

# Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems

Normann Decker<sup>1</sup>(✉), Philip Gottschling<sup>2</sup>(✉), Christian Hochberger<sup>2</sup>(✉),  
Martin Leucker<sup>1</sup>(✉), Torben Scheffel<sup>1</sup>(✉), Malte Schmitz<sup>1</sup>(✉),  
and Alexander Weiss<sup>3</sup>(✉)

<sup>1</sup> Institute for Software Engineering and Programming Languages,  
Universität zu Lübeck, Lübeck, Germany

{decker, leucker, scheffel, schmitz}@isp.uni-luebeck.de

<sup>2</sup> Rechnersysteme, Technische Universität Darmstadt, Darmstadt, Germany

{gottschling, hochberger}@rs.tu-darmstadt.de

<sup>3</sup> Accemic Technologies GmbH, Kiefersfelden, Germany  
aweiss@accemic.com

**Abstract.** This paper presents an approach for rapidly adjustable embedded trace online monitoring of multi-core systems, called RETOM. Today, most commercial multi-core SoCs provide accurate runtime information through an embedded trace unit without affecting program execution. Available debugging solutions can use it to reconstruct the run offline, but usually for up to a few seconds only. RETOM employs a novel online reconstruction technique that makes the program run available outside the SoC and allows for evaluating a specification formulated in the stream-based specification language TeSSLa in real time. The necessary computing performance is provided by an FPGA-based event processing system. In contrast to other hardware-based runtime verification techniques, changing the specification requires no circuit synthesis and thus seconds rather than minutes or hours. Therefore, iterated testing and property adjustment during development and debugging becomes feasible while preserving the option of arbitrarily extending observation time, which may be necessary to detect rarely occurring errors. Experiments show the feasibility of the approach.

## 1 Introduction

Software for resource-constrained environments demands for an application-specific and highly optimised implementation. Testing and debugging is challenging in this setting because of strong limitations regarding the acquisition and analysis of execution information. On one hand, comprehensive logging output provided by the software decreases the performance significantly and requires to anticipate the information needed in the debugging and testing process. On

---

This work is supported in part by the European Cooperation in Science and Technology (COST Action ARVI), the BMBF projects ARAMIS II with funding ID 01 IS 16025 and CONIRAS with funding ID 01 IS 13029, and the European Horizon 2020 project COEMS under number 732016.

© The Author(s) 2017

S. Cavalheiro and J. Fiadeiro (Eds.): SBMF 2017, LNCS 10623, pp. 179–196, 2017.

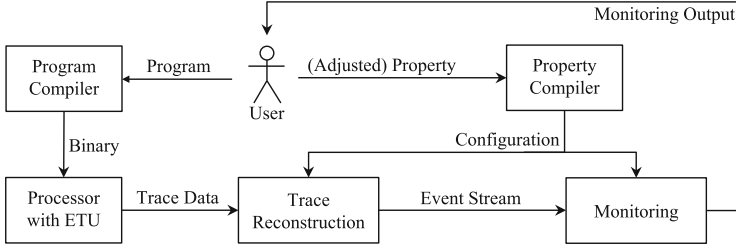
[https://doi.org/10.1007/978-3-319-70848-5\\_12](https://doi.org/10.1007/978-3-319-70848-5_12)

the other hand, runtime information can be observed dynamically using automatic code instrumentation, producing suitable program output, or standard breakpoint-based debugging features of the processor. The latter methods, however, are highly intrusive as they modify the software temporarily for the analysis or interrupt the execution. This is especially problematic for concurrent programs running on multi-core processors or real-time applications. Errors due to race conditions or inappropriate timing may be introduced or hidden.

To allow for a non-intrusive observation of the program trace, many modern microprocessors feature an *embedded trace unit (ETU)* [2, 12, 14, 28]. An ETU delivers runtime information to a debug port of the processor in a highly compressed format. State-of-the-art debugging solutions, such as ARM DSTREAM [3], allow the user to record this information for offline reconstruction and analysis. The essential disadvantage of this technology is, however, that traces can be recorded for at most a few seconds because high-performance memory with very fast write access is required to store the delivered information. For example, the ARM DSTREAM solution offers a trace buffer of 4 GB for a recording speed of 10 Gbit/s or more which means that the buffer can only hold data of less than four seconds. While the majority of errors can be found immediately within a short program trace, some of them may only be observable on long-running executions or under specific, rarely occurring (logical or physical) conditions. It is therefore desirable for the developer and maintainer to be able to monitor the program execution for an arbitrary amount of time during development and testing and even in the field after deployment.

**Contribution.** To overcome the limitations of current technology we propose a novel runtime verification methodology for evaluating long-term program executions that is suitable for development and debugging, testing, and in-field monitoring. Based on the runtime information provided by the ETU, we perform a real-time reconstruction of the program trace. The latter is evaluated with respect to a specification formulated by the user in the stream-based specification language TeSSLa [18]. To deliver sufficient performance for online analysis, both reconstruction and monitoring system are implemented using FPGA hardware.

FPGAs have become a very popular technology to implement digital systems. Designing digital circuits with FPGAs typically starts from hardware description languages like VHDL or Verilog. Synthesis software is responsible to map such designs to the elements available in an FPGA and then these elements must be positioned and routed on the FPGA fabric. Even for moderately large designs, this process can take hours. In case the design should run at high clock speed, this time is dramatically increased. Our monitoring system is therefore designed to not only evaluate a specific property specification. Instead, it builds on a flexibly and quickly configurable FPGA-based event processing platform described in [13]. We provide a tool chain for mapping TeSSLa specifications to the platform automatically within seconds. Formulating hypotheses, adapting property specification and checking them on the target system can be iterated quickly without time-intensive synthesis.



