Runtime Verification with TeSSLa on Enzian

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 245

Systems Group, Department of Computer Science, ETH Zurich

Runtime Verification with TeSSLa on Enzian

by

Pirmin Schmid

Supervised by

Prof. Timothy Roscoe, Dr. David Cock

February 2019 – August 2019

Informatik
Computer Science

Abstract

Runtime verification (RV) is a methodology to verify whether the behaviour of a (software) system satisfies the defined properties of a specification. This is of interest for safety-critical reactive systems. Multiple temporal logic systems, specification languages, and complete RV systems have been developed. Many of them work offline on collected program traces or logs.

Program traces of ARM CPUs can be accessed at runtime with the ARM CoreSight infrastructure. Other chip architectures offer similar trace functionality. High bandwidth requirements limit the amount of trace data that can be processed at runtime.

Field Programmable Gate Arrays (FPGA) can process data with high throughput. CPU/FPGA hybrid systems (like the Xilinx Zynq-7000 SoC with 2 ARMv7 cores) allow direct access to this CS trace on the FPGA. The new Enzian system of the Systems Group at ETH Zürich, which is under development, combines a server-grade 48 core Cavium, now Marvell, ThunderX SoC with a Xilinx Virtex UltraScale+ FPGA that are connected via cache coherency protocol on the last level cache (LLC) of the ThunderX.

The recently published Temporal Stream-based Specification Language (TeSSLa) is designed to allow processing on data streams, which is in particular suitable for data processing on FPGA. A prototype TeSSLa to Verilog compiler via Chisel – developed at the Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany – could be used in this project to build a new RV system.

This new RV system parses CS traces (new modules written in VHDL) and evaluates them with the embedded TeSSLa specification on FPGA, which allows online runtime verification of programs. An additional instrumentation library for glibc functions and incorporation of signals of the FPGA allow writing specifications even for CPU/FPGA hybrid applications. The instrumentation library wraps the program at load-time. Thus, no source code is required for the program-under-test.

The evaluation of the system includes runtime verification of 3 specifications in different application domains: 1) memory allocation (all allocated memory is deallocated); 2) event handler (maximum queueing time); 3) access to critical sections protected with locks in multi-threaded programs. Test programs were written to satisfy and violate the properties of the specifications. All evaluations were correct.

The system is compared with existing RV solutions in the thesis.

The prototype system on the Zynq board could not yet be transferred to the Enzian system due to external delays. A clear upgrade path for Enzian is described.

A critical bottleneck could be removed during this project. The TeSSLa InputAdapter could accept one multiplexed input channel, which required merging of all input streams. It could be extended to accept multiple such inputs from different sources in parallel. This required writing a timestamp driver to keep the TeSSLa computation network running. Removing this merge bottleneck has been crucial to scale the RV solution to multi-core systems like Enzian or even rack-scale runtime verification.

Acknowledgment

I would like to thank my advisors Dr. David Cock and Professor Dr. Timothy Roscoe of the Systems Group at ETH Zürich. Working in their team has been a dream come true. I appreciate the discussions, advice and feedback during the project, and the many things I could learn, not only on technical things.

I am very grateful for the excellent collaboration with Malte Schmitz and Daniel Thoma, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany. I appreciate the interesting Skype discussions during the semester and their testing of the prototype also on their board.

I dedicate this thesis to my parents. I am very grateful for the support.
I hope to get close to your wisdom sometime in life.

Contents

List of Figures viii

List of Tables ix

Listings x

1 Introduction 1
- 1.1 Focus 2
- 1.2 Main contributions 2
- 1.3 Thesis layout 3

2 Background 5
- 2.1 Runtime verification 5
- 2.2 ARM CoreSight 6
- 2.3 TeSSLa: Temporal Stream-based Specification Language 7
- 2.4 CPU/FPGA hybrid systems 8
  - 2.4.1 FPGA characteristics that affect the design of the RV system 8
  - 2.4.2 Zynq board 9
  - 2.4.3 Enzian system 9
- 2.5 Instrumentation 10

3 Related Work 13
- 3.1 Existing tracing and runtime verification infrastructure 13
- 3.2 TeSSLa implementations 13
- 3.3 Other CPU/FPGA hybrid implementations 14
- 3.4 Other runtime verification systems 15
- 3.5 Other verification systems 15

4 Design 17
- 4.1 Design overview 17
- 4.2 FPGA: design constraints and design decisions 17
- 4.3 Online and distributed runtime verification 19
- 4.4 AXI/TPIU controller and driver 20
- 4.5 CoreSight parsers 20

4.6 Timestamp driver and interface to TeSSLa . . . 20
4.7 TeSSLa specification . . . 21
4.8 Software tooling . . . 24
4.9 Instrumentation library . . . 24

5 Implementation 25
5.1 CoreSight parsers . . . 25
5.1.1 CS frame synchronizer and parser . . . 25
5.1.2 PTM parser . . . 26
5.1.3 ITM parser . . . 27
5.1.4 FTM parser . . . 27
5.2 Timestamp driver . . . 27
5.3 TeSSLa specification . . . 29
5.3.1 Some technical details of the inner working of TeSSLa on FPGA . . 29
5.3.2 TeSSLa compiler . . . 30
5.3.3 MultiInputAdapter . . . 30
5.3.4 OutputFilterAdapter . . . 31
5.4 AXI/TPIU controller and driver . . . 31
5.5 Software tooling . . . 31
5.6 Instrumentation library . . . 32

6 Evaluation and Benchmarking 33
6.1 Methods . . . 33
6.1.1 CoreSight configuration . . . 33
6.2 CS parser validation . . . 34
6.3 Instrumentation validation and benchmarking . . . 35
6.3.1 Validation . . . 35
6.3.2 Benchmarking . . . 37
6.4 Runtime verification with TeSSLa specifications . . . 40
6.5 Summary . . . 48

7 Discussion 49
7.1 Instrumentation overhead . . . 49
7.2 Limitations of the TeSSLa compiler prototype . . . 49
7.3 Related work . . . 51
7.4 Future work . . . 52
7.4.1 Runtime verification of CPU/FPGA hybrid applications . . . 52
7.4.2 Migration path to ThunderX / Enzian . . . 52
7.4.3 TeSSLa to Verilog compiler . . . 53

8 Conclusion 55

Appendices 55

A Reference TeSSLa Interfaces 57

B TeSSLa Language Snippets 61

C Timestamp Driver **65**

D trace_launch Options **71**

E Instrumentation Library **73**

F Programs for Trace Validation **75**

Bibliography **79**

List of Figures

2.1 Runtime verification 5
2.2 ARM CoreSight on Zynq board 6

4.1 FPGA modules: design overview 18
4.2 A simple TeSSLa computation network (Listing F.2) 21
4.3 Count macro: computation network (Listing 4.1) 22
4.4 TeSSLa computation network with unrolled map of capacity 8 23

5.1 Data flow in PTM parser 26

6.1 Experiment 6: trace density plots of instrumented and non-instrumented (baseline) program traces 37

List of Tables

2.1 Instrumentation opportunities 10

4.1 FPGA modules: clock frequencies and associated bandwidths 19

5.1 Multiplexed CS streams in frames: IDs used in CS configuration 26
5.2 Encoding of multiplexed TeSSLa event tuples (input / output) 29

6.1 Runtime verification with TeSSLa specifications: overview 41
6.2 Summary: instrumentation overhead 48

Listings

4.1 TeSSLa: count macro from stdlib 21
6.1 Filter macro 37
6.2 Memory allocation 41
6.3 Event handler: queueing time property, simple 42
6.4 Event handler: queueing time property, advanced 43
6.5 Critical section protected by lock 46
A.1 TeSSLa interface: reference input interface 57
A.2 TeSSLa interface: instrumentation library reference mapping 59
A.3 TeSSLa interface: reference output interface 60
B.1 TeSSLa: helper predicates 61
B.2 TeSSLa: function_call() macro 62
B.3 TeSSLa: fixed-capacity set 63
B.4 TeSSLa: fixed-capacity map 64
C.1 Timestamp driver 65
D.1 trace_launch options 71
F.1 Test program: nested_malloc 75
F.2 Relevant TeSSLa snippet used for nested_malloc 76

Chapter 1

Introduction

Runtime verification (RV) is a methodology to verify whether the behaviour of a system, e.g. a running software program, satisfies the defined properties of a specification [25]. This is of interest for safety-critical reactive systems as used in many embedded systems in cars, aeroplanes, or medical devices [27, 32]. Multi-core CPUs add to the complexity of possible system behaviours.

Other applications can be found in system security, in particular if users are allowed to execute foreign code, or many users are sharing the same computing infrastructure, e.g. in the cloud [24]. Deviations from specified patterns should be recognised as early as possible, or better prevented.

Runtime verification comes with a computational overhead that depends on various factors, e.g. the complexity of the specification. Due to this complexity, such verification can happen offline by analysis of logs or program traces. For many applications, online verification is desired while the system is running.

On ARM CPUs, the CoreSight (CS) infrastructure offers access to program traces at runtime, which can be processed by different cores of the same machine, or a different machine. However, program traces of CPUs can have large bandwidth requirements. Already a relatively slow Xilinx Zynq board (2 ARM cores at 1 GHz) can generate highly compressed raw processor traces at 260 MB/s (or more if additional options are activated).

Required bandwidth increases with more cores and higher clock frequency as e.g. in the 48 core Cavium, now Marvell, ThunderX SoC used in the new Enzian platform [16]. Scaling a runtime verification solution to a rack of fast servers with many cores creates severe bottlenecks if trace data cannot be processed early to reduce the data stream to relevant specification-defined signals [9].

Hybrid CPU/FPGA systems, such as the Zynq board or the Enzian system, provide an opportunity to implement the runtime verification in the Field Programmable Gate Array (FPGA). This allows direct local processing and data reduction without adding load to the CPUs. FPGA systems are well-suited for processing of data streams with high throughput if designed well [28].

The recently published Temporal Stream-based Specification Language (TeSSLa)[1] [10, 11] was chosen to be combined with a new CoreSight trace parser for FPGA, that was created in this thesis, to build a RV system for CPU/FPGA hybrids.

1.1 Focus

While the CoreSight ROM can be read on the ThunderX SoC, chip configuration itself – such as which cores to be traced – depends on a proprietary non-standard system. The ARM trusted firmware (ATF) for the SoC needs to be patched and a specific library needs to be used for configuration and to read traces. We did not get this patch and library until 4 months of this 6 month project have passed by. Additionally, several needed software and hardware components for the Enzian system are not yet available.

Due to these external circumstances and in accordance with the thesis supervisors, the focus of this thesis has been shifted from a direct implementation for the future Enzian system to a prototype on a Zynq board. This prototype provided the opportunity to solve many problems that are identical for the Enzian CPU/FPGA hybrid system.

The current system is fully functional on the Zynq board to parse CS traces and process them according to embedded TeSSLa specification on the FPGA board. The prototype could also be used on a Zed board by our colleagues in Lübeck.

Several key elements are prepared for Enzian. A clear upgrade path to Enzian is described in this thesis.

1.2 Main contributions

During the 6 months of this thesis project, a working system could be built for runtime verification of arbitrary binary programs running in a current Linux environment using a new CoreSight parser implementation in FPGA integrated with a TeSSLa specification processed in FPGA. The solution includes observation of long running multi-threaded programs, optional instrumentation of already compiled and linked programs, and contributes several insights in programming TeSSLa specifications.

The existing input interface for TeSSLa was expanded to accept multiple inputs in parallel. This is a critical step to scale TeSSLa to high bandwidth inputs of multiple program traces of a server-grade multi-core SoC like ThunderX. This extension could be accomplished with a new timestamp driver for the TeSSLa computation network.

The instrumentation library uses ITM stimuli of the CoreSight system. The ARM CS access library was extended to also integrate stimuli from FPGA into the CS stream (FTM), which is beneficial for CPU/FPGA hybrid applications. This additional data can be used to write more complex TeSSLa specifications. Several new TeSSLa macros were designed to accomplish these specifications.

Various examples were tested to illustrate runtime verification with this new system.

[1] TeSSLa has been developed at the Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany; `https://www.tessla.io` for information, documentation, and an online playground to test specifications

1.3 Thesis layout

Chapter 2 gives relevant background information for this thesis, and chapter 3 discusses related work. The design and implementation of the new RV system is shown in chapters 4 and 5, respectively.

The complete system is evaluated and benchmarked in chapter 6. This includes runtime verification of 3 properties in different application domains using TeSSLa specifications.

- Memory allocation: all allocated memory is deallocated
- Event handler: maximum queueing time
- Locks: access to critical section protected

The discussion (chapter 7) compares the results with related work, discusses current limitations and possible solutions, suggests future work, and in particular describes an upgrade path to a ThunderX / Enzian system. Chapter 8 concludes the thesis. The appendices give additional detail information.

Chapter 2

Background

2.1 Runtime verification

Runtime verification (RV) is a methodology to verify whether the behaviour of a system, e.g. a running software program, satisfies the defined properties of a specification (Figure 2.1). Output can be as simple as a boolean flag whether the trace satisfies the specification so far or a violation has been observed. In particular with advanced specification languages like TeSSLa, outputs can be arbitrarily detailed.

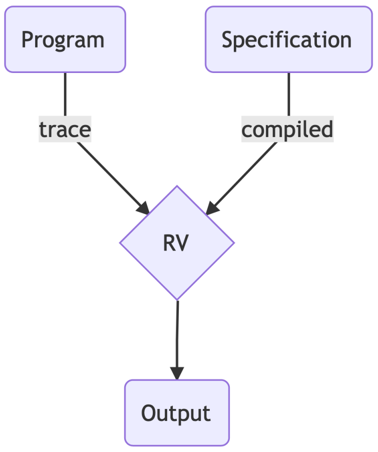

Figure 2.1: Runtime verification

Runtime verification must be distinguished from other forms of software verification like formal program verification with theorem provers, model checking, or static analysis of programs [20, 25]. The formal verification of the seL4 kernel is an excellent example of such a formal verification effort [23]. These methods aim to guarantee adherence to the given specification for all possible program traces. In contrast, runtime verification can only verify the program traces that are effectively observed at runtime.

However, there is a second side to this. Formal verification must assume an underlying execution model for the program to be correct. Runtime verification can detect violations against the specifications in case of bugs in this underlying execution model or attacks against the system, which may affect even theoretically correct/verified programs. Thus, there is a need for both types of verification [17].

Program testing (e.g. test-driven development and also fuzzing tools) is another method to reduce the amount of errors in a program. However, ”Testing shows the presence, not the absence of bugs” (quote from E. W. Dijkstra 1969 in [13]). There are conceptually inherent false-negatives in testing and false-positives in static analysis. Runtime verification can be combined with test cases or even fuzzing tools during development that explore a wide range of program behaviour.

Historical note: Conceptually, runtime verification is not restricted to software, and a wide definition of this concept has been used with other names as long back as humans have used tools. However, runtime verification has been formally defined now, and multiple logic system and specification languages / frameworks have been developed [25]. Various RV systems are discussed in section 2.3 and chapter 3.

2.2 ARM CoreSight

ARM processors are designed to offer trace information at runtime that can be used for debugging and verification: ARM CoreSight architecture [2, 5]. Such traces are also available for other CPU architectures, e.g. Intel Processor Trace [21].

The CoreSight (CS) system offers configuration options to select data sources (e.g. program trace of one or several CPU cores) and output sinks to read the data (Figure 2.2). An open source library, CoreSight Access Library (CSAL), is available for configuration [6]. Another library, OpenCSD, can be used to parse raw traces [26].

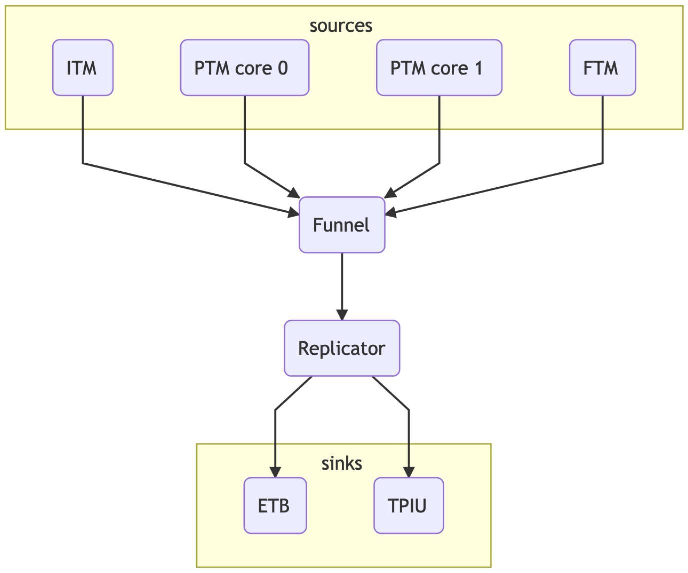

Figure 2.2: ARM CoreSight on Zynq board

The number and type of sources and sinks differ for different platforms. A Program Trace Macrocell (PTM) offers the program trace of a CPU core. Many systems offer an Instrumentation Trace Macrocell (ITM) that can be used to send values as software stimuli into the CS trace [2]. This system is used for the instrumentation library in this project. The Embedded Trace Buffer (ETB) is 4 KiB large and can be used to cache trace data of short programs.

In addition to other ARM SoC, the Xilinx Zynq board offers a source input that can be used by the FPGA to send values into the CS trace: Fabric Trace Monitor (FTM) [35]. The board also adds a second output: a Trace Port Interface Unit (TPIU). This interface is available in the FPGA via EMIO interface. It is used to read the raw trace for processing on the FPGA.

Trace data is highly compressed. The trace parser is expected to maintain an internal state to be able to decompress data such as sent address values. Typically, an ARM target system is debugged using a separate device,[1] that is connected with the target system via JTAG and with the host computer via USB or network connection. The entire source and binary codes are available on the host computer. Using the control flow graph (CFG) of the program, the debugger on the host computer can fully restore the program execution for arbitrary programs based on the compressed trace stream.

