni, ze hardwarová implementace watchdogu může snížit overhead potvrdilo.

Klíčová slova: Xilinx Zynq Ultrascale+, PREM, MPSoC, FPGA, trasování, ARM

Contents

Figures

Tables

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Baryshnikov**	Jméno: **Maxim**	Osobní číslo: **420064**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra řídicí techniky**		
Studijní program:	**Kybernetika a robotika**		
Studijní obor:	**Kybernetika a robotika**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Hardwarová podpora předvídatelné exekuce na vícejádrových procesorech

Název diplomové práce anglicky:

FPGA-based support for predictable execution model in multi-core CPU

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Xilinx, Zynq UltraScale+ MPSoC, Technical Reference Manual
[2] R. Pellizzoni et al., 'A Predictable Execution Model for COTS-Based Embedded Systems,' 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, Chicago, IL, 2011, pp. 269-279.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Sojka, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.01.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Michal Sojka, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Michael Šebek, DrSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Chapter 1
Introduction

Today's Automotive and Avionics industrial demands are suffering from the need for high-performance hardware to be incorporated in safety-critical systems. For example, Advanced Driver Assistant and Autopilot systems require powerful Graphics Processing Units for real-time environment tracking algorithms. Generally, the use of parallel algorithms in that field introduces many benefits in the sense of performance.

In attempts to make real-time embedded systems less expensive and more computationally powerful, a majority of researchers in the field are working on ways to incorporate Commercial-off-the-shelf (COTS) devices into safety-critical designs. One of the significant problems arising, while using a multi-core COTS hardware in real-time applications is the enormous unpredictability of shared resources competition. For instance, a common System-on-Chip (SoC) could contain CPUs, GPUs, and FPGAs all sharing the same interconnect and memory hierarchy, which leads to unwanted interferences. Thus, COTS devices cannot be used in systems with tightly deterministic time constraints without applying some third-party arbiters to the parts of a system where those shared resources present.

The European Project HERCULES[1], where CTU participates, uses Predictable Execution Model (PREM) [PBB$^+$11] in their automotive software stack. Their solution also includes a software-based guard that monitors the program execution in order to limit potential interference to other parts of the system. The monitoring ensures that the given time and memory budgets do not overrun but also introduces some overhead. The use of hardware-based guard instead may reduce the overheads that the software-based solution have. This thesis aims to prove that concept of hardware-based execution monitoring.

The thesis is structured as follows: Chapter 2 of this work is dedicated to the theoretical introduction into PREM's problematic. The main concepts and terms, which are used in the rest of this thesis are defined there.

[1] `http://hercules2020.eu/`

Chapter 3 introduces the reader to Xilinx Zynq Ultrascale+ platform and its abilities, focusing on hardware tracing capabilities. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit was given as a hardware platform for this work. Due to the general complexity and vast variety of features this platform provides, the investigation of it has taken the vast majority of the time spent on this project. This is the reason why Chapter 2 provides the information about features that were not applied in the final solution. However, those unemployed findings may serve well for future work purposes, and that is why they are kept there.

Chapter 4 explains the PREM Watchdog's implementation in details both from software part and hardware (FPGA) part points of view. The chapter aims to reason about the decisions that were made and about an implementation concept. Furthermore, it discusses the problems that were met on the way.

Finally, Chapter 5 describes the tests of developed functionality and overhead comparison with the software-based solution. The results are discussed in Conclusion.

Chapter 2

Theoretical Background

The following chapter provides some general explanation of the terms and concepts about the stated topic (Section 2.1). In Section 2.2, there is an overview of already implemented solutions. Finally, the part of the problem which this work aims to solve is discussed (Section 2.3).

2.1 Predictable Execution Model (PREM)

Predictable Execution model (PREM), proposed by Pellizzoni et al. [PBB$^+$11], is a way to execute safety-critical software in deterministic time on multicore systems. The primary target of struggle on such systems is cache misses while their amount strongly affects the worst-case execution time (WCET) with its indeterminism. PREM solves the mentioned issues by applying resource access scheduling on a given program. A program should have execution intervals which are stated either as **predictable** or **compatible**. Predictable intervals "are executed *predictably* and *without cache misses*"[PBB$^+$11], and cannot be preempted until the end of scheduling interval. Predictable intervals are then divided into sub-phases which are the following:

- Cache Prefetch
- Computations
- Cache Write-Back

During the cache **prefetch** phase, all the instructions and data needed for the computing stage are loaded into the cache which is shared among all cores. It is essential to avoid *self-eviction* of the cache, i.e., prefetching of the cache line does not rewrite the data fetched in the same phase. Then follows the **computation** phase where no cache misses could occur because all needed data are already there. Finally, after the computation is done the CPU posts the results back to the cache during the **write-back** phase.

Compatible intervals may follow after a sequence of predictable intervals. The task running in it is allowed to be preempted and have cache misses. It can not, however, block the execution indefinitely. For that reason, communication with peripheral devices should be restricted as much as possible during this phase.

The transformation of legacy code to PREM-compatible program requires a programmer to put pragmas (e.g., `predictable` code blocks, as stated in [PBB$^+$11]) which define single predictable intervals of a program and its limitations (e.g., time and memory budget). PREM real-time compiler should then put prefetch and write-back instructions at the beginning and the end of a predictable interval.

The intervals are then scheduled either on-line or off-line such that the memory access resource is shared between CPU and I/O peripherals exclusively during the non-preemptive intervals.

2.2 State of the Art

In the original paper ([PBB$^+$11]) which introduced PREM, the authors implement the single-core approach that is primarily focused on efficient and safe I/O peripherals distribution among tasks that are executed on CPU periodically. Figure 2.1 depicts an example of such schedule where memory access resource is shared exclusively between tasks τ_1 and τ_2 that do some computations and tasks $\tau_1^{I/O}$ and $\tau_2^{I/O}$ that work with I/O flows.

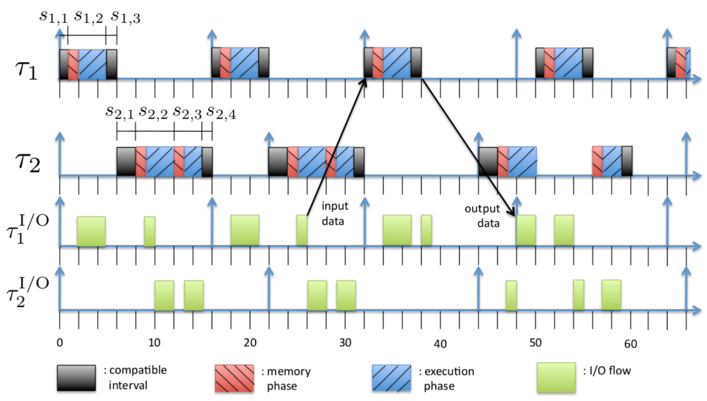


Figure 2.1: An example of PREM schedule. Source: [PBB$^+$11]

To achieve the PREM synchronization of I/O devices, R. Pellizzoni et al. introduce FPGA-based *real-time bridge* and *peripheral scheduler*. Figure 2.2 shows the hardware layout they used. It does not, however, cover multiple CPU setups. The further research focused more on incorporating PREM with multicore systems without the need of hardware-based arbiter, introducing the concept of multithreaded PREM scheduling based on fork-join principle [AP14]. Based on this and other works, J. Matějka et al. introduced

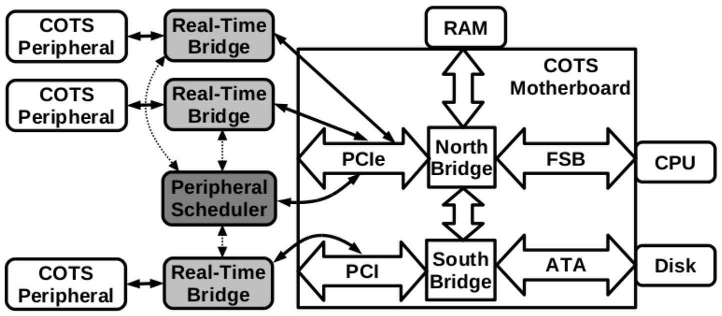


Figure 2.2: Real-Time I/O Management System proposed by R. Pellizzoni et al. Source: [PBB+11]

a complete PREM toolchain [MFS+18] consisting of a compiler for ARM and scheduling model which was successfully evaluated on Advanced-Driver Assistant-System (ADAS) scenarios (refer to Figure 2.3 and Figure 2.4).

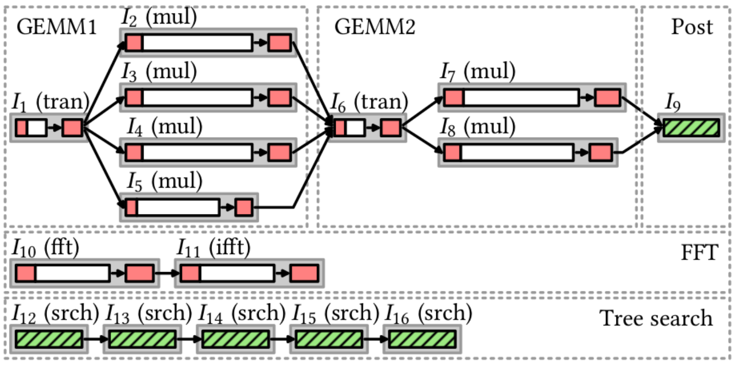


Figure 2.3: An ADAS-like scenario schedule presented as direct acyclic graph. In green: compatible intervals. In white: computation phases. In read: memory access phases. Source: [MFS+18]

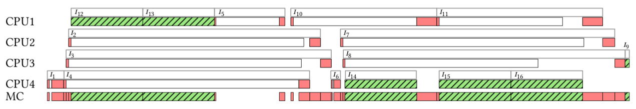


Figure 2.4: Gantt diagram of schedule depicted in Figure 2.3 of PREM intervals among multiple CPUs. The green ones depict computation phases. The read ones are memory access phases. Source: [MFS+18]

HERCULES project [Pro] also considers applying PREM in similar manner as proposed in [MFS+18] to achieve predictive execution of their ADAS framework on multicore systems. However, instead of using real-time bridges as [PBB+11] proposes they run that logic in a software arbiter such as VM or hypervisor, so they do not need to integrate additional hardware on a SoC.

[BBC+17]

2.3 The Problem Statement

As it was mentioned in the section, the HERCULES framework solution uses a hypervisor to drive PREM application's execution. In particular, it must be in charge of two tasks:

- Scheduling of execution intervals and phases
- Detecting misbehaving applications

The detection of the faulty behavior for a safety-critical program is unconditionally required. Not only due to an axiom that every software may have a bug but also because of safety-related certification process which every piece of control system's software should pass.