**Fig. 1.** General overview of the RETOM workflow cycle.

Figure 1 provides an overview of the proposed workflow based on our approach to *rapidly adjustable embedded trace online monitoring (RETOM)*. The user, e.g. the developer, tester, or maintainer, specifies the correct behaviour of the program under test based on program events such as a function call or variable access. The program is compiled and the binary is uploaded to the processor as usual. The property compiler automatically generates a corresponding configuration for the monitoring and trace reconstruction units that is then uploaded to the platform. When running the program on the processor, the monitoring platform reports the computed output stream to the user who can then use the information to adjust the program or the property.

We show the feasibility of RETOM in terms of a prototype implementation using the ARM CoreSight technology as specific but widely available variant of an ETU. A concurrent scenario is used to demonstrate the characteristics of parallel and time-critical applications and how corresponding runtime properties can be specified in TeSSLa and evaluated using our monitoring system.

**Related Work.** This paper focuses on runtime monitoring techniques which analyzes one particular program execution. For a general introduction into the field of runtime verification especially in comparison with static verification techniques such as model checking see [16, 17]. Non-intrusive observation of program executions is a long-standing issue [23] that becomes increasingly challenging with high circuit integration. On the other hand, integrated hardware extensions were described, e.g., in [29] and today many standard (“commercial off-the-shelf”) products feature advanced observation facilities [28]. Alternative approaches that aim at more powerful and flexible evaluation were developed based on programmable logic. Systems with on-chip programmable logic (SoPC) allow for direct observation and property evaluation by using specifically synthesised designs [26]. While this is appealing from a technical point of view, it introduces significant additional costs per unit. In [19] the authors propose similarly a partial reconfiguration of a (soft-core) processor. An external alternative based on a side channel is discussed in [22]. However, a lot of training is required in order to identify specific system behaviour. Extending the system by an external FPGA-based device using peripheral buses [25] seems more realistic, although it comes with the restriction that only the external communication on

**Table 1.** Comparison of hardware-supported monitoring frameworks. ETU refers to standard processors with embedded trace unit.

Framework	Non-intrusive	Online	Rapid adjustment	Trace source
QSTL mon. [15]	n/a	✓	–	n/a
P2V [19]	✓	✓	–	cust.
HidICE [5]	✓	✓	–	cust.
ptMTL mon. [24]	(✓)	✓	✓	cust./dbg. port
BusMOP [25]	✓	✓	–	periph. bus
SoPC monitoring [27]	–	✓	–	SoPC
ARM DSTREAM [3]	✓	–	✓	ETU
RETOM	✓	✓	✓	ETU

the used bus can be observed. A custom high-bandwidth trace interface is used in [5] to obtain trace data but the practical drawback is again, that this is not available in any standard product.

While many of these approaches have the merit of unbounded online evaluation, they are inconvenient in an iterative development or testing process because the properties to be evaluated are synthesised directly to programmable logic which is extremely time-consuming. The same applies to the monitor construction presented in [15]. A solution that allows for a rapidly adjustable evaluation of past-time MTL properties is given in [24]. Compared to ETU-based solutions, however, the used interfaces are not available on commonly available hardware or provide less runtime information, operate at low speeds and are, like JTAG, possibly intrusive. Table 1 provides an overview of related approaches.

This paper is organized as follows: In Sect. 2 we explain the online trace reconstruction on the FPGA. Section 3 describes TeSSLa, the language used to specify the monitors. How data flow graphs are constructed out of the specification and how monitors are synthesized on the FPGA is discussed in Sect. 4. Finally we present a case study in Sect. 5.

## 2 Trace Reconstruction

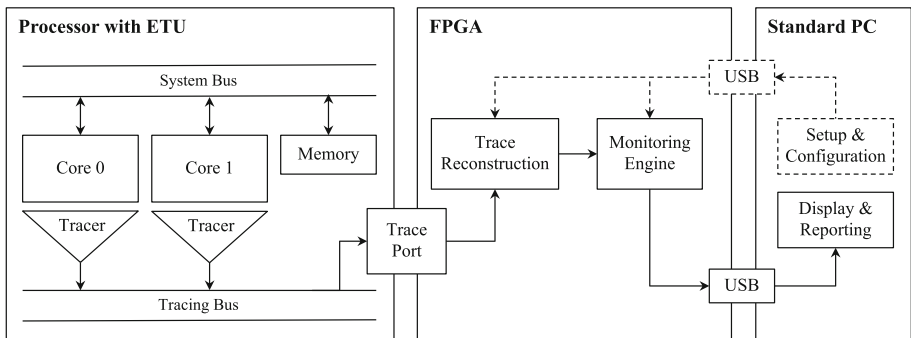
Figure 2 shows an overview of the RETOM setup: The cores of the multi-core processor are communicating with periphery, such as the memory, through the system bus. Every core is observed by its own tracer. The trace data is sent through the trace bus to the trace port without affecting the core. The trace bus is separated from the system bus and does not interfere with it. The trace port of the processor is connected to the FPGA on which the trace reconstruction and interpretation and the actual monitoring are located. The final monitoring output is displayed and reported on a standard PC connected via USB.