I refer to the online available documentation by ARM and Xilinx for the details of the binary formats of frame and packets that were used to write the CS parsers: Frame (Trace formatter, [5]), PTM [3], ITM [2], FTM [35].

Standard CS systems can be configured using CSAL. The CS ROM of the SoC provides key information needed for configuration.

The outside interface of the ThunderX CS system follows the CS standard, which can be read from the embedded CS ROM in the SoC. However, the internal configuration – e.g. which of the 48 cores to be active sources – is non-standard and proprietary.

2.3 TeSSLa: Temporal Stream-based Specification Language

Various specification languages and logic systems have been developed that can be used for runtime verification [25].

A formula written in linear temporal logic (LTL) can be translated into a Büchi automaton. This is used to analyse the collected program trace to determine whether the program execution satisfies the specification or violates it. Due to the complexity of these automata, this analysis typically happens offline after program execution; example [29].

The Temporal Stream-based Specification Language (TeSSLa) [10], developed at the Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany, offers an excellent option to write more complex specifications than with LTL. It was designed for specifying and analysing the behaviour of cyber-physical systems, where timing is a critical issue [10]. It is in particular well-suited for specifications in asyn-

[1] e.g. ARM DSTREAM: `https://www.arm.com/products/development-tools/debug-probes/dstream`, or an In-Circuit Emulator (ICE)

chronous settings. Time can be quantified and thus specific constraints be expressed in the language [10].

It extends a former language LOLA [12, 14] that introduced the idea of stream-based runtime verification and stream transformations specified via recursive equations [11]. TeSSLa extends these concepts from synchronous to additionally asynchronous events [11].

TeSSLa is well suited to run on FPGAs thanks to its design to process event streams in parallel. In principle, each event consists of a (timestamp, data value) tuple. In contrast to data stream processing systems, e.g. used for database applications [8, 28], timestamps are an integral and critical part of each event.

The implemented TeSSLa computation network (examples shown in Figures 4.2 to 4.4) needs monotonically increasing timestamps on all inputs to make progress. This led to subtle details that needed to be considered when the `InputAdapter` with one input was extended to handle multiple inputs without need to merge all data streams before input to TeSSLa network (implementation in subsection 5.2).

TeSSLa already had working implementations before the start of this project: 1) software implementation written in Scala that can be tested in an online playground, too; 2) an implementation in FPGA that differs from the implementation in this project. These implementations are described in section 3.2 (chapter Related Work).

Alternative use case

Additionally, trace information cannot only be used for verification or security purpose but also for resource monitoring and management. Such monitoring and decision algorithms can be written in TeSSLa for local data reduction and just forwarding of occasional management signals.

2.4 CPU/FPGA hybrid systems

2.4.1 FPGA characteristics that affect the design of the RV system

Data can be processed with high throughput and low energy consumption with a Field Programmable Gate Array (FPGA). However, specific characteristics need to be considered to achieve that goal [28]:

- relatively low clock frequency
- very wide data bus widths can be used
- parallel processing of multiple data streams
- clock frequency limited by longest path between registers (key words: setup time, combinational logic, holding time)

Therefore, computations need to be split into smaller processing steps that are connected together in a pipeline to achieve high clock frequencies.

Additionally, data should be processed in parallel. However, the design must assure that packets – that were processed in parallel by different parsers with different numbers of processing steps – remain synchronised as input for the downstream application such as the TeSSLa processing network. This was achieved by using an internal clock in the RV system that assigns a timestamp to CS frames in the module that reads the raw CS trace from the TPIU port.

2.4.2 Zynq board

A Xilinx ZC706 evaluation board for the Zynq-7000 XC7Z045 SoC[2] (short Zynq board; technical reference documentation [35, 36]) was used in this project. It integrates a processing system (PS; ARM Cortex A9 based application processor unit with 2 cores; ARMv7-A architecture; 1 GHz) and a programmable logic (PL; fabric, FPGA; speed grade -2) on a single die. Each side has 1 GB of RAM available.

The FPGA can access the ARM CoreSight stream of the PS via TPIU interface. Additionally, it can embed signals into the CS stream via FTM. The AMBA AXI interface was used to control the FPGA from a linux kernel module that provides a character device.

The system was configured to boot a Xilinx specific Linux kernel 4.4-xilinx[3] and an Ubuntu 18.04 LTS user environment. The kernel was compiled with the option to write processID into the Context ID register (CONTEXTIDR) using 24 bits above the 8 bit ASID).[4]

The system was accessed via SSH in the local network.

2.4.3 Enzian system

Enzian is a new CPU/FPGA hybrid platform that is being developed by the Systems Group at ETH Zürich [16]. It integrates a 48 core server-class ARM processor (Cavium, now Marvell, ThunderX[5]) with a large and fast FPGA (Xilinx Virtex UltraScale+[6] XCVU9P; speed grade -3).

ThunderX has a native interconnect for the cache coherency protocol that can connect two ThunderX in separate sockets at the last level cache (LLC). In Enzian, this interconnect is used to connect a ThunderX with the FPGA. FPGA modules are being developed in the group to enable this protocol on the FPGA. This interconnect enables a low-latency, high-bandwidth connection between both components in this CPU/FPGA hybrid system.

Enzian is an ideal platform to evaluate TeSSLa style runtime verification on realistic server and data center workloads.

Mainly due to lack of software support for the ThunderX CoreSight system with non-standard and proprietary configuration (section 1.1) the current RV system could not be developed for Enzian prototypes.

[2] `https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html`

[3] `https://github.com/Xilinx/linux-xlnx`; tag xilinx-v2016.2; this older kernel was used because a newer kernel matching the Vivado 2018.1 suite was not stable with occasional freezes.

[4] original kernel patch introducing this: `http://lists.infradead.org/pipermail/linux-arm-kernel/2011-July/057932.html`

[5] `https://www.marvell.com/server-processors/thunderx-arm-processors/`

[6] `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html`

2.5 Instrumentation

Instrumentation of programs can provide crucial additional information needed by the verification specification, such as memory location allocated by `malloc()` and released by `free()`, or mutex and thread identifiers for locks. Additionally, it may be required to send tag IDs into the instrumentation stream to follow data processing e.g. in distributed systems.

Table 2.1: Instrumentation opportunities

Type	Comment
source code [a]	direct call of instrumented functions
compile time [a,c]	program transformation e.g. with LLVM tooling
link time [b]	use link flags to define specific wrapper functions
load time with dynamic linking [d]	use `LD_PRELOAD` to dynamically wrap functions

[a] requires source code
[b] requires source code or at least unlinked objects
[c] e.g. implemented in [29] using the `CONTEXTIDR` register for instrumentation data
[d] only possible for functions of other dynamically linked libraries (`.so`)

There are various options to instrument a program (Table 2.1). The new instrumentation library of this project provides source code and load time instrumentation. It also allows a combination of both, i.e. function instrumentation via load time instrumentation and sending of additional signals via manual call of the signal function e.g. for debug purposes. It can be used for compile time and link time instrumentation with small modifications.

Different channels can be used to encode the instrumentation data. The `CONTEXTIDR` register works well for bare-metal program execution [29]. On Linux, this register cannot be modified directly from user space. Modifying the `CONTEXTIDR` register via kernel can be accomplished by a kernel patch providing an additional syscall or as piggy-back of an existing syscall [31], or by a kernel module that provides a device with such an `ioctl()` function[7].

For this project, it was advantageous to activate the ARM Instrumentation Trace Macrocell (ITM) [2] and send instrumentation data as ITM stimuli, which are parsed in the FPGA by a specific ITM parser. Advantages: 1) data is written to a memory-mapped register, which does not require a syscall; 2) no interference with the `CONTEXTIDR` register, that is already used to track processID and threadIDs of the test program.

When ITM signals worked so well, it made sense to write a Xilinx specific extension for the ARM CoreSight access library (CSAL) [6] that activates the Fabric Trace Macrocell (FTM) [35] too, which allows sending signals from the FPGA side into the same CS data stream. This is useful for CPU/FPGA hybrid applications either for debug purpose or runtime verification.

[7] `contextid_writer` kernel module for the zc706 platform in the "not_used_container" of the Tracing Master repo.

Instrumentation comes with overhead cost that is evaluated for the used instrumentation system. ETMv4 [4] offers the option to embed a data stream in addition to the program trace. Thus, separate instrumentation may become less relevant on such SoC at the cost of larger bandwidth requirements for the CS stream and more complex parser logic on the FPGA. Dependent on the specific requirements, an additional instrumentation pathway may still be beneficial to encode arbitrary signals independently of program and data traces.

Chapter 3

Related Work

3.1 Existing tracing and runtime verification infrastructure

The current project builds on the already existing infrastructure in the Systems Group to collect raw CS traces via ETB and TPIU (chapters 4 and 5).

In an earlier MSc thesis, CS traces could be collected and verified against a specification in past-time linear temporal logic (ptLTL) [29]. CS traces were collected via JTAG interface form a Panda board using DSTREAM and ARM DS-5 infrastructure. Traces were parsed in software. ptLTL formulas were compiled to Büchi automata, which were used for verification of traces.

The instrumentation library was attached via code transformation during compilation using LLVM. Instrumentation signals were sent to the CS trace via CONTEXTIDR register modifications (mcr and isb instructions).

3.2 TeSSLa implementations

There is a software reference implementation of TeSSLa that has been used for various projects [10]. The playground on `https://www.tessla.io` offers several examples and interactive exploration of the language.

There already exists an FPGA implementation of TeSSLa that can handle a limited subset of the language (COEMS project) [11]. To avoid time consuming re-synthesis of a specification for FPGA, this solution embeds multiple modules (processing units) in FPGA that can be reconnected quickly. This solution is proprietary and closed source. No details on inner working or detailed limitations of the system could be read in the literature nor be provided by our collaboration partners.

In contrast, the new FPGA implementation in this project aims to re-synthesize each TeSSLa specification for FPGA and embed it directly with the CS parser system with the goal to allow more complex specifications and higher bandwidth. This is advantageous for multi-core systems and even rack-scale runtime verification. In addition, this system is planned to become open source.

Even without knowledge of the inner workings of the already existing FPGA solution, it is easy to deduce for a given FPGA with limited resources: more complex specifications and/or more bandwidth for the raw CS trace can be achieved with a specialised solution for one TeSSLa specification than a generic system that can be readjusted for multiple specifications on the fly.

One specification expressed in concrete Verilog modules offers more optimization options to the Vivado synthesis and implementation routines (e.g. place and route) than a set of independent generic modules that are re-wired for new specifications.[1] This advantage can be enhanced by future TeSSLa compilers that will optimize already before Verilog code generation (section 7.2 for suggestions).

Creating a specialised FPGA implementation for each specification comes at a cost: synthesis of a bit file takes about 20-30 minutes with Vivado 2018.1 on an Intel Core i7-8750H 2.2 GHz with 32 GB DDR4 RAM and SSD. Thus, both types of FPGA implementations of TeSSLa make sense in different application scenarios.

3.3 Other CPU/FPGA hybrid implementations

An FPGA assisted instrumentation of programs running on Linux adds an additional syscall to the Linux kernel to send instrumentation signals to the CS trace [31]. Instrumentation data is embedded into the CONTEXTIDR register similar to the solution used in [29]. A syscall is needed because the required assembly instructions (mcr and isb) cannot be used in user mode. The FPGA part parses the CS stream from TPIU. However, no other trace data is used in the publication. Instrumentation overhead is reported as 30 μs with the syscall.[2] In case another syscall can be instrumented, a piggy-back instrumentation can be used by patching this syscall, which reduces the measured overhead to 0.014 μs.

CS program traces have been used to integrate monitoring for Code Reuse Attacks (CRA) into CPU/FPGA hybrid systems [24]. In addition to the program trace, memory access needs to be observed in the FPGA, which is not (yet) implemented in the solution presented in this thesis. ROP and JOP attacks can be detected implementing known algorithms for this purpose.

In a presentation, a CS frame parser is described [30]. No additional processing of CS macrocell data is described.

[1] Please note: complexity can be achieved by a generic solution as well by e.g. running an instruction script step by step. However, this reduces the bandwidth for the raw CS trace. The advantage: new specifications can be loaded quickly.

[2] A Zed board was used in the paper with 2 ARM cores at 667 MHz using a Yocto Linux kernel version $\geq$ 4.9. For comparison with measured data in the evaluation part of this thesis, the overhead of a syscall was tested on the used Zynq board with the Xilinx kernel 4.4, too. Here it was 0.43 μs. This large difference is not explained by slower clock frequency of the board alone. Dependent on which kernel was used precisely in [31] patches against speculative execution side-channel attacks and other factors could play a role, too.

3.4 Other runtime verification systems

Due to processing overhead, several solutions connect the target SoC under test with external computers or specialised devices. This category includes solutions that connect the DSTREAM device to the JTAG output of the board, e.g. the already discussed solution [29].

A solution uses an external emulator that is synced with the target SoC [7]. The FPGA emulator (or ASIC for faster target SoC) is connected with the target SoC and emulates all CPU cores, all bus master interfaces and the memory of the target SoC. This reduces the processing overhead that needs to be done on the computer used for analysis of the data.

Signal temporal logic (STL) assertions have been translated to monitors / runtime verification systems running on FPGA [22]. Several signals from external hardware were used for evaluation. Due to the hardware specific setting, the system was restricted to past and bounded future temporal operators interpreted over discrete time.

Many solutions have been described for pure software solutions, e.g. [19]. Many RV frameworks have been created, as e.g. listed in [25].

TeSSLa must not be confused with TESLA [1], Temporally Enhanced System Logic Assertions, a different RV tool that allows users to insert assertions into a program to test various properties.

3.5 Other verification systems

Runtime verification needs to be distinguished from other forms of verification, such as formal verification with theorem provers or static analysis tools. These types of verification offer different advantages and disadvantages [17]. As discussed in section 2.1, there is a need for both types of verification.

Chapter 4

Design

4.1 Design overview

The full runtime verification infrastructure of this project consists of two parts:

- **FPGA:** CS parsers, timestamp driver, and TeSSLa specification are embedded in an AXI/TPIU controller module
- **Software:** TeSSLa to Verilog compiler via Chisel, Linux drivers to configure CS and AXI/TPIU module, launch programs, parse output, and test/helper programs

The overall design of the FPGA modules is shown in Figure 4.1. Three clock domains are used: AXI/TPIU controller 200 MHz, CS parser modules 125 MHz, TeSSLa specification 50 MHz. These frequencies were deduced by the design constraints discussed below. Data streams are connected across clock domains with asynchronous FIFOs. Some use input:output width ratios of 2:1 to achieve higher bandwidth.

Important constraints, that influenced the design, are discussed in the next section. The following sections show the design of the individual modules of the RV system.

4.2 FPGA: design constraints and design decisions

The design of the full RV system including CS parsers and TeSSLa specification on FPGA is guided by recommendations for FPGA systems (subsection 2.4.1). Additionally, constraints of the board (FPGA architecture and speed grade; subsection 2.4.2) influenced the amount of processing that can be achieved in a single processing step.

As a rule of thumb, a CS trace leads in average to 1 bit / instruction [3]. Thus, an initial input bandwidth estimate was made of 250 MB/s for 2 cores at 1 GHz. Bandwidth requirements increase with additional options (e.g. branch broadcast mode) and decrease with filtering of trace parts that are not of interest.

Thus, an effort was made to keep AXI/TPIU clock frequency at 200 MHz to allow an input bandwidth of 400 MB/s for raw trace data (Table 4.1). The next effort was made to keep CS parser clock frequency at 125 MHz to sustain this required bandwidth. Each parser can write a (timestamp, data) tuple in one clock cycle to the FIFO that connects to

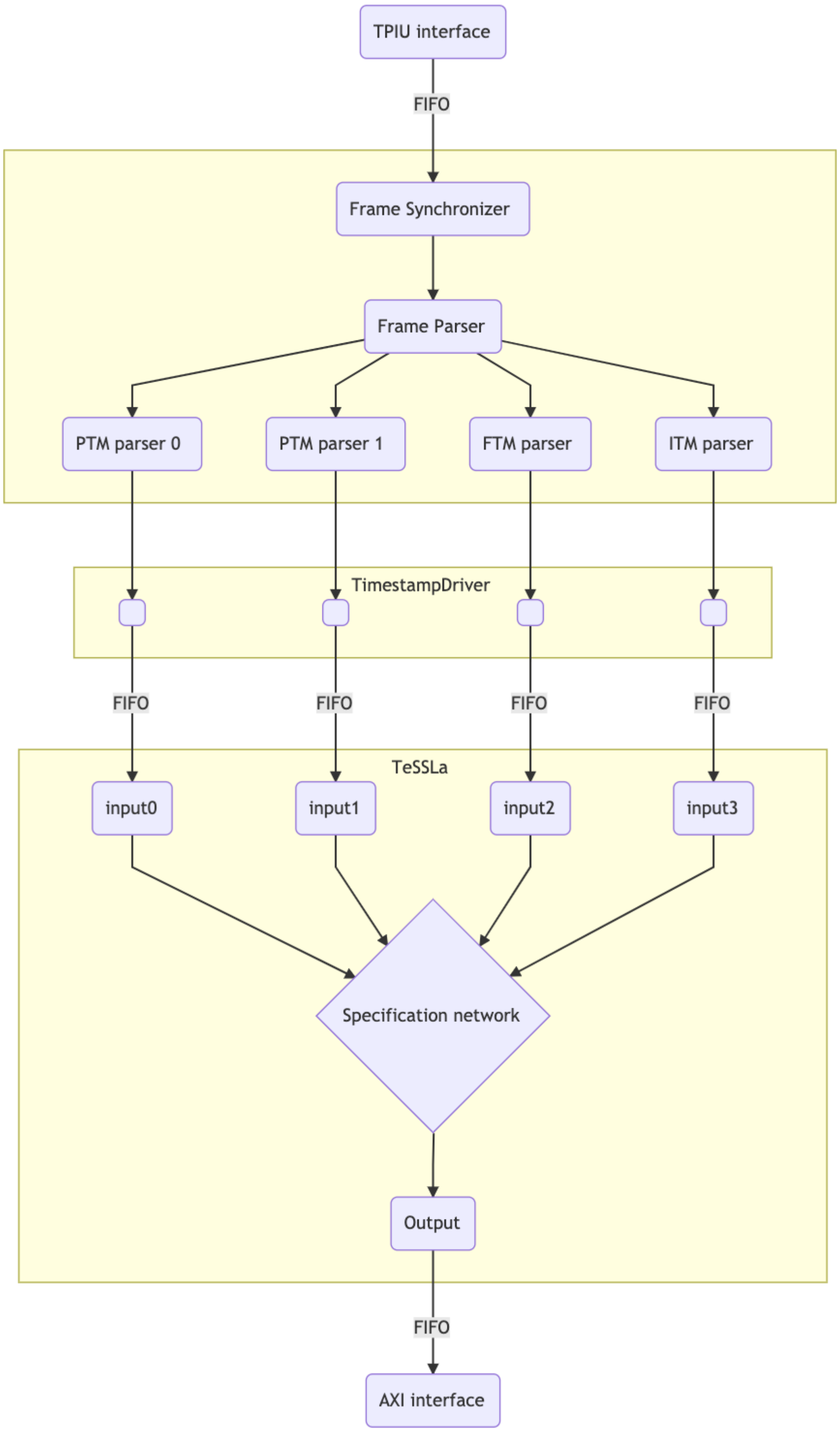

Figure 4.1: FPGA modules: design overview

TeSSLa. The parsers can be adjusted to different filters that reduce the number of events sent to the input of the TeSSLa computation network, if needed.

