# A Distributed Hardware Monitoring System for Runtime Verification on Multi-Tile MPSoCs

MARCEL METTLER, DANIEL MUELLER-GRITSCHNEDER, and ULF SCHLICHTMANN,
Chair of EDA, Technical University of Munich

Exhaustive verification techniques do not scale with the complexity of today's multi-tile Multi-processor Systems-on-chip (MPSoCs). Hence, runtime verification (RV) has emerged as a complementary method, which verifies the correct behavior of applications executed on the MPSoC during runtime.

In this article, we propose a decentralized monitoring architecture for large-scale multi-tile MPSoCs. In order to minimize performance and power overhead for RV, we propose a lightweight and non-intrusive hardware solution. It features a new specialized tracing interconnect that distributes and sorts detected events according to their timestamps. Each tile monitor has a consistent view on a globally sorted trace of events on which the behavior of the target application can be verified using logical and timing requirements. Furthermore, we propose an integer linear programming-based algorithm for the assignment of requirements to monitors to exploit the local resources best. The monitoring architecture is demonstrated for a four-tiled MPSoC with 20 cores implemented on a Virtex-7 field-programmable gate array (FPGA).

CCS Concepts: • **Hardware** → **Safety critical systems**; • **Computer systems organization** → *Embedded hardware*; • **Theory of computation** → *Modal and temporal logics*; • **Networks** → *Network on chip*;

Additional Key Words and Phrases: Runtime verification, tracing, networks-on-chip, MPSoCs, LTL

**ACM Reference format:**
Marcel Mettler, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. 2020. A Distributed Hardware Monitoring System for Runtime Verification on Multi-Tile MPSoCs. *ACM Trans. Archit. Code Optim.* 18, 1, Article 8 (December 2020), 25 pages.
https://doi.org/10.1145/3430699

## 1 INTRODUCTION

Application fields such as robotics, telecommunication, and autonomous driving have an ever increasing demand for more computing power. In order to meet this rising demand, tile-based architectures are used to scale up the number of cores on a single chip, known as multi-tile Multi-processor Systems-on-Chip (MPSoCs). Due to the complexity of these MPSoCs, traditional verification methods comprising of theorem proving, model checking, and testing reach their limits [14, 18, 25]. In order to address this problem, additional methods are deployed to monitor system specifications at runtime. Those techniques are studied in the field of *Runtime Verification* (RV).

Partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project Number 146371743—TRR 89 Invasive Computing.
Authors' addresses: M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann, Chair of EDA, Technical University of Munich, Arcisstr. 21, Munich, Germany, 80333; emails: {marcel.mettler, daniel.mueller, ulf.schlichtmann}@tum.de.

1544-3566/2020/12-ART8
https://doi.org/10.1145/3430699

RV is a lightweight verification technique in which the system under verification is *instrumented* to extract its current status based on events. The trace of events is then analyzed by a *runtime monitor*, which infers a positive verdict if the monitored requirement is fulfilled and a negative verdict if not [10].

RV is used pre-deployment for verification, debugging, and testing, but it can also be used post-deployment for fault detection, identification, and recovery (FDIR) [18]. For example, in the automotive industry, the development partnership AUTomotive Open System ARchitecture (AUTOSAR) [1] proposes safety mechanisms for embedded software including logical and timing supervision of the target application. *Logical supervision* is used to verify the control flow of an application in order to detect synchronization errors between software elements. In contrast, *timing supervision* verifies the timing between code sections or threads in order to detect deadlocks, livelocks, and the incorrect allocation of execution times. These safety mechanisms are based on a post-deployment use of runtime monitors and RV methods. Of course, the same runtime monitors can also be used pre-deployment to detect common concurrency bugs and analyze the temporal behavior of an application. Hence, the runtime monitors, which implement both safety mechanisms, form a powerful tool to improve the system. Yet, when applications are executed in a distributed fashion on large-scale, multi-tile MPSoCs, additional challenges arise for using RV methods: firstly, every processor generates a partial event trace of the application such that no global trace is available per se. Additionally, when monitors are distributed over the system, a second challenge is to assign the runtime requirements to the local monitors. For this purpose, we propose a decentralized monitoring architecture to address these challenges. It is—to the best of our knowledge—the first hardware-based RV approach that is capable of verifying runtime requirements such as logical and timing supervision for applications executed in a distributed fashion on large-scale, tile-based MPSoCs.

The key concept behind the monitoring architecture is a Network-on-Chip (NoC) tracing interconnect, SortNoC, which is specifically designed to sort partial event traces based on timestamps and to distribute them in the monitoring architecture. Thus, the distributed monitors have a consistent view on a globally sorted trace of events, which enables them to verify requirements on applications, running in a distributed fashion. Furthermore, we formulate the assignment of requirements to monitors as an integer linear programming (ILP) problem. Using an ILP solver, it is possible to maximize the number of simultaneously verifiable requirements by exploiting the available resources best. The key contributions of our work can be summarized as follows:

- A decentralized monitoring architecture for the verification of runtime requirements for applications executed in a distributed fashion on large-scale, tile-based MPSoCs. The design is hardware-based and, hence, non-intrusive as well as suitable for pre- and post-deployment use.
- SortNoC, an NoC-based tracing interconnect for large-scale monitoring architectures. Compared to a regular NoC, its specialized design improves area and latency, and offers broadcasting capabilities.
- The ILP-based algorithm to assign requirements to monitors to efficiently use local monitoring resources.

We evaluate the monitoring architecture in a multi-tile MPSoC with four tiles, having five Leon3 cores each, implemented on a Virtex-7 field-programmable gate array (FPGA). The fully-configured monitoring architecture introduces an overhead of 10.8% in terms of lookup tables (LUTs) and 30.6% in terms of flip-flops (FFs). This is significantly lower than the overhead of other non-intrusive verification approaches like DiaSys [33]. Furthermore, we demonstrate that the proposed tracing NoC scales better for large-scale systems compared to a naive approach using a regular NoC when considering the network latency and area. Finally, we present typical use cases of the monitoring system and show the benefits of the ILP-based requirement assignment algorithm compared to a greedy assignment policy.

The remainder of this article is structured as follows: First, we present the related work in Section 2. Subsequently, Section 3 introduces our monitoring architecture. The implementation of the monitoring system is discussed in Section 4, followed by a requirement assignment algorithm in Section 5. Experimental results are discussed in Section 6, and Section 7 concludes the article.

## 2 RELATED WORK

The implementation of runtime verification methods can be conducted in software (SW) and hardware (HW). SW-based verification approaches execute additional software for the instrumentation and the verification of target applications. Although those approaches benefit from the expressiveness of programming languages, they typically introduce a significant performance overhead [22] and change the temporal behavior of the application [9]. In contrast, HW-based approaches introduce dedicated monitoring hardware, which is less expressive but minimizes or completely avoids performance overheads. In the following, we focus on HW-based approaches because they are better suited for embedded systems.

The first class of HW-based approaches synthesizes formal specifications into a hardware description language: Lu and Forin [20] introduce a property specification language (PSL) to Verilog compiler, which translates assertions into loadable extensions for the extensible Microprocessor without Interlocked Pipelined Stages (eMIPS) architecture. Similarly, Solet et al. [29] synthesize past-time linear temporal logic (ptLTL) specifications on an FPGA fabric, which is integrated alongside a micro-controller. Thus, all instrumented variables must be mapped to FPGA registers, which significantly increases the access latency. They further evaluate this approach for the RV of a real-time operating system (RTOS) kernel [30, 31]. Here, a performance overhead of 16.2–33% was found. Additionally, there exist different tools such as LamaConv [16] and ltl2mon [5] that convert specifications into three-valued LTL (LTL3) monitors, which can directly be implemented as a finite-state machine (FSM). In general, the synthesized monitors are either implemented on an FPGA, resulting in high–access latencies, or directly in HW, restricting the configurability of the system.

The second class pre-processes monitoring data on-chip and forwards it to a host machine for further post-processing: Hochberger and Weiss present a hidden in-circuit emulator (HidICE [15]) that replicates the internal states of the target system using the read data from peripheral components. The replicated internal states of the emulator can be accessed by a host machine to gain insights into the system behavior. Decker et al. [7] developed an online analysis platform, which processes trace data of an MPSoC in real-time. After execution, the collected data is available for further offline post-processing. Similarly, Wagner et al. [33] present a **diagnosis system, DiaSys,** for MPSoCs that executes a diagnosis application to process events at runtime according to a dataflow application. The result is then sent to a host machine to enable insights in the program execution to reveal concurrency bugs. While these approaches are especially useful for testing and debugging pre-deployment, in-situ approaches are required to perform RV post-deployment.

The third class uses reconfigurable monitoring hardware: Seo and Lysecky [28] introduce a non-intrusive runtime monitoring methodology (NIRM) for the verification of control flow and timing requirements on single-core architectures. Mettler et al. [21] extend this approach for inter- and intra-thread requirements on embedded MPSoCs using a hierarchical monitoring architecture. The intra-thread requirements are verified on local monitors and all inter-thread requirements are verified on a single global monitor limiting the scalability of the architecture. In contrast, Nassar et al. [22] introduce a nonuniform verification architecture (NVUA) for the verification of a parameterized finite-state automaton. To achieve this goal, it uses a directed acyclic graph (DAG) to organize a population of monitors. In order to apply events on the distributed data structure, a directory-based coherence protocol is implemented. Although this enables to maintain a coherent view on the checker population, to our understanding, there are no measures taken to apply events in their order of detection. A generalized tracing methodology is introduced by Seo and Kurdahi [27]. The methodology consists of a compatibility layer for tracing interfaces of different single-core architectures, a filter layer for the detection of events, and a verification layer that checks the requirements using Micron's automata processor (AP) [8]. Alternatively, Reinbacher et al. [26] introduce an HW accelerator for the verification of ptLTL specifications. The accelerator consists of a parallel monitoring architecture where each monitor checks an item of the temporal specification using atomic checkers.