In this paper we use the ARM Coresight [2] debugging technology as a widely available example of an ETU, which is included in every current ARM

processor (Cortex M, R and A). In particular, we use the Program Flow Trace (PFT) [1] to acquire trace data of the operations executed by the ARM processors.

As stated in the PFT manual [1] the “PFT identifies certain instructions in the program, and certain events, as waypoints. A waypoint is a point where instruction execution by the processor might involve a change in the program flow.” With PFT we only observe as waypoints conditional and unconditional direct branches as well as all indirect branches and all other events, e.g. interrupts and other exceptions, that affect the program counter other than incrementing it. In order to save bandwidth on the trace bus, the Program Flow Trace Protocol (PFTP) does not report the current program counter address for every cycle. Especially for direct branches, the target address is not provided but only the information whether a (conditional) jump was executed or not. The full program counter address is sent irregularly for synchronization (I-Sync message). In case of an indirect branch those address bits that have changed since the last indirect branch or the last I-Sync message are outputted.

For RETOM we employ an online (real time) trace-reconstruction method implemented on the FPGA hardware [30,31]: From a static analysis of the binary running on the CPU we know all the jump targets of conditional direct jumps and can store those in a lookup table in the memory of the FPGA. Due to the high parallelism of the FPGA, we can split the trace data stream and reconstruct the program trace using the lookup table. The trace data stream can be split at the synchronization points that contain the full program counter address. A FIFO buffer stores the trace data stream until we reach the next synchronization point. The buffer must be able to store at least the trace data between two synchronization points. For further processing we then immediately filter the reconstructed trace by comparing the reconstructed addresses to a list of addresses, called *tracepoints*, that correspond to the input events used in the TeSSLa specification to be evaluated. This comparison is realized by adding an additional tracepoint flag to the lookup table. After putting the slices back



**Fig. 2.** Overview of the RETOM setup. Operations of the cores are traced by the ETU, the trace is then reconstructed, filtered and monitored on the FPGA.

together in the right order we end up with a stream of tracepoints. Every tracepoint contains an ID and a timestamp. The timestamp is either assigned by the ARM processor if cycle accurate tracing is enabled or during the reconstruction on the FPGA otherwise. Cycle accurate tracing is only available for certain processor architectures, because it requires high bandwidth on the trace port in order to attach timing information to every message. This trace-reconstruction approach can also be used for execution time measurement [9, 10].

Note that PFT traces logical addresses used in the CPU before the memory management unit (MMU) translates them to physical addresses, which are used to address certain cells in the memory. Because logical addresses are used in the program binary and by the CPU, RETOM does not need to handle physical addresses.

In a typical multithreaded application, we have multiple threads running on different cores and multiple threads running on the same core using any kind of scheduling. While we can distinguish instructions traced from the different CPUs, we have to consider the actual thread ID in order to distinguish different threads running on the same core. This information is provided by a so-called context ID message [2], sent every time when the operation system changes the context ID register of the CPU. The logical addresses for different threads might be exactly the same, because the MMU is reconfigured in the context switch to point to another physical memory region. If we see a context switch to another process, we have to change the lookup table for the program flow reconstruction information.

### 3 Specification of Trace Properties

In order to specify correctness properties as well as to describe the computation of statistical and numerical metrics based on the trace data, we use TeSSLa<sup>1</sup>. This temporal stream-based specification language is described and analyzed in detail in [18] and was specifically designed for program traces derived from ETUs. TeSSLa reasons over asynchronous input streams by deriving new streams from the input streams. This key concept supports both, the computation of metrics and specifying desired behavior of the observed program trace.

TeSSLa can be seen as an asynchronous extension of the stream based language LOLA [8]. LOLA is based on synchronous streams, but as we want to observe multi-core systems, we can not assume synchronization between the streams coming in from different cores. Because of that, TeSSLa uses asynchronous streams as underlying model, similar to Signal Temporal Logic (STL) [20]. However, TeSSLa provides rich data domains that allow for formulating quantitative specifications computing statistics and numerical temporal metrics. Furthermore, STL lacks a clean separation of the evaluation (expressed explicitly in terms of dependencies) and the data manipulation (expressed by each individual operation).

---

<sup>1</sup> For more information on TeSSLa see <http://www.isp.uni-luebeck.de/tessla>.

### 3.1 Syntax and Semantics of TeSSLa

TeSSLa supports signals and event streams, a concept which has already been used for example for the definition of Timed Regular Expressions [4]. Let in the following  $\mathbb{T}$  be a suitable time domain, e.g.  $\mathbb{Q}$ . An *event stream* is a partial function  $\eta : \mathbb{T} \rightarrow D$  where  $D$  is a data domain. This partial function is only allowed to be defined for finitely many timestamps in a finite interval. We call the set of time points at which an event stream  $\eta$  is defined  $E(\eta)$ . The set of all event streams over  $D$  is denoted by  $\mathcal{E}_D$ .

In addition to the definition in terms of partial functions, an event stream  $\eta \in \mathcal{E}_D$  can be naturally represented as a timed word with a sequence  $s_\eta = (t_0, \eta(t_0))(t_1, \eta(t_1)) \cdots \in (E(\eta) \times D)^\infty$  ordered by time ( $t_i < t_{i+1}$ ) and containing all event points, i.e.  $\{t \mid (t, v) \text{ occurs on } s_\eta\} = E(\eta)$ .