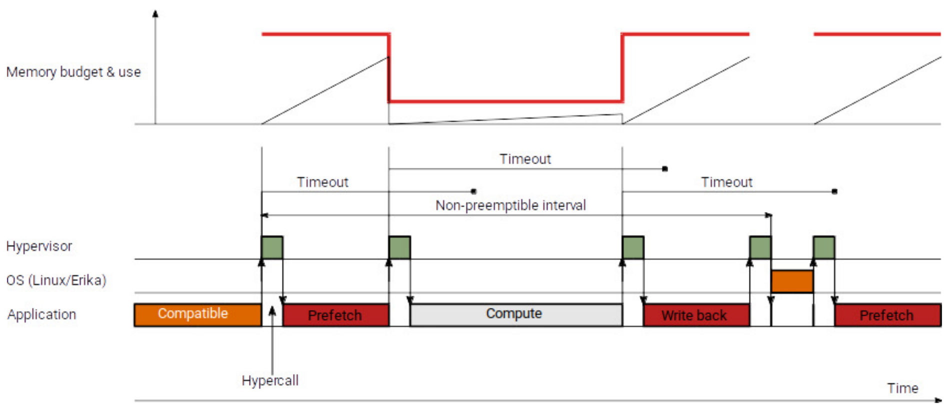


Figure 2.5: HERCULES Memory access scheduling & supervision. Source: [MSH17]

The system setup consists of a PREMized application which runs in userspace of Linux/Erika OS and Jailhouse hypervisor[BBC+17] . The whole execution process goes as follows[MSH17] (See Figure 2.5):

1. At the beginning of every non-preempted interval phase or compatible interval the application issues a hypercall.
2. When the hypervisor receives the call, it becomes aware of:
   a. the end of the previous phase
   b. the start of the current phase
   c. the memory and time budget that the current phase requests to have.
3. if the hypervisor detects the memory budget overrun during the application's execution, it signals about it and then acts as predefined in case of a critical fault.

Unfortunately, this solution has a drawback that affects the whole system performance. Experiments with that setup show (available here [Gai17]) that the hypercall lasts **21.72 μs** in average and up to **58.42 μs** in the worst case. That time could be improved with the use of some hardware instead of the hypervisor. Minimally, the watchdog functionality may be implemented in FPGA. Thus, the remainder of that thesis will discuss the ways how to achieve it.

Chapter 3
Hardware Platform Overview

This Chapter describes the capabilities of hardware given for the experiments with PREM. In attempts to mitigate the overhead mentioned at the end of Section 2.3, one should determine the right tools to achieve that, and this is the purpose of existence of this Chapter. In particular, the investigation here focuses on abilities of non-invasive tracing of DRAM access events for multiple CPU cores.

Section 3.1 presents the general overview of the given hardware. It also describes Trace subsystems available on the chip. Section 3.2 scopes out the ARM's CoreSight framework, its building blocks, and their roles. This section also provides a short overview of software tools available for working with that toolset. The last Section (3.3) describes ARM's Performance Monitoring Unit.

3.1 Zynq MPSoC's capabilities

The thesis specification suggests using Xilinx Zynq MPSoC Ultrascale+ platform for the PREM's experiments. It is reasonable proposition because the platform is cost-efficient, powerful, and it covers a broad range of applications. Moreover, Xilinx claims that Zynq MPSoC Ultrascale+ EG devices will find their use in Aerospace application and Zynq MPSoC Ultrascale+ EV are ideal for automotive tasks such as ADAS. In this work, however, the EG SoC is used, but the difference between them is only the fact that EV chips have an integrated H.264 / H.265 video codec.[bADASA]

3.1.1 Zynq MPSoC Overview

The top architecture overview of Zynq MPSoC Ultrascale+ is presented in Figure 3.1. That Figure shows all available processing units, I/O devices, platform controllers, etc. The points of interest, however, are Application Processing Unit (APU) and Programmable Logic (PL) and their interconnect with DRR4-type memory. APU consists of 64-bit Quad-core Cortex-A53 ARMv8 multiprocessing CPU, 1MB of L2 cache and Snooping Control Unit

(SPU) which cares about direct transfers between per-CPU L1 caches on purpose of maintaining cache coherence.The L2 cache is 16-way set-associative; Also, SPU supports Accelerator Coherency Port (ACP) port, which one could use to have I/O coherency of PL design, or even to have own L2 caches in PL coherent with the rest of the system (full-coherency mode). [Incc]

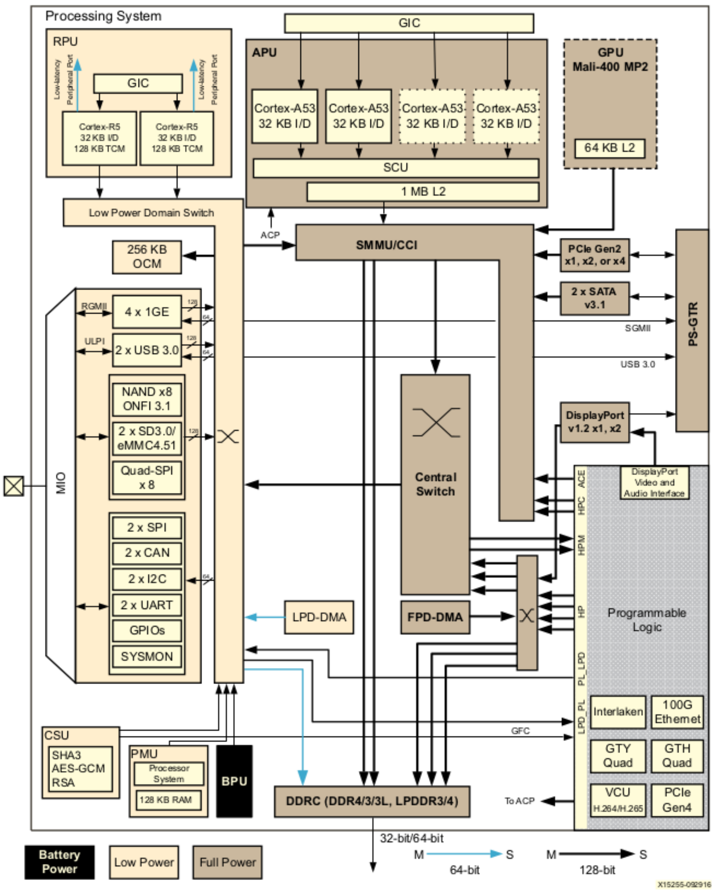


Figure 3.1: Zynq UltraScale+ MPSoC Top-Level Block Diagram. Source: [Incc]

All components are interconnected with AMBA-compliant network through Advanced eXtensible Interfaces (AXI). This is a Network-On-Chip which consists of peer-to-peer connected devices in a master-slave manner; some of them are switches and bridges, which not only provide many-to-many connectivity but also synchronize signals coming from the different clock and power domains [Limf]. Figure 3.2 shows the interconnect in details.

Certain parts of the memory interconnect (especially the Cache Coherent Interconnect (CCI), DDR controller and QoS-400 Regulator) support Quality-of-Service (QoS) which could provide memory bandwidth throttling for the selected paths in the system. The master ports of these devices may be configured to give Low Latency (High Priority), High Throughput (Best

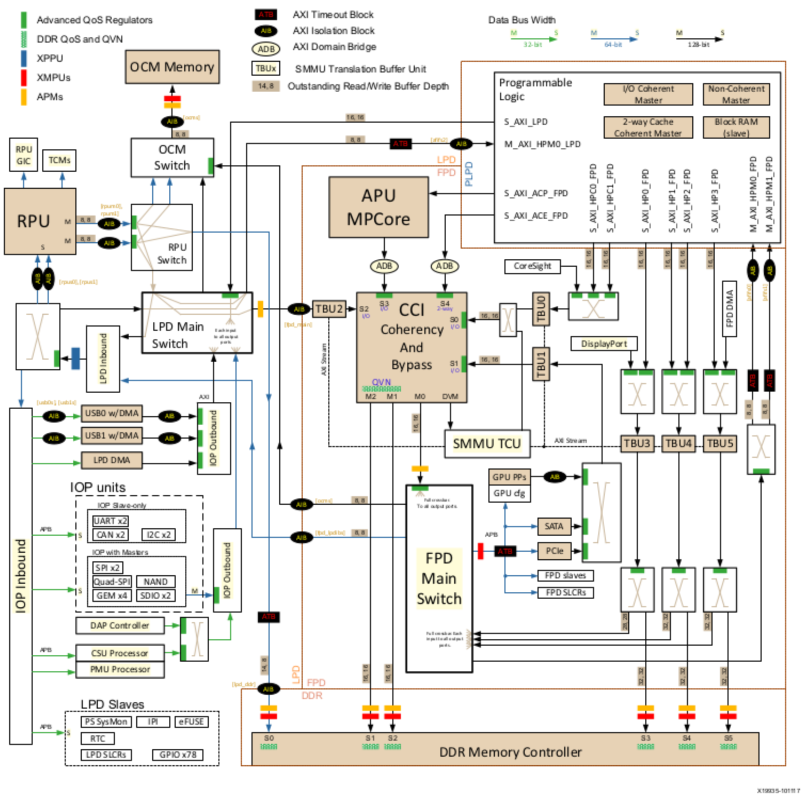


Figure 3.2: Zynq UltraScale+ MPSoC Top-Level AXI Interconnect Architecture. Source: [Incc]

Effort) or Isochronous access. This could be programmed either statically, or be controlled dynamically from PL.[Incc] The feature may be used to bound DRAM memory accesses in a manner of, e.g., MemGuard project ([YYP$^+$13]), or in the collaboration of MemGuard and PREM where the throttling was applied on compatible phases of PREM model ([HSH17]).

PL could use several master and slave AXI ports to access the memory shared with Processing System (PS). The communication is also possible through multiplexed I/O interface (MIO) and extended multiplexed I/O interface (EMIO), PS-to-PL, and PL-to-PS interrupts.