Table 4.1: FPGA modules: clock frequencies and associated bandwidths

Module	Frequency	Info	Bandwidth
TPIU	200 MHz	4 B / 2 cycles	400 MB/s
CS parser in each parser out	125 MHz	4 B / cycle	500 MB/s up to 125 M events/s
TeSSLa in TeSSLa out	50 MHz	one input for each parser one 64 bit output	25 M events/s per input 25-50 M events/s, 400 MB/s
output cache FIFO	200 MHz	32 bit width (AXI), 512 KiB	800 MB/s
to SD card	–	online mode (section 4.3)	$\leq$ SD card write speed

With the current TeSSLa compiler prototype, the generated Verilog modules cannot be implemented for the Zynq board with clock frequencies above 50 MHz. Because TeSSLa reads timestamp and data value of the (timestamp, data) tuple in 2 separate clock cycles, a maximum input of 25 M events/s can be achieved per input adapter.

Thus, it has been crucial – not only for later multi-core systems but already for the present Zynq board – to enable TeSSLa to read from multiple input streams in parallel. This additionally removes the critical bottleneck needed for the prototype TeSSLa system: merging all streams in correct timestamp order into one multiplexed input stream.

The limited output bandwidth of the TeSSLa specification may only be a limitation for initial testing specifications that mainly provide raw packet data (addresses, contextIDs, instrumentation signals) as output. Typical TeSSLa specifications for RV reduce large inputs to small outputs. As an extreme, a specification may limit its output to the boolean result whether the property is satisfied or violated by the running program.

TeSSLa's OutputAdapter can output multiple data values for one time stamp. Thus, more than 25 M events/s can be written (upper bound 50 M events/s) with a clock cycle of 50 MHz. The output FIFO cache in the AXI/TPIU controller is fast enough to cache all output up to its size (currently 512 KiB).

4.3 Online and distributed runtime verification

The RV system offers an option "online mode" (option `-z` in `trace_launch`) to write these outputs to a file while the program-under-test is running. Thus, larger outputs than the cache size can be collected with output bandwidth limitation of the SD card.[1] This limitation is not a problem for many typical specifications that reduce the raw CS trace by several orders of magnitude. Additionally, one thread is running on core 0 to handle this writing.

[1] A future option of this mechanism could directly use e.g. a TCP socket instead of a file allowing higher bandwidth.

Main advantage of this option: the output of the TeSSLa specification can be read while the program-under-test is running. This allows online RV verification of a running program.

This output stream can be forwarded to another computer, e.g. a dedicated specification server that runs an additional specification on such reduced data streams of multiple devices. Such a distributed RV system allows rack-scale runtime verification.

4.4 AXI/TPIU controller and driver

An AXI interface [33] is used to connect and control the embedded FPGA modules on the current Zynq platform. The FPGA module runs the needed state machines to work with this protocol. Additionally, it controls the TPIU interface that provides the raw CS trace. This module has been part of the existing Tracing Master infrastructure, written by David Cock, that could collect raw CS traces from TPIU.

In the current solution, the raw CS trace is piped through CS parsers and TeSSLa specification instead of direct output. Some additional control and management mechanisms were added.

4.5 CoreSight parsers

The TPIU interface of the Zynq 7000 system provides the CS trace as a multiplexed byte stream in form of 16 bytes long frames. These frames contain multiplex information and the actual trace payload of all activated CS streams, i.e. program trace for both cores, and optionally ITM and FTM data.

Demultiplexed trace streams are parsed by specific parsers in parallel (implementation in section 5.1). Timestamps are created by a counter in the Frame Synchronizer. Thus, events of all parsers remain synchronised independent of different parsing pipeline lengths in the different parsers. Additionally, with the known clock frequency (currently 125 MHz), absolute time can be derived from timestamps.

Granularity of timestamp packets in the CS trace is too coarse to be used as timestamps for TeSSLa. However, timestamp packets are parsed, and the parsers could be adjusted to forward the received values in separate event streams to TeSSLa, if required.

4.6 Timestamp driver and interface to TeSSLa

The TeSSLa stream processing network needs progressing timestamps on all input streams even if no data is available. After extending the TeSSLa input interface to accept more than one `InputAdapter` und thus allow parallel input of multiple trace streams without need to merge them first, a timestamp driver had to be created (subsection 5.2 for important implementation details).

Although this driver needs to have a look at all incoming data streams, it does not merge them. Thus, it will scale to larger number of input streams, in particular of more cores to be observed on e.g. ThunderX SoC.

4.7 TeSSLa specification

FPGAs are in particularly well suited for stream data processing in parallel such as the computation network that is defined by a TeSSLa specification (background in 2.3). The simple test/demo specification (Listing F.2), that is also used in the first part of the evaluation, creates this computation network (Figure 4.2).

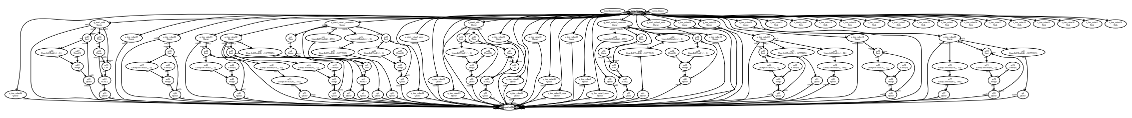

Figure 4.2: A simple TeSSLa computation network (Listing F.2)

For a detailed introduction to this language, I refer to `https://www.tessla.io` that offers documentation and also an interactive playground to test specifications on traces and in instrumented C programs for runtime verification. The syntax resembles the syntax of Scala.

The currently used prototype compiler[2] transforms a TeSSLa specification via Chisel to a Verilog module. It was modified and extended to work well for this project.

A TeSSLa specification is compiled to the TeSSLa core language [10] first. These small computation steps are connected in a computation network to accomplish the full functionality of a specification. A simple macro `count()` from the stdlib is used here for illustration. It counts the number of events (Listing 4.1). The generated directed computation graph (Figure 4.3) consists of computation and queue nodes.[3]

Listing 4.1: TeSSLa: count macro from stdlib

```
in event: Events[Int]

def count[A](a: Events[A]) = c where {
    def c: Events[Int] = merge(last(c, a) + 1, 0)
}

def event_count = count(event)

out event_count
```

The prototype compiler can already compile a large amount of the TeSSLa language. However, several functions (like `delay()`) are not implemented yet. Additionally, sets and maps – that are fully supported in the software version of TeSSLa – are not available in the compiler. Please note, sets and maps must be instantiated with predefined capacity in an FPGA implementation.

[2] provided by Malte Schmitz and Daniel Thoma, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck

[3] Additionally, there is an `InputAdapter`. The `OutputAdapter` is instantiated as `OutputAdapter0`, the `OutputfilterAdapter` as `OutputAdapter` (implementation details in subsection 5.3.4 for details).

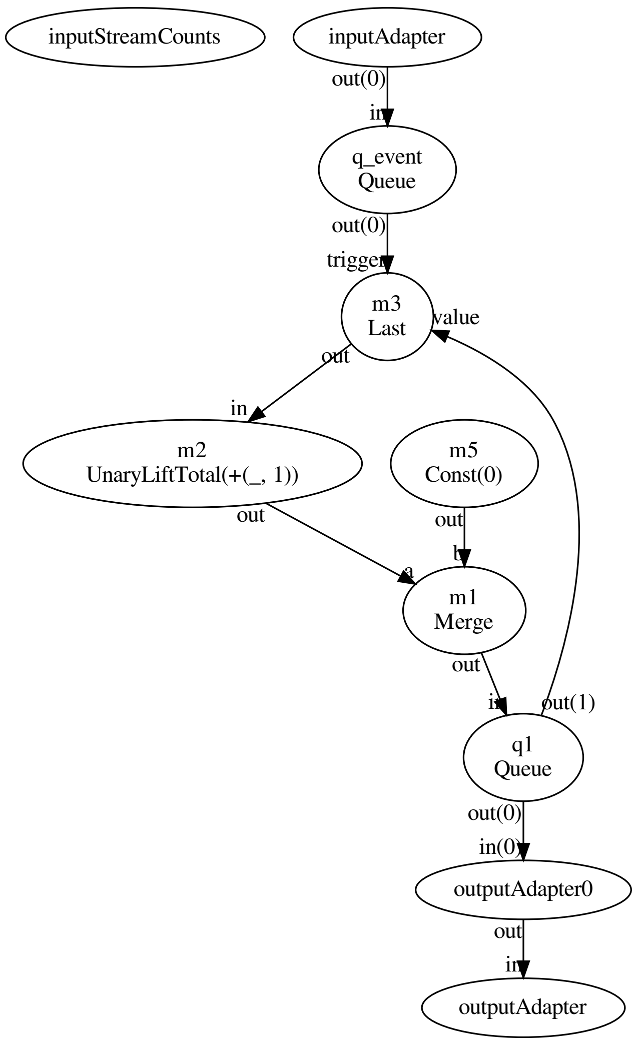

Figure 4.3: Count macro: computation network (Listing 4.1)

To write advanced specifications for testing, Daniel Thoma provided a fixed-capacity set implementation written as a TeSSLa macro (Listing B.3). It unrolls during compilation.

This was then extended to a fixed-capacity map implementation (Listing B.4) that is useful for other specifications (example with unrolled map of capacity 8: Figure 4.4).

Work for a new TeSSLa to FPGA compiler has already started in Lübeck. It will also improve on issues that were observed during the evaluation of this system. Additionally, a standard library will provide important macros for many use cases.

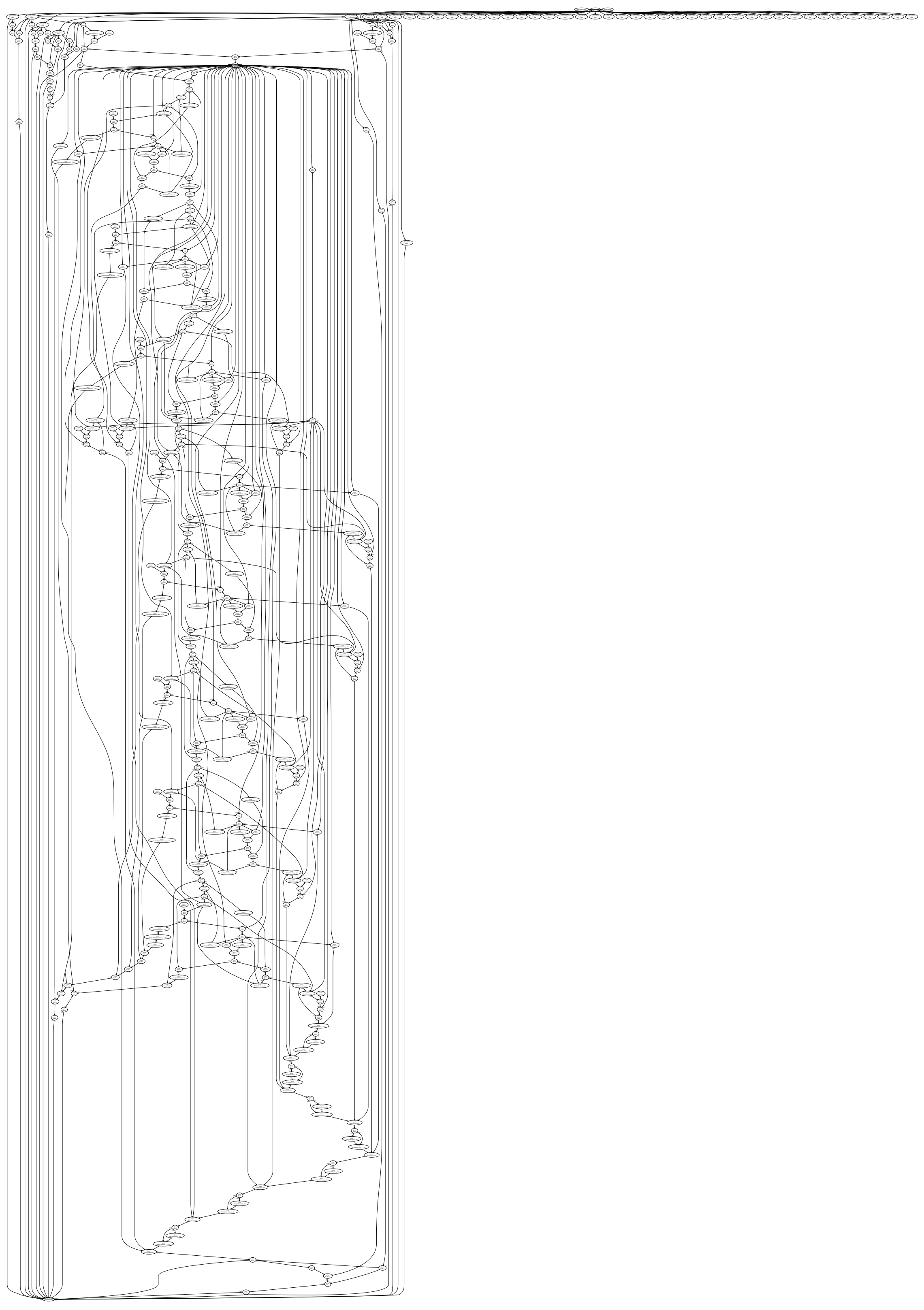

Figure 4.4: TeSSLa computation network with unrolled map of capacity 8. Event stream flows from top to bottom.

4.8 Software tooling

The runtime verification system runs on a current Linux environment (Ubuntu 18.04 LTS) with an older kernel (4.4-xilinx). After loading of the .bit file for the FPGA and the kernel module to access it, test programs can be launched with various configuration options with `trace_launch` (Appendix D). This includes optional instrumentation of the binary without need to recompile from source.

Additional tools include an output parser, a timestamp generator, and various test programs and test specifications.

4.9 Instrumentation library

The program trace parser decompresses branch addresses from the CS trace and forwards them to the specification. With this information, specifications can be written to e.g. track order and timing of function calls during runtime.

However, more detailed information is needed to e.g. track memory allocation/deallocation and/or mutex/lock/barrier use in multi-threaded programs. For this, an instrumentation library has been created that wraps these typical functions of glibc and sends instrumentation data – e.g. address of an allocated/freed memory region – via ITM stimulus.

Chapter 5

Implementation

The implementation of this project is based on the existing Tracing Master infrastructure written by David Cock that allowed collection of raw CS traces via ETB and TPIU to be processed offline. All FPGA modules (VHDL of the parsers and Verilog of compiled specification) are embedded into this AXI/TPIU controller. Asynchronous FIFOs are used to connect the different clock domains. Various modifications and extensions were made to the kernel module and the tooling software, as well.

The prototype TeSSLa compiler consists of different parts (TeSSLa to core TeSSLa, core Tessla to Chisel, TeSSLa synthesis) that were provided by Malte Schmitz and Daniel Thoma, University of Lübeck. Several modifications and extensions were made, which were made available for optional integration into the upstream code base.

5.1 CoreSight parsers

Parsers were written in VHDL to demultiplex and decompress the CS byte stream from TPIU interface to events as needed by the TeSSLa specification for processing. The overall data flow is shown in Figure 4.1.

5.1.1 CS frame synchronizer and parser

The Trace Port Interface Unit (TPIU) provides the CS byte stream in 4 byte words [35]. Frames of 16 bytes contain multiplex information and payload data (Trace formatter, [5]). Periodic synchronization signals `0xFFFFFFFF` are sent to resynchronize frame parsers, if needed.

The `Frame Synchronizer` reads and caches the received words from TPIU and forwards only complete frames to the frame parser. It resynchronizes, if needed. Additionally, it increments a 48 bit wide counter with each clock cycle. This timestamp information is forwarded with each frame through the entire pipeline. Thus, it does not matter that some downstream parsers need more processing steps than others.

The `Frame Parser` decodes the specified bit pattern of the frame [5] to demultiplex the payload using the defined IDs (Table 5.1). From each frame, 0 to max 15 B of data are sent to each of the specific parsers via synchronous FIFO.

Table 5.1: Multiplexed CS streams in frames: IDs used in CS configuration

ID	Value
Core 0	0x10
Core 1	0x11
ITM	0x6F
FTM	0x70 to 0x7F

5.1.2 PTM parser

The parser maintains a buffer that is appended when new data is received (0 to 15 B every 4 clock cycles) and reduced/shifted when complete packets are parsed. It is built as a processing pipeline (Figure 5.1).

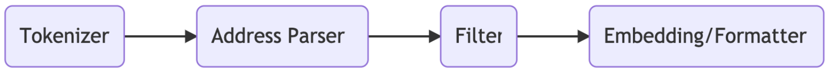

Figure 5.1: Data flow in PTM parser

Each of these processing modules of the pipeline consists of several small processing steps in a pipeline to keep clock frequency high (subsection 2.4.1 for relevant FPGA characteristics; consecutive design constraints in section 4.2).