In contrast to event streams, a *signal* defines a value for every point in time. It is a piece-wise constant function  $\sigma : \mathbb{T} \rightarrow D$  that can be represented as an event stream of value update events and can thus only change its value a finite number of times within a finite time interval. We denote the set of time points at which the value of a signal changes by  $\Delta(\sigma)$ . The set of all signals over a data domain  $D$  is denoted by  $\mathcal{S}_D$ . Section 5 provides some practical examples of signals and event streams.

**Structure of TeSSLa Specifications.** The syntax of TeSSLa is inspired by existing stream-based specification languages like LOLA [8] and the underlying concept of functional reactive programming [11]. TeSSLa is built around the basic concept of deriving internal or output streams by applying functions to input streams or already derived internal streams. Because it is designed to be readable with prior knowledge of C-style programming languages, the derived streams are defined in an imperative manner. Consider the following example of a TeSSLa specification where we assume two input streams, an event stream  $e$  whose events are counted and an event stream  $trigg$  which is used as trigger.

```
define numberOfEvents := eventCount(e)
define triggerInLast2Sec := inPast(trigg, 2s)
define error :=
  filter(e, triggerInLast2Sec && numberOfEvents < 5)
out error
```

It defines the signal *numberOfEvents* as the result of applying the function *eventCount* :  $\mathcal{E}_D \rightarrow \mathcal{S}_{\mathbb{N}}$  to the input stream  $e$ . At every time point  $t \in \mathbb{T}$  the signal provides the number of events that occurred on  $e$  up to  $t$ . Also, the signal *triggerInLast2Sec* is true as long as *trigg* had an event at most two seconds ago. Further, an event occurs on the event stream *error* whenever an event occurs on  $e$  while during the past two seconds an event occurred on *trigg* and the number of events on  $e$  has not reached the limit of five, i.e. the event on  $e$  is not filtered out by the *filter* function. For readability, type annotations can be omitted and are inferred at compile time. The semantics of a TeSSLa specification is a mapping from a set of input streams to a set of output streams and the keyword **out**

defines *error* to be one of the latter. These are visible outside of the monitor and can be used for further processing or be presented to the user.

TeSSLa does not allow for recursive definitions of streams in any way. This leads to a large library of built in functions which incorporate specific recursive functionality. The big advantage of this approach is that the dependency graph of a TeSSLa specification is a directed acyclic graph. In combination with restricting real-time operators to refer only to the current and past events, this enables us to use more effective algorithms for synthesizing a specification onto an FPGA which leads to greater flexibility. The concrete process of doing so is described in Sect. 4.

By providing a set of built-in functions we can use an optimized translation for FPGA synthesis. Consider the idea of summing up the values of the events of an event stream. If the user would define this in a recursive fashion, the evaluation on the FPGA would typically consist of an adder and a delay unit storing the result of the adder such that it is used with the next input event of the stream to be summed up. By using a specialized function called *sum*, just one operation unit needs to be synthesized onto the FPGA that internally stores the last output in a register and adds it to the next value of the event stream.

Next, a selection of important functions available in TeSSLa is provided. An in-depth discussion of the TeSSLa design and an exhaustive list of available functions can be found in [18].

**Available Functions.** There are five different types of functions in TeSSLa: simple arithmetic functions, aggregations, stream manipulators, timing functions and temporal property functions.

*Simple arithmetic functions* combine multiple input signals with an arithmetic operation into one output signal. All operations available in common programming languages can also be used in TeSSLa, for example a function *add* for point-wise summation of the value of two signals or a function *mul* for multiplication. More complex calculation functions in TeSSLa are *aggregations*. These generally take event streams as input and produce a signal. For example the function  $sum : \mathcal{E}_{\mathbb{N}} \rightarrow \mathcal{S}_{\mathbb{N}}$  computes the sum of the data of all events on an event stream and always outputs the current sum. Variants for other additive types, like rational numbers  $\mathbb{Q}$  or time points  $\mathbb{T}$ , are also available; polymorphism is resolved at compile time. The function *eventCount* counts the events on an event stream that occurred until a certain point in time, ignoring the values carried by events. Another important function in this category is *mrval* :  $\mathcal{E}_D \times D \rightarrow \mathcal{S}_D$  that computes the *most recent value* of an event stream. It returns the value of the last event that happened or the default value given as second parameter as long as no event occurred, yet. With *mrval* one can transform event streams into signals and then apply arithmetic functions on them.

Conversely, *sampling functions* convert signals into event streams. The function *changeOf* :  $\mathcal{S}_D \rightarrow \mathcal{E}_D$  returns an event stream with an event at those points in time where the value of the input signal changes. The function *sample* :  $\mathcal{S}_D \times \mathcal{E}_D \rightarrow \mathcal{E}_D$  samples a signal clocked by an event stream and thus returns for every input event an event containing the values of the signal at the respective point in time.



With *stream manipulators* one can split and combine streams. Typical functions are *filter*, that works like a mask and deletes events by a certain criteria, and *merge*, that combines two event streams into one. Constructs like if-then-else are also stream manipulators essentially combining filter and merge functionality.

With *timing functions* one can refer to the past or future given a (real) time offset. The functions *delay* and *shift* are delaying a signal for a certain amount of time or shifting the values of the events of an event stream by a certain number of events, respectively. The two functions *inPast* and *inFuture* let us describe if an event happened on an event stream a certain amount of time in the past or future, respectively.