3.1.2 Tracing capabilities

Zynq Ultrascale+ provides wide abilities to trace the whole SoC. For example, integrated AXI Performance Monitors (APM) could monitor some of those connections. There are 4 APMs, and they could count the metrics at nine points of memory interconnect (See Figure 3.2). They could be used for the use-cases as obtaining latency metrics, read/write throughputs, count AXI bus events, debug AXI peripherals. (citation). APM is also available as an Intelectual Property (IP) block for FPGA to analyze an AXI traffic of implemented logic.[Inca]

There are three mods in which APM may operate. In the mode called **Advanced** APM can do *Event Logging*, where the specified events are stored in FIFO and then exported via AXI-Stream interface, and *Event Counting*, where the integrated metric counters are set on some event type. In **Profile Mode**, APM works similarly as in Advanced - Counting mode, but the metrics are predefined. The **Trace Mode** of APM shares the same idea as Profile Mode does: This is a simplified easy-to-use version of Advanced feature, in this case that is Event Logging. APM IP could emit an interrupt which may be set up on the overflow of metric counters or to signalize if the tracing FIFO is full.[Inca]

The important note about APM placed in SoC is "The PS-based APMs implement the advanced mode without error logging or the AXI Stream features."[Incc]

Xilinx also introduces the Fabric Trace Macrocell (FTM) (placed on the scheme in Figure 3.3) that allows cross-triggering between PS and PL. It has 32-bit GPIO from/to PL and four input/output trigger channels. The typical use-case of it is to start capturing in Integrated Logic Analyzer IP[Incb] placed in PL after some hardware event occurred at PS. The other option is to stimulate trace events with a trigger from PL. FTM is CoreSight-compliant device. For CoreSight description see the following Section 3.2.

3.2 ARM's CoreSight Framework

Zynq Ultrascale+ MPSoC platform implements the ARM CoreSight SOC-400 Trace & Debug components. This system provides an opportunity for non-invasive tracking of events coming from various devices on the chip. Figure 3.3 shows the complete layout of all CoreSight components available on Zynq Ultrascale+ MPSoC. This work heavily uses the features of CoreSight to follow the memory access events due to non-intrusiveness of those tools and ability to export the needed information at runtime (See Chapter 4).

CoreSight devices are memory-mapped, and every such device must contain a set of specification-defined identification registers. It allows either software or hardware trace analyzer to detect the topology. A trace analyzer should know a physical address of CoreSight ROM table, where it should find the offsets of either devices or other ROM tables.[Lim13]

The trace data transfer consists of the following stages. *Trace units* (i. e. *trace sources*) emit a *trace packet* that contains trace unit's ID and encapsulated data. Then, the packet flows through *trace links* (which are connected by AMBA Advanced Trace Bus (ATB)) to *trace sinks*. The trace is then read from the sink by trace analyzer and decoded.

Those software libraries and drivers may be useful when working with CoreSight:

- CoreSight Access Library (CSAL) [GCAL]
- Linux CoreSight driver - already in kernel's mainline
- OpenCSD - An OpenSource CoreSight Decoding library [GOAosCTDl]

CSAL provides C API for programming of almost all CoreSight components available on the market both from bare metal and Linux environment. This work uses that tool for CoreSight components configuration.

Linux CoreSight driver integrates the CoreSight system with standard performance evaluation tool - **perf**. Linaro's OpenCSD is able then to work together with perf to get a human-readable tracing of the kernel and user-space program execution.[GOAosCTDl]

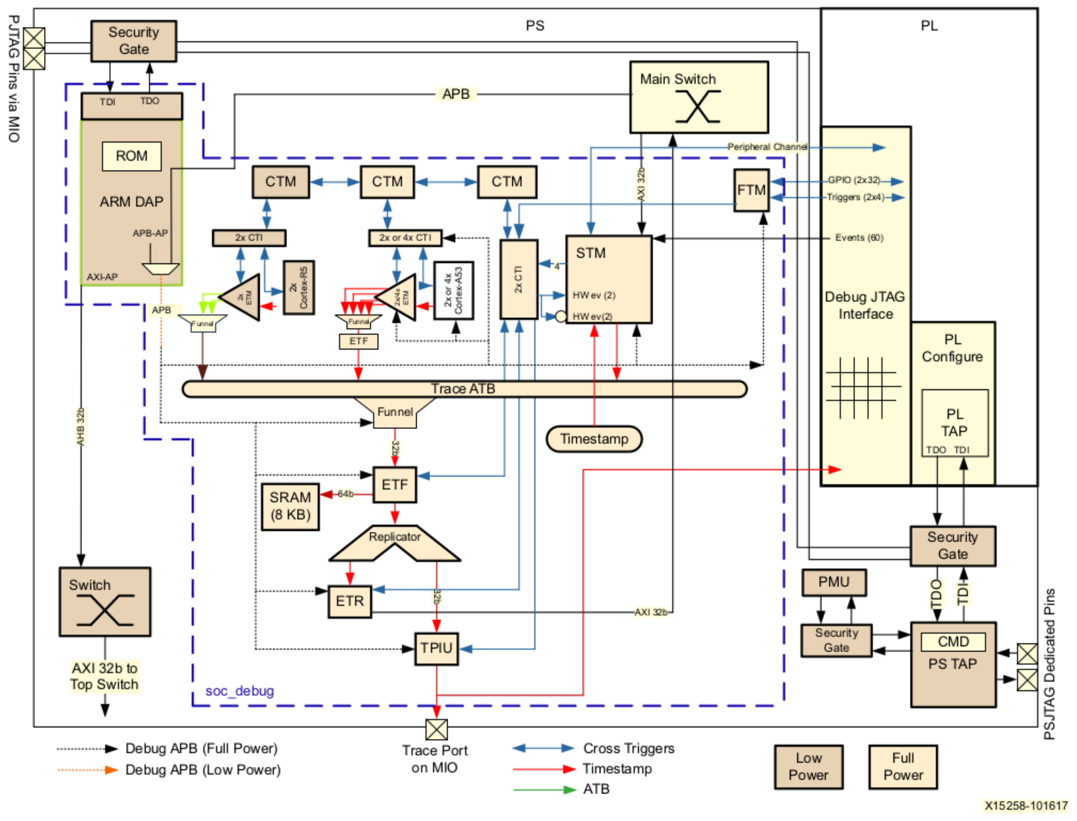


Figure 3.3: Zynq UltraScale+ MPSoC Debug Block Diagram. Source: [Incc]

3.2.1 Trace Sources

Embedded Trace Macrocell

Embedded Trace Macrocell (ETM) is a trace system element which targets on producing program traces. Those traces are generated at program's run-time. However, on purpose not to overload trace streams, ETM may be configured to notice only particular *trace elements* (i.e., events or event sequences). That filtering is highly customizable.

ETM is tightly coupled with CPU's core. Thus, every CPU core has a separate ETM. The hardware platform used in this work employs 4 Cortex-A53 cores and 2 Cortex-R5 cores (See overview in Figure 3.3), and R5's ETMs differ from the other ones in provided options. For instance, A53's ETMs do not support Data tracing. The remaining of the section focuses mainly on abilities of A53's ETM which implement the ETMv4 architecture.

Trace elements could be ([Incc]):

- Instruction address match
- Indirect branches and direct branches
- Instruction barrier instructions
- Exceptions
- Changes in processor instruction set state
- Changes in the processor security state
- Context-ID register changes
- Entering to debug state
- Cycle count during the traced parts
- Global system timestamps
- Target addresses for taken direct branches
- Trace control events such as:

    Trace synchronization packets

    Indicators of speculative execution for some instruction, if such event occurs

Those events are generated by so-called *trace resources*. They are configurable subsystems of ETM. Here is an overview of what resources Cortex-A53's ETM has. ([Limd],[Lime])

- External inputs

  These are input signals that other resources (Counters and resource selectors) may process to generate a trace event. Cortex-A53's ETM has 30 inputs: 4 Cross-Trigger inputs + 26 events from PMU. 4 of them can be selected.

- External outputs

  There are 4 of them. All are wired to Cross-Trigger interface. Any event from other resources may signal on an output; this is configured through Resource selectors (see below).

- Address comparators

  4 address comparators are available on Cortex-A53's ETM. They may be used to signal on a single preset address of the instruction which processor executed, or they may be used in pairs to create trace events when the address of the instruction is in (or out of) the preset range. The important notice here is that an address comparator also reacts on instructions executed speculatively.

- Single-shot comparator

  To mitigate the problem of noticing speculatively executed instructions in case of Address Comparator's use, the single-shot comparator is introduced. The examined ETM provides only one such trace resource. A Single-shot comparator chooses a single address or range address comparators to follow. When an instruction noticed by them is actually (non-speculatively) executed by a processor, the Signal-shot comparator fires. There is also a possibility to set that unit to reset after every fire so that it will be a "multiple-shot" one.

- Context identifier and Virtual context identifier comparators

  They may be associated with Address comparators or used on its own. They react on a particular Context or Virtual Context IDs respectively.

- Counters

  Two decrementing counters are available. They may be used to count events on other resource units. A user may set initial and reset value for them, so every time a counter reaches zero it fires. The self-reload mode is also possible: counter resets with the provided value every time it reaches zero.

- Sequencer

  Provides a programmable 4-state machine to react on sertian sequence of events preprogrammed as state machine transitions.

- ViewInst unit provides the functionality of filtering of instruction trace events.

Resource selector units are used to interconnect the resources between each other. Figure 3.4 explains the concept. The example of resources configuration is provided in Chapter 4.

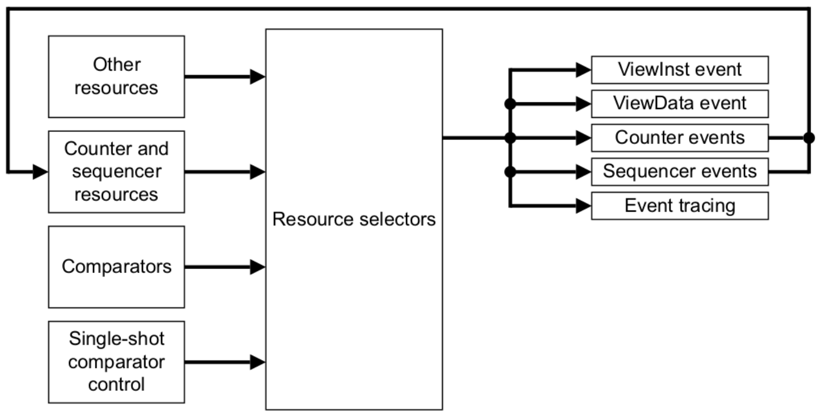


Figure 3.4: ETM's resource selection overview. Source: [Lime]

System Trace Macrocell

The System Trace Macrocell provides an opportunity to trace HW events, and a printf-style debug/trace option. On Zynq MPSoC, only PL events (60 of them) are connected to the HW event interface.[Incc]

Printf-style tracing means that STM can generate trace packets when software writes a message to STM registers. Combination of address and data in that message activate some "stimulus" port in STM, so STM bursts a packet associated with that port.