Tokenizer. The tokenizer can handle all PTMv1.1 packet types as specified [3]: A-sync, I-sync, Atom, Branch address, Waypoint update, Trigger, Context ID, VMID, Timestamp, Exception return, Ignore.

Some packets have a defined fixed payload size; for other packets, this payload size is determined dynamically based on the packet data. Parsed packets are forwarded to the address parser. Each packet carries the timestamp of the frame it was extracted from.

Address Parser. Addresses are compressed in the CS program trace in `Branch address` and `Waypoint update` packets. `I-sync` packets always provide a full 32 bit address. The address parser needs to maintain an internal state to decompress addresses correctly: last known address and current ISA mode (arm, thumb, jazelle). With this information, full 32 bit addresses are added to address carrying packets. Other packets are forwarded.

Filter. Not all packets are of interest for the TeSSLa specification. Currently, addresses of `Branch address` and `Waypoint update` packets and contextID from `I-sync` and `Context ID` packets are used. The filter can be adjusted to other requirements.

Embedding/Formatter. This part of the pipeline encodes the address and contextID information into event tuples as defined below (Table 5.2). Multiplex address 0x00 is used for addresses, 0x01 for contextID.

Due to compression, several events can be encoded in a single frame. To assure strictly monotonic time increments, the 48 bit timestamp is extended by a time extension of 4 bits, that is used for additional small increments if multiple packets arrive that were derived from the same frame.

An asynchronous FIFO with am input:output width ratio of 2:1 is used for data transfer to TeSSLa. Thus, timestamp and data value can be written to the FIFO in one clock cycle.

5.1.3 ITM parser

The ITM parser works like the PTM parser. However, fewer and less complex packets need to be parsed.

Tokenizer. The tokenizer can handle all ITM packet types as specified [2]: Synchronization (identical with A-sync of PTM), Overflow, SWIT, Timestamp, Reserved. SWIT packets contain the data of the stimuli that were issued by software, e.g. due to instrumentation.

Embedding/Formatter. The 32 ITM ports are mapped to multiplex addresses 0x00 to 0x1F for TeSSLa.

5.1.4 FTM parser

The Fabric Trace Monitor (FTM) is an extension of the internal CS infrastructure by the Xilinx Zynq-7000 SoC [35]. It allows the FPGA implementation to send 32 bit data words into the CS stream. This is convenient for instrumentation and debug purpose of e.g. hybrid CPU / FPGA applications.

Using a 4 bit ATID address, up to 16 different signal ports can be defined in the FPGA application. However, only one (address, data) tuple can be sent per clock cycle. Thus, an FPGA application has to synchronize signals itself if different modules want to send such signals to different ATID addresses concurrently.

This ATID address is combined with the provided FTM ID number in CS configuration.[1] With the deliberately chosen 0x70 as base address, FTM IDs 0x70 to 0x7F are available in the CS frame.

Due to the nature of multiplexed data streams in CS frames, it is best to instantiate a separate FTM parser for each of these FTM IDs. Thus, only one FTM address (ATID 0x0, thus ID 0x70) is used.

The FTM parser works like the PTM parser. However, fewer and less complex packets need to be parsed.

Tokenizer. The tokenizer can handle all FTM packet types as specified [35]: Synchronization (identical with A-sync of PTM), Trace, Trigger, Cycle count, Overflow, First.

Embedding/Formatter. The FTM data is mapped to multiplex addresses 0x00 for TeSSLa.

5.2 Timestamp driver

The TeSSLa event stream processing network of the specification requires strictly monotonic increments on all input channels to make progress. This was implicitly guaranteed by the original `InputAdapter` of the TeSSLa compiler prototype.

[1] The ARM CSAL library was extended for this to allow detection and configuration of Xilinx FTM. Details are in subsection 5.5

An extension was built during this project to avoid a bottleneck of merging all parsed event streams before sending them to the TeSSLa network (`MultiInputAdapter` below). However, computation stopped if not all inputs were used (e.g. only one core, or no FTM/ITM data).

Thus, this timestamp driver was introduced between the outputs of the various CS parsers and embedding the information into the asynchronous FIFOs to TeSSLa (Figure 4.1). Despite looking at all data streams, there is no merging of data. Thus, it will scale to more inputs (e.g. program traces of more cores).

It is designed to send additional timestamps on idling streams (broom wagon principle), which allows the TeSSLa network to make progress. Several variants have been explored during this project.[2] Subtle changes can have large effects.

The current correctly-working solution consists of a co-design of 3 modules: this timestamp driver, the new `MultiInputAdapter` and modified `InputAdapter` in TeSSLa (subsection 5.3.3).

At any clock cycle, a parser output may have a valid (timestamp, value) event tuple (indicated by asserted `wr_en` for the FIFO) or no event (not asserted `wr_en`). If none or all of the parsers have a valid event, nothing is changed. If some parsers have a valid event, an artificial "broom wagon" event is provided to the FIFOs of parser outputs without event. Therefore, at any clock cycle, events are written to all FIFOs or none.

This "broom wagon" event consist of a timestamp that is calculated from observed timestamps and is guaranteed to be lower than any timestamp a parser could send in the future. The data value uses mux address 0xFF that is defined to be filtered in the modified `InputAdapter`. Because the "broom wagon" timestamp is guaranteed to be lower than any future true timestamp of all parsers, there is the possibility that it could be lower than an already sent timestamp for a particular parsed event. Thus, the `MultiInputAdapter` has a filter to remove such timestamps that would violate the required monotonicity.

Please note: All parsers are designed to only provide strictly monotonically incrementing timestamps. Thus, any timestamp violating this property, must have been a "broom wagon" timestamp, which is safe to be removed.

And finally, at the end of the trace, the timestamp driver sends an additional final timestamp push to the TeSSLa network via all FIFOs to allow it processing the last input from the traces.

This solution has the advantage that event flow through the TeSSLa network comes as close to the original InputAdapter as possible.[3] The added functionality of multiple inputs

[2] An initial design implemented in Chisel as a module instantiated in the `MultiInputAdapter` had a subtle bug (occasionally dropping one address during bursts) that is related to requirements of the `InputAdapter` to be directly connected with a special "fwft" FIFO (subsection 5.3.1), which prevents easy chaining of modules.

[3] Having lots of incrementing timestamps flowing into the computation network on all outgoing edges of the TeSSLa `InputAdapter` has shown to be a critical property for proper working of the computation network. During development, "smarter" timestamp drivers that sent "broom wagon" events only after some idle time (in the false attempt to limit the number of additional events sent to the TeSSLa network), and already filtered timestamps (to avoid the need for a filter in `MultiInputAdapter`) led to halts of the computation network. In retrospective, the gained insights of these explorations make lots of sense.

without need to merge first is a critical step for scaling of the TeSSLa network to program traces of multi-core CPUs like ThunderX.

Despite looking simple (Appendix C), several subtle details had to be considered. Due to the co-design of 3 modules, overall effect needs to be considered if modifications are made in any of them.

5.3 TeSSLa specification

5.3.1 Some technical details of the inner working of TeSSLa on FPGA

For a given TeSSLa specification, a computation network is generated in which event streams can be processed. Each event consists of a (timestamp, data value) tuple. For technical reasons, timestamp and data are pushed separately over this computation network, timestamp first and data value second. Thus, each movement of of such a tuple takes at least 2 clock cycles. It may take longer if no progress can be made in a particular network node (e.g. caused by merging in the output adapter).

Table 5.2: Encoding of multiplexed TeSSLa event tuples (input / output)

Type	Flag	Description [bits]	
Timestamp	1 [61]	padding [60:53]	timestamp [52:0]
Data value	0 [61]	mux address [60:53]	data value [52:0]

For external connection of this computation network, each input and output adapter can multiplex multiple event streams (Table 5.2). Each 62 bit wide word represents either a timestamp or multiplexed data value. Currently, a mux address width of 8 bit was chosen to allow up to 256 multiplexed streams.

Timestamp width 53 bits was chosen to accommodate a 48 bit counter (lasting approx. 26 days at 125 MHz), plus 4 bit time extension, plus 1 bit to embed this unsigned int in the signed int used in TeSSLa internally. Data width was chosen to be equal because data values must be able to hold a timestamp. Finally, a max. limit of 64 bit was used as constraint to allow an easy 2:1 FIFO connection with the outside AXI interface (32 bit data port).

These settings were chosen specifically for this project and may be adjusted for other projects.

As a technical detail: Due to its design, the `InputAdapter` may de-assert a ready signal upon presentation of the data value. This happens e.g. when a data value could be sent to a specific edge but then the next timestamp, which needs to be sent to all edges, cannot be forwarded. Thus, the `InputAdapter` makes specific use of the optional first-word-fall-through (fwft) mode of the connected FIFO [34] with different semantics than standard ready/valid signals used in chainable modules. Specifically, the `ready` signal of the `InputAdapter` should be connected directly to the `rd_en` control of the fwft FIFO, which allows the required semantics.

5.3.2 TeSSLa compiler

The compilation process of a TeSSLa specification to a Verilog module currently consists of 2 steps.

1. `tessla2chisel` compiles the TeSSLa specification to a Chisel program, internally creating a program in TeSSLa core language first.
2. `tessla-synthesis` provides pre-defined TeSSLa specific Chisel modules to run this program and finally create a Verilog module.

This `Specification.v` Verilog module is then used when building the entire FPGA bit-stream file using Xilinx Vivado 2018.1.

Key modifications made to the TeSSLa compile system:

1. Allow manual definition of the multiplex address width instead of auto-deduction based on arity of muxed streams
2. Sorting of input and output labels first before assigning IDs to have consistent mappings of input and output streams
3. `MultiInputAdapter` with required `TimestampDriver` on CS parser side
4. patched `InputAdapter`
5. `OutputFilterAdapter`

5.3.3 MultiInputAdapter

The original prototype system allowed only one multiplexed input (Table 5.2). While data values were forwarded to the specific edge of the computation network (implemented as queue) defined by the mux address, timestamps are forwarded to all edges connected with the `InputAdapter`. This implicitly guarantees that computation progress can be made in the entire network.

However, this would require merging all parsed CS streams (2x PTM, FTM, ITM; on larger systems many more) into one event stream, which creates a bottleneck.

Thus, a `MultiInputAdapter` was created, that internally instantiates a separate `InputAdapter` module for each input and then uses offsets to map demultiplexed streams to queues with associated IDs.

This simple design lead to a complete stop of the computation network if not all inputs were providing inputs (e.g. inactive ITM or FTM). Thus, the `Timestamp Driver` was added (subsection 5.2) that guarantees progress on the entire computation network. The `MultiInputAdapter` was extended to filter out "broom wagon" timestamps issued by the `Timestamp Driver` if they referred to an earlier time than already received with a former true event. Thus, monotonicity is guaranteed in the TeSSLa network. This co-design is described in detail in subsection 5.2.

The `InputAdapter` was patched to drop data values with mux address 0xFF to avoid invalid data values associated with the driver timestamps to flow into the computation network.

5.3.4 OutputFilterAdapter

The original `OutputAdapter` of the received prototype is a masterpiece. It accomplishes proper merging of all defined output streams into one output that is then sent back to the AXI/TPIU controller.

To reduce bandwidth needed for the TeSSLa output, a small state machine was added as an `OutputFilterAdapter` Chisel module to forward only timestamps if at least one data value follows. This additional computation node is instantiated between the `OutputAdapter` and the effective output of the specification module.

5.4 AXI/TPIU controller and driver

This entire processing pipeline (Figure 4.1) is embedded in the already existing AXI/TPIU controller FPGA module and kernel module driver (by David Cock). Some extensions and modifications were made to add new functionality and configuration options.

The processed output can be read from the character device `/dev/axi_tpiu` instantiated by the `tpiu_emio_ctrl` kernel module. The device offers a control and status interface in `/sys/class/misc/axi_tpiu`. A software library exists that can access these parts directly from C.

5.5 Software tooling

The entire trace collection and runtime verification system runs on a current Ubuntu 18.04 LTS system. For stability reasons an older kernel (4.4-xilinx) is used.

trace_launch The entire CS configuration, control of the AXI/TPI module with the FPGA processing pipeline, preloading of the instrumentation library, and data collection is handled by this `trace_launch`. Thus, this existing program was extended a lot and offers various options now (Appendix D).

Extended CS Access Library (CSAL) A modified and extended fork of ARM CSAL [6] is used for CS configuration. It has been extended earlier for the modified configuration procedure that allows platform configuration with a Prolog script instead of direct C code.[4] To use this script configuration in the instrumentation library too without embedding a full parser, the script is compiled to a byte code at build time that is then executed during CS initialization.

The library has been extended to provide detection and configuration of FTM in addition to ETM/PTM and ITM. This extension is specific for the Zynq-7000 SoC from Xilinx. However, it only uses information that is already provided by the CS ROM and by the configuration script, which is platform specific in all cases. Thus, it can be modified easily for other platforms that provide such functionality.

parse_tessla_output The output of processed CS trace is in the binary format as described above (Table 5.2). This helper tool parses this binary output to text in various formats.

[4] Tracing Master repo for details.

5.6 Instrumentation library

Writing properties for runtime verification with TeSSLa is limited if only address jumps (e.g. jumps to functions) and contextID switches are available. More interesting properties need e.g. the address of allocated/deallocated memory blocks, specific mutex/lock/barrier information (threadID, lockID), or tracking tags in distributed applications.

There are various options to instrument programs to provide such data (Table 2.1). The instrumentation library is written to allow all of them with minor modifications. It embeds instrumentation data via ITM into the CS trace, which is then parsed by the ITM parser.

The most convenient way of instrumenting a program is accomplished by adding the `-P` option to `trace_launch`. It then modifies the environment of the program under test to use `LD_PRELOAD` with the `instrumentation_wrapper.so` library. All currently instrumented functions are listed in Appendix E

For memory allocation/deallocation: the address of the memory block is sent as instrumentation signal.

For mutex/lock/barrier instrumentation: threadID and mutexID are merged into one 32 bit word. In the current configuration, 256 threads can be observed (8 bits). The `pthread_t` typed thread id is mapped to an identifier in [0, 255] using a hash map[5] in the instrumentation library. New ids are created with each `pthread_create()` and used for the instrumented mutex/lock/barrier function calls. The lower 24 bit of the pointer to the mutex/lock/barrier are merged into the full 32 bit instrumentation word. Alternatively, a hashmap lookup can also be used for this mapping.

The TeSSLa specification can separate this info in the specification for specific use (Listing A.2).

This solution was chosen to have atomic signals sent to the ITM, which guarantees a unique event timestamp in TeSSLa specification. Alternatively, handling multiple separate signals, each with different timestamp but semantically associated with one event, would be tricky.

The instrumentation overhead is benchmarked and discussed in the evaluation chapter. Because of this overhead, spin-locks are not instrumented in default instrumentation setting. However, the instrumentation library offers a flag to include them, too.

As a technical note: writing an instrumentation wrapper for memory allocating functions requires the wrapper to provide a temporary workaround memory allocator until initialization is complete. Looking up the original symbols with `dlsym()` triggers a memory allocation, which would run into an endless recursion until stack overflow without such an added allocator.[6]

[5] uthash [18] was chosen for this implementation for various reasons: 1) it has worked very reliably in an earlier project [15], 2) permissible open-source license, 3) easy-to-integrate header-only library, 4) simple adjustment of its internal memory management to not interfere with wrapped glibc `malloc()` and `free()` functions in the instrumentation library.

[6] Tracing Master repo for details.

Chapter 6

Evaluation and Benchmarking

Various aspects of the implemented RV system were evaluated in three groups of experiments.

- correct parsing of the raw CoreSight (CS) stream (section 6.2)
- instrumentation correctness and overhead (section 6.3)
- RV using TeSSLa specifications for three properties in different domains (section 6.4)

6.1 Methods

All experiments were run on the same Zynq zc706 board with Zynq 7000 SoC, Linux kernel 4.4-xilinx, and Ubuntu 18.04 LTS. The kernel was patched to write processID into the CONTEXTIDR register (option during build process).

Experiments were run as root and with deactivated Address Space Layout Randomization (ASLR). Each experiment configuration was run 3 times. Results are shown as raw counts, mean $\pm$ SD for time, or average and peak for throughput.

Experiments were sequentially numbered from 1 to 20. For space reasons, the experiment descriptions here are summarised. The experiment numbers refer to the detailed experiment description that allow full reproducibility, raw data collection, and detailed analysis description in the thesis report repository.

6.1.1 CoreSight configuration

Several details of CS configuration are of importance to understand some findings. As default for all tests (unless mentioned otherwise), CS was configured to filter for 1) address space of the test program (text section), and 2) for processID of the test program, which is encoded in the CONTEXTIDR register. This already reduces the raw CS trace to the parts that are typically of interest when tracing a program in user space.

For testing multi-threaded programs (pthread library), the filtering for processID must be deactivated (`-p` option in `trace_launch`) because created threads have a different ID in the CONTEXTIDR register on which this filter is based on. Otherwise, only the trace of the main thread is collected.

CS is configured with branch broadcast option activated to receive direct jump addresses to functions, which is an absolute requirement to define TeSSLa specifications without instrumentation. This leads to longer traces (no E atoms). Without this option active, direct function calls would only be encoded as an E atom. Trace overhead (and thus raw trace throughput) varies on actual program code. However, raw traces show that there are already many `I-sync` packets (each 9 bytes) due to ISA changes between arm and thumb code and returning from filtered program sections when glibc functions are called. Thus, overhead was observed to be around 10% for tested programs.

With given clock frequency of the ARM cores, the theoretical bandwidth limit of 400 MB/s (TPIU clock at 200 MHz) was never reached. A max. throughput of 260 MB/s was observed with the used configurations. No packets of the raw trace were dropped, as long as the used TeSSLa specification reduced the input stream to fit all into the output FIFO or not be throttled by the output to disk (online mode, section 4.3).