Finally there is the generic *monitor* function. This function provides a closed scope for specifying properties in different propositional temporal logics. For example, LTL or SALT [7] can be used with classical (finitary) semantics or more informative ones like LTL<sub>3</sub> [6]. Especially for the last one, this closed scope is needed because the LTL formula has to be processed in a complex way to build the monitor. Hence it has to be known what exactly belongs to the formula. The input consists of a set of boolean signals as propositions and an arbitrary event stream as clock for stepping the monitor. The type of the output stream depends on the output type of the used semantics.

### 3.2 Observation Specification

With the observation specification we can define in the TeSSLa specification certain streams based on the tracepoints generated by the online trace reconstruction. Such an observation can be defined on three different levels: (1) On the level of the C code, (2) on the level of the binary and (3) on the level of the processor. Because in the end we need to define the tracepoints for the trace reconstruction in terms of logical addresses in the binary, we need to translate the code level and the processor level to the binary level. On the binary level we can simply define streams with an event each time a given logical address is executed. On the code level we can define streams with an event each time a function is entered or left or each time a certain line of code gets executed. This information can be translated to the execution of logical addresses in the binary using the debug information in the binary. On the processor level we can for example specify streams with an event each time a floating point instruction is executed. This could be translated to the execution of logical addresses in the binary by simply analysing the binary for all floating point operations and listing all their addresses. In this paper we will use the following TeSSLa functions to define streams on the code level:

- `functionCalls("<file>:<function>")` creates an event each time the function with the specified name in the given file is entered,
- `functionReturns("<file>:<function>")` creates an event for leaving the function and
- `codeLine("<file>:<line>")` creates an event each time the given line in the given file is executed.

## 4 Monitor Synthesis and FPGA Implementation

The observation specification is compiled into tracepoint declarations as already sketched in the previous section. Unique IDs are assigned to every tracepoint. The first stage of the trace evaluation is a filter that creates the logical streams based on the tracepoint IDs attached to the events generated by the trace reconstruction unit. As depicted in Fig. 2 on page 5 in order to monitor a certain property the generated tracepoints for that property must be configured in the reconstruction engine on the FPGA using the PC interface.

In our setup the FPGA fulfills three major functions. First, it realizes the reconstruction explained in Sect. 2. Second, it implements the monitor system that evaluates the reconstructed trace stream as described in the next sections. Third, it provides a softcore processor as a communication interface to the host system for configuration and monitor evaluation.

### 4.1 Merging Data Flow Graphs

Each TeSSLa specification (monitor) produces a new control and dataflow graph (CDFG) that can be transformed into a datapath (DP), i.e. the hardware implementation that executes the operations given by its CDFG on the FPGA. To be able to check all specifications in parallel one would assemble all specified monitors into a directly synthesized monitor system consisting of different DPs, one for each specification. This approach has a major drawback: As soon as only one monitor specification changes the whole system has to be resynthesized. Long FPGA-synthesis time, however, would render the interactive RETOM workflow impossible in which TeSSLa specifications are adapted frequently. To overcome this problem, we follow [13] by merging several CDFGs into one super CDFG with reconfiguration capabilities. Now the monitoring system consists of multiple instances of the same reconfigurable DP, which can implement at least all of the previously specified monitors. This is even more flexible: It is no longer necessary to know how many monitors of a certain type are required in the monitor system as now all DPs can be reconfigured to implement the desired monitor.

Consider the two CDFGs  $CDFG_A$  and  $CDFG_B$  given in Fig. 3a and b. We want to merge those CDFGs into a new CDFG that can implement both of them. It can be seen that the CDFGs contain identical operations among each other. These operations can be shared instead of adding every operation from both graphs into the new one. Finding a preferably large amount of sharable operations is essential for merging two CDFGs. A higher amount of shared operations reduces the resulting CDFG size and thereby, reduces the resulting hardware resources required for the DP.

Therefore, we have to create a matching in which every operation of  $CDFG_A$  matches to either exactly one or no other node in  $CDFG_B$ . We use a generated compatibility graph (CG) as described in a formal way by Moreano et al. in [21]. Here, compatible matches are represented as edges between them. We search for a preferably large fully connected subgraph in the CG, also known as a maximal clique. This clique only contains matchings that can be applied simultaneously

and do not conflict. The resulting merged CDFG (CDFG<sub>M</sub>) is given in Fig. 3c. Operations that are used by both input CDFGs are filled.

### 4.2 Implementing Datapaths

A CDFG can be translated into a hardware description language – more precisely, a Verilog module to implement the configurable datapath. The CDFG has to be preprocessed, as there are some premises. In Fig. 3c it is clear, that the output of the CDFG can only have one input, either the “less than” or the finite state machine (FSM). Therefore, multiplexers are inserted at every operand input that has more than one predecessor. These allow later configuration at runtime to select the desired functionality. To further increase the degree of freedom, constants are never hardcoded into the module. As they change most often, they are replaced by configurable registers, so that their value can be changed quickly during runtime. The FSM is implemented as a microprogrammable state machine whose behavior only depends on the context of a memory. This context can be exchanged during runtime as well and thereby, offers a huge amount of flexibility.

The resulting DP copes without any kind of control logic. It works like a pipeline that can accept new data at its input in every clock cycle. Hence, the amount of time to calculate a result is constant and determined by the number of pipeline stages in the DP. At last, a configuration interface that connects all configurable elements is added.