Besides the verification features of the RV architecture, its compatibility with the development workflow of safety-critical systems is an important feature as well. Safety critical systems are designed according to international standards, such as the functional safety standard 26262 of the International Organization for Standardization (ISO). for the automotive industry. The standard includes functional safety assessment and audit processes that lead to functional safety requirements. Heffernen et al. [13] use the ISO 26262 to define functional safety requirements as monitoring properties in LTL for an automotive gearbox control system. Using a runtime monitor, they are able to verify the safety requirements throughout the lifetime of the system but provide no implementation suitable for large-scale systems. As more and more electrical control units (ECUs) are integrated into one single large-scale central system, monitoring architectures must scale accordingly to verify functional safety properties. While DiaSys and NUVA are the only scalable approaches, neither of them exploits the full potential of RV, suitable for pre- and post-deployment use cases. DiaSys is especially designed for a pre-deployment use only. It moves the trace analysis only partially on-chip, which makes a post-deployment use impossible. While NUVA can also be used post-deployment, it supports the verification of a single parameterized finite-state automaton only. Especially on larger-scale systems, this is not sufficient. Thus, we integrate multiple runtime monitors in each tile in order to obtain a scalable architecture. Furthermore, the two approaches support timing supervision neither for pre-deployment nor for post-deployment use cases. This is a major shortcoming as RV is especially used in real-time systems where timing requirements are essential. In a decentralized system, the events, which are detected on different locations, must be sorted based on their detection order to reliably detect concurrency bugs. Hence, the RV architecture of a decentralized system must entail a sorting mechanism. Such a mechanism is neither implemented in DiaSys nor in NUVA. We address this shortcoming by SortNoC, a scalable tracing interconnect that inherently sorts detected events based on their timestamp.

## 3 DECENTRALIZED HARDWARE-BASED MONITORING ARCHITECTURE FOR MULTI-TILE MPSOCS

In this section, we give an overview of the monitoring system that overcomes the scalability shortcomings of today's hardware approaches. First, we present our RV methodology, followed by a formal definition of the supported runtime requirements, and a discussion on the benefits and limitations of this monitoring approach.

### 3.1 Architectural Design

As already stated, RV is a lightweight verification technique that verifies a particular system behavior against a set of requirements. Compared to traditional verification approaches such as model checking and testing, RV approaches carefully balance the coverage of the verification strategy against the computational effort. This is possible by instrumenting the system under scrutiny by

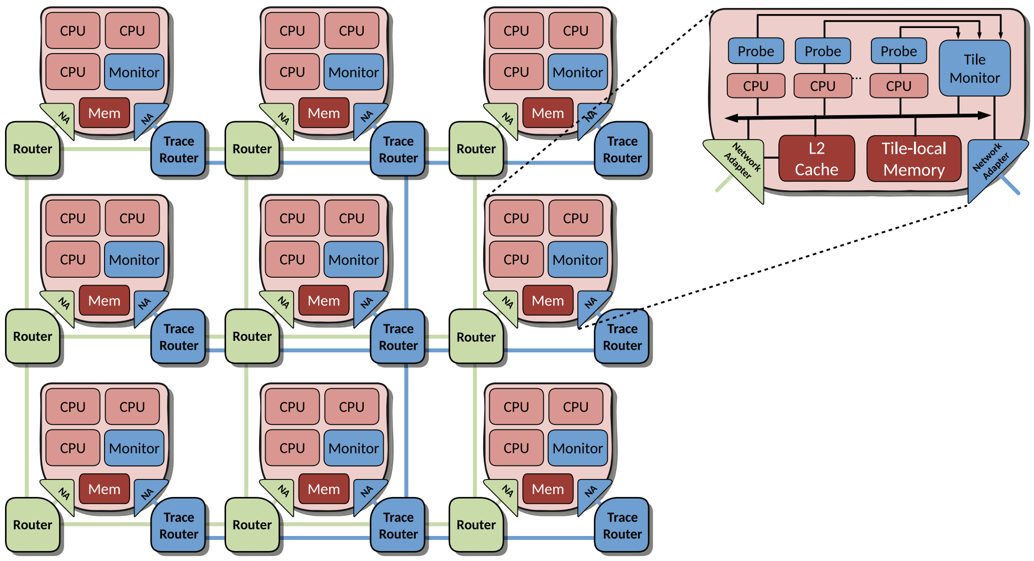


Fig. 1. The decentralized monitoring architecture integrated in a tiled MPSoC.

a set of probes that extract an event trace. The event trace is then forwarded to a runtime monitor that checks formal requirements on the trace and returns a verdict.

In large-scale systems, the number of supported requirements and thereby the number of monitors must scale with the size of the system. Francalanza et al. [11] survey different monitor organizations for decentralized systems. Here, the monitor organization is mostly determined by the verification strategy of requirements that are global in nature. In this scenario, probes on different tiles detect events that are required for the verification of the same requirement. As a result, either the local event traces must be aggregated, or the monitors must communicate with each other.

For large-scale MPSoCs, requirements must be tested for applications running in a distributed fashion across several tiles of multi-processor subsystems. Hence, we face the challenge to verify requirements that are global in nature. With the development of the tracing interconnect SortNoC, we decided to aggregate the local event traces, which is simpler to realize (i.e., expected to consume fewer resources) and enables an arbitrary assignment of requirements to monitors. This results in a decentralized monitoring setup, where a set of probes is attached to the processors in the system. Each probe generates a local event trace. The event trace is then aggregated and distributed to all monitors in the system. Our specific setup is illustrated in Figure 1. Here, we instrumented the instruction trace of each central processing unit (CPU) by a probe. The probes of each tile send the probe-local event traces to the respective tile monitors, where the probe-local event traces are aggregated to tile-local event traces and injected into the SortNoC. Finally, the SortNoC aggregates the tile-local event traces to a single totally ordered event trace, which is distributed to all monitors. In the monitors, the specifications are then verified based on a globally sorted event trace.

## 3.2 Supported Runtime Requirements

Another key feature of the monitoring architecture is the expressiveness of its supported requirements. In HW, one needs to find a good tradeoff between the expressiveness and the HW overhead. Inspired by AUTOSAR [1], we decided to support logical and timing requirements as well as range checks and checks on power corridors. They are already expressive enough to formulate complex functional safety requirements but come at the same time with a reasonable HW overhead to provide a sufficient number of monitors in the architecture. The runtime requirements are formally defined together with the event trace in the following.

We define all runtime requirements on a set of events $E$, where each event $e$ has an event ID $i_e$. Due to the limited hardware resources of the monitors, the bit width of an event ID $W_I$ is restricted by the runtime monitors. To ensure that the number of supported events still scales with the size of the system, this notation is extended by a cluster ID $i_c$, which identifies the monitoring cluster, in which an event is used. A monitoring cluster is a consolidation of tile monitors that verify requirements based on the same set of events. Thus, it is possible to use the same event IDs for different monitoring clusters, without any interference. As a result, the maximal number of supported events in the system is given by Equation (1), where $N_{TM}$ corresponds to the number of monitors in the system.

$$N_E = N_{TM} \cdot 2^{W_I} \tag{1}$$

In this case, each monitor forms its own cluster and is assigned to a unique cluster ID $i_c$. If all monitors are assigned to the same cluster, the number of supported events is limited by $N_E = 2^{W_I}$. Furthermore, each event is assigned to a timestamp $t$, which allows SortNoC to sort the events by their detection order. Together, the event ID, the timestamp, and the cluster ID define a trace element $s_i \in S$ by a 3-tuple $(i_e, t, i_c)$. In the proposed monitoring architecture, we support logical, temporal, as well as result range and power corridor requirements, which are discussed in the following.

*3.2.1 Logical Requirements.* Logical requirements verify the control flow of applications based on detected events and thus, reveal synchronization errors between software elements. We define a logical requirement $l \in L$ by a finite deterministic automaton using a 6-tuple $(\Upsilon_l, E_l, \delta_l, \upsilon_0, \upsilon_t, \upsilon_f)$:

— $\Upsilon_l$ is a finite set of states.
— $E_l$ is a finite set of accepted events, with $E_l \subseteq E$.
— $\delta_l$ is a transition function $\delta_l : \Upsilon_l \times E_l \to V_l$.
— $\upsilon_0$ is the initial state, with $\upsilon_0 \in \Upsilon_l$.
— $\upsilon_t$ is the acceptance state, with $\upsilon_t \in \Upsilon_l$.
— $\upsilon_f$ is the failure state, with $\upsilon_f \in \Upsilon_l$.

The automaton can either be constructed manually or synthesized from LTL requirements using the open source tool LamaConv [16]. Furthermore, it follows the notation of a 3-valued semantic and issues a verdict $z_F$ based on its current state $\upsilon_c$.

$$z_F = \begin{cases} \top & \upsilon_c = \upsilon_t \\ \bot & \upsilon_c = \upsilon_f \\ ? & \text{else} \end{cases} \tag{2}$$

An example of a logical requirement is given in Equation (3) as the LTL formula and in Figure 2 as an automaton synthesized out of the LTL formula.

$$e_{exit} \vee \left(e_{opened} \rightarrow (\neg e_{exit} \, \mathcal{U} \, e_{closed})\right) \tag{3}$$

Here, $e_{open}$ always must be followed by $e_{closed}$ till $e_{exit}$ was detected. As long as neither $\upsilon_t$ nor $\upsilon_f$ is reached, neither a positive nor a negative verdict can be issued because, at this time, the monitor does not know whether the requirement is going to be fulfilled or violated. Therefore, the monitor issues ? until the requirement is accepted or violated. Such a requirement can be used to verify at runtime that critical tasks close their resources after use.

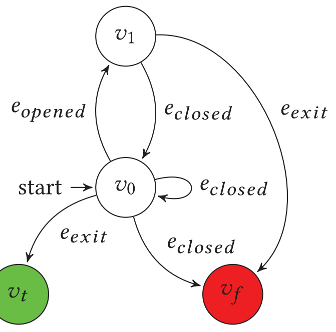


Fig. 2. An example of an automata-based requirement.

*3.2.2 Timing Requirements.* Timing requirements verify the real-time behavior of an application based on the latency between two events and reveal deadline misses, insufficient task progression, deadlocks, as well as livelocks. A timing requirement $f \in F$ is defined by a 4-tuple $(e_{start}, e_{stop}, T_{min}, T_{max})$, where $e_{start}$ is the event that starts the timer, $e_{stop}$ is the event that stops the timer, and the interval $[T_{min}, T_{max}]$ corresponds to the defined latency range. Similar to the automata-based requirements, timing requirements follow a 3-valued semantic that issues a verdict $z_F$ based on the measured latency $t_f$.

$$z_F = \begin{cases} \top & t_f \in [T_{min}, T_{max}] \\ \bot & t_f \notin [T_{min}, T_{max}] \\ ? & \text{measurement not complete} \end{cases} \tag{4}$$

A simple timing requirement used in small embedded systems is set via windowed watchdog timers. The software has to regularly "pat" the dog and reset the windowed watchdog timer within a timing window, e.g., to check that a task is running within a fixed time period. For complex applications running on a tile-based MPSoC, a complex set of such requirements can be defined to monitor the progress and deadlines of real-time-critical applications.