Overhead will be larger for programs without calls to glibc and no or only few ISA mode changes between arm and thumb code. Thus, branch broadcast can be deactivated (`-b` option in `trace_launch`).

6.2 CS parser validation

These experiments were designed to establish correct address and contextID parsing by the newly written PTM parser. OpenCSD [26] `trc_pkt_lister` was used as software reference parser for raw CS trace streams.

Experiments 3 and 4: correct address and contextID

As a first task, output of the CS parser / TeSSLa system was compared with the raw CS traces as collected via ETB and via AXI/TPIU using a patch to bypass the processing pipeline and output raw traces.[1]

Methods. A simple recursive test program `nested_malloc` (Listing F.1) was used. This program is used for many of the experiments shown here. Several other programs have also been tested during development and testing of the CS parser. Core affinity was set to core 1. Raw CS traces were collected 1) from ETB and 2) from TPIU with bypass, and then parsed with `trc_pkt_lister` from OpenCSD project.

A simple TeSSLa specification was used that forwards addresses and contextIDs (identity functions), their counts, and time differences. The relevant snippet is shown in Listing F.2, which is embedded after the reference interface (Appendix A) and out definitions for all 52 outputs. It is the default test/demo TeSSLa specification in the Tracing Master repository.

Different address filter option configurations were compared in experiments 3 and 4 to establish initial outputs to be matching the parsed raw trace without filters to the final state with activated filters in the PTM parser and TeSSLa specification as used in the default configuration:

[1] experiment 1 was an initial raw trace collection; experiment 2 was an earlier validation test that revealed a subtle bug in an earlier version of the timestamp driver that was embedded in the `MultiInputAdapter` leading to a redesign of the timestamp driver (subsection 5.2).

- `I-sync packets` provide address and contextID information; address information is used to update the internal parser state but not forwarded to the specification (explanation below); instead the contextID is forwarded
- active `OutputFilterAdapter` (subsection 5.3.4)

Results. Raw traces from ETB and TPIU showed 55 `I-sync`, 183 `Branch address`, and 15 `Atom` packets on core 1 after initial `A-sync` packets for both cores. Traces are fully reproducible. Addresses are equal in all traces (deactivated ASLR). All 13 calls to the recursive `allocate()` function (at 0x10514) are in the trace. ContextID values differ between the traces (as expected) but are equal within one trace (single-threaded program).

The different configurations with different filter options worked correctly.

Conclusion. Results of CS parser match parsed output of raw trace using software reference implementation. The filters worked correctly. Filter configuration was set to the default configuration mentioned above.

Experiment 5: continuous writing of output to disk (online mode)

Earlier experiments were run with data collection of the TeSSLa output in the output FIFO (512 KiB), which is larger than the ETB (4 KiB) but not sufficient for longer running programs. Thus, a new output mode was implemented in `trace_launch` (option `-z`) that uses a separate thread on core 0 to write the content of the output FIFO to a file while the test program is running. This additionally allows online observation of e.g. longer running programs by reading this written file in parallel. This can be used for online and distributed RV (section 4.3)

Methods. The same test program and TeSSLa specification were used as in experiments 3 and 4. This TeSSLa output and the output of the bypass method (writing raw CS trace to disk) were tested and compared with results in the earlier experiments.

Results. Identical results for raw traces and TeSSLa output with this new method, except of expected different contextID values and small differences in the absolute timestamps, as expected.

6.3 Instrumentation validation and benchmarking

Evaluation of program instrumentation tests various parts of the RV solution: instrumentation library including wrapper, CS configuration to write ITM stimuli, ITM parser in the CS parser, and correct processing of instrumentation events in the TeSSLa specification. It consists of validation of correct functionality and benchmarking of the instrumentation overhead in running programs.

6.3.1 Validation

Experiments 6 to 8: instrumentation signals

Methods. In experiments 6 and 7, the default TeSSLa specification and the same test program (`nested_malloc`) from experiment 4 were used. In experiment 8, a modified version was used that avoided any `printf()`.

Two configurations were used for comparison: without instrumentation as baseline and with preload instrumentation (option `-P` in `trace_launch`).

Results. Experiments 6 to 8: the instrumented version showed all 12 signals for `malloc()` and for `free()` with correct addresses for the pointers (reverse order in free as expected). Thus, the instrumentation is correct. As before, all 13 expected calls of the recursive `allocate()` function were in the output.

The jump addresses for `malloc()` and `free()` in glibc were also available as before, of course with different addresses in the instrumented version due to the wrapper function. However, as a technical detail: each branch address is found only 11 times in the trace (also as before). The first jump to each glibc function is not a direct jump but the address is first resolved in the Procedure Linkage Table (PLT) mechanism.

Thus, even when knowing the glibc function addresses in case of deactivated ASLR, not all glibc calls could be tracked with function addresses alone. This is a limitation that needs to be remembered when writing TeSSLa specifications with such purpose.

Experiment 6: While this experiment was not be designed to benchmark the instrumentation overhead, it was still expected that the execution of the instrumented program takes longer than the non-instrumented program. The init phase to the first event for TeSSLa was longer (13.0 $\pm$ 6.0 ms, range 9.5 to 19.9 ms) with instrumentation than without (2.3 $\pm$ 0.1 ms) due to the constructor running in the instrumentation wrapper, as expected. However, the run time after that surprisingly was shorter in the instrumented program (0.37 $\pm$ 0.02 ms) than without instrumentation (0.6 $\pm$ 0.03 ms). This puzzling result was confirmed with additional tests.

A "trace density plot" (Figure 6.1)[2] gave a good hint for the explanation that was then clarified with experiments 7 and 8. While overall program execution does not relevantly differ between instrumented run and baseline, there is a longer delay in the non-instrumented program version early in the program trace.

Experiment 7: different configurations of the instrumentation and CSAL libraries were explored to reduce their outputs to stderr and stdout. In conclusion, the default library was changed to avoid all `printf()` during initialization. CSAL library was left unchanged.

Experiment 8: Using the modified test program without `printf()` as well, the longer init phase of the instrumented program remained (as expected) with 8.413 $\pm$ 0.957 ms compared to 2.338 $\pm$ 0.091 ms (non-instrumented) to the first event. The average relative run time after initialization was almost equal for both configurations 0.736 $\pm$ 0.035 ms, 95% CI [0.696, 0.776] with instrumentation compared to 0.758 $\pm$ 0.029 ms, 95% CI [0.725, 0.790] without.

Conclusion. Instrumentation signals are correctly sent and received. A puzzling side observation of experiment 6 could be resolved.

[2] Program execution can be observed at high time resolution. With current CS parser frequency of 125 MHz, there are discrete time intervals of 8 ns. Timestamps assigned to CS frames differ by at least 32 ns (16 B frames; max. input to CS parser 4 B / clock cycle).

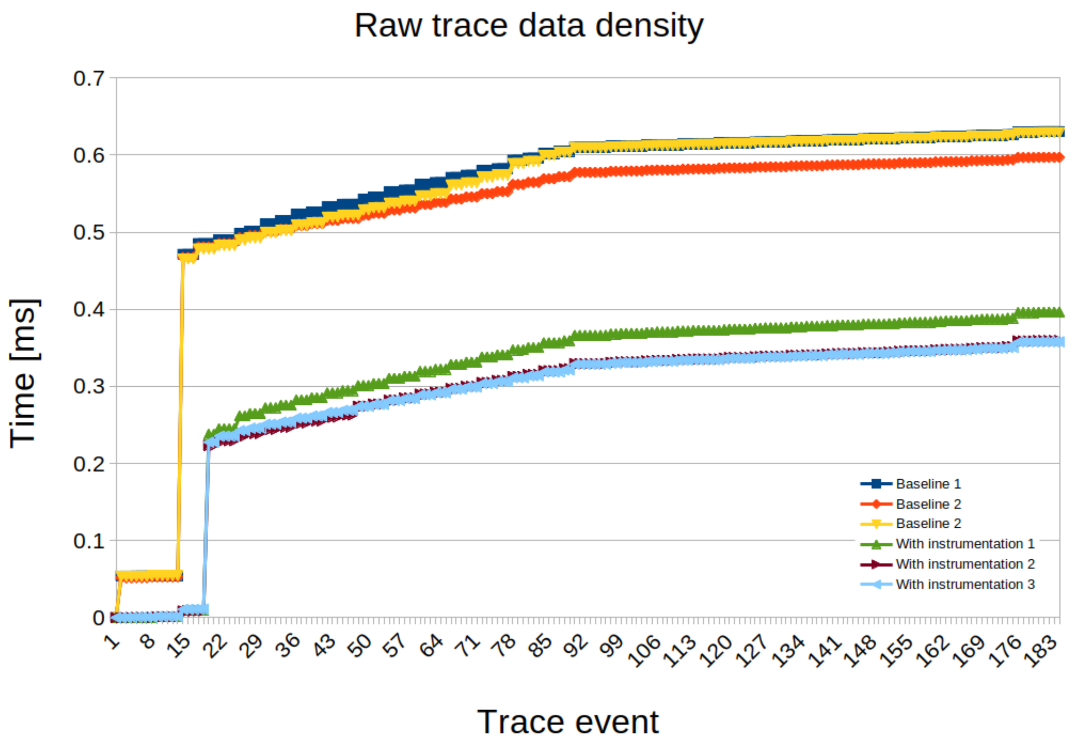

Figure 6.1: Experiment 6: trace density plots of instrumented and non-instrumented (baseline) program traces. Relative time from first trace event is shown on y axis in relation to event number on x axis. The flatter the curve, the more events per time.

6.3.2 Benchmarking

These experiments were designed to run for several seconds or even minutes generating several GB of raw CS trace data to be parsed and processed by the TeSSLa specification. Thus, filters were used in the specification to reduce the size of the output after processing.

TeSSLa can lift functions defined for values to apply them on event streams. This filter (Listing 6.1) effectively reduces the number of events to one event per 100'000 input events .

Listing 6.1: Filter macro

```
def filtered_malloc_count = filter(malloc_count,
                                   malloc_count % 100000 == 0)
```

Similar filters were used for the other instrumentation events. For addresses of the program trace, only one count per 1×10^6 addresses is forwarded.

With this, several GBs of raw trace could be processed to an output of several KiB without dropping any data of the raw trace.

Experiment 9: malloc/free single-threaded

Methods. A test program repeated 100 times: 1×10^6 times allocating 8 B memory and then free this memory. Time was measured inside of the program using `clock_gettime()` using the monotonic clock source. Each configuration was run 3 times. The program was run with affinity for core 1.

Results. Without instrumentation, the program run in average for 41.8 s, generating a 4 GB long raw trace; average raw CS trace throughput 97.3 MB/s, peak 121.8 MB/s, no drops. Thus, an average `malloc() / free()` call took 0.209 ± 0.000 μs. This average includes very short calls that can be resolved in glibc directly and calls that need syscalls to increase the heap.

With instrumentation, the program run in average for 118.5 s, generating a 5.5 GB long raw trace; average raw CS trace throughput 47.0 MB/s, peak 51.6 MB/s, no drops. All 100×10^6 `malloc()` and `free()` calls were reported with the ITM instrumentation. Filters were chosen to detect even the lack of one signal. An average instrumented `malloc() / free()` call took 0.593 ± 0.000 μs. Thus, average instrumentation overhead for `malloc() / free()` instrumentation is 0.384 μs.

In each configuration, over 700×10^6 addresses were parsed from core 1 and processed in TeSSLa, i.e. 16.7×10^6 addresses/s in the baseline configuration. Time measurements in the program and derived from first and last event in the processed trace matched (as expected).

Discussion. Relative instrumentation cost for more typical programs, that do not only consist of `malloc() / free()`, will be lower. Nevertheless, there is room for optimization in the instrumentation code that is e.g. using regular CS access library calls to send stimuli to ITM. If the final write to the memory-mapped register `*(unsigned int volatile *)(d->local_addr + off) = data;` (code from CSAL) was done directly in the instrumentation wrapper, several function calls in the library could be saved. This is not implemented yet. Main focus was on achieving reliable instrumentation first.

Experiment 10: malloc/free multi-threaded

Methods. Two threads run in parallel, one on each core. Each repeats 50 times: 1×10^6 times allocating 8 B memory and then free this memory, as in experiment 9. The main goal of this test was to see whether the instrumentation works correctly even in multi-threaded programs.

To collect program traces of all threads, the `-p` option was used in `trace_launch` that disables filtering based on contextID. Otherwise, only the trace of the main thread would be collected.

Results. Without instrumentation, the program run in average for 26.8 s, generating a 4.4 GB long raw trace; average raw CS trace throughput 165.2 MB/s, peak 190.2 MB/s, no drops. Over 330×10^6 addresses were parsed from each of the cores. Exact number differ (interleaving). No perfect speed-up of 2 is expected due to the implementation of `malloc()` and `free()`.

With instrumentation, the program run in average for 68.8 s, generating a 5.8 GB long raw trace; average raw CS trace throughput 82.4 MB/s, peak 96.7 MB/s, no drops. All

100×10^6 `malloc()` and `free()` calls were reported with the ITM instrumentation. Even one missing signal would have been detected.

Experiments 11 and 18: lock/unlock, no contention

Methods. This test creates 255 threads to fill the internal hash map (section 5.6). A barrier was used to allow these thread creations to be complete before proceeding. Then, 3 nested mutex locks are locked and unlocked 10×10^6 times with an inner critical section incrementing a counter. This locking/unlocking happens single-threaded in the main thread with affinity to core 1 to avoid any contention. Time measurement as mentioned before.

Time and traces were collected in 3 configurations: a) no instrumentation, b) slower instrumentation with 2 hashmap lookups, c) faster instrumentation with 1 hashmap lookup (default). The same filtering TeSSLa specification was used as in experiments 9 and 10.

Please note: the instrumentation sends 2 signals for successfully acquired locks: lock request and lock acquired. This can be used for some TeSSLa specifications. However, the expensive hashmap lookup happens only once.

Results. All 30×10^6 lock request, lock acquired and unlock signals were received in both instrumentation options. Note: already missing one signal would be detected in the used configuration. Results were reproducible.

A peak raw trace throughput was observed during the average 4.6 s run without instrumentation: 260.3 MB/s, average 256.0 MB/s for the 1.1 GB raw trace, no drops. Instrumented versions took significantly longer: average 72.5 s (slow instrumentation), 50.7 s (faster instrumentation) with lower CS raw trace throughput of 20.1 MB/s and 35.3 MB/s for the 1.5 and 1.7 GB long traces, respectively.

Lock is very fast without contention: average 0.077 ± 0.000 μs for each `pthread_mutex_lock` / `pthread_mutex_unlock` call. With slow instrumentation 1.203 ± 0.010 μs, average instrumentation cost 1.126 μs. With faster instrumentation 0.846 ± 0.003 μs, average instrumentation cost 0.769 μs.

Discussion. The discussion of this relevant instrumentation overhead is in section 7.1. During development of experiment 17, the instrumentation library needed to be adjusted: the unlock signal had to be moved from after actual unlock function to before the function (i.e. into the critical section). Otherwise, due to the measured relevant overhead, sometimes unlock and lock acquired signals of different threads were in wrong order. Experiment 18 repeated experiment 11 with the new instrumentation library. Equal results (as expected in the setting without contention).

Experiments 12 and 19: lock/unlock, with contention

This experiment evaluates whether all instrumentation signals can be received even in case of many threads running in parallel on both cores, and to evaluate the effect of contention on the instrumentation overhead. 255 threads were created on both cores.

Each was repeating 100'000 times: acquire all 3 locks, increment a global variable in the critical section, unlock all 3 locks.[3]

Methods. The experiment was run in 2 configurations: a) without instrumentation, b) with default (faster) instrumentation. Each configuration was run 3 times. The program was launched with `-p` option active to collect the program traces of all threads.

Results. All 76.5×10^6 lock request, lock acquired and unlock signals were received. Already one missing signal would have been detected.

Average of `pthread_mutex_lock` / `pthread_mutex_unlock` calls was longer in this experiment due to the created contention in lock calls: 0.120 $\pm$ 0.004 μs without instrumentation; 0.909 $\pm$ 0.000 μs with instrumentation. Average instrumentation cost remained stable with 0.789 μs.

Discussion. The discussion of this relevant instrumentation overhead is in section 7.1. This experiment was repeated as experiment 019 after the instrumentation library change for experiment 017. Also here, signals were received correctly. Measured instrumentation overhead was equal.

Experiment 13: Overhead of a syscall

This experiment was run to compare instrumentation overhead in the current instrumentation library using ITM stimuli with other signal paths, e.g. `CONTEXTIDR` via kernel [31]. Thus, overhead of a syscall was measured for the specific Zynq board with specific Linux kernel and user environment including glibc as used for all other experiments.

Methods. A test programm called a non-existing syscall number 1024 1×10^6 times; in a second test `getpid()` was called 1×10^6 times. Time measurement as described before. In the used glibc 2.27 `getpid()` is not cached by glibc; it is also not listed in vDSO for ARM.

Results. One average empty syscall (nr 1024) takes 0.431 $\pm$ 0.000 μs (n=4); one average `getpid()` takes 0.490 $\pm$ 0.000 μs (n=4). This makes sense for the small amount of computation that is done in `getpid()`

6.4 Runtime verification with TeSSLa specifications

Three runtime verification scenarios were designed with TeSSLa specifications and test programs that satisfy or violate the specification (Table 6.1). The detailed properties are described in each experiment. Additionally, the experiments were designed to illustrate some pitfalls when writing TeSSLa specifications and to illustrate different usage modes of the RV system.

Properties defined in TeSSLa specifications are evaluated many times during program execution reporting the evaluation for the program trace up to this point in time. The number of evaluations depends on the defined specification based on processing of the

[3] While the lock order matters in nested locks, the unlock order does not matter as long as a released lock is not acquired again. To have a bit more interesting contention, the nested locks were released in the same order as acquired (as e.g. used for list traversal) and not in reverse order (as typically released with RAII in C++ or `synchronised` blocks in Java).