### 4.3 Programming Monitors

After loading the monitor system onto the FPGA, the context for programming one of the monitors has to be created. During the preprocessing phase for

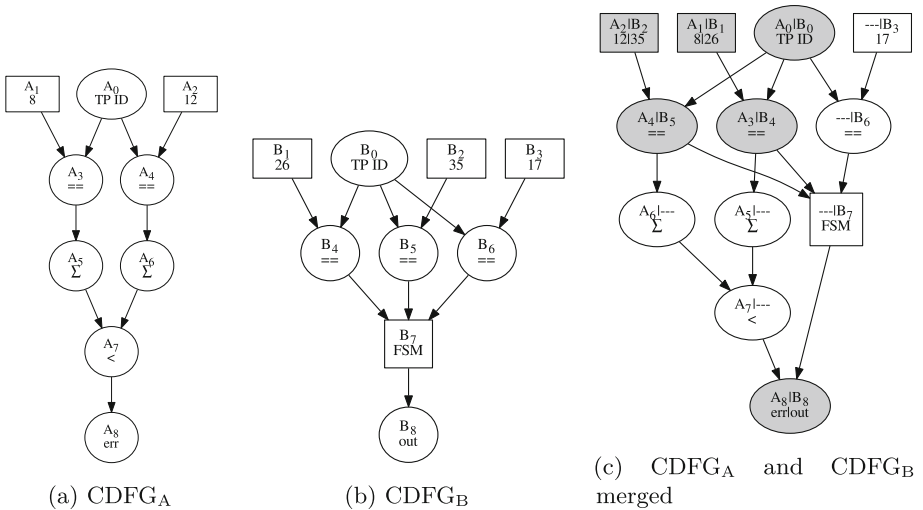
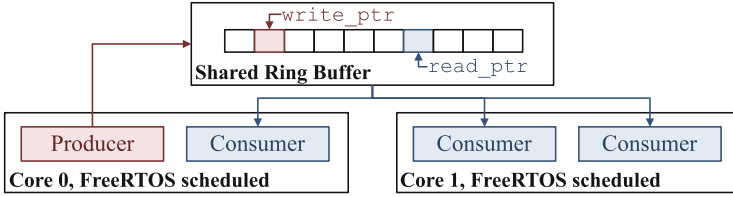


Fig. 3. Example of two CDFGs (a, b) merged into one (c)



**Fig. 4.** Overview of the ring buffer scenario. The producer and consumer threads are distributed over two cores. On core 0 the producer and one consumer is located, on core 1 two consumers are located. The ring buffer is located in a memory section shared between the two cores.

generating the Verilog module, additional information about configurable operations is stored. From that information and the CDFG to program, it can be calculated which operation is executed on which resource on the DP. Another matching is constructed for the edges. The input CDFG does not necessarily need to be a CDFG of the merging set. As shown in [13] disjoint problems can be matched on already synthesized datapaths when the required resources are available.

As each edge match automatically implies two node matches, it is sufficient to find a complete matching for the edges. The matching is said to be complete when every edge in the input CDFG is matched to an edge in the implemented DP. It is automatically constrained by the node matchings. When an edge match matches operation  $A_i$  onto resource  $R_j$ , no other operation may be matched on this resource. If one complete matching is found, the search can be stopped, as all complete solutions are of equivalent quality. Neither resources nor processing time can be reduced at this point as the DPs are already synthesized.

From a complete matching the context for the elements can be extracted: a register's value is then determined by the constant that matches to it. Multiplexers use the incoming edge to determine if their control signal must be 0 or 1. The microprogram for the FSM can be generated from the states, transitions, and output values of the FSM that was created by the TeSSLa compiler for a monitor function. Programming a DP with this context turns it into an active monitor.

## 5 Case Study

We have implemented a multi-core program to show the feasibility and flexibility of our RETOM approach. We used a dual-core ARM Cortex-A9 processor with a clock frequency of 866 MHz embedded in a Zynq-7000 SoC that provides us easy access to the processor's trace port. The trace reconstruction and monitoring took place on separate FPGAs with clock frequencies of about 200 MHz.

Our case study is a concurrent producer/consumer setting written in C. The architecture can be seen in Fig. 4. The C-file *core0.c* runs on core 0 containing the producer and one consumer as well as a start and stop mechanism for the

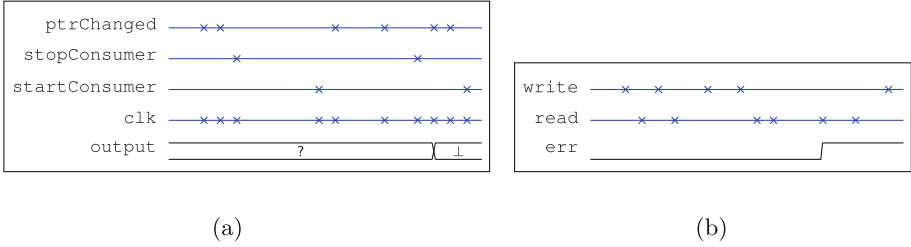


Fig. 5. Evaluation of the TeSSLa streams on an example run.

consumers on both cores. The C-file *core1.c* runs on core 1 containing two identical consumer threads. We use the FreeRTOS scheduler independently on both physical cores to run multiple threads per core. The producer writes elements into a ring buffer and the three consumers read these elements from the buffer. After an element is read, the read pointer (*read\_ptr*) is moved to the next element by the consumer that reads it. Each time the producer writes an element to the buffer it increments the write pointer (*write\_ptr*).