*3.2.3 Result Range and Power Corridor Requirements.* Traditionally, logical and timing requirements are defined based on events that are fired on specific checkpoints, i.e., program counter addresses. We generalize this approach by further event conditions on which the logical and timing requirements can be defined, which are presented in the following.

A *result range* requirement defines a valid range for the results of arithmetic operations. Formally, this requirement is defined by 3-tuple $(pc, dtype, [result_{min}, result_{max}])$, where $pc$ corresponds to the program counter address of the arithmetic operation, $dtype$ to the dataype of the result and $[result_{min}, result_{max}]$ to the valid result range. Whenever a result outside of this range appears, an event is generated. Thus, it is possible to not only verify the control flow of an application but also the application data.

A *power corridor* requirement defines a valid power corridor in which the power consumption of a core must lay. Formally, this specification is defined by a power interval $[P_{min}, P_{max}]$. Thus, it is possible to detect spikes in the power consumption, which might disturb nearby analog circuits within critical sections of the application.

## 3.3 Benefits and Limitations

In this section, we discuss the consequence of the design decisions we presented in Section 3.1 and 3.2, which lead to the benefits but also to some limitations of the monitoring architecture.

Using a hardware-based implementation of the monitoring system comes with major benefits in terms of non-intrusiveness and low fault detection latencies. There is no performance penalty on the application performance except at the start, when the monitoring system is configured with the application-specific requirements. Due to a pure hardware implementation, the number and the type of the integrated monitors are already defined at the design time of the system and not necessarily at the design time of the application. Hence, it must be carefully decided upon system design time which probes have to be integrated into the system. The proposed architecture offers here the benefit that also further event detectors, in addition to those we discuss in Section 4, can easily be integrated into the probes.

Another clear benefit of the proposed architecture is its capability to generate a globally sorted trace of events using SortNoC as the heart of the monitoring architecture. SortNoC enables that distributed monitors can check runtime requirements of applications running in a distributed fashion on complex multi-tiled MPSoCs. SortNoC comes with a throughput limitation in the event trace of one event per cycle per system (EPCPS). Considering a frequency of 1 $GHz$, this corresponds to 1 billion events per second. On a large-scale MPSoC with 100 cores, on average every 100th cycle, an event could be triggered. With an $IPC = 0.7$ (an optimistic value for a Leon3 core), every 70th instruction could trigger an event. This is more than any software implementation could provide under a reasonable performance overhead constraint.

Due to the non-intrusiveness of the design, the architecture is also applicable to perform pre-deployment testing and debugging tasks as well as post-deployment supervision. Nevertheless, it should be noted that the system does not provide all features of well-known tracing and debug approaches such as ARM CoreSight [2]. The implementation of those features are out of the scope of this article. Nevertheless, in practice the traditional tracing features could be integrated into our monitoring methodology to form a single monitoring architecture for all runtime verification tasks performed pre- and post-deployment.

## 4 COMPONENTS OF THE MONITORING ARCHITECTURE

This section presents the implementation details of the proposed monitoring system for tiled MPSoCs. The architecture is composed of three types of components, which are illustrated in blue in Figure 1: a set of probes $P$, a set of tile monitors $TM$, and a set of tracing routers $R$. A *probe* is assigned to each core in a compute tile. It extracts events from the trace data provided by the CPU in order to reduce the data volume. The events are then concatenated with their corresponding cluster ID and the current timestamp to form event trace elements that are forwarded to the network adapter of the *tracing router* via the *tile monitor*. Using a specialized router design, it is possible to distribute the event trace elements to all tiles and sort them according to their timestamps. Using the cluster IDs, the tile monitors filter the globally sorted event trace for trace elements on which their monitors operate. If a monitor returns a negative verdict, an interrupt will be raised and the operating system (OS) can address the detected violation. The following sections give a detailed overview of the individual hardware components.

### 4.1 Probe

In this section, we present the implementation of a probe with a set of event detectors, which are suitable to check a range of typical requirements for embedded applications. Nevertheless, some applications might require additional detectors, which can be added similarly. The architecture of our probe, illustrated in Figure 3, is composed of a timestamp generator, a set of event detectors, an arbitration logic, and a configuration block. The *timestamp generator*, illustrated in purple, is made up of a free-running counter that issues a timestamp $t \in [0, 2^{W_t} - 1]$ with a bit width $W_t$. The bit width $W_t$ is chosen large enough to identify the correct order of event trace elements,

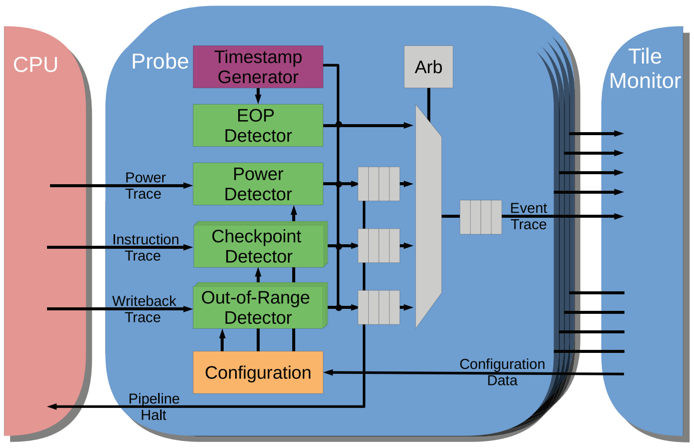


Fig. 3. The architecture of a probe component.

even in highly disordered traces. Furthermore, all timestamp generators need to be synchronized as part of the boot sequence of the system to guarantee a correct ordering in the trace routers. On an FPGA implementation the synchronization can be achieved by the global set reset (GSR) signal, which is already pre-routed in FPGAs. However on an application-specific integrated circuit (ASIC), the synchronization measure should be implemented differently: During the boot sequence of the operation system, an event could be triggered that resets each of the timers. At this point in time, no other event is present in the SortNoC, which is why the latency of the event to each of the tile monitors is known. Given the latency, it is now possible to reset each timestamp counter such that all counters are synchronized at the beginning.

The *detectors*, illustrated in green, analyze the incoming data traces from the CPU in order to reduce the data volume to an amount, which is processable at runtime. Thus, each detector issues an event ID $i_e$ and a corresponding cluster ID $i_c$ when it detects a configured pattern in its input trace. The current design of our probe supports four different detector types, which are introduced in the following:

*EOP Detector.* The end of period (EOP) detector issues an event ID $i_{EOP}$ when the timestamp generator overflows. Thus, the timers in the tile monitor are able to compute the latency between two events using the corresponding timestamps and the number of EOP events. As the EOP event has an exceptional role, no tile monitor can filter it out. Therefore, it is sufficient to implement the EOP detector only in one probe in the complete monitoring architecture.

*Power Detector.* The power detector analyzes the power trace of the assigned CPU and issues an event ID when a power value lies outside of a predefined power corridor. This is especially helpful in large-scale systems where strict thermal requirements need to be fulfilled. Furthermore, it can be used to detect power spikes, which might disturb neighboring analog circuits. In a mixed-signal ASIC with an analog front-end, the sensor data could be disturbed by a power spike. With this detector, the disturbance can be detected and the system can repeat the measurement.

*Checkpoint Detector.* A set of checkpoint detectors is used to monitor the execution progress of the application. Each detector issues an event ID when an instruction at a predefined program counter (PC) address, which corresponds to the checkpoint at a specific program location, is detected in the instruction trace. This enables the monitors to check timing and control-flow requirements of the target application.

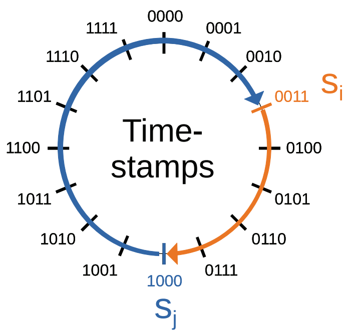


Fig. 4. A graphical representation of all timestamps in a binary number wheel for $W_t = 4$. Marked are the timestamps of two trace elements $s_i$ and $s_j$ whose detection order is to be determined. Here, the orange arrow shows the timestamp pattern if $s_i$ was issued first and the blue arrow shows the timestamp pattern if $s_j$ was issued first. Given a large enough $W_t$, the shorter arrow will always correspond to the actual detection order. As a result, an arbitration schema should prioritize $s_i$ over $s_j$.

*Out-of-Range Detector.* Each out-of-range detector analyzes the writeback trace and issues an event ID in order to check whether the result of a predefined arithmetic operation lies within a configured range. As a result, monitors are enabled to adjust their specifications for different runtime scenarios or to perform sanity checks on arithmetic operations. Currently supported data types are: float, double, signed integer, and unsigned integer.

The *arbitration logic*, illustrated in gray, buffers the generated trace elements $s_i^{(p)} = (i_e, t, v)$ of probe $p \in P$ from all detectors except from the EOP detector because EOP events occur so infrequently that a dedicated buffer is not required. Given the, by design unlikely, case that one of the buffers is full, the pipeline of the CPU is halted in order to prevent a loss of trace elements. After the buffer stage, the trace elements are then combined to a probe-local trace $S^{(p)}$ using a timestamp-based arbitration scheme. The arbitration scheme analyzes the differences between two timestamps $t_i = \Pi_t(s_i)$ and $t_j = \Pi_t(s_j)$ of two trace elements $s_i$ and $s_j$ according to Figure 4. As illustrated in the binary number wheel, there are two possible detection orders for the events, which are illustrated by the orange arrow and by the blue arrow. Assuming a large enough $W_t$, the smaller arrow always corresponds to the correct detection order. Algorithm 1 computes the length of the blue arrow by the difference $d$ between the timestamps $t_i$ and $t_j$. If the length of the arrow is less than then timestamp bound $T_b$, which equals the midpoint of the timestamp interval $[0, 2^{W_t} - 1]$, it also must be smaller than the orange arrow. This is due the fact that the sum of the arrow length is equal to the interval length $2^{W_t}$. As a result, $s_i$ was issued before or simultaneously to $s_j$ if $d$, i.e., the length of the blue arrow, is smaller than or equal to $T_b$; and $s_i$ was issued after $s_j$ if $d$ is larger than $T_b$.