Table 6.1: Runtime verification with TeSSLa specifications: overview

Experiment	Scenario	Methods
14	Memory allocation	preload instrumentation, fixed-capacity set
15	Event handler, simple	direct tracking of function calls
16	Event handler, advanced	additionally fixed-capacity maps
17	Critical section and locks	instrumentation, fixed-capacity sets

trace stream in TeSSLa. The final value reports whether the property holds for the entire program.

Experiment 14: Memory allocation

Property. All allocated memory blocks are deallocated at the end of the program run. Such a property can be expressed in LTL as:

$$\Box(x = malloc(\_) \longrightarrow \Diamond free(x))$$

TeSSLa. A fixed-capacity set is used to track addresses of memory allocations. Addresses are added to the set at allocation and removed upon deallocation. The set size is used to define the property in combination with overflow detection. A trivial specification that only counts the number of allocations and deallocations would be insufficient, of course.

As an additional side-property, the specification tracks whether `malloc()` would return the same address before it has been deallocated with `free()`.[4] This property is only valid if no overflow of the set has happened, similar to main property.

Limitation: In the current prototype version of the TeSSLa compiler, the capacity of the set had to be limited to 8. The discussion of this important limitation is in section 7.2.

Listing 6.2 shows the key components of the specification using a fixed-capacity set (Listing B.3) and helper predicate `none()` (Listing B.1).

Listing 6.2: Memory allocation

```
-- instrumentation signals: in_malloc and in_free
def key = merge(in_malloc, in_free)
def allocated = lookup(key, time(in_malloc) == time(key), 8)

def n_allocations = allocated._1
def overflow_flag = allocated._2
def overflow = overflow_flag == 0

def never_overflow = none(overflow)
def all_allocations_freed = n_allocations == 0 && never_overflow
```

[4] This property came as an example for the fixed-capacity set from Daniel Thoma. It was then embedded into the the `none()` helper predicate (Listing B.1).

```
-- side property: "double malloc", i.e. malloc() returns an address
-- before it has been deallocated with free()
-- note: only valid if no overflow
def changed = prev(n_allocations) != n_allocations
def double_malloc = !changed && time(in_malloc) == time(key)
def never_double_malloc = none(double_malloc)

-- from stdlib
def prev[T](events: Events[T]) = last(events, events)
```

Methods. A modified `nested_malloc` program allows four test configurations.

- default: 8 memory allocations and deallocations
- missing free: one memory block is not deallocated
- additional allocation after freeing the initial ones (not violating the set capacity)
- overflow: 12 memory allocations with given capacity of 8

Instrumentation signals of the preloaded instrumentation wrapper were used in the TeSSLa specification.

Results. All instrumentation signals were received. The defined property was satisfied by the default configuration. Both property violations and the overflow were detected correctly. Also the side-property ″double malloc″ was evaluated correctly.

Experiment 15: event handler, simple

Setting. Multiple client threads enqueue requests into a thread-safe queue implementation. A single-threaded event handler processes all requests similar to e.g. user interface event handlers. One request type (named A) generated by one client thread is tracked in detail: request, start processing, finished processing. Configuration guarantees that no new request A is requested before the former has finished processing. Either 3 or 19 other threads request another type simulating any number of other request types.

Property. Queueing time of all requests type A $\leq$ 5 ms.

TeSSLa. A simple solution is shown in Listing 6.3 using helper predicate `all()` (Listing B.1) and a macro to detect function calls in the address streams of both cores (Listing B.2). The function addresses can be found with the `nm` tool. A tool is provided in the project to determine the binary encoding of time given in either s, ms, μs or ns.

Listing 6.3: Event handler: queueing time property, simple

```
-- adjust addresses to compiled program binary
def request_a = function_call(0x00010eb0)
def start_a = function_call(0x00010c3c)
def finished_a = function_call(0x00010c54)

def simple_queueing_time =
  time(start_a) - last(time(request_a), start_a)
```

```
def simple_response_time =
  time(finished_a) - last(time(request_a), finished_a)

-- queueing time <= 5 ms
def p = simple_queueing_time <= 0x989680
def simple_property = all(p)
```

Methods. The test program (as described above) was designed to satisfy the property with 3 additional threads and to violate it with 19 threads on the 2 core Zynq board. Other configurations (service time of the request, request interval for the threads) were kept identical in both configurations. Request interval was identical for all threads: 20 ms. This guaranteed that a request A was finished before a new request was issued. Request type A was requested 10 times.

Results. Time measurements were correct. Queueing time for request type A in configuration with 3 additional threads was in range 1.1 to 3.4 ms, satisfying the property; with 19 additional threads in range 8.9 to 12.0 ms, violating the property. The property was evaluated correctly for both configurations.

Discussion. An important limitation of the current prototype compiler is discussed in section 7.2 (`delay()` functionality).

Experiment 16: event handler, advanced

Setting. Identical setting as in experiment 15 with the exception that the request interval for all threads is reduced from 20 ms to 2 ms. Thus, some requests type A are issued before former requests have finished or even started processing (long queue). This experiment was designed to illustrate a pitfall of the simple specification and to show a solution by using additional tags.

Property. Queueing time of all requests type A $\leq$ 5 ms.

TeSSLa. The simple time calculation used above fails in this scenario by under-estimating queueing and response time. When evaluating queueing time for e.g. request number 3 at time point `time(start_a)` already request number 4 or 5 may have been issued. Thus, `last(time(request_a), start_a)` contains the time of this newer request. The same applies to response time.

To keep track of events that belong to each other, i.e. request, start, finished for specific request number, the fixed-capacity set provided by Daniel Thoma (Listing B.3) was extended to a fixed-capacity map (Listing B.4). This map implementation has similar limitations as the set implementation (section 7.2 for discussion). In this example, the count macro (Listing 4.1) was used to provide tags. Alternatively, tags could be provided by the program using e.g. instrumentation signals.

Listing 6.4: Event handler: queueing time property, advanced

```
-- adjust addresses to compiled program binary
def request_a = function_call(0x00010eb0)
def start_a = function_call(0x00010c3c)
def finished_a = function_call(0x00010c54)
def request_a_count = count(request_a)
def start_a_count = count(start_a)
def finished_a_count = count(finished_a)

def key1 = merge(request_a_count, start_a_count)
def value1 = time(key1)
def store_kv_pair1 = time(request_a_count) == time(key1)
def tracking_map1 = lookup_map(key1, value1, store_kv_pair1, 8)

def key2 = merge(request_a_count, finished_a_count)
def value2 = time(key2)
def store_kv_pair2 = time(request_a_count) == time(key2)
def tracking_map2 = lookup_map(key2, value2, store_kv_pair2, 8)

def queueing_time_with_map = filter(time(start_a) -
  tracking_map1._2, time(tracking_map1._2) == time(start_a))

def response_time_with_map = filter(time(finished_a) -
  tracking_map2._2, time(tracking_map2._2) == time(finished_a))

-- queueing time <= 5 ms
def q = queueing_time_with_map <= 0x989680
def property_with_map = all(q)
```

Methods. As described in experiment 15.

Results. All time calculations with the map method were correct. As soon as requests type A were enqueued before the processing of earlier requests had started (for queueing time) or finished (for response time), these time intervals were too short as explained above. The actual subtraction of time was correct, of course. The property was evaluated correctly for all configurations.

Discussion. An important limitation of the current prototype compiler is discussed in section 7.2 (`delay()` functionality).

Technical note. The TeSSLa specifications for experiment 15 and 16 were in one TeSSLa file: 211 lines including interface code (Listing A.1), output definitions, and comments. It compiled via Chisel to 41'236 lines of Verilog code. Combined with CS parser code and AXI/TPIU controller, 56% of available LUT and 29% of available BRAM were used in the FPGA.

Experiment 20: Critical section protected by lock

This experiment was deliberately designed to illustrate current limitations in the instrumentation library and TeSSLa functionality in the prototype compiler.[5] Future implementations will improve.

[5] Technical notes: 1) As can be seen in Listing 6.5, testing locks needs some complexity in the specification. Thus, only while developing the specification for this experiment, a subtle problem was detected in the

Setting. 8 threads are concurrently calling a function `critical_section()`, which increments a shared variable by one with each call. The shared variable is initialised to 0. A barrier lets the threads wait until all are ready. Each thread repeats 128 times: acquire lock, call this function once, release lock.

As a configuration option to violate the property, the main thread calls `critical_section()` once without lock at some time while all 8 threads are working.

Property. Critical section only executed when lock is acquired and only executed once per lock.

TeSSLa. Different ways can be chosen to implement this property in a TeSSLa specification. For this example at the current state of the TeSSLa compiler and instrumentation infrastructure, several sub-properties are implemented in the specification to implement the property:[6]

- max. one lock is used
- lock is held by max. one thread
- critical section not accessed if no lock is held
- critical section only called once per lock; this implies that another thread does not access the critical section while another thread holds the lock
- all these sub-properties must be satisfied all the time for the property output of the specification to indicate true

Thus, an additional thread does not access a critical section while no lock is held nor when a lock is held, which implies all the time. The TeSSLa specification is shown in Listing 6.5.

There are two limitations at the moment: 1) Tracking more than one lock would require more advanced features, like maps of sets. Thus, the initial check that only one lock is used. Such additional data structures – or other functions to allow such tracking – may be implemented in the future.

2) Of course, the specification would be strengthened significantly if it could also directly assert that the call to the critical section was made by the identical thread as the thread that just acquired the lock. This can be achieved by using the contextID values from CS `I-sync` and `Context ID` packets, that are parsed in the PTM parser and sent as inputs to the TeSSLa specification for each core separately. However, the current version of the instrumentation library does not embed the core ID into the instrumentation signal, which

instrumentation library. Due to the relevant computing time in the instrumentation, sending the ITM signal after unlock had to be moved to before the actual unlock operation, i.e. into the critical section. Thus, potentially affected experiments of this change (experiments 11 and 12) were repeated as experiments 18 and 19 (section 6.3). 2) This experiment was first run as number 17. It had a small bug in the specification. The `waiting_set` used the acquired instead of requested event to set the store-key flag. This set was only used for illustration/demonstration purpose and does not affect the property, which was correctly evaluated already in experiment 17. Nevertheless, the experiment was repeated as experiment 20 that also reports this information correctly.

[6] A trivial specification (count number of lock calls and compare with number of critical section calls) would not suffice for the property.

would be required to correlate it with a contextID in the TeSSLa specification. This is an option to consider for the future.[7] However, it adds to to the instrumentation overhead.

Specifications that work with function call mapping inside of the specification (Listing B.2 used in experiments 15 and 16) could use the information from contextID already now to map specific calls to specific threads. Program trace information from each core is provided separately to the TeSSLa specification.

Listing 6.5: Critical section protected by lock

```
-- 1) track critical section --
def critical_section = itm_value13

-- for illustration
def cs_count = count(critical_section)

-- a workaround: types of both arguments in last()
-- must match at the moment
def cs_as_bool = first(true, critical_section)

-- 2) assure that max. one lock is used: required --
def locks_set = lookup(in_lock_request_addr, true, 3)
def locks_count = locks_set._1
def locks_assert = all(locks_count <= 1)

-- 3) track threads waiting for locks: optional, for illustration --
def waiting_key = merge(in_lock_request_tid, in_lock_acquired_tid)
def waiting_set =
  lookup(waiting_key,
         time(in_lock_request_tid) == time(waiting_key), 8)
def waiting_count = waiting_set._1
def waiting_overflow = waiting_set._2 == 0
def waiting_never_overflow = none(waiting_overflow)

-- 4) track threads currently holding the lock: required --
--    note: count must be <= 1 at all time
def locked_key = merge(in_lock_acquired_tid, in_unlock_tid)
def locked_set =
  lookup(locked_key,
         time(in_lock_acquired_tid) == time(locked_key), 8)
def locked_count = locked_set._1
def locked_overflow = locked_set._2 == 0
def locked_never_overflow = none(locked_overflow)
```

[7] An early test version of the instrumentation library actually used separate core specific ITM ports for each instrumentation signal using twice as many ports on the 2 core system. `sched_getcpu()` was used to determine the core inside of the instrumentation wrapper. This functionality was removed because it would not scale to more cores and to avoid additional instrumentation overhead. However, with different instrumentation channels than ITM, individual instrumentation signals could be implemented for each core even on a multi-core system such as Enzian.

```
def locked_assert = all(locked_count <= 1)

-- 5) assert that critical_section is only accessed
--    if one lock is held
def cs_with_lock = last(locked_count, critical_section) == 1
def cs_assert1 = all(cs_with_lock)

-- 6) assert that critical_section is only called once
--    when lock is held
--    note: dependent on thread interleaving, an unlocked
--    access to the critical section can happen while
--    another thread is holding a lock.
def cs_allowed = {
    -- this is a simple version that works with one lock
    -- an instance needs to be used for each lock,
    -- which requires e.g. maps
    -- see *_addr streams to distinguish locks
    def acquired_lock = first(true, in_lock_acquired)
    def released_lock = first(false, in_unlock)
    def cs_started = first(false, critical_section)
    merge(cs_started, merge(released_lock, acquired_lock))
}

def current_cs_allowed = last(cs_allowed, cs_as_bool)
def cs_assert2 = all(current_cs_allowed)

-- 7) combine assertions and required side-conditions
def cs_assertion = cs_assert1 && cs_assert2 &&
                   locks_assert && locked_assert

-- optional: to have fewer output signals
def filtered_cs_assertion = filter(cs_assertion, cs_as_bool)
```

Methods. The mentioned test program was run 3 times with each configuration.

Results. In all runs, the defined property could be evaluated correctly under the limitations discussed above.

6.5 Summary

During evaluation, correct parsing of raw CS traces and instrumentation signals could be demonstrated.

Instrumentation overhead could be determined for wrapped memory allocation and mutex functions (summary in Table 6.2; default instrumentation settings; data collected and analysed in experiments 9-12, 18, 19).

Table 6.2: Summary: average instrumentation overhead. SD < 0.005 μs for all averages; n=3. O:D ratio = Overhead:Direct function call ratio.

Functions	Direct [μs]	Instrumented [μs]	Overhead [μs]	O:D ratio
malloc/free	0.209	0.593	0.384	1.8
lock/unlock, no contention	0.077	0.846	0.769	10.0
lock/unlock, some contention	0.120	0.909	0.789	6.6

And finally, complete runtime verifications could be illustrated for three properties in different application domains (Table 6.1). Test programs were designed to satisfy or violate the defined properties. TeSSLa specifications could be used to evaluate the properties on FPGA while the programs were running on the CPU. All evaluations were correct.

Chapter 7

Discussion

7.1 Instrumentation overhead

Benchmarking (section 6.3) has revealed a relevant instrumentation overhead in the order of average malloc/free call itself but about 10x the absolute time of lock/unlock calls (Table 6.2). This overhead is compared to other solutions below in section 7.3. It significantly affects runtime of programs that consist only of such calls like the benchmark programs. However, relative cost of instrumentation may be less pronounced in typical programs where also other processing is done in addition to the instrumented calls.

Nevertheless, there is room for optimization in the instrumentation library. The primary focus has been a reliably working system. Optimisation opportunities:

Direct writing into mapped register. Currently, CSAL library functions are used to write stimuli into the memory-mapped register. The instrumentation wrapper could be modified to write directly into the registers, which would save several function calls.

Instrumentation directly in wrapper. To allow flexible use of the instrumentation library also in other scenarios like compile-time instrumentation (Table 2.1), the instrumentation wrapper calls the instrumentation function of the library. Also this call could be prevented by directly have all implementations inside of the wrapper.

For mutex/lock/barrier. To merge thread ID and mutex/lock/barrier ID into one signal, a hashmap lookup needs to be done at each instrumentation call. The currently used library uthash [18] was chosen for the advantages mentioned in the implementation (section 5.6). However, there are much faster libraries that could be used instead.

Additionally, in systems like Enzian, a different instrumentation signal path needs to be used than ITM (upgrade path in subsection 7.4.2). This gives the opportunity to even avoid such a hashmap lookup completely. This was another reason why no additional effort was put into optimising this instrumentation solution for speed at the moment.

7.2 Limitations of the TeSSLa compiler prototype

The current TeSSLa compiler prototype can be used successfully to run many TeSSLa specifications in this CPU/FPGA hybrid RV solution (section 6.4). However, it cannot compile all language features yet.

Events in the future. For example, `delay()` could not yet be used. This function is required to write powerful specifications that can define events in the future. For example: The event handler specifications used in the evaluation (Listings 6.3 and 6.4) recognize the prolonged queueing time at the moment when processing of the event starts. Using `delay()`, a future event can already be defined at the time when the request event is observed, which can then be used to recognize such a property violation at the time when the processing should have started and not when it actually started.[1] This is the desired functionality for an online RV system. There is a nice `Stimulus Response Pattern` example on `www.tessla.io` that illustrates this functionality.

Data structures. This limitation includes also important data structures, like sets and maps, that are available in the software version of TeSSLa but not yet in the prototype compiler. The currently used TeSSLa macros for fixed-capacity set (provided by Daniel Thoma; Listing B.3) and fixed-capacity map (Listing B.4) are clear workarounds at the moment until a better solution is available in the upcoming new compiler.

Both constructs do not scale beyond capacity 15 (set) and 8 (map) at the moment. One cause of this limitation (in tessla2chisel) is already fixed in the next version of the compiler. However, when creating the examples, I had also to reduce the set capacity to 8 because larger sets could not be implemented at clock frequency 50 MHz (negative slack). Thus, there is another upper bound in the generated Chisel/Verilog code.

Additionally, they have limited functionality at the moment. Reading removes the element from sets and maps. Thus, it would need to be stored again to implement updates of a set, which adds complexity.

In contrast to software implementations, maps and sets must have a fixed capacity for synthesis on FPGA. However, instead of implementing them in TeSSLa itself, they could be implemented as Chisel modules that can then be instantiated with configured capacity from default map and set data structures in TeSSLa. Such a direct implementation in Chisel offers a lot of room for optimization compared to the current workarounds in TeSSLa. It could also add more features (like updates) to provide full semantics of typical software sets/maps.