We introduced a bug in *core1.c* such that the section where the ring buffer is read and the read pointer is moved is no longer thread exclusive. This leads to a data race which we want to detect using RETOM.

**Property (a).** We want to check if the start and stop mechanism for the consumers works. Therefore, we use TeSSLa to specify a monitor which checks that when all consumers are stopped, the read pointer must not be changed anymore until they are started again:

```

define ptrChanged := merge(codeLine("core0.c:27"),
                             codeLine("core1.c:27"))
define stop := functionCalls("core0.c:stopConsumers")
define start := functionCalls("core0.c:startConsumers")
define clk := merge(stop, ptrChanged, start)
define output :=
    monitor("always (stop implies
            (not (ptrChanged) until start))",
            step := clk)
out output
    
```

The two *codeLine* streams reference the code line in which the read pointer is moved on. *stop* and *start* reference a call to the respective function. All these streams have an event whenever the piece of code referenced by them is executed. The clock stream *clk* is defined to be used to step the monitor. An example run for this property can be found in Fig. 5a. Here, the LTL<sub>3</sub> semantics is used and therefore the monitor outputs ? as long as the property can still be fulfilled and violated, while ⊥ or ⊤ occurs as soon as the property is certainly violated or fulfilled, respectively.

**Property (b).** We want to check multi-processing of elements in the ring buffer: If the consumers process more elements than the producer writes we spot a bug. Hence, we compare the number of observed read and write accesses to the buffer:

```
define write := codeLine("core0.c:37")
define read0 := codeLine("core0.c:24")
define read1 := codeLine("core1.c:24")
define err := eventCount(read0) + eventCount(read1)
           > eventCount(write)
out err
```

The input streams contain an event if an element is written or read, respectively. Then *err* is defined by counting the number of written elements and the number of read elements. If more elements are read from than written to the buffer, some elements have been processed twice. The diagram in Fig. 5b shows an example evaluation of the streams.

For both properties (a) and (b) it is necessary to observe the system for an arbitrary amount of time because errors can occur randomly due to scheduling and timing differences on the cores. This means that the time when the error may occur also varies per execution. Because of that it is not feasible to just log data and evaluate that to find a possible bug. Also, a non-intrusive observation method is crucial for this property, because intrusiveness would change the timing of the code execution.

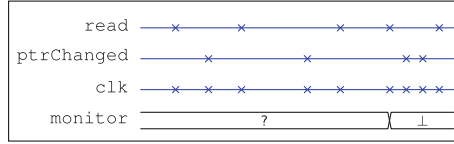
If we synthesize the monitors for property (a) and (b) on the FPGA, connect the FPGA to the processor running the ring buffer example and execute the program, we detect a violation of the property (b). The time needed to detect this violation differs for every execution due to scheduling reasons. Property (a) always produces ? which means that no error occurred yet. Property (b) states that some elements in the ring buffer are processed multiple times. To investigate this issue further we write another property to check if a data race already happens locally on one of the cores.

**Property (c).** We observe the accesses to the memory and to the read pointer to see if, after one consumer thread accessed the memory, another one accesses the memory before the read pointer is moved. This property can be expressed as follows in TeSSLa:

```
out doubleRead(
  read := codeLine("core0:24"),
  ptrChanged := codeLine("core0:27"))
```

where `doubleRead` is a macro defined as follows:

```
macro doubleRead(read, ptrChanged) := {
  define clk := merge(read, ptrChanged)
  monitor("always(read implies
    next(not(read) until ptrChanged)"),
    step := clk)
}
```



**Fig. 6.** Evaluation of the TeSSLa streams of Property (c) on an example run.

Using this macro the property can be expressed for core 1 by changing *core0* to *core1*. All macros are fully expanded by the TeSSLa compiler before the monitor synthesis. The diagram in Fig. 6 shows an example run.

With RETOM we can now adjust the monitor system on the FPGA to check property (c) without the need to re-synthesize the FPGA.

As shown in Fig. 4, there is only one consumer thread on core 0, so on that core we only check if this consumer does not read an element twice. But on core 1, we found the data race because one of the consumer threads sometimes read the ring buffer before the other one increments the read pointer.

In the properties (a), (b) and (c) tracepoints happen rather seldomly during the program execution with an average event rate of about 1 kHz, because the main filtering happens already in the tracepoint matching during the trace reconstruction. Nevertheless with the RETOM approach one can also monitor high-frequency events like quantitative analysis on how many certain CPU instructions are performed. The synthesized monitoring pipeline on the FPGA can process a new external event with every clock cycle. With a clock frequency of 200 MHz the monitors are capable of processing up to 200 million events per second. For the properties described above we needed 196 lookup tables (LUTs) and 414 flip-flops (FFs). As a comparison, the Virtex 7 xc7vx485t that we used has 303600 LUTs and 607200 FFs available. Hence, one could synthesize on one FPGA about 1400 monitors of the size we used in this case study, all checking possibly different properties in parallel.