Furthermore, detection patterns and event IDs of the detectors are stored in a *configuration unit*, illustrated in orange, which is configured by the tile monitor. Thus, it's possible to reconfigure the probes according to the needs of the target application.

## 4.2 Tile Monitor

The architecture of a tile monitor is illustrated in Figure 5. It consists of a configuration memory, a network adapter, and a set of monitors (APs and timers). The *configuration memory*, illustrated in orange, is a 36 $Kb$ SRAM, which stores different probe configurations. A request to load a configuration can be issued via the Advanced Peripheral Bus (APB) interface. Thus, the OS needs to access the system bus only once to reconfigure a complete probe if needed. In order to prevent

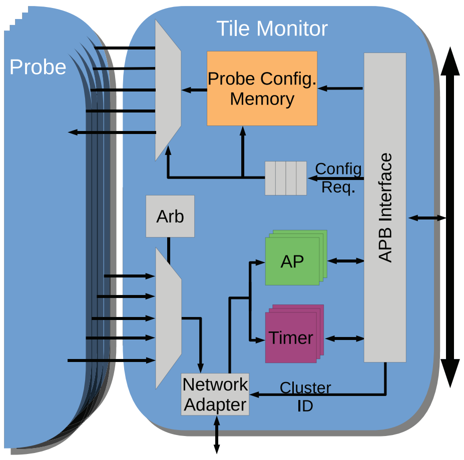


Fig. 5. The architecture of a tile monitor.

**ALGORITHM 1:** Timestamp-based Arbitration

```
1: const. $T_b = 2^{W_t - 1} - 1$
2: function GET_FIRST($s_i, s_j$)
3:     $d \leftarrow \Pi_t(s_i) - \Pi_t(s_j)$
4:     if $d \leq T_b$ then
5:         return $s_i$
6:     else
7:         return $s_j$
8:     end if
9: end function
```

contentions between subsequent requests, each request is stored in a first-in, first-out (FIFO) buffer till all prior requests are processed.

The probe-local event traces $S^{(p)}$ of the connected probes are combined to a monitor-local event trace $S^{(m)}$ of monitor $m \in TM$ using the same arbitration logic as in Section 4.1. As a result, the *network adapter* injects a locally sorted event trace $S^{(m)}$ into the tracing interconnect where the traces of all tile monitors are combined to a single globally sorted event trace $S$. In the network adapter, the cluster ID of each trace element is compared to the cluster ID of the monitor. Given the case that both IDs are equal, the trace element is forwarded to the runtime monitors. Thus, it is possible to create a cluster of monitors that operate on the same events by assigning them the same cluster ID.

The *monitors*, consisting of APs and timers, operate on the filtered event trace to verify the configured runtime requirements. A detailed description of their functionality is given in the following:

*Automata Processor (AP).* Each AP, illustrated in green, is used to verify a logical requirement $l \in L$. We implement each automaton in a small 18 $Kb$ on-chip SRAM block. Here, the memory is addressed using the conjunction of the detected event ID $i_e$ and the ID of the current state $v_c \in \Upsilon_l$ of the automaton. The accessed data corresponds then to the ID of the next state $v_n \in \Upsilon_l$ of the automaton. If this state corresponds to a failure state, the AP will issue a negative verdict and raise an interrupt. The required size of the memory block $M_{AP}$ can be computed using Equation (5),

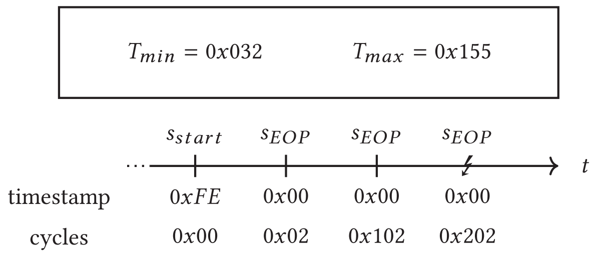


Fig. 6. A timing violation of $T_{max}$ illustrated on a timeline.

where $W_{i_v}$ corresponds to the bit width of a state ID and $W_{i_e}$ to the bit width of an event ID.

$$M_{AP} = W_v \cdot 2^{(W_{i_v} + W_{i_e})} \tag{5}$$

*Timer.* Each timer, illustrated in purple, verifies a timing requirement $f \in F$. After $s_{start}$ has been detected in the event trace, a counter is incremented for every $s_{EOP}$. The counter is stopped as soon as $s_{stop}$ has been detected. Thus, the timers are able to compute the latency between two trace elements using Equation (6), where $N_{EOP}$ corresponds to the number of detected EOP events in the event trace.

$$T = \Pi_t(s_{stop}) - \Pi_t(s_{start}) + N_{EOP} \cdot 2^{W_t} \tag{6}$$

Given the case that $s_{stop}$ is not generated, a violation of the max timing requirement can unambiguously be detected as soon as Equation (7) is violated.

$$T_{max} \geq (N_{EOP} - 1) \cdot 2^{W_t} \tag{7}$$

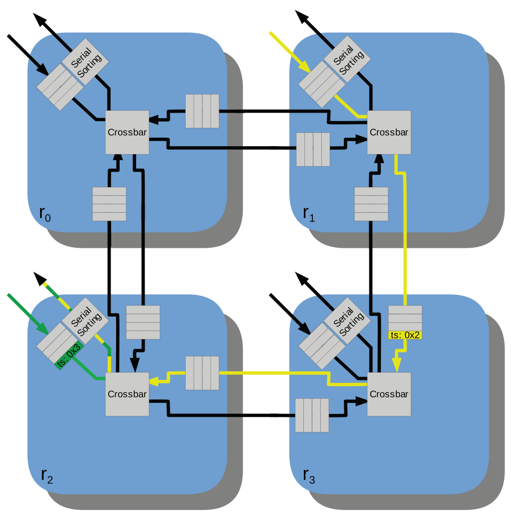
In this condition, $N_{EOP}$ must be decremented by one as the latency between $s_{start}$ and the first $s_{EOP}$ remains unknown. Considering the example in Figure 6, the first $s_{EOP}$ is already detected after $0x02$ cycles. As a result, the violation of $T_{max} = 0x155$ can not be detected before the third $s_{EOP}$. This results in a maximal detection latency of $2^{W_t} - 1$ cycles for a missed $T_{max}$ requirement.

## 4.3 SortNoC—A New Tracing Interconnect Architecture

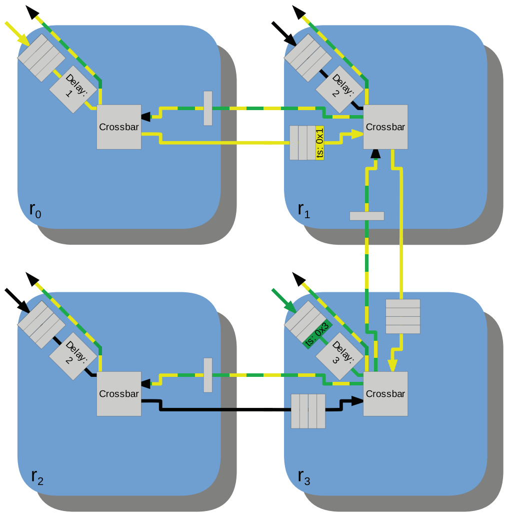
The monitor interconnect is the core of the decentralized monitoring architecture. Here, the trace elements of the individual tile monitors are distributed and sorted to provide a consistent view on the event order. To minimize the hardware overhead of the interconnect, each packet in the architecture consists of a single flit that stores a single trace element. As a result, the interconnect architecture is free of deadlocks by design. Furthermore, the presented NoC architectures use a lightweight router design with no virtual channels and a single stage pipeline to minimize the network latency.

In this section, we discuss two approaches to design such an interconnect architecture: First, a naive approach using a regular NoC architecture with attached sorting IPs is introduced. To overcome the high hardware overhead and the missing broadcasting capabilities, a custom NoC architecture, SortNoC, is introduced thereafter.

*4.3.1 Regular NoC Architecture with Attached Sorting IPs.* A naive approach to distribute and sort trace elements is illustrated in Figure 7(a). It consists of a regular NoC architecture with sorting IPs at the local output port of each router. Furthermore, Figure 7(a) shows two trace elements, $s_g = (i_{e_g}, 3, i_{c_g})$, illustrated in green, and $s_y = (i_{e_y}, 2, i_{c_y})$, illustrated in yellow, which have been injected into the NoC architecture. Even though $s_y$ got injected before $s_g$, $s_g$ will reach the shared destination $r_2$ before $s_y$ because of the lower hop distance. Thus, a sorting IP must be attached at the local output port of each router to sort the trace elements correctly. The serial sorting IP, inspired



(a) A naive interconnect approach for decentralized monitoring architectures.



(b) A specialized NoC architecture to distribute and sort event trace elements.

Fig. 7. Two NoC architectures that sort event trace elements based on their timestamp.

by Lee and Tsai [17], and Perez-Andrade et al. [24], uses a shift register architecture (SRA), which supports the operations shift left, shift right, load, and initialize on which the sorting operations *insert* and *read out* are constructed. An insert operation is executed when a new trace element arrives at the local output port of the router. In this process, the element is compared with all other trace elements in the SRA using Algorithm 1 to determine its correct position in the SRA. The time a trace element must remain in the NoC architecture to ensure a sorted trace is defined by $t_o$. At that point, it is expected that all earlier injected trace elements have arrived at the sorting IP. Thus, we compare the first trace element in the SRA to a reference trace element $s_{ref}$ with a reference timestamp $t_{ref} = t - t_o$ using Algorithm 1, where $t$ corresponds to the currently issued time of the timestamp generator and $t_o$ to a fixed offset between both. If the first trace element in the SRA was found to be issued before or concurrently to the reference element, it is read out.