Processing bandwidth. With a maximum event rate of 25 M events/s per input adapter (Table 4.1), the specification network is the bottleneck at the moment. A future version of the compiler can improve (some suggestions in subsection 7.4.3).

On the other hand, filters in the CS trace parsers, in particular for PTM/ETM traces, can be adjusted. Currently, for evaluation of the initial system, the PTM parser forwards all decompressed addresses of `Branch address` and `Waypoint update` packets to the specification.[2] The parser is prepared to use additional filters, e.g. for address ranges. Many of the transmitted addresses are not needed by actual specifications. This can relevantly reduce the number of events that are sent to the TeSSLa computation network.

[1] Worst-case scenario: a property violation might be missed completely if processing never starts. Other workaround solutions to detect this might also evaluate this property too late for online RV.

[2] The address of `I-sync` packets is not forwarded but only used to adjust the internal state of the parser. ContextID information of the `I-sync` packets is forwarded instead.

Therefore, for practical applications, the bottleneck is not as severe as it seems to be when comparing the CS parser output and TeSSLa input bandwidths for event rates (Table 4.1).

7.3 Related work

Software TeSSLa implementation

The software version of TeSSLa can e.g. be accessed on `www.tessla.io`. As mentioned above, the current compiler prototype cannot yet handle all language features. However, a new compiler version is being built in Lübeck at the time.

Existing FPGA TeSSLa implementation

There exists already a TeSSLa implementation in FPGA that uses predefined modules, which are reconfigured to define the specification (section 3.2). Because this solution is closed source and proprietary, no comparisons could be made with this system. Conceptually, the RV solution of this thesis should be able to allow more complex specifications and/or higher CS raw trace bandwidth than configurable modular system (section 3.2 for explanation). Synthesising specific FPGA solutions for each TeSSLa specification has the disadvantage that it takes much more time than a reconfiguration of a system. However, once a specification is synthesised, its bit file can be loaded to the FPGA almost as fast as such a reconfiguration.

Alternative instrumentation path via CONTEXTIDR

Instrumentation overhead for glibc functions (malloc/free) is 0.384 μs with the current library using ITM. An alternative instrumentation using CONTEXTIDR via syscall has been described [31] (section 3.3 for details). A raw syscall takes 0.431 μs on the Zynq board with currently used Linux version and user environment (experiment 13 in section 6.3).

Thus, the currently used instrumentation method is faster even without the optimizations mentioned above (section 7.1).

The also mentioned method of adding the CONTEXTIDR modification to an existing syscall (piggy-back), which has an overhead of only 0.014 μs (on the slower Zed board used in the paper [31]), cannot be applied to instrumentation of glibc functions that may not issue a syscall with each call.

Other CPU/FPGA hybrid solutions

The CPU/FPGA hybrid solution mentioned above [31] can be used to collect instrumentation data of programs that can be used for runtime verification in any software system offline afterwards. The solution presented in this thesis, additionally forwards addresses and contextID values from the program trace, ITM and FTM signals. Additionally, it embeds a TeSSLa specification on the FPGA that allows online runtime verification while the program is running.

CS program traces have been used to integrate monitoring for Code Reuse Attacks (CRA) into CPU/FPGA hybrid systems [24]. ROP and JOP attacks can be detected implementing known algorithms for this purpose. In this solution, memory access was observed in addition to the program trace. This is not yet available in the current RV system of this thesis. Conceptually, TeSSLa could be used to write such applications. However, the current prototype compiler is too limited.

Other runtime verification solutions

Correct evaluation of three different specifications in different domains could be demonstrated for the implemented RV system (section 6.4). Thus, it is a working RV system as other described RV systems [1, 7, 10, 11, 19, 22, 25, 29].

Most solutions run in software offline after the program. The RV solution presented here can evaluate the specification online while the program is running. It can also write the output data stream of the TeSSLa to disk while the program under test is running. Thus, true online monitoring of an even longer running program is possible.

Additionally, this output can be forwarded via network connection to a larger specification server that collects and processes such reduced data streams of many systems in a rack or even server farm. Also on such a system, processing can be implemented in TeSSLa, creating a hybrid distributed specification system that scales.

7.4 Future work

7.4.1 Runtime verification of CPU/FPGA hybrid applications

The current RV system can already include signals from FPGA into the CS stream and use this information in a TeSSLa specification. It has been used during development to send internal signals of the parsers to the output stream.

Thus, not only software but CPU/FGPA hybrid applications can be verified by TeSSLa specifications. Such an application could not yet be tested.

7.4.2 Migration path to ThunderX / Enzian

Many of the gained insights on the Zynq board prototype are key for the future port to a more complex CPU/FPGA hybrid system like Enzian. This includes in particular the method to scale TeSSLa's input port to multiple inputs without merging (subsection 5.2). The following upgrade path includes various steps that I had planed for this thesis already but could not execute due to delays.

1. Collect CS traces from an Enzian prototype. The source codes for a patched version of the ARM trusted firmware (ATF) and of the received trace collecting library (with patches to compile) are in our internal code repository.
2. Transfer of trace data to the FPGA side. A PCI express connection between CPU and FPGA sides can be used as a first start. Mapped memory, that is using the cache coherency protocol of the CPU/FPGA interconnect, will provide more bandwidth and lower latency. There are additional options as well.
3. The CS streams are not multiplexed in frames as on the Zynq board. Thus, no demultiplexing will be needed. The various CS traces can be connected with the

various parsers. However, the input on the FPGA needs to be adjusted to translate the input stream into the required buffer format (count, buffer) and data transfer frequency (currently set to once per 4 clock cycles) as required by the parsers. Additionally, a timestamp needs to be provided to the parsers similarly to the one generated in the Frame Synchronizer.

4. The Enzian system does not offer ITM nor FTM infrastructure as on the Zynq board. However, there are clear options to integrate also instrumentation data and FPGA fabric data into the system. As a suggestion: memory mapping via cache coherency protocol can be used for instrumentation data. This can start simple as one memory mapped "register" and can be extended to multiple such registers providing multiple ports even for each traced CPU. Of course, parsing needs to be adjusted. An (asynchronous) FIFO can be used for fabric inputs.
5. The current PTM parser needs to be extended to the ETMv4 [4] specification, which has additional features. As a first step, focus should be on the program trace alone. To keep clock frequency high or increase it, I planned adding additional states in this parser to handle long packets in separate computation steps. Additional data parts can be added.
6. An extended version of the Timestamp Driver, including a suggestion for a new time/data width is already in our internal code repository (folder fpga64).
7. The current tessla2chisel compiler and tessla-synthesis can easily be adjusted to the new number of inputs and to the wider time/data types. A new version of the compiler may increase input event rate (suggestions in subsection 7.4.3). Higher clock frequencies should be tested in all cases on the faster FPGA architecture used in Enzian.

Of course, any step needs to be adjusted to findings that will only be available after running some tests. I have learned that there may always be smaller to larger surprises.

7.4.3 TeSSLa to Verilog compiler

Work has already started at the Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany for a new TeSSLa compiler. Some new features will be influenced by experiences gained in this project with the prototype compiler.

Optimization of Chisel modules. It is worthwhile to analyse the currently used Chisel modules in detail. If the clock frequency limiting modules can be modified into small pipelines of smaller processing steps, then the clock frequency of the entire specification can be increased leading to a relevant bandwidth increase. This work may already have started with the new compiler.

Optimization of TeSSLa to Chisel compilation. There is optimisation potential in the TeSSLa to core TeSSLa and core TeSSLa to Chisel compilation steps. This will enable more complex specifications.

Direct Chisel implementations of macros. Several complex macros are implemented in TeSSLa itself at the moment. This includes the workaround fixed-capacity set (Listing B.3) and map (Listing B.4) macros. More efficient implementations can be written in Chisel. However, to instantiate such modules, the compilation process needs to be adjusted.

At the moment, a TeSSLa specification is compiled to core TeSSLa language first, which is then used for compilation to Chisel. The initial compilation step needs to be adjusted to forward e.g. maps, sets, and other defined macros *as is* to the Chisel compiler. In this case, even regular maps and sets of TeSSLa could be used in the specification as they are used in the software version of TeSSLa. These could be translated to fixed-capacity variants with configurable capacity.

Such direct implementations could be applied to other complex macros as well, in particular for macros of the new TeSSLa standard library, which is being developed in Lübeck at the moment.

Chapter 8

Conclusion

The presented solution in this thesis offers a complete and working runtime verification system on Xilinx Zynq-7000 SoC using ARM CoreSight with CS trace parsing and verification with a TeSSLa specification on the FPGA.

Programs can be instrumented using ITM. Additional fabric traces of the FPGA can be integrated into the CS stream and processed by the CS parser and specification.

A solution could be found to feed multiple event streams of different sources in parallel into the TeSSLa stream processing network. This is a critical step to scale this system towards multi-core systems like Enzian.

Specifications in the experiments were designed specifically to explore the boundaries of the current instrumentation library and TeSSLa compiler. Future upgrades of the TeSSLa compiler – currently being developed at the Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Germany – will allow more complex specifications to run.

Three different specifications of different domains were tested on the current RV system with TeSSLa specifications and programs satisfying and violating the properties. All property evaluations were correct.

The system can be used for online runtime verification. The output of the TeSSLa specification can be read and forwarded to other computers, e.g. a dedicated specification server, while the program-under-test is running. This can be used for distributed RV, which is another critical step for rack-scale runtime verification.

An upgrade path has been shown for the Enzian CPU/FPGA hybrid system.

Overall, the current RV solution is a step stone towards the larger goal of rack-scale runtime verification [9].

Appendix A

Reference TeSSLa Interfaces

The `tessla2chisel` compiler maps input and output labels in alphabetical order to internal data stream indices. All TeSSLa specifications must adhere to the reference input interface (Listing A.1).

Instrumentation data can be accessed with the reference instrumentation mapping (Listing A.2).

Output labels can be chosen arbitrarily. However, they are multiplexed into the output interface in alphabetical order. The current system, therefore, uses an explicit mapping to dedicated output labels to make this order clearly visible (Listing A.3).

Listing A.1: TeSSLa interface: reference input interface

```
-- version 2019-07-30
-- input interface ------------------------------------------------------
in etm0_addr: Events[Int]
in etm0_value1_contextid: Events[Int]
in etm0_value2: Events[Int]
in etm0_value3_error: Events[Int]

in etm1_addr: Events[Int]
in etm1_value1_contextid: Events[Int]
in etm1_value2: Events[Int]
in etm1_value3_error: Events[Int]

in ftm_value0: Events[Int]
in ftm_value1: Events[Int]
in ftm_value2: Events[Int]
in ftm_value3_error: Events[Int]

in itm_value00: Events[Int]
in itm_value01: Events[Int]
in itm_value02: Events[Int]
in itm_value03: Events[Int]
in itm_value04: Events[Int]
in itm_value05: Events[Int]
in itm_value06: Events[Int]
in itm_value07: Events[Int]
in itm_value08: Events[Int]
in itm_value09: Events[Int]
in itm_value10: Events[Int]
in itm_value11: Events[Int]
in itm_value12: Events[Int]
in itm_value13: Events[Int]
in itm_value14: Events[Int]
in itm_value15: Events[Int]
in itm_value16: Events[Int]
in itm_value17: Events[Int]
in itm_value18: Events[Int]
in itm_value19: Events[Int]
in itm_value20: Events[Int]
in itm_value21: Events[Int]
in itm_value22: Events[Int]
in itm_value23: Events[Int]
in itm_value24: Events[Int]
in itm_value25: Events[Int]
in itm_value26: Events[Int]
in itm_value27: Events[Int]
in itm_value28: Events[Int]
in itm_value29: Events[Int]
in itm_value30: Events[Int]
in itm_value31: Events[Int]
in itm_value32_error: Events[Int]

-- contextID consist of 24 bits processID/threadID and 8 bits ASID
-- note: processID should be unique per process
-- however, testing has shown that it is even unique per thread in
-- multi-threaded programs (see pthread)
-- in default configuration, the PTM parser sends contextID
-- only if it has changed

def etm0_thread_id = (etm0_value1_contextid >> 8) & 0x00ffffff
def etm0_asid = etm0_value1_contextid & 0x000000ff

def etm1_thread_id = (etm1_value1_contextid >> 8) & 0x00ffffff
def etm1_asid = etm1_value1_contextid & 0x000000ff
```

Listing A.2: TeSSLa interface: instrumentation library reference mapping

```
-- version 2019-07-30
-- reference mapping of current instrumentation library --------------
-- memory: address of allocated / freed memory block

def in_malloc = itm_value00
def in_free = itm_value01


-- concatenation of thread-id and mutex/lock/barrier address
-- mask 0xff000000 contains mapped thread-id [0,255]
-- mask 0x00ffffff contains 24 bit representation of the
-- mutex/lock/barrier address
-- precise address representation depends on configuration of
-- the instrumentation library:
-- see CAT_USE_FAST_MODE flag

def in_lock_request = itm_value02
def in_lock_acquired = itm_value03
def in_lock_not_acquired = itm_value04
def in_unlock = itm_value05

def in_rd_lock_request = itm_value06
def in_rd_lock_acquired = itm_value07
def in_rd_lock_not_acquired = itm_value08

def in_wr_lock_request = itm_value09
def in_wr_lock_acquired = itm_value10
def in_wr_lock_not_acquired = itm_value11

def in_barrier_wait = itm_value12
def in_info = itm_value30
def in_error = itm_value31


-- splitting tid and mutex/lock/barrier address examples:
-- technical note: right shift must happen *before* masking
-- 0xff000000 for direct masking cannot be handled properly in
-- the TeSSLa compiler pipeline at the moment

def in_lock_request_tid = (in_lock_request >> 24) & 0x000000ff
def in_lock_request_addr = in_lock_request & 0x00ffffff
def in_lock_acquired_tid = (in_lock_acquired >> 24) & 0x000000ff
def in_lock_acquired_addr = in_lock_acquired & 0x00ffffff
def in_unlock_tid = (in_unlock >> 24) & 0x000000ff
def in_unlock_addr = in_unlock & 0x00ffffff
```

Listing A.3: TeSSLa interface: reference output interface

```
-- output: can be adjusted; outputs 000 to 255 can be used ----------------
out output000
out output001
out output002
out output003
out output004
out output005
out output006
out output007
out output008
out output009
out output010
out output011
out output012
out output013
out output014
out output015
out output016
out output017
out output018
out output019
out output020
-- etc. for all used outputs
```

Appendix B

TeSSLa Language Snippets

Listing B.1: TeSSLa: helper predicates

```
def all(p: Events[Bool]) = c where {
  -- all events must be true
  -- first result with first event
  def c: Events[Bool] = merge(last(c, p), true) && p
}

def none(p: Events[Bool]) = c where {
  -- no true events
  -- first result before any events
  def c: Events[Bool] = merge(last(c, p) && (!p), true)
}

def some(p: Events[Bool]) = c where {
  -- at least one event must be true
  -- first result with first event
  def c: Events[Bool] = merge(last(c, p), false) || p
}

def some_with_init(p: Events[Bool]) = c where {
  -- at least one event must be true
  -- first result before any events,
  -- i.e. captures the case that no event was received at all
  def c: Events[Bool] = merge(last(c, p) || p, false)
}
```

Listing B.2: TeSSLa: function_call() macro

```
-- syntax follows function_call() macro on tessla.io playground,
-- which can take function names as argument
def function_call(function_address: Int) = c where {
  -- filter first and merge second to avoid huge bottleneck
  def a = filter(etm0_addr, etm0_addr == function_address)
  def b = filter(etm1_addr, etm1_addr == function_address)
  def c = merge(a, b)
}
```

Listing B.3: TeSSLa: fixed-capacity set

```
-- fixed-capacity set
-- TeSSLa macro provided by Daniel Thoma, University of Luebeck,
-- Germany
-- max capacity 15 with current prototype compiler

def prev[T](events: Events[T]) = last(events, events)

def lookup(key: Events[Int], f: Events[Bool], size: Int):
    (Events[Int], Events[Int]) = {

  def l: Events[Int] = last(reg, key)

  def add = f && l == -1
  def found = l != -1 && key == l
  def remove = !f && found

  def reg = merge(if add then key else if remove then -1 else l, -1)

  static if size == 0 then
    (default(nil[Int], 0), filter(const(0, key), f))
  else {
    def result = lookup(key, f && !(add || found), size - 1)
    (result._1 + if reg != - 1 then 1 else 0, result._2)
  }
}


-- this macro is used for malloc() / free() examples, e.g. as

def mallocAddress = merge(in_malloc_0, in_malloc_1)
def freeAddress = merge(in_free_0, in_free_1)

def t = merge(mallocAddress, freeAddress)

def allocated = lookup(t, time(mallocAddress) == time(t), 15)

def changed = prev(allocated._1) != allocated._1
def doubleMalloc = !changed && time(mallocAddress) == time(t)
```

Listing B.4: TeSSLa: fixed-capacity map

```
-- fixed-capacity map
-- an extension of the fixed-capacity set macro
-- max capacity 8 with current prototype compiler
-- limitation: keys and values must be > 0 at the moment
-- this macro is used for advanced TeSSLa examples
-- with event_handler test framework
--
-- input: if f == true:  add key/value pair to map
--                false: lookup & removal of key
-- returns ._1: current map size
--         ._2: if f == true:  always returns -2
--                   false: returns value if key was found, 0 otherwise
--         ._3: 0 if overflow, nothing otherwise
-- version 2019-07-25

def lookup_map(key: Events[Int], value: Events[Int],
    f: Events[Bool], capacity: Int):
    (Events[Int], Events[Int], Events[Int]) = {

  def lk: Events[Int] = last(regk, key)
  def lv: Events[Int] = last(regv, key)

  def add = f && lk == -1
  def found = lk != -1 && key == lk
  def remove = !f && found

  def regk = merge(if add then key else if remove then -1 else lk, -1)
  def regv = merge(if add then value else if remove then -1 else lv, -1)

  static if capacity == 0 then
    (default(nil[Int], 0), default(nil[Int], 0),
     filter(const(0, key), f))
  else {
    def result = lookup_map(key, value, f && !(add || found),
                            capacity - 1)
    (result._1 + if regk != -1 then 1 else 0,
      if lk == key then lv else
          if result._2 > 0 then result._2 else -2,
      result._3)
  }
}
```

Appendix C

Timestamp Driver

The timestamp driver (written in VHDL) is needed to push the processing in the TeSSLa network forward even for inputs without events for longer periods of time. Despite looking very simple now, there are subtle details that needed to be considered in the implementation. It is part of a co-design of 3 modules: this timestamp driver, the new `MultiInputAdapter` and modified `InputAdapter` in TeSSLa (subsection 5.2) for details. The driver is embedded between the CS parsers (PTM, ITM, FTM) and their corresponding asynchronous FIFOs to the TeSSLa specification.