The example of STM use is presented in FreeRTOS board support package (see Listing 3.1). When the operating system enters in some function representing a system call (such as a task switch), the event's ID is written to STM's address. A developer could then analyze those events along with timestamps using a software (e.g., Xilinx SDK IDE, see screenshot in Figure 3.6 and the necessary setup option in Figure 3.5) that knows which trace IDs are defined for which syscalls. Some guides are provided at [UXSFAuS]

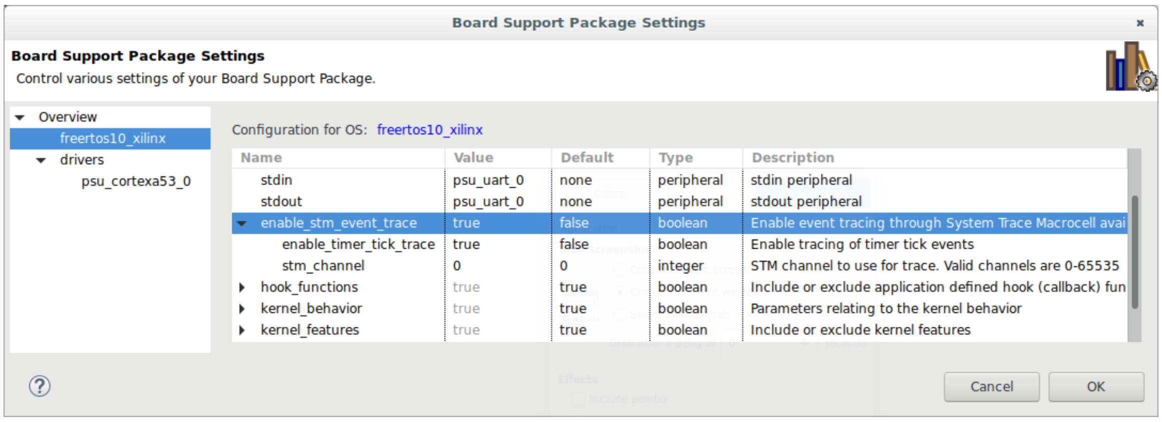
Listing 3.1: The parts of FreeRTOSSTMTrace.h which presents how FreeRTOS uses STM for tracing

```
#define STM_BASE                        0xf8000000

#define FREERTOS_EMIT_EVENT(id)         Xil_Out8(STM_BASE +
    (FREERTOS_STM_CHAN * 0x100), id)
#ifdef EXEC_MODE32
#define FREERTOS_EMIT_DATA(data) Xil_Out32((u32) (STM_BASE +
    (FREERTOS_STM_CHAN * 0x100) + 0x18), (u32) data)
#else
#define FREERTOS_EMIT_DATA(data) Xil_Out64((u64) (STM_BASE +
    (FREERTOS_STM_CHAN * 0x100) + 0x18), (u64) data)
#endif

...

#ifndef traceINCREASE_TICK_COUNT
/* Called before stepping the tick count after waking from
    tickless idle sleep. */
#define traceINCREASE_TICK_COUNT( x ) {                          \
 FREERTOS_EMIT_EVENT(FREERTOS_INCREASE_TICK_COUNT);           \
 FREERTOS_EMIT_DATA(x);                                       \
}
#endif
...
```


Board Support Package Settings

Board Support Package Settings

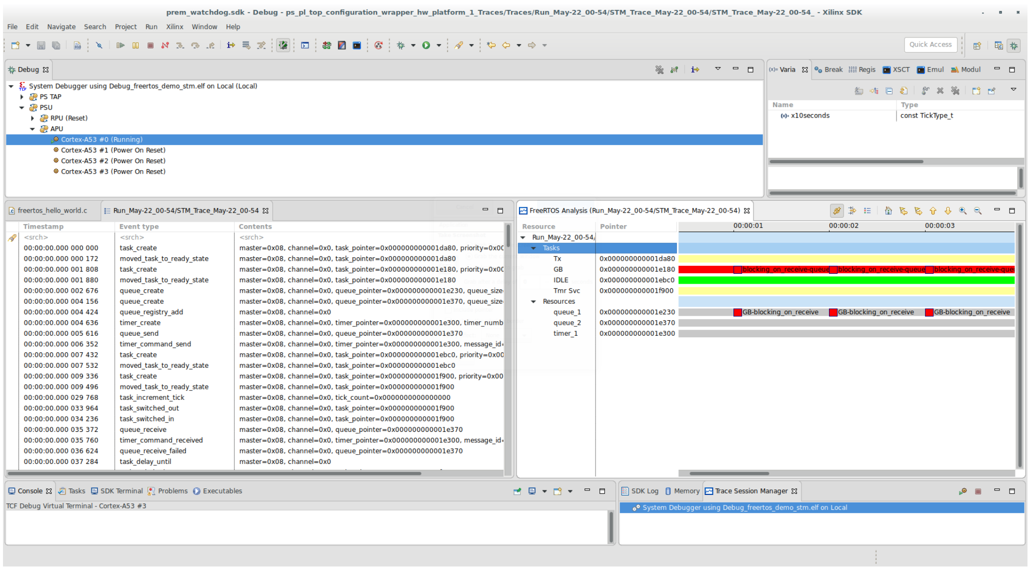
Control various settings of your Board Support Package.

- Overview
  - freertos10_xilinx
  - drivers
    - psu_cortexa53_0

Configuration for OS: freertos10_xilinx

Name	Value	Default	Type	Description
stdin	psu_uart_0	none	peripheral	stdin peripheral
stdout	psu_uart_0	none	peripheral	stdout peripheral
enable_stm_event_trace	true	false	boolean	Enable event tracing through System Trace Macrocell avai
enable_timer_tick_trace	true	false	boolean	Enable tracing of timer tick events
stm_channel	0	0	integer	STM channel to use for trace. Valid channels are 0-65535
hook_functions	true	true	boolean	Include or exclude application defined hook (callback) fun
kernel_behavior	true	true	boolean	Parameters relating to the kernel behavior
kernel_features	true	true	boolean	Include or exclude kernel features

Figure 3.5: The necessary option setup in FreeRTOS BSP for enabling STM tracing in Xilinx SDK IDE.



prem_watchdog.sdk - Debug - ps_pl_top_configuration_wrapper_hw_platform_1_Traces/Traces/Run_May-22_00-54/STM_Trace_May-22_00-54/STM_Trace_May-22_00-54_ - Xilinx SDK

```
System Debugger using Debug_freertos_demo_stm.elf on Local (Local)
  PS TAP
  PSU
    RPU (Reset)
    APU
      Cortex-A53 #0 (Running)
      Cortex-A53 #1 (Power On Reset)
      Cortex-A53 #2 (Power On Reset)
      Cortex-A53 #3 (Power On Reset)
```

```
Name            Type
x10seconds      const TickType_t
```

Run_May-22_00-54/STM_Trace_May-22_00-54

```
Timestamp              Event type                  Contents
<srch>                 <srch>                      <srch>
00:00:00.000 000 000   task_create                 master=0x08, channel=0x0, task_pointer=0x000000000001da80, priority=0x00
00:00:00.000 000 172   moved_task_to_ready_state   master=0x08, channel=0x0, task_pointer=0x000000000001da80
00:00:00.000 001 808   task_create                 master=0x08, channel=0x0, task_pointer=0x000000000001e180, priority=0x00
00:00:00.000 001 880   moved_task_to_ready_state   master=0x08, channel=0x0, task_pointer=0x000000000001e180
00:00:00.000 002 676   queue_create                master=0x08, channel=0x0, queue_pointer=0x000000000001e230, queue_size
00:00:00.000 004 156   queue_create                master=0x08, channel=0x0, queue_pointer=0x000000000001e370, queue_size
00:00:00.000 004 424   queue_registry_add          master=0x08, channel=0x0
00:00:00.000 004 636   timer_create                master=0x08, channel=0x0, timer_pointer=0x000000000001e300, timer_numb
00:00:00.000 005 616   queue_send                  master=0x08, channel=0x0, queue_pointer=0x000000000001e370
00:00:00.000 006 352   timer_command_send          master=0x08, channel=0x0, timer_pointer=0x000000000001e300, message_id=
00:00:00.000 007 432   task_create                 master=0x08, channel=0x0, task_pointer=0x000000000001ebc0, priority=0x00
00:00:00.000 007 532   moved_task_to_ready_state   master=0x08, channel=0x0, task_pointer=0x000000000001ebc0
00:00:00.000 009 336   task_create                 master=0x08, channel=0x0, task_pointer=0x000000000001f900, priority=0x00
00:00:00.000 009 496   moved_task_to_ready_state   master=0x08, channel=0x0, task_pointer=0x000000000001f900
00:00:00.000 029 768   task_increment_tick         master=0x08, channel=0x0, tick_count=0x0000000000000000
00:00:00.000 033 964   task_switched_out           master=0x08, channel=0x0, task_pointer=0x000000000001f900
00:00:00.000 034 236   task_switched_in            master=0x08, channel=0x0, task_pointer=0x000000000001f900
00:00:00.000 035 372   queue_receive               master=0x08, channel=0x0, queue_pointer=0x000000000001e370
00:00:00.000 035 760   timer_command_received      master=0x08, channel=0x0, timer_pointer=0x000000000001e300, message_id=
00:00:00.000 036 624   queue_receive_failed        master=0x08, channel=0x0, queue_pointer=0x000000000001e370
00:00:00.000 037 284   task_delay_until            master=0x08, channel=0x0
```

FreeRTOS Analysis (Run_May-22_00-54/STM_Trace_May-22_00-54)

```
Resource            Pointer               00:00:01                      00:00:02                      00:00:03
Run_May-22_00-54
  Tasks
    Tx              0x000000000001da80
    GB              0x000000000001e180    blocking_on_receive-queue     blocking_on_receive-queue     blocking_on_receive-que
    IDLE            0x000000000001ebc0
    Tmr Svc         0x000000000001f900
  Resources
    queue_1         0x000000000001e230    GB-blocking_on_receive        GB-blocking_on_receive        GB-blocking_on_receive
    queue_2         0x000000000001e370
    timer_1         0x000000000001e300
```

TCF Debug Virtual Terminal - Cortex-A53 #3

System Debugger using Debug_freertos_demo_stm.elf on Local

Figure 3.6: FreeRTOS tracing in Xilinx SDK IDE.

3.2.2 Trace Links and Sinks

Funnels & Replicators

Replicators are non-programmable elements of the trace system which simply transfer the input trace on outputs. Funnels, however, are a little bit more sophisticated. They combine trace from multiple inputs into one output trace. One can configure input ports of a funnel to be enabled/disabled and to have a priority. The fixed priority scheme is then applied on inputs [Limc]. Block scheme of its principle is in Figure 3.7.

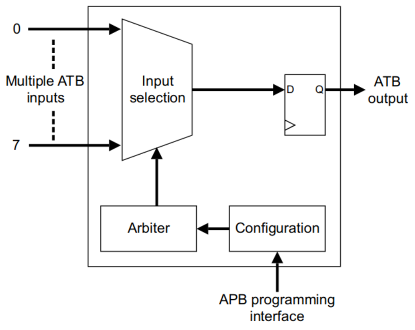


Figure 3.7: Funnel block diagram. Source: [Limc]

Trace Memory Controller

This CoreSight IP may present in one of three types:

- Embedded Trace Buffer (ETB) - stores trace in a circular buffer on SRAM
- Embedded Trace FIFO (ETF) - functions as a queue for trace packets for a reason of trace bandwidth normalization
- Embedded Trace Router (ETR) - is a *trace sink* (endpoint of a trace bus). It sends trace packets to main memory through AXI.