The sorting IP comes with two major design parameters, which are discussed in the following:

— Timestamp offset $t_o$:
The offset between the issued timestamps $t$ of the timestamp generator and the reference timestamp $t_{ref}$ in the sorting IP must be chosen to the maximal latency $l_{max}$ in the NoC architecture to ensure that no trace element arrives at the sorting IP after a subsequently generated trace element has already been read out. However, $l_{max}$ depends strongly on the traffic pattern and thus on the target application, which is unknown at the design time of the chip. As a result, a conservative offset $t_o$ must be chosen.

— SRA depth:
The depth of the SRA must be chosen according to Equation (8), where $o_k$ corresponds to the trace element rate at the local output port of router $r_k$ and $t_o$ to the timestamp offset.

$$N_{depth} = o_k \cdot t_o \tag{8}$$

Again, both parameters highly depend on the target application, which is unknown at the design time of the monitoring architecture. Thus, also for $N_{DEPTH}$, a conservative value must be chosen.

In summary, the NoC architecture has the drawback that both design parameters must be chosen conservatively, which significantly increases the hardware overhead of the monitoring architecture. Furthermore, it is not possible to broadcast trace elements over the complete NoC architecture. Thus, all requirements defined on a shared event must be assigned to the same tile monitor, limiting the configurability of the monitoring system. In order to address these drawbacks, we propose SortNoC in the following.

*4.3.2 SortNoC Architecture.* SortNoC, illustrated in Figure 7(b), is a specialized NoC architecture to distribute and sort event trace elements. The architecture sorts the trace elements along a forward path and broadcasts them along a backward path. In the *forward path*, all trace elements are sent to a predefined target router (here, $r_3$). Even though any router could be chosen as target router, for larger architectures, central routers are especially promising because of their short hop distances to all other routers. To sort the trace elements along this path, an arbitration scheme based on Algorithm 1 is implemented in the crossbars. This approach requires that concurrently issued trace elements arrive simultaneously at a crossbar. Therefore, each router consists of a delay stage at its local input port. The required delay for each router is given by Equation (9), where $N_{diam}$ corresponds to the diameter of the NoC topology and $N_{h2t}$ corresponds to the number of hops to the target router.

$$N_{delay} = N_{diam} + 1 - N_{h2t} \tag{9}$$

As a result, the monitor-local event traces $S^{(0)}$ of $TM_0$ and $S^{(1)}$ of $TM_1$ arrive simultaneously at the crossbar of $r_1$ where they are sorted to an intermediate event trace. The intermediate event trace arrives then simultaneously with the remaining event traces $S^{(2)}$ of $TM_2$ and $S^3$ of $TM_3$ at the crossbar of $r_3$ where they are interleaved to a globally sorted event trace $S$. Thus, the forward path can be seen as a cascading of timestamp-based arbiters. In the backward path, the globally sorted trace is broadcast to the local output ports of all routers. By design there are no possible contentions along this path. Thus, it is possible to replace the input FIFOs along this path by simple registers. The routing paths to the target router are assigned statically and the broadcast paths use the same links in the opposite direction. Thus, large mesh typologies would consist of a significant number of unused links. These links can be removed to reduce the hardware overhead to a minimum.

In Figure 7(b), two trace elements are injected into the NoC architecture: $s_y = (i_{e_y}, 1, i_{c_y})$, illustrated in yellow, and $s_g = (i_{e_g}, 3, i_{c_g})$, illustrated in green. Even though $s_y$ got injected at the local input port of $r_3$, the corresponding delay stages ensure that $s_y$ arrives earlier at the crossbar of $r_3$ to ensure a correct ordering of the trace elements. In the backward path, both trace elements are then broadcasted to all local output ports with the result that all tile monitors, configured with the respective cluster IDs, get informed about the detected events. In summary, SortNoC is a lightweight alternative to the naive approach. Compared to the sorting IP, the delay stages are easy to implement, and their functionality does not depend on hard-to-choose design parameters. Furthermore, this approach broadcasts the trace elements to all routers. This allows tile monitors to verify requirements defined by common events at different tiles.

## 5 ILP-BASED ASSIGNMENT OF REQUIREMENTS TO MONITORS

The assignment of requirements to tile monitors has a considerable influence on the usable monitoring resources. While the number of events within a tile monitor is limited by $2^{W_{ie}}$, the number of available events in the monitoring system is defined by $N_{TM} \cdot 2^{W_{ie}}$. In order to exploit all available events, each monitor must use a unique cluster ID to operate on its own subset of events. If two monitors operate on the same cluster ID, they process the same events and thus reduce the number of usable events in the system by $2^{W_{ie}}$. As a result, requirements defined on the same events must be assigned to the same monitors to maximize the number of usable events in the system.

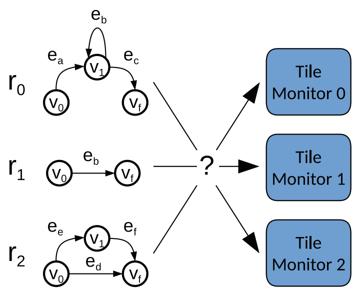


Fig. 8. A set of requirements that must be assigned to the tile monitors.

An example for the assignment problem is illustrated in Figure 8. Here, three requirements $r_0$, $r_1$, and $r_2$ must be assigned to three monitors $m_0$, $m_1$, and $m_2$. Equation (10) gives a naive assignment for this task, where each requirement is assigned to a different tile monitor.

$$U_1 = \{(r_0, m_0), (r_1, m_1), (r_2, m_2)\} \tag{10}$$

In this example, $r_0$ and $r_1$ are defined on the same event $e_b$. Therefore, $m_0$ and $m_1$ must share the same cluster ID to build a cluster of monitors that processes the same events. Hence, only $2 \cdot 2^{W_I}$ out of $3 \cdot 2^{W_I}$ events remain usable in the monitoring system. A better solution for the assignment is given in Equation (11).

$$U_2 = \{(r_0, m_0), (r_1, m_0), (r_2, m_1)\} \tag{11}$$

In this solution, each event is used by a single monitor, respectively. Thus, each monitor can use a unique cluster ID and process different events, which increases the number of usable events in the system to $3 \cdot 2^{W_I}$.

However, it is not always possible to assign requirements, which are defined on a shared event, to the same monitor. Considering an upper limit of one requirement per monitor in Figure 8, only $U_1$ out of the given mappings remains feasible. Therefore, it is necessary to find a feasible mapping of requirements to tile monitors that maximizes the number of usable events in the monitoring system. Even though this problem is similar to the multiple knapsack problem, there is a major difference. In this problem, it is possible to combine monitors, i.e., knapsacks, to generate a cluster of monitors that share the same verification ID. As a result, the cluster can be seen as an individual tile monitor, which provides the sum of available monitoring resources. However, the number of supported events in a monitoring cluster remains at $2^{W_{ie}}$. Thus, the number of monitors included in a cluster should be minimized. In the following, we formulate the assignment problem by a dependency graph and propose an ILP-based assignment algorithm to solve it.

## 5.1 Problem Formulation

An intuitive approach to formalize the problem is an acyclic undirected graph $G = (E, D)$ defined by the set of events $E$ and a relation $D$, describing the dependencies between the events. In this context, two events are defined to have a dependency if both are part of the same runtime requirement. As a result, the events in each component $H = (E_H, D_H)$ of $G$, with $E_H \in E$ and $D_H \in D$, can be monitored independently. In order to assign the individual components to the tile monitors, a notion for their corresponding cost is required. We define the cost $C_H$ of a component $H$ by a 3-tuple $(N_{F,H}, N_{L,H}, N_{E,H})$, where $N_{F,H}$ corresponds to the number of logical requirements in which the events of $H$ are used; $N_{L,H}$ corresponds to the number of timing requirements in which the events of $H$ are used; and $N_{E,H}$ corresponds to the number of events

**ALGORITHM 2:** Construction of all component cost tuples in $G$

```
function GET_COMP_COST_TUPLES($L, F$)
    $\mathcal{X} \leftarrow \emptyset$
    for $f \in F$ do
        $E_f \leftarrow \{\Pi_{e_{start}}(f), \Pi_{e_{stop}}(f)\}$
        $H \leftarrow$ complete_graph($E_f$)
        $X_H \leftarrow (H, (0, 1, 0))$
        $\mathcal{X} \leftarrow$ add_tuple($\mathcal{X}, X_H$)
    end for
    for $l \in L$ do
        $E_l \leftarrow \Pi_E(l)$
        $H \leftarrow$ complete_graph($E_l$)
        $X_H \leftarrow (H, (1, 0, 0))$
        $\mathcal{X} \leftarrow$ add_tuple($\mathcal{X}, X_H$)
    end for
    for $X_H \in \mathcal{X}$ do
        $C_H \leftarrow \Pi_C(X_H)$
        $C_H \leftarrow (\Pi_F(C_H), \Pi_L(C_H), |H|)$
        $X_H \leftarrow (H, C_H)$
    end for
    return $\mathcal{X}$
end function

function ADD_TUPLE($\mathcal{X}, X_H$)
    $\mathcal{X}_n \leftarrow \emptyset$
    for $X_{H'} \in \mathcal{X}$ do
        if share_events($H, H'$) then
            $H \leftarrow$ compose($H, H'$)
            $C_H \leftarrow C_H + C_{H'}$
            $X_H \leftarrow (H, C_H)$
        else
            $\mathcal{X}_n \leftarrow \mathcal{X}_n \cup \{X_{H'}\}$
        end if
    end for
    return $\mathcal{X}_n \cup \{X_H\}$
end function
```

in $H$. This mapping between a component and its cost is defined by a tuple $X_H = (H, C_H)$ with $X_H \in \mathcal{X}$, which is used by the ILP-based requirement assignment.

Algorithm 2 constructs the set of all tuples $\mathcal{X}$ using the logical requirements $L$ and the timing requirements $F$. In a first step, the algorithm iterates over all timing requirements and generates a complete graph $H$ out of the start and stop events of each requirement. The graph is then mapped to a cost tuple that indicates a resource usage of one timer. At this step, the cost of events $N_{E,H}$ is neglected as otherwise the computation of the composed cost of an existing component and a new clique could not be performed by an addition since shared events would be counted twice. Thus, the cost of events is calculated at the end of the algorithm. Afterward, the component cost tuple $X_H$ is inserted in the existing set of component cost tuples $\mathcal{X}$ calling *add_tuple*($\mathcal{X}, X_r$). This function iterates over all existing component cost tuples and checks whether the corresponding graphs share an event with $X_H$. In this case, it updates the component graph $H$ by the composition of both graphs and the cost tuple $C_H$ by the addition of both cost tuples. Component cost tuples that do not share an event with $X_H$ are unified with a new set of component cost tuples $\mathcal{X}_{new}$. The union of $\mathcal{X}_{new}$ and $X_H$ corresponds then to the new set of all component cost tuples $\mathcal{X}$.