Listing C.1: Timestamp driver

```
-- version 2019-07-18

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tessla_timestamp_driver is
port (
    -- control interface
    CLK       : in std_logic;
    RST       : in std_logic;

    -- FIFO control signals (RST, WR_CLK, WR_RST_BUSY, FULL)
    -- are connected outside of this module

    -- input
    IN_0_WR_EN   : in std_logic;
    IN_0_DIN     : in std_logic_vector(123 downto 0);

    IN_1_WR_EN   : in std_logic;
    IN_1_DIN     : in std_logic_vector(123 downto 0);

    IN_2_WR_EN   : in std_logic;
    IN_2_DIN     : in std_logic_vector(123 downto 0);
    IN_3_WR_EN   : in std_logic;
    IN_3_DIN     : in std_logic_vector(123 downto 0);

    -- output:
    OUT_0_WR_EN   : out std_logic;
    OUT_0_DIN     : out std_logic_vector(123 downto 0);

    OUT_1_WR_EN   : out std_logic;
    OUT_1_DIN     : out std_logic_vector(123 downto 0);

    OUT_2_WR_EN   : out std_logic;
    OUT_2_DIN     : out std_logic_vector(123 downto 0);

    OUT_3_WR_EN   : out std_logic;
    OUT_3_DIN     : out std_logic_vector(123 downto 0)
);
end tessla_timestamp_driver;

architecture behavioral of tessla_timestamp_driver is

signal sync_needed : std_logic;
signal sync_in     : std_logic_vector(123 downto 0);
signal sync_out    : std_logic_vector(123 downto 0);

-- to have x clock cycles @ CS_PARSER_CLK use
-- x << 4, due to 4 bit timestamp extension
-- x should be > the number of cycles the longest parser needs to
-- process a frame
-- for compatibility, x should be a multiple of 4
-- currently x = 1024 => x << 4 = 16384
constant delta_t : unsigned(52 downto 0) := to_unsigned(16384, 53);

-- mainly for debugging purpose
constant dummy_data : std_logic_vector(31 downto 0) := X"00C0FFEE";

-- tracking for the final push: somewhat arbitrary config
-- note: it will send the last seen timestamp + final_delta_t
-- as defined below
constant final_wait_t       : natural := 12500000;
-- wait for 100 ms at 125 MHz
constant final_stop_t       : natural := final_wait_t + 1;
signal final_wait_counter   : natural;

-- just add 1 clock cycle to have an increment
-- thus: 1 << 4 = 16;
constant final_delta_t      : unsigned(52 downto 0)
```

```
                                         := to_unsigned(16, 53);
signal final_last_timestamp : unsigned(52 downto 0);
signal final_needed         : std_logic;
signal final_out            : std_logic_vector(123 downto 0);


-- this method must match exactly the TeSSLa encoding used in the parsers
pure function extract_time(v : std_logic_vector(123 downto 0))
return unsigned is
    variable r : unsigned(52 downto 0);
begin
    r := unsigned(v(52 downto 0));
    return r;
end;

-- this method must match exactly the TeSSLa encoding used in the parsers
pure function embed_time(t : unsigned(52 downto 0))
return std_logic_vector is
    variable r : std_logic_vector(123 downto 0);
begin
    r := (others => '0');

    -- set address
    r(122 downto 115) := X"FF"; -- default: 0xFF; for debugging: 0x03

    -- set data
    r(93 downto 62) := dummy_data;

    -- set time
    r(61) := '1';
    r(52 downto 0) := std_logic_vector(t);
    return r;
end;


pure function create_broom_wagon(v : std_logic_vector(123 downto 0))
return std_logic_vector is
    variable r : std_logic_vector(123 downto 0);
    variable t : unsigned(52 downto 0);
begin
    t := extract_time(v);
    if t > delta_t then
        t := t - delta_t;
    else
        t := to_unsigned(1, 53);
    end if;
    r := embed_time(t);
    return r;
end;

begin

sync_needed <= IN_0_WR_EN or IN_1_WR_EN or IN_2_WR_EN or IN_3_WR_EN;

-- give priority to slower parsers because they have lower timestamps
-- 1. PTM
-- 2. FTM == ITM

sync_in <= IN_0_DIN when IN_0_WR_EN = '1' else
           IN_1_DIN when IN_1_WR_EN = '1' else
           IN_2_DIN when IN_2_WR_EN = '1' else
           IN_3_DIN;

sync_out <= create_broom_wagon(sync_in);

OUT_0_WR_EN <= IN_0_WR_EN or sync_needed or final_needed;
OUT_1_WR_EN <= IN_1_WR_EN or sync_needed or final_needed;
OUT_2_WR_EN <= IN_2_WR_EN or sync_needed or final_needed;
OUT_3_WR_EN <= IN_3_WR_EN or sync_needed or final_needed;

OUT_0_DIN <= IN_0_DIN when IN_0_WR_EN = '1' else
             sync_out when sync_needed = '1' else final_out;
OUT_1_DIN <= IN_1_DIN when IN_1_WR_EN = '1' else
             sync_out when sync_needed = '1' else final_out;
OUT_2_DIN <= IN_2_DIN when IN_2_WR_EN = '1' else
             sync_out when sync_needed = '1' else final_out;
OUT_3_DIN <= IN_3_DIN when IN_3_WR_EN = '1' else
             sync_out when sync_needed = '1' else final_out;

track_time : process(CLK) is
begin
    if rising_edge(CLK) then
        if RST = '1' then
            -- tracking
            final_last_timestamp <= (others => '0');
            final_wait_counter <= 0;

            -- output
            final_needed <= '0';
            final_out <= (others => '0');
        else
            if sync_needed = '1' then
                final_last_timestamp <= extract_time(sync_in);
                final_wait_counter <= 0;
            else
                if final_wait_counter = final_wait_t then
                    final_needed <= '1';
                    final_out <= embed_time(final_last_timestamp
                                 + final_delta_t);
                    final_wait_counter <= final_wait_counter + 1;
                elsif final_wait_counter = final_stop_t then
                    final_needed <= '0';
                else
                    final_wait_counter <= final_wait_counter + 1;
                end if;
            end if;
        end if;
    end if;
end process;

end behavioral;
```

Appendix D

trace_launch Options

`trace_launch` is the central program for CoreSight configuration, binary start with optional instrumentation, and output collection during/after the run. Configuration options are shown in Listing D.1.

Listing D.1: trace_launch options

```
Usage: trace_launch [-t] [-d] [-b] [-p] [-z] [-c <nr>] [-s <nr>]
                    [-P] [-I] [-F] [-T] [-i <init.cx>]
                    [-o <output dir>] <program to launch> [<arg>, ...]

main config
  -t  Route trace via TPIU, not ETB.
  -d  Enable verbose debugging output.
  -b  Disable branch broadcast (default is ON)
  -p  Disable processID/contextID matching (default ON is optimal for
      single-threaded programs; OFF is better for multi-threaded
      programs (memo: pthread)
  -z  online ('infinity') mode; automatically activates -t;
      continuously read output FIFO to <output dir>/cstrace.bin

core nr and sync period settings
  -c  use core <nr> for <program to launch>; can be used multiple
      times (set of cores); core affinity is set to traceable
      cores, if not defined
  -s  TPIU synchronization every <nr> CS frames [1, 4096];
      default 64 corresponds to once per 1024 B

trace collection config
  -P  Enable LD_PRELOAD instrumentation wrapper for <program>;
      also activates -I; trace_launch must be launched with
      build folder as PWD (memo: PRELOAD)
  -I  Enable ITM
  -F  Enable FTM: activates both, CS FTM source input and FTM packet
      generation on FPGA board via zynq_emio_ctrl driver
```

```
path settings
  -i  <init.cx> bytecode file of a CS configuration;
      value of ENV variable INIT_CS_FILENAME is used if not provided
      ENV variable is required for -P option
  -o <output dir>: this folder is created; output files go there
```

Appendix E

Instrumentation Library

Instrumented library functions of glibc:

- from malloc.h including workaround memory allocator during init phase: malloc, calloc, realloc, free
- from malloc.h without workaround: reallocarray, memalign, valloc, pvalloc
- from pthread.h: pthread_create to create thread ID mapping
- from pthread.h: mutex and rwlock functions
- from pthread.h: optional spinlock functions (inactive in default setting)
- from pthread.h: barrier_wait

Appendix F

Programs for Trace Validation

Listing F.1: Test program: nested_malloc

```
#include <malloc.h>
#include <stdio.h>
#define N 12
#define ALLOC_SIZE 4096

void allocate(int n) {
    if (n <= 0) return;
    int *m = malloc(ALLOC_SIZE);
    if (m == NULL) {
        printf("error:␣no␣malloc\n");
        return;
    }
#ifndef QUIET
    printf("n␣%d;␣stack␣address␣&m␣%p;␣allocated␣heap␣address␣m␣%p\n",
          n, &m, m);
#endif
    allocate(n-1);
    free(m);
#ifndef QUIET
    printf("n=%d;␣free(m)\n", n);
#endif
}

int main(void) {
#ifndef QUIET
    printf("main()␣␣␣␣␣at␣%p\n"
           "allocate()␣at␣%p\n"
           "malloc()␣␣␣at␣%p\n"
           "free()␣␣␣␣␣at␣%p\n",
           main, allocate, malloc, free);
#endif
```

```
    allocate(N);
    return 0;
}
```

Listing F.2: Relevant TeSSLa snippet used for nested_malloc

```
def etm0_addr_time = time(etm0_addr)
def etm1_addr_time = time(etm1_addr)

def etm0_addr_delta = etm0_addr_time - last(etm0_addr_time, etm0_addr)
def etm1_addr_delta = etm1_addr_time - last(etm1_addr_time, etm1_addr)

def etm0_contextid_time = time(etm0_value1_contextid)
def etm1_contextid_time = time(etm1_value1_contextid)

def etm0_contextid_delta = etm0_contextid_time -
    last(etm0_contextid_time, etm0_value1_contextid)
def etm1_contextid_delta = etm1_contextid_time -
    last(etm1_contextid_time, etm1_value1_contextid)


def count[A](a: Events[A]) := c where {
    def c: Events[Int] := merge(last(c, a) + 1, 0)
}

def etm0_addr_count = count(etm0_addr)
def etm1_addr_count = count(etm1_addr)

def etm0_contextid_count = count(etm0_value1_contextid)
def etm1_contextid_count = count(etm1_value1_contextid)

def malloc_count = count(in_malloc)
def free_count = count(in_free)
def barrier_count = count(in_barrier_wait)

-- counts only mutex locks; other lock types work identically
-- option: signals can be merged before processing, if desired
def lock_request_count = count(in_lock_request)
def lock_acquired_count = count(in_lock_acquired)
def unlock_count = count(in_unlock)


-- output example: many values are reported here for testing;
-- typical specs may have fewer outputs

def output000 = etm0_addr
def output001 = etm0_addr_count
def output002 = etm0_contextid_count
def output003 = etm0_thread_id
def output004 = etm0_asid

def output005 = etm1_addr
def output006 = etm1_addr_count
def output007 = etm1_contextid_count
def output008 = etm1_thread_id
def output009 = etm1_asid

-- use -t flag in parse_tessla_output for time output streams
def output010 = etm0_addr_delta
def output011 = etm1_addr_delta
def output012 = etm0_contextid_delta
def output013 = etm1_contextid_delta

-- arbitrary values can be transmitted via ITM; see test/itm_test program;
-- show selection here
def output014 = itm_value20
def output015 = itm_value21
def output016 = itm_value22
def output017 = itm_value23
def output018 = itm_value24
def output019 = itm_value25

def output020 = ftm_value0

-- mapped instrumentation interface
def output021 = in_malloc
def output022 = in_free
def output023 = in_lock_request
def output024 = in_lock_acquired
def output025 = in_lock_not_acquired
def output026 = in_unlock
def output027 = in_rd_lock_request
def output028 = in_rd_lock_acquired
def output029 = in_rd_lock_not_acquired
def output030 = in_wr_lock_request
def output031 = in_wr_lock_acquired
def output032 = in_wr_lock_not_acquired
def output033 = in_barrier_wait

def output034 = in_lock_request_tid
def output035 = in_lock_request_addr
def output036 = in_lock_acquired_tid
def output037 = in_lock_acquired_addr
def output038 = in_unlock_tid
def output039 = in_unlock_addr
def output040 = in_info
def output041 = in_error

def output042 = malloc_count
def output043 = free_count
def output044 = barrier_count
def output045 = lock_request_count
def output046 = lock_acquired_count
def output047 = unlock_count


-- TeSSLa system info / error messages
def output048 = etm0_value3_error
def output049 = etm1_value3_error
def output050 = ftm_value3_error
def output051 = itm_value32_error
```

Bibliography

[1] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Brooks Davis, and Ilias Marinos. TESLA: Temporally enhanced system logic assertions. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014.

[2] ARM. *CoreSight Components Technical Reference Manual*. reference DDI 0314H, ARM Ltd., 2009.

[3] ARM. *CoreSight Program Flow Trace Architecture Specification*. reference IHI 0035B, ARM Ltd., 2011.

[4] ARM. *ARM Embedded Trace Macrocell Architecture Specification*. reference IHI 0064D, ARM Ltd., 2016.

[5] ARM. *ARM CoreSight Architecture Specification*. reference IHI 0029E, ARM Ltd., 2017.

[6] ARM. CoreSight Access Library (CSAL), 2017. `https://github.com/ARM-software/CSAL`, accessed: 2019-08-07.

[7] Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime Verification for Multicore SoC with High-quality Trace Data. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):18:1–18:26, April 2013.

[8] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining Outputs in Modern Data Analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, August 2016.

[9] David Cock. *Litmus Testing at Rack Scale*. The 2016 Workshop on Multicore and Rack-scale Systems MaRS'16 at EuroSys 2016, 2016.

[10] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. TeSSLa: Temporal Stream-Based Specification Language. In Tiago Massoni and Mohammad Reza Mousavi, editors, *Formal Methods: Foundations and Applications*, pages 144–162, Cham, 2018. Springer International Publishing.

[11] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 43–63, Cham, 2018. Springer International Publishing.

[12] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, June 2005.

[13] Edsger W. Dijkstra. Software Engineering Techniques. *Report on a conference in Rome, Italy October 27 to 31, 1969 sponsored by the NATO Science Committee*, page 16, 1969.

[14] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A Stream-Based Specification Language for Network Monitoring. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 152–168, Cham, 2016. Springer International Publishing.

[15] S. Di Girolamo, P. Schmid, T. Schulthess, and T. Hoefler. SimFS: A Simulation Data Virtualizing File System Interface. In *33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19)*. IEEE, May 2019.

[16] Enzian group. Enzian System, 2018. `http://enzian.systems`, accessed: 2019-08-07.

[17] Dilian Gurov, Klaus Havelund, Marieke Huisman, and Rosemary Monahan. Static and Runtime Verification, Competitors or Friends? *ISoLA 2016, LNCS 9952*, pages 397–401, 2016.

[18] Troy D. Hanson and Arthur O'Dwyer. uthash, 2006. `https://troydhanson.github.io/uthash/`, accessed: 2019-08-07.

[19] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *First Workshop on Runtime Verification (RV'01). Theoretical Computer Science*, 55(1), 2001.

[20] IEEE. IEEE Standard for System, Software, and Hardware Verification and Validation. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017)*, pages 1–260, Sep. 2017.

[21] Intel. Processor Tracing, 2013. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, accessed: 2019-08-07.

[22] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovie. From signal temporal logic to FPGA monitors. In *MEMCOD*, pages 218–227, 09 2015.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification

of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[24] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Trans. Des. Autom. Electron. Syst.*, 22(3):52:1–52:25, April 2017.

[25] Martin Leucker. Teaching Runtime Verification. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 34–48, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[26] Linaro. OpenCSD - An open source CoreSight Trace Decode library, 2015. `https://github.com/Linaro/OpenCSD`, accessed: 2019-08-07.

[27] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems - Safety*. Springer, 1995.

[28] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, August 2009.

[29] Andrei Pârvu. *Program Trace Capture and Analysis for ARM*. ETH Zürich, Systems Group, Master's Thesis Nr. 153 supervised by Prof. Timothy Roscoe and Dr. David Cock, 2016.

[30] Albert Schulz. Implementierung eines TPIU-Streamdekoders in VHDL, 2016. presentation, `https://tu-dresden.de/ing/informatik/ti/vlsi/ressourcen/dateien/dateien_studium/dateien_lehstuhlseminar/vortraege_lehrstuhlseminar/folder-2016-04-20-4701759038/presentation.pdf?lang=de`, accessed: 2019-08-07.

[31] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Arnab Kumar Biswas, Vianney Lapôtre, and Guy Gogniat. A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components. *CoRR*, abs/1812.01667, 2018.

[32] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1:172–179(7), October 2007.

[33] Xilinx. *AXI Reference Guide*. reference UG761, Xilinx Ltd., 2012.

[34] Xilinx. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide*. reference UG953, Xilinx Ltd., 2017.

[35] Xilinx. *Zynq-7000 SoC Technical Reference Manual*. reference UG585, Xilinx Ltd., 2018.

[36] Xilinx. *ZC706 evaluation board for the Zynq-7000 XC7Zo45 SoC User Guide*. reference UG954, Xilinx Ltd., 2019.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Runtime Verification with TeSSLa on Enzian

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schmid	Pirmin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Gwatt, August 14, 2019	P. Schmid

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.