Zynq MPSoC has 2 ETFs (8 KB) and 1 ETR. All these elements have programmable interfaces and are connected to CTI in purpose to start/stop trace or signal about buffer overflows etc.

Trace Port Interface Unit

This is also a final point of trace bus. It outputs trace data to an external device which decodes and stores/analyses the trace. Its outputs signals are TRACEDATA(32-bit width, could be reduced), TRACECTL (service signals), TRACECLK(250 MHz by default, may be provided externally). Before delivering the trace out, TPIU reformats it “re-associates trace sources' IDs with trace data”[Limc], to provide better bandwidth and an “ability for a trace decoder to resynchronize on frame boundary”[Limc]. To enable PL output (16-bit width data at MIO and 32-bit data at EMIO ports): “The TPIU.EXTCTL_OUT_Port register must be set to output trace into the PL.”[Incc]

3.2.3 Embedded Cross-Trigger

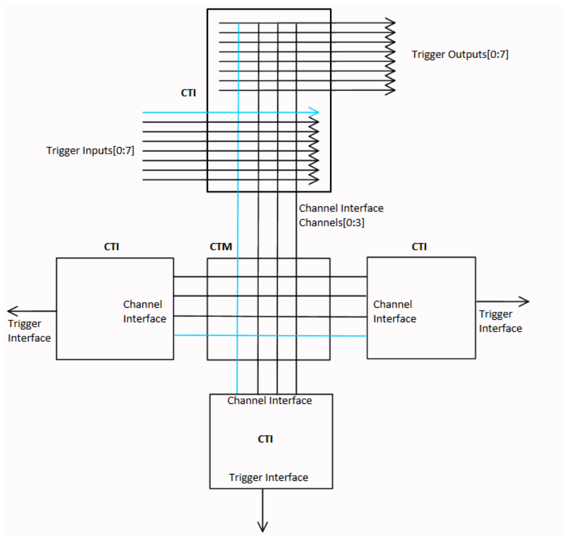


Figure 3.8: ECT block diagram. “The CTI at the top is configured to propagate the trigger event on Trigger Input 0 to Channel 0.” Source: [UXSCT]

The Embedded Cross-Trigger (ECT) system distributes trigger signals between all trace and debug elements. Two main elements are presented here: Cross-Trigger Interface (CTI) and Cross-Trigger Matrix (CTM).

CTI is in charge of mapping signals between input/output ports and input/output channels. The mapping is configured through CTI's registers (internal logic is presented in Figure 3.9). To propagate the signal from internal channel outside, CTIGATE register bits should be set. There is also an opportunity to set some channels active from software, or, to send a trigger (channel pulse) through CTIAPPSET/CTIAPPCLEAR, which is sometimes helpful for debug purposes. Every Zynq MPSoC's CTI has 8 input and 8 output trigger signals (but some are reserved).

CTM broadcasts signals to other CTIs. The channels coming from CTI are combined in OR manner. Figure 3.8 illustrates the example of trigger propagation. Zynq MPSoC's CTMs have 4 channels.

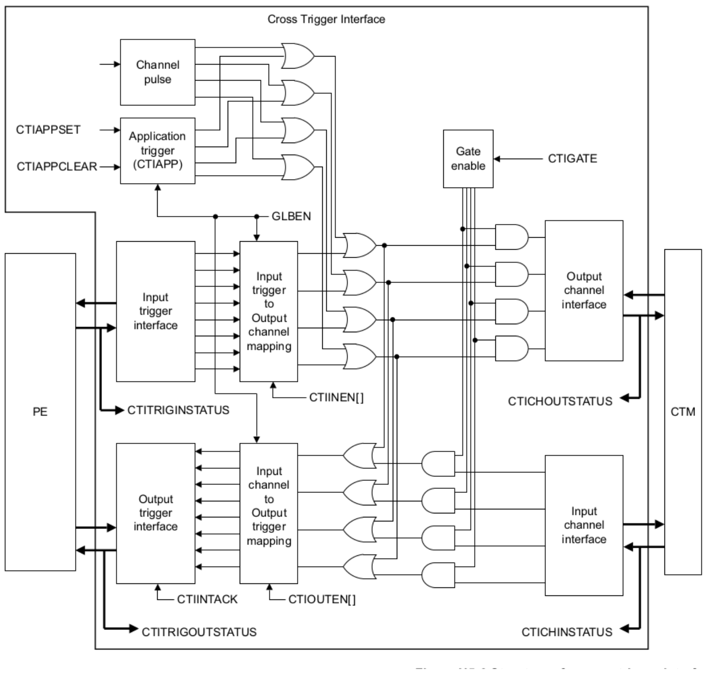


Figure 3.9: CTI internal logic overview. Source: [Lima]

Channel interface consists of two pairs of input and output wires (CHIN/-CHOUT, CHINACK/CHOUTACK) when acknowledge is asynchronous, and contains additional wire CHCLK when the interface implemented as synchronous. An "asynchronous interface uses a basic 4-phase handshaking protocol"[Limb].

There are 8 CTIs on Zynq MPSoC: 2 are for cores of RPU unit, other 4 are connected to Cortex-A53 cores (each on its own core), and the last 2 are for the whole SoC. Every Cortex A53 CTI accepts signals from ETMs (External outputs), PMU (PMUIRQ), and the debug request from CPU. For out triggers, it has the ETM's external inputs wired, may send an interrupt, and debug halt command to CPU. The next CTI, SoC's one, handles FTM and STM triggers (in and out). The other SoC's CTI triggers ETFs and TPIU. More precise information on ECT wiring for Zynq MPSoC is provided here - [UXSCTiZUM].

3.3 ARM's Performance Monitoring Unit

Performance monitoring unit enables a user to count selected processor events such as memory bus accesses, cache accesses, exceptions and so on. Every core of APU has its own PMU so that one could have information about such events per every core separately. It is useful especially in case of trying to implement per-processor memory access monitoring.

PMU available at Cortex A53 has one 64-bit wide clock counter, which is often used for measuring the execution time of a program part, and six event counters (32-bit wide). Event counters may be configured on a specific event type (the full list is available). The counters count up and could be set to emit an interrupt (PMUIRQ) on overflow, meaning – when the counter becomes equal to zero.

22 events from PMU are wired to ETM's external inputs so that the ETM may use these as trace events. This facility is used in this work as described in Chapter 4.

Chapter 4
PREM Watchdog Implementation

4.1 Concept

The aim of creating the PL Watchdog system is to non-invasively monitor the number of memory accesses made by APU cores per every PREM phase execution. The two possible on-chip hardware provide such functionality: APM (See Section 3.1.2) and PMU (See Section 3.3). APM, however, does not fit as a tool because of its inability to differentiate the initiators of a memory request. It can measure the overall traffic, e.g ., for read/write byte count coming from CCI master, but it has no tracing points which are closer to cores than that.

From the variety of memory access events that PMU provides (cite armv8), `L2_CACHE_REFILL` was chosen to be followed. `L2_CACHE_REFILL` defined as "Each read from or write to the cache that causes a refill from outside the Level 1 and Level 2 caches"[Lima], what is a result of the L2 cache miss. In PREM, it is essential to reduce cache misses on a shared resource to a minimum, because it influences other users (CPU) of that shared resource. So the mentioned memory budget for PREM phase may be represented as the number of permitted cache misses.

The next problem to solve is how to deliver the event from PMU to PL. The investigation done in Section 3.3 narrows the following possibilities:

1. Software or hardware could directly read the event counter value. This option does not fit due to the possible influence on the system being watched.

2. Catch PMUIRQ routed through the Generic Interrupt Controller to the PL. A breaking of the rule of non-invasiveness is also possible here.

3. The other option is to route PMUIRQ using core's CTI directly into PL. Even though the PL has an interface to deliver the CTIINT signal directly, writing to CTIACK (check the name) register should be done

to acknowledge it. A trigger would not fall without the acknowledgment so that the next event will be missed.

4. Use ETM to react on the preset PMU event and then propagate it through ECT all the way up to FTM's interface. Refer to the CoreSight System overview in Figure 3.3.

Also, the watchdog logic should be aware of a memory limit for the currently monitored PREM phase. The only possibility to send the limit value to the PL is to use AXI memory interconnect, so the PL logic will appear as a memory mapped device. This information may then be delivered with a PREMized program, i.e., instructions that will write that value into PL's memory are called at the beginning of every PREM interval.

The Figure 4.1 provides an overview of the final concept. The Figure 4.1 illustrates that every APU's core has its Counter in PL (**X** denotes the CPU ID: e.g., the CPU**3** is connected to ETM**3**, and CTI**3** is connected to CTM at the **third** channel and so on). Counter X counts and acknowledges triggers through FTM's Trigger interface. Counter **X** is an AXI slave, so the message about Limit for processor **X** is delivered from SW through memory mapped interface. Finally, every Counter sends an interrupt through the provided PL to PS interrupt delivery interface in case of the given limit is overflown.

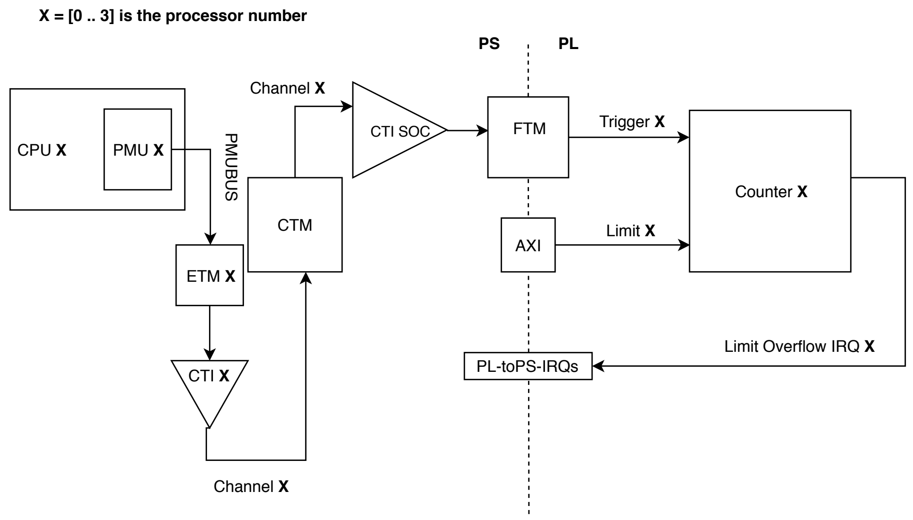


Figure 4.1: The implementation concept overview.