After the timing requirements, the logical requirements are processed similarly. A complete graph is generated out of the accepted events of the automata definition and a cost tuple is generated that indicates a resource usage of one AP. The resulting component cost tuple $X_H$ is then composed with the existing set of component cost tuples in $\mathcal{X}$ calling *add_tuple*$(\mathcal{X}, X_H)$.

At the end, the event cost $N_{E,H}$ of all component cost tuples $X_H$ is updated by the cardinality of the graph components. The result of the algorithm is a set of component cost tuples $\mathcal{X}$ that can be used by the following ILP-algorithm to solve the assignment problem.

## 5.2 ILP Formulation

An optimal assignment of component cost tuples to tile monitors allows designers to verify a larger number of requirements or more complex requirements due to the higher number of available events. To achieve this, we propose an ILP-based assignment that maximizes the number of available events for the given resource constraints of the monitoring hardware. The number of supported events per tile monitor $N_{E'} = 2^{W_I}$ is given by the supported event ID width $W_{i_e}$ in an AP. Furthermore, the number of available AP resources and timer resources is given by $N_A$ and $N_T$. In the ILP, the monitor clusters can be modeled by one monitor that receives the resources of all other monitors in the cluster. Thus, the number of available resources of a monitor $m \in TM$ is given by $N_{A,m}$ and $N_{T,m}$. Formally, we describe this behavior by a relation matrix $P_{N_{TM} \times N_{TM}}$, where a monitor $m$ provides its resources to monitor $n$ if $p_{m,n} = 1$. Similarly, the assignment of components cost tuples is modeled by a matrix $Q_{N_{\mathcal{X}} \times N_{TM}}$ where a component cost tuple $X_H$ is assigned to a monitor $m$ if $q_{H,m} = 1$.

The objective of the ILP is to maximize the number of available events in the monitoring architecture. Therefore, the number of monitors providing its resources to itself must be maximized such that the number of clusters is minimized. Formally, the objective is given by Equation (12).

$$\max \sum_{m \in TM} p_{m,m} \tag{12}$$

To ensure that each monitor provides its resources to only one monitor (including itself), each row of $P_{N_{TM} \times N_{TM}}$ must contain exactly one assignment.

$$\forall_{m \in TM} \sum_{n \in TM} p_{m,n} = 1 \tag{13}$$

Similarly, each component cost tuple must be assigned to exactly one tile monitor.

$$\forall_{X_H \in \mathcal{X}} \sum_{m \in TM} q_{H,m} = 1 \tag{14}$$

The available timing and AP resources a monitor gets provided is modeled by the equality constraints in Equations (15) and (16), respectively.

$$\forall_{m \in TM} \sum_{n \in TM} p_{n,m} N_A = N_{A_m} \tag{15}$$

$$\forall_{m \in TM} \sum_{n \in TM} p_{n,m} N_T = N_{T_m} \tag{16}$$

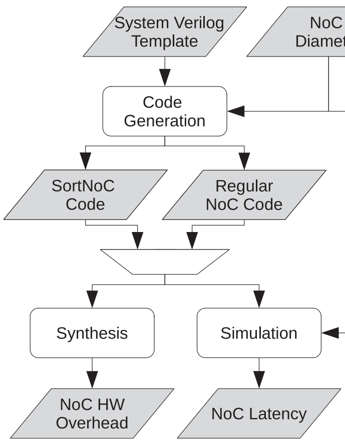


Fig. 9. NoC Simulation Framework.

Finally, the assigned component cost tuples must be restricted by the number of available resources for each monitor. Equations (17)–(19) model this using inequality constraints for each resource type, respectively.

$$\forall_{m\in TM} \sum_{X_H\in\mathcal{X}} q_{H,m}\Pi_F(C_H) \leq N_{A_m} \tag{17}$$

$$\forall_{m\in TM} \sum_{X_H\in\mathcal{X}} q_{H,m}\Pi_L(C_H) \leq N_{T_m} \tag{18}$$

$$\forall_{m\in TM} \sum_{X_H\in\mathcal{X}} q_{H,m}\Pi_E(C_H) \leq N_{E'} \tag{19}$$

After the ILP is solved, the cluster ID of each monitor can be derived from $P_{N_{TM}\times N_{TM}}$ while $Q_{N_{\mathcal{X}}\times N_{TM}}$ describes the component to monitor assignments. Using this information, it is possible to assign unique IDs to events that are used in the same monitor cluster and, thus, generate probe and tile monitor configurations.

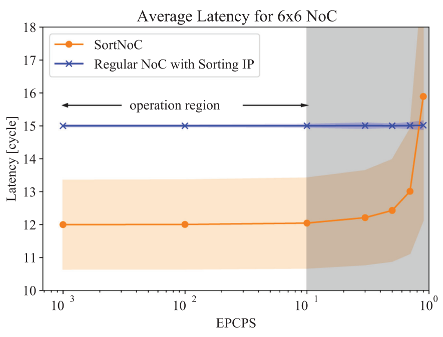
## 6 EXPERIMENTAL RESULTS

In this section, we first evaluate the performance of the proposed SortNoC and, subsequently, demonstrate the monitoring architecture in hardware on multiple case studies. Finally, we compare the ILP-based assignment algorithm with a greedy assignment policy and present the hardware overhead of the monitoring system.

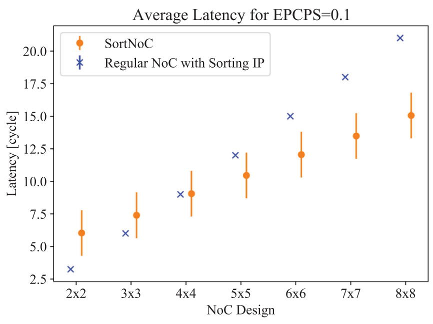
### 6.1 Evaluation of the SortNoC Architecture

The performance of the tracing NoCs, discussed in Section 4.3, is evaluated using the simulation framework illustrated in Figure 9. Based on System Verilog templates, we generate the code of the SortNoC and the regular NoC architecture for different diameters, which are then synthesized in the following to obtain the HW overhead of the architecture. Additionally, we simulate the architectures for 10,000 randomly injected trace elements for different EPCPS ratios. For the timestamp offset of the sorting IPs, we choose $t_o = \max(2, 1.5 \cdot N_{diam})$ to ensure a correct order of the trace elements in the naive tracing approach.

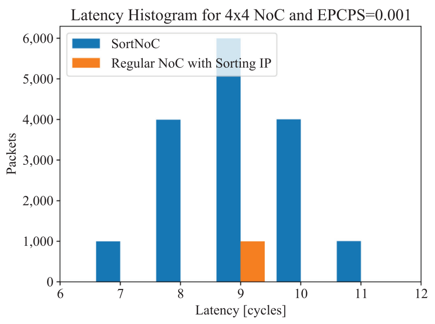
Figure 10(a) shows the average latency and the standard deviation of a 6x6 architecture for different EPCPS ratios. The regular NoC with sorting IP shows a constant latency and a very low standard deviation over all EPCPS ratios because the minimal time a trace element remains in



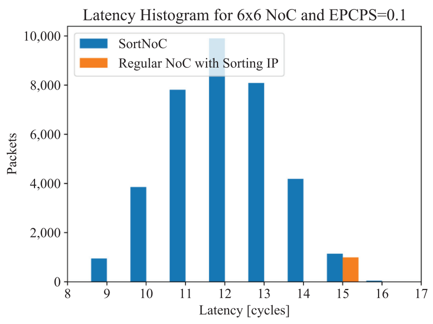
(a) The average latency and the standard deviation for a fixed 6x6 NoC architecture.



(b) The average latency and the standard deviation for a fixed injection rate of 0.1 EPCPS.



(c) The latency histogram for 4x4 NoC architectures with an injection rate of 0.001 EPCPS.



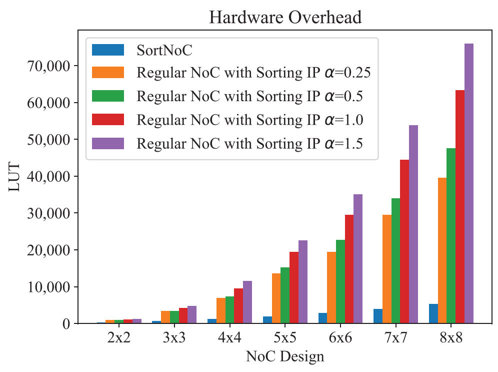
(d) The latency histogram for 6x6 NoC architectures with an injection rate of 0.1 EPCPS

Fig. 10. A comparison between the latencies of both NoC approaches.

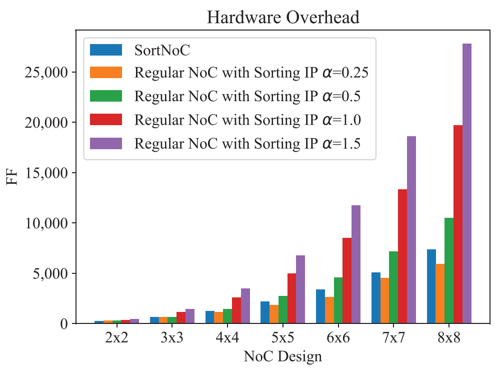
the NoC architecture is defined by the timestamp offset $t_o$. In contrast, SortNoC shows a lower latency with an exponential increase for large EPCPS ratios. Although the throughput of SortNoC is limited by 1 EPCPS, for typical event detection ratios in RV (mili- and microseconds [6, 12]), a throughput of up to 0.1 EPCPS is sufficient even for large-scale systems. A detailed discussion on this limitation is presented in Section 3.3.