## 6 Conclusion

In this paper we proposed non-intrusive online monitoring for multi-core systems. Our approach RETOM utilises the embedded trace unit (ETU) of the system under test, which allows non-intrusive observation not only for collaborative software with debug statements, but for arbitrary software. With online monitoring one can react almost immediately to events of interest without having any limits regarding the execution length of the system under test. Using the stream-based specification language TeSSLa we can express correctness properties as well as statistics and numeric metrics, both with support for real-time operations. The control and data flow graph (CDFG) created from a TeSSLa specification contains no cyclic dependencies which simplifies its realization on FPGA hardware. By using merged CDFGs on the FPGA we can change the currently evaluated TeSSLa specification without the need of re-synthesizing the FPGA. This rapid adjustment is suitable for a debugging workflow where the

user incrementally updates the specification based on the last monitoring output in order to understand the system under test. We have shown the feasibility of RETOM in a case study involving three properties spotting a race condition in a multi-core system by detecting a bug due to a long time observation of the system. With the possibility to interactively adjust the specification, one can have iterative debugging sessions in order to find more specific causes based on previous results. The next step to show the feasibility in a broader scale would be an industry case study which we are planning for the future.

**Acknowledgements.** We thank Jannis Harder and Sebastian Hungerecker for their work on TeSSLa, its compiler and the case study.

## References

1. ARM Limited: ARM IHI 0035B: CoreSight Program Flow Trace: PFTv1.0 and PFTv1.1 - Architecture Specification, Issue B, March 2011
2. ARM Limited: ARM IHI 0029B: CoreSight™ Architecture Specification v2.0, Issue D (2013)
3. ARM Limited: DS-5 ARM DSTREAM User Guide Version 5.27 (2017)
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. *J. ACM* **49**(2), 172–206 (2002)
5. Backasch, R., Hochberger, C., Weiss, A., Leucker, M., Lasslop, R.: Runtime verification for multicore SoC with high-quality trace data. *ACM Trans. Des. Autom. Electr. Syst.* **18**(2), 18:1–18:26 (2013)
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* **20**(4), 14:1–14:64 (2011)
7. Bauer, A., Leucker, M., Streit, J.: SALT—structured assertion language for temporal logic. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 757–775. Springer, Heidelberg (2006). [https://doi.org/10.1007/11901433\\_41](https://doi.org/10.1007/11901433_41)
8. D’Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: *TIME*, pp. 166–174. IEEE (2005)
9. Dreyer, B., Hochberger, C., Lange, A., Wegener, S., Weiss, A.: Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In: *WCET. OASICS*, vol. 55, pp. 4:1–4:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
10. Dreyer, B., Hochberger, C., Wegener, S., Weiss, A.: Precise continuous non-intrusive measurement-based execution time estimation. In: *WCET. OASICS*, vol. 47, pp. 45–54. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
11. Eliot, C., Hudak, P.: Functional reactive animation. In: *Proceedings of ICFP 2007*, pp. 163–173. ACM (1997)
12. Freescale Semiconductor, Inc.: P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual, Rev. F (2012)
13. Gottschling, P., Hochberger, C.: ReEP: a toolset for generation and programming of reconfigurable datapaths for event processing. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 141–149 (2017)
14. Intel Corporation: Intel(R) 64 and IA-32 Architectures Software Developer’s Manual (2016)



15. Jakšić, S., Bartocci, E., Grosu, R., Ničković, D.: Quantitative monitoring of STL with edit distance. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 201–218. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-46982-9\\_13](https://doi.org/10.1007/978-3-319-46982-9_13)
16. Leucker, M.: Teaching runtime verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 34–48. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-29860-8\\_4](https://doi.org/10.1007/978-3-642-29860-8_4)
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. *J. Log. Algebraic Program.* **78**(5), 293–303 (2009)
18. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams (2017). unpublished
19. Lu, H., Forin, A.: Automatic processor customization for zero-overhead online software verification. *IEEE Trans. VLSI Syst.* **16**(10), 1346–1357 (2008)
20. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30206-3\\_12](https://doi.org/10.1007/978-3-540-30206-3_12)
21. Moreano, N., Borin, E., de Souza, C., Araujo, G.: Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **24**(7), 969–980 (2005)
22. Moreno, C., Fischmeister, S.: Non-intrusive runtime monitoring through power consumption: a signals and system analysis approach to reconstruct the trace. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 268–284. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-46982-9\\_17](https://doi.org/10.1007/978-3-319-46982-9_17)
23. Nutt, G.J.: Tutorial: computer system monitors. *SIGMETRICS Perform. Eval. Rev.* **5**(1), 41–51 (1976)
24. Reinbacher, T., Függer, M., Brauer, J.: Runtime verification of embedded real-time systems. *Form. Methods Syst. Des.* **44**(3), 203–239 (2014)
25. Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: this time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-89247-2\\_4](https://doi.org/10.1007/978-3-540-89247-2_4)
26. Shobaki, M.E., Lindh, L.: A hardware and software monitor for high-level system-on-chip verification. In: ISQED, pp. 56–61. IEEE Computer Society (2001)
27. Solet, D., Béchenec, J., Briday, M., Faucou, S., Pillement, S.: Hardware runtime verification of embedded software in SoPC. In: SIES, pp. 171–176. IEEE (2016)
28. Stollon, N.: On-Chip Instrumentation: Design and Debug for Systems on Chip, 1st edn. Springer, London (2010). <https://doi.org/10.1007/978-1-4419-7563-8>
29. Tsai, J.J.P., Fang, K., Chen, H., Bi, Y.: A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. Softw. Eng.* **16**(8), 897–916 (1990)
30. Weiss, A., Lange, A.: Trace-data processing and profiling device. EP Patent EP 2873983 A1, May 2015
31. Weiss, A., Lange, A.: Trace-data processing and profiling device. US Patent 9286186 B2, March 2016

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