4.2 CoreSight System Configuration

To configure all the CoreSight devices mentioned in Section 4.1, CSAL is used. It adds some abstraction above the memory mapped registers to make the programmer's work easier. Moreover, the support of both Linux and bare metal environment makes the configuration code portable.

The CSAL workflow starts with calling `cs_init()` function which initializes internals of the framework. Then, a device should be registered using the `cs_device_register(cs_phys_addr_t addr)`, where physical device address must be provided as an argument. That function unlocks the device and saves a handle into library's internal structure. CoreSight device unlocking must involve several writes to ARM's registers; the exact algorithm is provided here [Limb]; without the unlocking, the device does not accept writes.

The following sections describe the configuration of single CoreSight blocks to make the modeled concept work.

PMU

The only configuration which every PMU must have is calling `cs_pmu_bus_export()` function. This sets the fifth bit in PMCR register of PMU to allow the event export to ETM.

ETM

The diagram showing the combination of ETM's resources used is provided in Figure 4.2. Firstly, ETM device must be cleared from any previous settings by calling `cs_etm_clean()`. Then, the new configuration structure (`cs_etmv4_config_t`) is created, which consists of register values that soon will be set on ETM. Those registers are configured as follows (refer to ETM register description in [cite]):

1. External Input Select Register bits [4:0] are set with `L2D_CACHE_REFILL` event number + 4 because the first four event numbers mean ETM input triggers. This opens external input 0.

2. Resource Selector 2 Register (may be any from 2nd to 8th because the first two are reserved for other use) is set with the number of external input (0) and the type of resource to select (external input group is set to 0). This selects the External Input 0.

3. Then, one must configure the Event Control 0 Register to connect Resource Selector 2 with External Output 0. Bits [3:0] are set with Resource Selector number.

After all the configuration is done, it must be written to ETM by calling `cs_etm_config_put()`, and ETM must be enabled with `cs_etm_disable_programming()`.

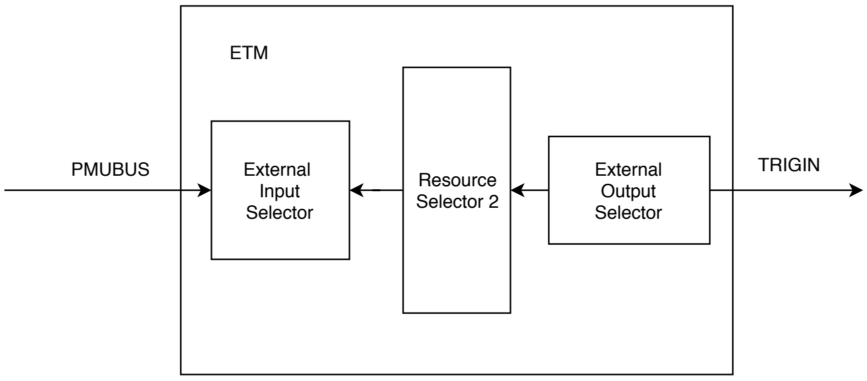


Figure 4.2: ETM's configuration.

ECT configuration

Near-to-CPU CTI is configured to send a signal from Trigger input 4 (which comes from ETM) to channel **X**, where **X** denotes the number of CPU. Thus, every CPU occupies its own channel in CTM matrix. The CTI which is connected to FTM maps channels 0-3 to trigger outputs 0-3 which are connected to FTM. And, FTM does not require any configuration.

4.3 Programmable Logic Design

The main idea behind the logic that is placed in FPGA is to count triggers and signal if the preset limit exceeded. However, to make the solution more effective from a software point of view, it was decided that PREM Watchdog logic should contain three counters for three phases and an ability to switch between phases. This might reduce the maintenance overhead between them. Thus, it should be possible to set all limits at the beginning of PREM compatible interval, and then simply signal to the logic about the current phase (with a write into phase register).

Figure A.2 depicts the whole three-counter logic. `PHASE` register bits [1:0] chooses the current phase, i.e., which counter is currently enabled. Counters counts up, incrementing every time the trigger is high and have not yet been acknowledged. Their values are compared with `LIMIT_PREF`, `LIMIT_COMP`, and `LIMIT_WB` registers. If some value is greater than the limit, the interrupt is set high. Listing 4.1 presents the part of VHDL code implementing this logic.

Listing 4.1: The PREM Watchdog logic.

```
-- Add user logic here
ctr_rst <= '1' when phase = "00" else '0';

-- enable counter for one clock cycle on HIGH level of trigger;
ADDER_EN: for ph in 0 to N_CNTRS -1 generate
 process( S_AXI_ACLK ) is
 variable trigack_pending : std_logic := '0';
 begin
  if (ctr_rst = '1' ) then
   ctr(ph) <= (others => '0');
  else
   if rising_edge( S_AXI_ACLK ) then
    if (unsigned(phase) = (ph + 1)) and (U_PMU_TRIGIN = '1') and
  (trigack_pending = '0') then
     ctr(ph) <= ctr(ph) + 1;
     trigack_pending := '1';
    else
     trigack_pending := U_PMU_TRIGIN;
    end if;
   end if;
  end if;
 end process;
end generate ADDER_EN;

U_LIMIT_IRQ <= '1' when (nor_reduce(phase) = '0') and
( ctr(PREF_CNTR) > pref_limit or
 ctr(COMP_CNTR) > comp_limit or
 ctr(WB_CNTR)   > wb_limit) else '0';

pref_ctr <= ctr(PREF_CNTR);
comp_ctr <= ctr(COMP_CNTR);
wb_ctr   <= ctr(WB_CNTR);

U_PMU_TRIGOUT <= U_PMU_TRIGIN;
-- User logic ends
```

The memory mapped registers require the logic to have AXI interface implemented. Xilinx Vivado IDE has the feature to generate AXI peripheral IP source template: **Create and Package new IP** > **Create AXI4 peripheral**. Due to the simplicity of the device, AXI-Lite interface was chosen. After the generation, the template was successfully integrated with the logic presented in Listing 4.1.

The register for PREM counters are documented in Table 4.1. The whole FPGA design is provided in Figure A.1. The sources of the Vivado project are provided in attachments: (`vivado-and-xsdk/prem_watchdog/`).All the logic works at 100MHz clock.

Name	Offset	Access	Description
PHASE	0x0	RW	Bits [1:0] represents the current phase, i.e., which counter is enabled. "00" – configuration phase: Counters are zeroed and stopped, IRQ is cleared. "01" – prefetch phase: CNTR_PREF is active, all others do not count. "10" – compute phase: CNTR_COMP is active, all others do not count. "11" – writeback phase: WB_COMP is active, all others do not count.
LIMIT_PREF	0x4	RW	Sets limit for CNTR_PREF.
LIMIT_COMP	0x8	RW	Sets limit for CNTR_COMP.
LIMIT_WB	0x10	RW	Sets limit for CNTR_WB.
CNTR_PREF	0x14	RO	Gives the current value of prefetch phase counter.
CNTR_COMP	0x18	RO	Gives the current value of compute phase counter.
CNTR_WB	0x1C	RO	Gives the current value of writeback phase counter.

Table 4.1: PREM Watchdog registers overview. All are 32-bit wide

4.4 Event Masking Problem

FTM trigger interface is asynchronous, and the accepted trigger must be acknowledged. Otherwise, it remains HIGH, so next trigger signal will not be noticed. For the reason of counting events as fast as it is possible, the instantaneous acknowledgment was realized by connecting `TRIGIN` signal directly to `TRIGACK` (see Listing 4.1). However, the signal propagation itself through ECT system lasts some time. The time of the trigger acknowledgment was measure using Vivado IDE and Integrated Logic Analyzer IP (ILA) [?]. The ILA was connected to the net between FTM's `TRIGIN` and `TRIGOUT` port, and it had the sampling frequency equaled 100 MHz (as the rest of the system has). Then, the trigger signal was stimulated from software. Figure4.3 presents the appeared waveform.

The measurement shows that it takes at least 30 ns (three samples of ILA at 100 MHz) for the acknowledged trigger to go LOW. Experiments showed, that reporting once on 8 events almost mitigate the mentioned problem.

The following modification was introduced to reduce the number of lost

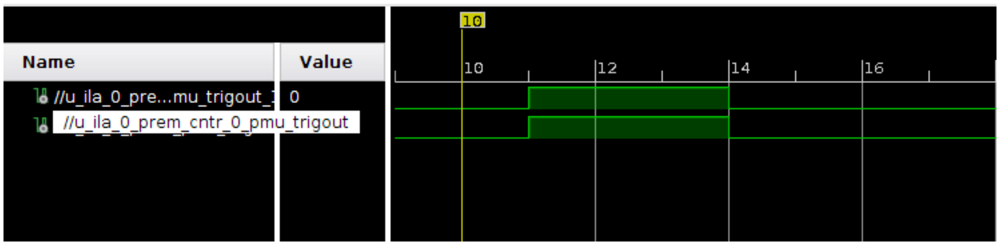

Name
Value
//u_ila_0_pre...mu_trigout_1
0
//u_ila_0_prem_cntr_0_pmu_trigout
10
10
12
14
16

Figure 4.3: The measured time of the Trigger acknowledgment. The sampling period is 10 ns.

events. The counter in ETM is used as an event buffer. 8 events are buffered before sending one signal to trigger system which means that 8 events are occurred. Further two resources were added to ETM's configuration to achieve this (see Figure 4.4 and refer to register reference in [Lime]) – Counter and another Resource Selector 4. In this configuration, Counter 0 is set to decrement when an event on Resource Selector 2 occurs (bits [7:0] of Counter Control Register) and to self-reload when reaching zero (bit 16 of Counter Control Register). The both Counter Reload Value Register and Counter Value Register are set to (8 - 1). Then, Resource Selector 4 selects Counter 0 (the same manner as described in Section 4.2, but the group is different – `0b0010`). And, finally, Resource Selector 4 is added to fire at the output through Event Control 0 Register.

This modification obviously requires the software developer to be aware of it when he sets or reads the limit values or counter values. But all of that is easily mitigated by simply multiplication/division on a predefined constant (e.g., in sources attached to that work `ETM_EVENTS_BUFF_NUM` constant is used for that purpose).