Figure 10(b) shows the average latency and its standard deviation for an average trace element injection ratio of 0.1 EPCPS and different NoC sizes. The proposed SortNoC scales better for large NoC architectures than the regular NoC architecture with sorting IP. While our approach responds to temporal peaks in the injection ratio with a brief increase in latency, the design parameters of the regular NoC with sorting IP must be designed for the worst case to ensure the correct order of events at all times. Thus, a conservative choice of the timer offset $t_o$ of the sorting IP is required, which affects the NoC performance throughout the simulation.

In Figure 10(c) and (d), we evaluate the NoC latency in a histogram for a low EPCPS ratio of 0.001 and a relatively high EPCPS ratio of 0.1. In both latency histograms, it can be seen that the overall number of packets, i.e., trace elements, in the SortNoC architecture is significantly higher than in the regular NoC architecture. This behavior results from the properties of the target router. It broadcasts all trace elements to all tile monitors, which increases the overall number of trace elements in the NoC by a factor of $N_{TM}$. The latency in the regular NoC architecture is defined by the timer offset $t_o$ in the sorting IP. As a result, the packet latency is fixed unless multiple trace elements with the same timestamp are inside one sorting IP. In contrast, the latency in the SortNoC



(a) The hardware overhead in terms of LUTs.



(b) The hardware overhead in terms of FFs.

Fig. 11. The hardware overhead of both tracing NoC approaches.

depends on the hop distance of the packets and on the EPCPS rate. As a result, the latencies show a broader distribution in the histogram. In Figure 10(c), both NoC approaches show a similar average latency while a larger standard deviation can be observed for the packets in the SortNoC. However, for larger architectures, SortNoC outperforms the regular NoC architecture with the sorting IP for almost all packets as shown in Figure 10(b). This confirms the observation from Figure 10(b) that SortNoC scales better with the size of the system.

We evaluate the area overhead of both approaches using the synthesis results of Vivado 2018.3. We synthesize the naive tracing approach for different depths of the sorting IP.

$$N_{depth} = \alpha \cdot N_{diam}, \text{ with } \alpha \in \{0.25, 0.5, 1, 1.5\} \tag{20}$$

The hardware overhead in terms of LUTs is illustrated in Figure 11(a) and in terms of FFs in Figure 11(b). SortNoC utilizes 73% to 94% fewer LUTs than the regular NoC with sorting IP since the NoC topology implements fewer links and fewer FIFOs. Furthermore, the sorting IP is mostly implemented using LUTs, which results in a high overhead. In contrast, the delay stage requires more FFs than the sorting IP. Thus, the regular NoC with Sorting IP uses fewer FF for $\alpha = 0.25$ than SortNoC. Nevertheless, the overall overhead of FFs and LUTs is significantly lower for the proposed tracing NoC than for the naive approach.

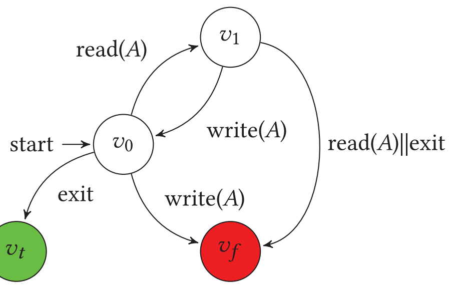
## 6.2 Demonstration of Runtime Verification Case Studies

We demonstrate the monitoring architecture, in the configuration presented in Section 4, on a 2x2 tiled MPSoC with five Leon3 cores per tile on a Virtex-7 FPGA. The power consumption of each of the 20 cores is evaluated by a power emulator similar to the embedded power temperature monitor (ePTMon) proposed by Listl et al. [19]. As there exist no standardized benchmarks for the evaluation of monitoring systems, we evaluate the system on two typical debugging case studies and on one performance benchmark for high-performance computers.

*6.2.1 Case Study 1: Data Race.* A prominent cause for the undesired behavior of an application is a race condition. In a race condition, the behavior of the application depends on the timing or the sequence of uncontrollable events. This situation typically arises if two threads have uncontrolled access to shared resources. An example for this situation, i.e., a data race, is illustrated in Figure 12(a). Here, two threads consecutively read and write from a resource *A*, such that the modification of *thread_1* is overwritten by *thread_0*. Race conditions are especially hazardous because they do not necessarily manifest themselves as a bug during the testing phase. A detection of a race time

| ***thread_0*** | ***thread_1*** |
|---|---|
| *read(A)* | |
| | *read(A)* |
| | |
| | *write(A)* |
| *write(A)* | |

(a) Two threads competing for a shared resource $A$.



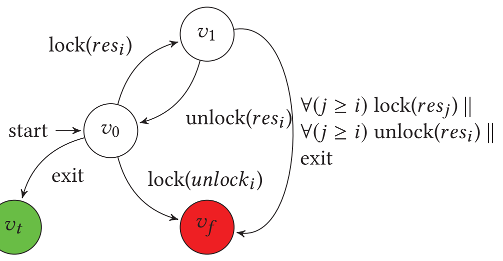
(b) A logical requirement to detect a data race.

Fig. 12. An example for a data race and the corresponding logical requirement.

time

| ***thread_0*** | ***thread_1*** |
|---|---|
| *lock($res_1$)* | |
| | *lock($res_0$)* |
| *lock($res_0$)* | |
| | *lock($res_1$)* |
| *unlock($res_0$)* | |
| | *unlock($res_1$)* |
| *unlock($res_1$)* | |
| | *unlock($res_0$)* |

(a) Two threads using a complementary locking order for shared resources.



(b) A logical requirement to protect the locking order for a resource $res_i$.

Fig. 13. An example for a deadlock and the corresponding logical requirement.

condition by static code analysis is proven to be NP-hard [23]. Therefore, one might want to protect the critical sections by runtime verification supplementary to the locking mechanisms in software. The logical requirement for the protection of the critical section is illustrated in Figure 12(b) as an automata. This requirement can be used to debug data races at design time. Furthermore, the requirement can be used to protect the locking mechanism after deployment. If the data race manifests itself as a bug, the monitoring architectures raises an interrupt to the operation system, which can then initiate an appropriate reaction (e.g., rerun the involved threads). To demonstrate this case study in HW, we wrote a short application containing the race condition illustrated in Figure 12(a). Each of the threads was mapped to a different tile in the large-scale architecture. Using the requirement in Figure 12(b), it was possible to detect when the race condition manifests itself in a bug.

*6.2.2 Case Study 2: Deadlock.* Another prominent example for a race condition is the deadlock illustrated in Figure 13(b). Here, two threads are competing for two shared resources in opposite order. As a result, both threads are blocked from execution. Such a behavior can be prevented by a strict locking order. The corresponding logical requirement is illustrated in Figure 13(b) as an automaton for an arbitrary resource $res_i$. The requirement not only prevents a data race for the lock, it also ensures that no lock with a lower priority will be acquired, which guarantees the strict locking order required to prevent deadlocks. Alternatively, a generic approach to detect deadlocks is a timing requirement $(e_{start_{t\_i}}, e_{stop_{t\_i}}, 0, T_{max})$, which defines an upper bound for the execution time of thread $t\_i$. Again the requirements can be used to debug deadlocks pre- and post-deployment. As existing approaches such as DiaSys and NUVA do not support timing requirements, they do not support deadlock detection either pre- or post-deployment.

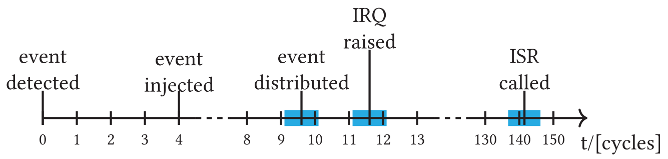


Fig. 14. The detection latency of the monitoring architecture illustrated on a timeline.

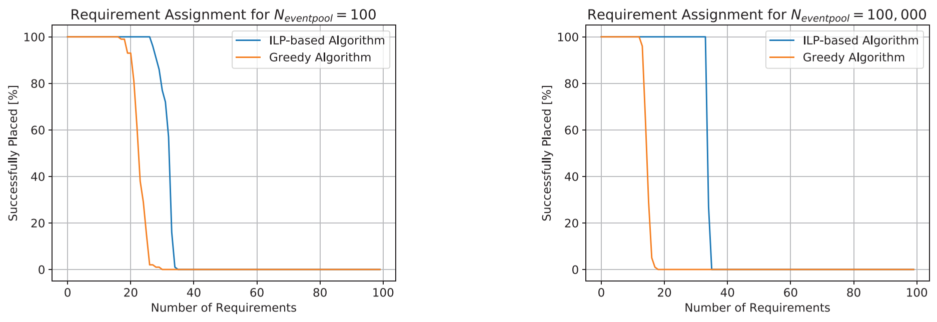
We also demonstrated this case study in HW. Again, each of the threads was mapped to a different tile in the large scale architecture and the monitoring system detected the deadlock with each of the requirements illustrated in Figure 13(b).

*6.2.3 Embarrassingly Parallel Benchmark.* "Embarrassingly Parallel" is the name of one benchmark in the NAS Benchmark suite [4] that has been designed to evaluate the performance of supercomputers and large-scale systems. We decided to use the embarrassingly parallel benchmark due to its high load on all cores in the system. Therefore, the benchmark can be seen as a stress test for the complete system including the monitoring architecture. In this demonstration, we verify the major branches and the synchronization points of the application. To demonstrate the RV architecture, we add synchronization and control-flow bugs to the code, which are then detected at runtime. Furthermore, we use one of the timers to measure the detection latency between the event causing an interrupt and the execution of the interrupt service routine (ISR).

The results are illustrated on the timeline in Figure 14, where the marked regions correspond to the standard deviations of the measured times. The internal latencies in the monitoring architecture are obtained by simulations. On average, it takes 141 cycles from the detection of a wrong event until the ISR is called. The other benchmarks of this benchmark suite are not presented in this article because they would not generate further value.