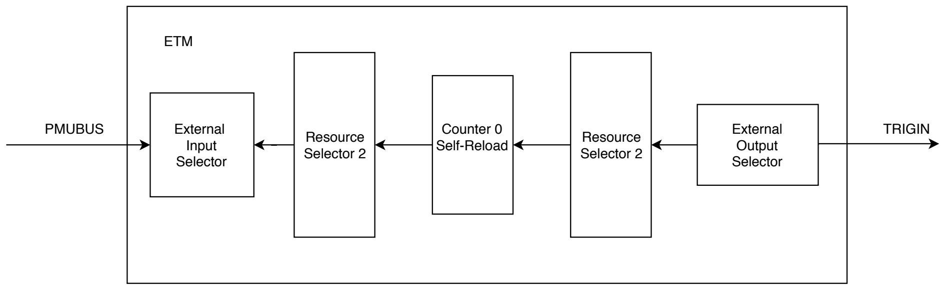


Figure 4.4: The modified ETM's configuration concept overview.

4.5 PL Logic Driver API

Listing 4.2: The header file of the software driver for the implemented PL logic.

```c
#ifndef SRC_PREM_COUNTER_H_
#define SRC_PREM_COUNTER_H_

#include <stdint.h>

#define PHASE_WHEN_USED_AS_COUNTER 1
#define PMUBUS_EVENT (21U + 4U)

#define PREM_PHASE_CONF 0b00
#define PREM_PHASE_PREF 0b01
#define PREM_PHASE_COMP 0b10
#define PREM_PHASE_WB   0b11

#define ETM_EVENTS_BUFF_NUM 8U

typedef uint32_t prem_phase_t;

typedef struct prem_conf {
uint32_t lim_prefetch;
uint32_t lim_compute;
uint32_t lim_writeback;
} prem_conf_s;

void prem_configure(uint32_t cpu, prem_conf_s * config);
void prem_set_phase(uint32_t cpu, prem_phase_t phase);
void prem_print_state(uint32_t cpu);

void cs_prem_count_init();
void cs_prem_count_percpu_init(uint32_t cpu);
void _deprecated_pl_count_reset(uint32_t cpu);
uint32_t _deprecated_pl_count_read(uint32_t cpu);

#endif /* SRC_PREM_COUNTER_H_ */
```

The driver interface for the proposed hardware is simple (See Listing 4.2).

To employ that logic in a PREM application, one should place the function calls at the beginning of PREM phases, e.g. `prem_set_phase(1, PREM_PHASE_WB)` to let the PREM watchdog know that write-back phase counter for CPU1 should count now. At the very beginning of program execution, CoreSight logic should be initialized with call firstly `cs_init()` to init CSAL, then `cs_prem_count_init()` and then `cs_prem_count_percpu_init()` for every CPU that should be monitored. The `prem_configure()` function must be called at the beginning of PREM interval: counters will be reseted and limits are set for the next three phases.

Chapter 5

Performance Evaluation

5.1 Precision evaluation

Regardless of problems, mentioned in Section 4.4, the PREM Watchdog must work correctly with a reasonable and determinable tolerance. The simple testbench was proposed to evaluate that. It simulates random-write memory accesses of various working set size (WSS), including the single and multi-threaded use-case. The test goes as follows:

1. The CoreSight system is set up as stated in Section 4.2 with modifications proposed in Section 4.4 .
2. The PMU units (for every processor) are set up to count the same event that PREM watchdog counter monitors.
3. Set PREM watchdog at some phase, does not matter which one. The correctness of phase switching is not tested here, only the counter precision matters.
4. At the start of every benchmark session, meaning – for every WSS tested, reset both PMU counter and PREM watchdog (by setting its phase at "00", and then back at predefined phase).
5. At the end of benchmark session, read the both PMU and PREM watchdog's counters value and calculate the absolute difference.
6. Present the results.

The source code of the proposed tests is provided in attachments to this work (`vivado-and-xsdk/prem_watchdog/prem_watchdog.sdk/test-Membench-linux`).

Tests were launched in one-, two-, three- and four-thread configurations inside Linux OS, every thread accessed the memory of WSS from 1KB up to 18MB. The results are presented in Figure 5.1, where the absolute difference of PMU value from PREM value is denoted as an "Error". The absolute frequencies of appearing of Error calculated across every launch measurements are the values at Y-axis. The error generally grows with the frequency of counted events (cache misses), and this could be seen from the Figure 5.1: Involving more threads leads to more cache misses. Thus, bigger Error values occur more frequently. With the number of cache misses measured the maximum error frequency grows also.

It could be seen from the Figure 5.1 that worst-case error measured in tests is 38 lost events. Most likely, only two events are lost. The average value of error from all measurements is 10 events, which is explainable. 8 events are buffered, so it is expected to lost 8 events in the worst case. From the measurements, it is also follows, that the error value is negletable due to WSS at that the error occurs. For instance, WSS=16MB and error in 38 cache misses events. Assuming, that 1 cache miss means fetching, e.g. 512B, the uncertainty is only (512 * 38) / (16 * 1024 * 1024) ≈ 0.1 % of WSS.

The observational error also influences the results of the benchmark (e.g., it is not possible to refresh all counters at once, so during the refresh of one counter other may increment.).

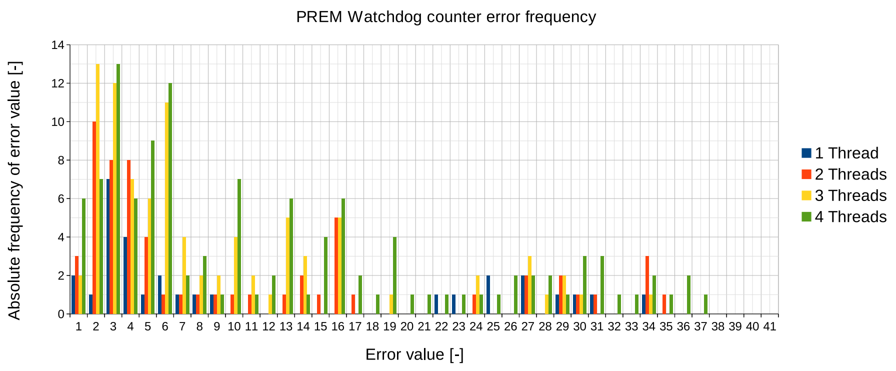


Figure 5.1:

5.2 Comparison with software-only watchdog

The primary source of overhead what the introduced implementation should compete with is a hypercall from Jailhouse hypervisor. Hypercall is an expensive operation. It was mentioned in Section 2.2 that Paolo Gai demonstrated [Gai17] the hypercall jitter on their full-loaded setup as the following: **5.47** μs minimum, **21.72** μs in average and up to **58.42** μs maximum. Those values, however, are quite pessimistic, because author of the experiment mentioned that their setup does not yet run PREM model, and, obviously their setup is loaded with ADAS-like task.

In case of FPGA-based watchdog implementation, the overheads are the writes to phase register at the beginning of every phase and writes of limits at the beginning of the predictable interval. Thus, it should be measured how long does it take for software in Linux to write to four registers at PL, to write to one register at PL, and, possibly, to read four registers at once or the only one. The source code of the simple application that does so is provided in attachments (`vivado-and-xsdk/prem_watchdog/prem_watchdog.sdk/test-regs-app`). The application runs in Linux user-space.

In this test, much lower values of hypercall time are received. Jailhouse hypervisor cell contains a bare metal program, which measures the hypercall time. That program is presented in Listing 5.1. The rest of the system (Linux OS) has not been specifically loaded in any way.

The results are presented in Table 5.1. As it could be seen, even in the case of pure hypercall time (without any side load nor work done into actual hypercall) the PL register writes are much faster (**2.88** times in average). Hence, the software overhead is much less in case of HW implemented watchdog.

	Read 1 reg [ns]	Write 1 reg [ns]	Read 4 reg [ns]	Write 4 reg [ns]	Hypercall [ns]
min	330	100	1040	140	369
average	373.4	101.4	1231.3	143.8	415
max	380	110	1250	520	719

Table 5.1: The comparison of PL Watchdog with hypervisor watchdog.

Listing 5.1: The hypercall measurement code.

```c
#define REPEAT 100
void inmate_main(void)
{
    int i;
    printk("Initializing the timer...\n");
    u64 sum=0,min = 9999, max = 0;
    for (i = 0; i < REPEAT; i++)
    {
        before = timer_get_ticks();
        jailhouse_call_arg2(9,99999,1);
        after = timer_get_ticks();
        long actual = timer_ticks_to_ns(after-before);
        sum += actual;
        if(actual > max)max = actual;
        if(actual < min)min = actual;
        printk("time was %6ld ns\n",actual);
    }
    printk("min -> %6ld ns\n",min);
    printk("max -> %6ld ns\n",max);
    printk("avg -> %6ld ns\n",sum/REPEAT);
    while (1)
        asm volatile("wfi" : : : "memory");
}
```

Chapter 6
Conclusion

This thesis has studied the problem of implementing execution monitoring mechanism on using Xilinx Zynq Ultrascale+ platform. The mechanism supports application of Predictable Execution Model (PREM) on that platform. Based on this investigation, the FPGA logic behaving as PREM memory budget monitor was successfully implemented. The implemented hardware logic in pair with specifically configured on-SoC subsystems can non-invasively count cache misses which occur at individual cores of the multiprocessor system.

The overheads that the proposed solution brings to the system were evaluated. The results show that in case of using FPGA-based memory watchdog maintenance takes 2.88 times less than the hypervisor-based solution requires in average (the hypercall time). Hence, the statement that HW-based guard may decrease the overhead of PREM application when compared to the software-based guard is proven.

The presented implementation solves the problem of missing some monitored events when their frequency is too high. This was caused by propagation delays in the SoC's debug system. The solution to this problem was implemented and tested, however, reliability of the solution needs to be further investigated due to limitations of available measurement methods.

As for the future work, it might be worth investigating an alternative approach of using CoreSight Trace Port Interface connected to the Programmable Logic (FPGA). This would require implementing trace stream decoder in the FPGA, but it will allow to count multiple types of memory events at once. Also beginning of PREM phases could be detected using Embedded Trace Macrocell's Address comparators instead of register writes, which might (or might not) have even less overhead.

Appendix A

Bibliography

[AP14] A. Alhammad and R. Pellizzoni, Time-predictable execution of multithreaded applications on multicore systems, 2014 Design, Automation Test in Europe Conference Exhibition (DATE), March 2014, pp. 1–6.

[bADASA] Xilinx Camera based Advanced Driver Assistance Systems (ADAS), [online]`https://www.xilinx.com/applications/automotive/adas.html`, Accessed: 2018-05-17.

[BBC+17] Paolo Burgio, Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli, Michal Sojka, Přemysl Houdek, Andrea Marongiu, Paolo Gai, Claudio Scordino, and Bruno Morelli, A software stack for next-generation automotive systems on many-core heterogeneous platforms, Microprocessors and Microsystems **52** (2017), 299 – 311.

[Gai17] Paolo Gai, Multi-os demo on nvidia jetson tx1 with jailhouse hypervisor, erika enterprise 3 and linux, [online]`https://www.youtube.com/watch?v=skIcAkXfNWQ`, 2017, Accessed: 2018-05-16.

[GCAL] GitHub - CoreSight Access Library, [online]`https://github.com/ARM-software/CSAL`, Accessed: 2018-05-16.

[GOAosCTDl] GitHub - OpenCSD - An open source CoreSight(tm) Trace Decode library, [online]`https://github.com/Linaro/OpenCSD`, Accessed: 2018-05-16.

[HSH17] P. Houdek, M. Sojka, and Z. Hanzálek, Towards predictable execution model on arm-based heterogeneous platforms, 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE), June 2017, pp. 1297–1302.

[Inca] Xilinx Inc, AXI Performance Monitor v5.0 LogiCORE IP, Product Guide (PG037).

[Incb] ______, Integrated Logic Analyzer v6.1 LogiCORE IP, Product Guide (PG172).

[Incc] Xilinx Inc., Zynq MPSoC Ultrascale+, Technical Reference Manual v1.7.

[Lima] ARM Limited, ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile, howpublished = Technical Reference Manual.

[Limb] ______, ARM coresight Architecture specification v2.0, (IHI0029D).

[Limc] ______, ARM Coresight Components Technical Reference Manual, Technical Reference Manual.

[Limd] ______, ARM Cortex-A53 MPCore Processor Technical Reference Manual, Technical Reference Manual.

[Lime] ______, ARM Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.3, 2017.

[Limf] ______, Corelink NIC-400 Network Interconnect, Technical Reference Manual.

[Lim13] ______, CoreSight Technical Introduction, Tech. report, 2013.

[MFS$^+$18] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu, Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution, Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (New York, NY, USA), PMAM'18, ACM, 2018, pp. 11–20.

[MSH17] Joel Matějka, Michal Sojka, and Zdeněk Hanzálek, Hypervisor structure & predictable application scheduling, Retrieved form Michal Sojka, 2017.

[PBB$^+$11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley, A predictable execution model for cots-based embedded systems, Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (Washington, DC, USA), RTAS '11, IEEE Computer Society, 2011, pp. 269–279.

[Pro] Hercules Project, [online]`https://hercules2020.eu/`, Accessed: 2018-05-16.

[UXSCT] Using Xilinx SDK - Cross Triggering, [online]`https://www.xilinx.com/html_docs/xilinx2017_4/SDK_Doc/SDK_concepts/concept_cross_triggering.html`, Accessed: 2018-05-16.

[UXSCTiZUM] Using Xilinx SDK - Cross-Triggering in Zynq UltraScale+ MPSoC, [online]`https://www.xilinx.com/html_docs/xilinx2017_4/SDK_Doc/SDK_references/reference_cross-trigerring-zynqmp.html`, Accessed: 2018-05-16.

[UXSFAuS] Using Xilinx SDK - FreeRTOS Analysis using STM, [online]`https://www.xilinx.com/html_docs/xilinx2017_4/SDK_Doc/SDK_tasks/sdk_freertos_analysis.html`, Accessed: 2018-05-16.

[YYP+13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2013, pp. 55–64.

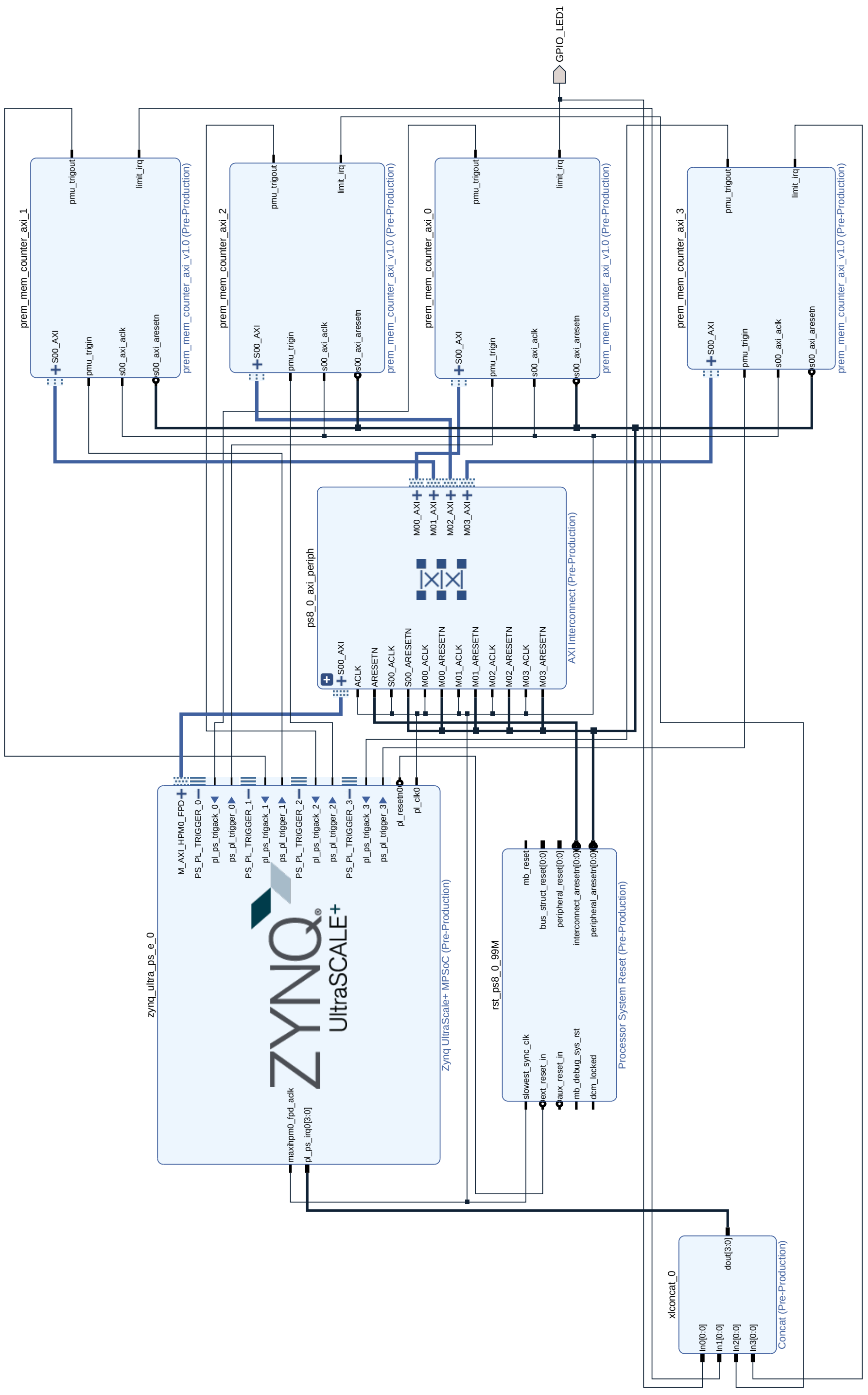


Figure A.1: The complete PL design.

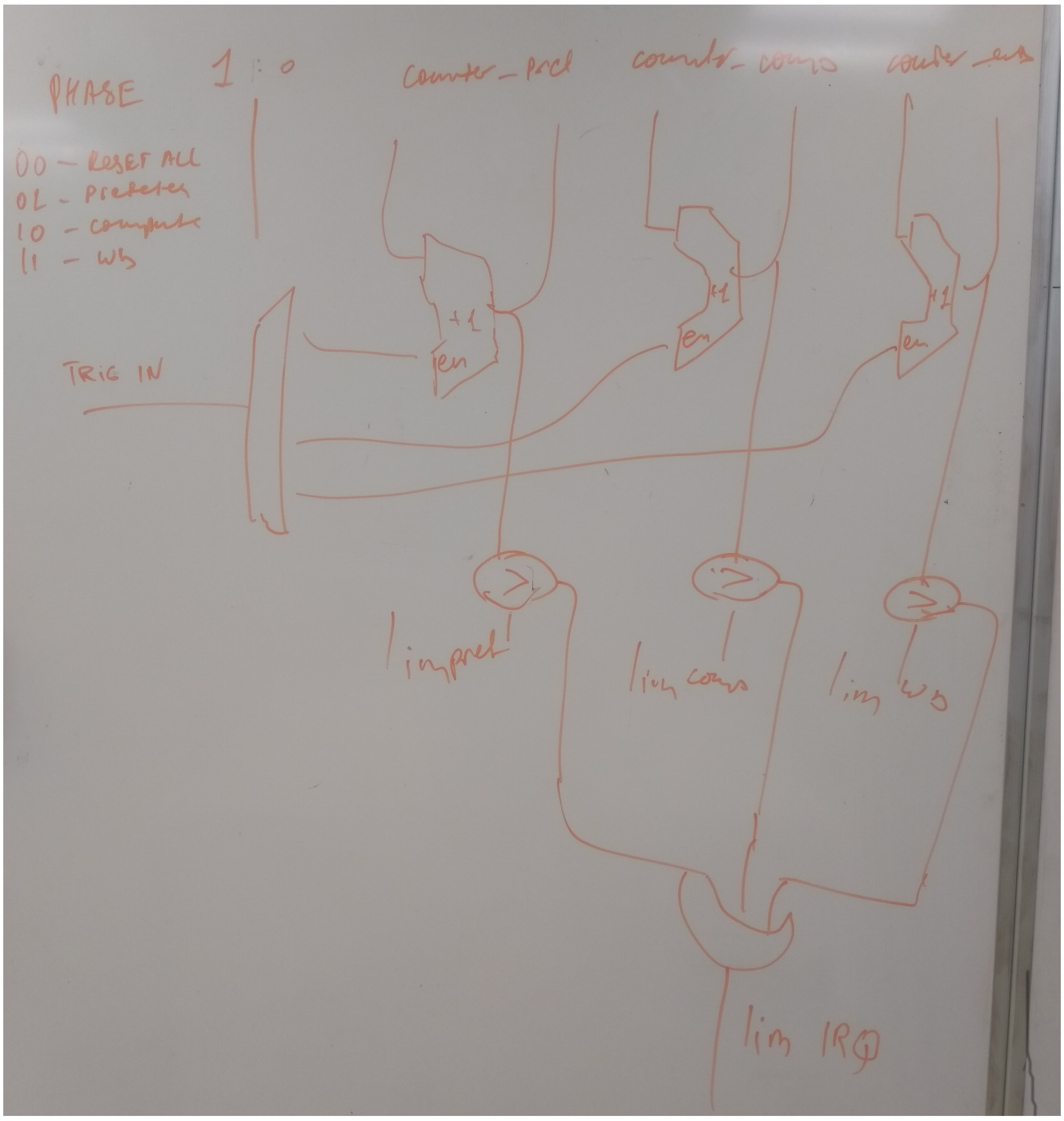


Figure A.2: The PL logic concept.