## 6.3 Requirement Assignment

In order to evaluate the performance of the ILP-based assignment algorithm presented in Section 5, we compare it to a greedy assignment policy. The greedy assignment policy follows assignment Algorithm 3. Here, the algorithm goes through all requirements (either logical or timing) and tries to assign each to one of the monitors. The function *try_place* returns true if the monitor resources of $tm$ are sufficient to incorporate requirement $r$ and apart from that false. If a requirement $r$ could not be placed in any of the monitors, an error is returned. If the requirement was successfully placed in one of the monitors, the algorithm checks whether some of the monitors must be merged to one monitor cluster due to shared events between the monitors. Both algorithms are evaluated for 100 synthetic requirements, which have been generated randomly out of an event pool. The timing requirements are generated by randomly choosing two events out of the event pool that serve as start and stop events. For the logical requirements, we first determine a random number of states with an average of four and, subsequently, a random number of edges with an average of four as well. Each edge is triggered by a random event out of the event pool. Requirements, which are generated from a small event pool, are likely to have many shared events while requirements, which are generated from a large event pool, are unlikely to have shared events. In Figure 15, we present results for both scenarios. The graphs show that the ILP-based assignment algorithm is superior to the greedy assignment policy independently of the event pool size and therefore, independently from the number of shared events between the requirements. However, it can be seen that the ILP-based assignment algorithm is especially promising for a low number of shared events, which is also expected for real applications. Here, the algorithm must carefully decide



(a) Low number of shared events, i.e. an event pool size of 100.

(b) High number of shared events, i.e. an event pool size of 100,000.

Fig. 15. A comparison between the ILP-based assignment algorithm and a greedy assignment policy.

which assignment minimizes the cost function. For a large number of shared events, almost all monitors must be assigned to one large cluster anyway, which gives the algorithm few options to optimize the assignment.

**ALGORITHM 3:** Greedy Requirement Assignment Policy

```
1: function ASSIGN_REQ_GREEDY($R$)
2:     for $r \in R$ do
3:         assigned ← False
4:         for $tm \in TM$ do
5:             if try_place($r, tm$) then
6:                 assigned ← True
7:                 break
8:             end if
9:         end for
10:        if not assigned then
11:            return Error
12:        end if
13:        merge_mon_if_neccessary($TM$)
14:    end for
15: end function
```

## 6.4 Monitoring Overhead

Our monitoring architecture is non-intrusive. Therefore, only a hardware overhead but no performance overhead needs to be considered. The required hardware resources to implement an example configuration of the monitoring architecture are illustrated in Table 1. Here, each checkpoint and each out-of-range detector supports 32 different events. Furthermore, 32 timers and 32 APs are implemented on each tile monitor in the system. According to the needs of the system architecture, this number could be increased or reduced. The overall hardware overhead of the monitoring system corresponds to 10.8% in terms of LUTs and 30.6% in terms of FFs. Thus, the overhead is significantly lower than other non-intrusive verification approaches like DiaSys [33], which introduces an overhead of more than 50% in LUTs and FFs. In contrast to our monitoring architecture, DiaSys does not provide error detection capabilities, which makes our lower overhead even more noteworthy. In general, the implementation of debug units and runtime verification hardware is

Table 1. The Resource Usage of the Monitoring Architecture and the Base System

| Module | LUTs | FFs | RAM36s | RAM18s |
|---|---|---|---|---|
| Base System | 471,918 | 231,980 | 270 | 1,480 |
| 1 Tile | 126,370 | 71,656 | 67 | 402 |
| 2x2 NoC | 8,981 | 9,517 | 0 | 0 |
| Monitoring System | 51,268 | 71,188 | 4 | 128 |
| 2x2 SortNoC | 1,013 | 266 | 0 | 0 |
| 1 Probe | 1,441 | 2,492 | 0 | 0 |
| 1 Probe (incl. EOP) | 1,720 | 2,494 | 0 | 0 |
| 1 Tile monitor | 3,454 | 4,000 | 1 | 32 |
| 1 AP | 5 | 0 | 0 | 1 |
| 1 Timer | 40 | 117 | 0 | 0 |

expensive in terms of hardware overhead. Even standard debug units, e.g., for the OpenProcessor Platform [3] or an RISCV core [32], introduce area overheads of around 10%. Nevertheless, this area is well invested, as no modern processors can be programmed efficiently without debug support. We believe that the area overhead for runtime verification will be also well invested, as it is becoming increasingly difficult to verify all possible scenarios at design-time when developing safety-critical software for complex many-core systems.

## 7 CONCLUSION

In this article, we presented a decentralized monitoring architecture for multi-tile MPSoCs that support the verification of logical and timing requirements. The architecture consists of SortNoC, a new tracing interconnect that generates a globally sorted event trace on which the runtime requirements can be verified. Furthermore, a requirement assignment algorithm was proposed to exploit the resources of the monitoring architecture best. In our evaluation, we showed the benefits of the proposed tracing interconnect, we demonstrated the architecture on a Virtex-7 FPGA, and we presented the advantages of our assignment algorithm.

## REFERENCES

[1] AUTOSAR. 2016. Overview of Functional Safety Measures in AUTOSAR.

[2] ARM. [n.d.]. CoreSight. Retrieved from https://www.arm.com/products/silicon-ip-system.

[3] S. Bach. 2008. Design and Implementation of a Debugging Unit for the OpenProcessor Platform. Seminar Paper.

[4] D. Bailey et al. 1991. The Nas parallel benchmarks. *International Journal of High Performance Computing Applications* 5 (Sept. 1991), 63–73.

[5] A. Bauer, M. Leucker, and C. Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 14.

[6] J. O. Blech, Y. Falcone, and K. Becker. 2012. Towards certified runtime verification. In *Formal Methods and Software Engineering*, Toshiaki Aoki and Kenji Taguchi (Eds.). Springer, Berlin, 494–509.

[7] N. Decker et al. 2018. Online analysis of debug trace data for embedded systems. In *Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 851–856.

[8] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (Dec. 2014), 3088–3098.

[9] S. Fischmeister and P. Lam. 2010. Time-aware instrumentation of embedded software. *IEEE Transactions on Industrial Informatics* 6, 4 (Nov. 2010), 652–663.

[10] A. Francalanza et al. 2017. A foundation for runtime monitoring. In *Runtime Verification*, Shuvendu Lahiri and Giles Reger (Eds.). Springer International Publishing, Cham, 8–29.

[11] A. Francalanza, Jorge A. Pérez, and César Sánchez. 2018. *Runtime Verification for Decentralised and Distributed Systems*. Springer International Publishing, Cham, 176–210.

[12] F. Gorostiaga and C. Sánchez. 2018. Striver: Stream runtime verification for real-time event-streams. In *Runtime Verification*, C. Colombo and M. Leucker (Eds.). Springer International Publishing, Cham, 282–298.

[13] D. Heffernan, C. Macnamee, and P. Fogarty. 2014. Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties. *IET Software* 8, 5 (2014), 193–203.

[14] T. L. Hinrichs, A. Prasad Sistla, and L. D. Zuck. 2014. Model check what you can, runtime verify the rest. In *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday (EPiC Series in Computing)*, Andrei Voronkov and Margarita Korovina (Eds.), Vol. 42. EasyChair, 234–244.

[15] C. Hochberger and A. Weiss. 2008. A new methodology for debugging and validation of soft cores. In *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications.* 551–554.

[16] Institute for Software Engineering and Programming Languages. [n.d.]. LamaConv—Logics and Automata Converter Library. Retrieved from https://www.isp.uni-luebeck.de/lamaconv.

[17] C. Lee and J. Tsai. 1995. A shift register architecture for high-speed data sorting. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 11 (1995), 273–280.

[18] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *Journal of Logic and Algebraic Programming* 78, 5 (May/June 2009), 293–303.

[19] A. Listl, D. Mueller-Gritschneder, F. Kluge, and U. Schlichtmann. 2018. Emulation of an ASIC power, temperature and aging monitor system for FPGA prototyping. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS).* 220–225.

[20] H. Lu and A. Forin. 2008. Automatic processor customization for zero-overhead online software verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 10 (Oct. 2008), 1346–1357.

[21] M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann. 2020. Runtime monitoring of inter- and intra-thread requirements on embedded MPSoCs. In *Proceedings of the 2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)* (Jan. 2020).

[22] A. Nassar, F. J. Kurdahi, and W. Elsharkasy. 2015. NUVA: Architectural support for runtime verification of parametric specifications over multicores. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES).* 137–146.

[23] R. H. B. Netzer and B. P. Miller. 1990. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing.* 93–97.

[24] R. Perez-Andrade, R. Cumplido, C. Feregrino-Uribe, and F. Martin Del Campo. 2009. A versatile linear insertion sorter based on an FIFO scheme. *Microelectronics Journal* 40, 12 (2009), 1705–1713.

[25] S. M. K. Quadri and S. U. Farooq. 2010. Article: Software testing - goals, principles, and limitations. *International Journal of Computer Applications* 6, 9 (Sept. 2010), 7–11.

[26] T. Reinbacher, J. Geist, P. Moosbrugger, M. Horauer, and A. Steininger. 2012. Parallel runtime verification of temporal properties for embedded software. In *Proceedings of the 2012 IEEE/ASME 8th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications.* 224–231.

[27] M. Seo and F. Kurdahi. 2019. Efficient tracing methodology using automata processor. *ACM Transactions on Embedded Computing Systems* 18, 5s (Oct. 2019).

[28] M. Seo and R. Lysecky. 2018. Non-intrusive in-situ requirements monitoring of embedded system. *ACM Transactions on Design Automation of Electronic Systems* 23, 5 (Aug. 2018), 58:1–58:27.

[29] D. Solet, J. Béchennec, M. Briday, S. Faucou, and S. Pillement. 2016. Hardware runtime verification of embedded software in SoPC. In *Proceedings of the 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES).* 1–6.

[30] D. Solet, J. Béchennec, M. Briday, S. Faucou, and S. Pillement. 2018. Hardware runtime verification of a RTOS kernel: Evaluation using fault injection. In *Proceedings of the 2018 14th European Dependable Computing Conference (EDCC).* 25–32.

[31] D. Solet, S. Pillement, J. Béchennec, M. Briday, and S. Faucou. 2018. HW-based architecture for runtime verification of embedded software on SoPC systems. In *Proceedings of the 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS).* 249–256.

[32] N. Velguenkar. 2018. *The Design of a Debugger Unit for a RISC Processor Core.* Master's thesis. Rochester Institute of Technology (RIT).

[33] P. Wagner, T. Wild, and A. Herkersdorf. 2016. Improving SoC insight through on-chip diagnosis. *CoRR* abs/1607.04549 (2016). arxiv:1607.04549

Received May 2020; revised October 2020; accepted October 2020