Non-Intrusive Runtime Monitoring for Manycore Prototypes

Fabian Lesniak, Nidhi Anantharajaiah, Tanja Harbaum, Juergen Becker
{lesniak,nidhi,harbaum,becker}@kit.edu
Institute of Information Processing Technology (ITIV), Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

ABSTRACT

Rapid prototyping is widely used, essential technique for developing novel computing architectures. While simulation-based approaches allow to examine the design under test, the observability of FPGA-based prototypes is limited as they can behave like a black box. However, for verification and design space exploration purposes it is crucial to obtain detailed information on the internal state of such a prototype. In this work we propose an architecture to gather detailed internal measurements during execution and extract them from the design under test without impacting its runtime behavior. It is specifically designed for low resource usage and minimal impact on timing, leaving more resources for the actual prototyped system. Our proposed architecture offers several different interface modules for various signal sources, including register capturing, event counters and bus snooping. We present an estimate of achievable bandwidth and maximum sample rate as well as a demanding case-study with a tiled manycore platform on a multi-FPGA prototyping platform. Experimental results show up to 32 million 4-byte measurements per second, saturating a gigabit Ethernet connection. The monitoring system has proven to be very useful when working with an FPGA-based manycore prototype, as it is an essential tool to reveal incorrect behavior and bottlenecks in hardware, operating system and applications at an early stage.

CCS CONCEPTS

• **Hardware** → **Methodologies for EDA**; **Reconfigurable logic and FPGAs**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

KEYWORDS

Prototyping, Monitoring, Non-intrusive debugging, Data aquisition

ACM Reference Format:
Fabian Lesniak, Nidhi Anantharajaiah, Tanja Harbaum, Juergen Becker. 2022. Non-Intrusive Runtime Monitoring for Manycore Prototypes. In *Proceedings of Rapid Simulation and Performance Evaluation for Design (RAPIDO '23).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

RAPIDO '23, January 16–18, 2023, Toulouse, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The development of novel computing architectures requires in-depth verification and thorough validation. Especially when targeting ASICs, the *first time right* principle should be followed [11]. New components of any System on Chip (SoC) are therefore tested extensively using hardware simulation. Most errors on component or unit level can be ruled out at this stage. However, the interplay of different components and the system as a whole is not verified. Cycle accurate system simulators such as GEM5 [3] offer a highly configurable framework optimized for CPU architectures. Such simulators provide high accuracy, but are often not suitable for comprehensive verification of manycore designs due to the high complexity of a large system simulation [1].

In general, simulation performance is improved by increasing the level of abstraction. Virtual platforms, for example, do not attempt cycle-accurate simulation. Instead, they make use of high-level models of the hardware design [10]. This is a valid approach for tasks like design space exploration or functional verification, but it omits important details of the hardware design. It is therefore desirable to verify cycle-accurate behavior to increase the test coverage. Rapid prototyping techniques can be used to solve to this issue. A common approach is to implement the full architecture on an FPGA system. While not reaching the full performance of an ASIC realization, the execution is cycle-accurate by design and still faster when compared to simulation.

An FPGA-based prototype offers interfaces for programming, debugging and communication. FPGA bitstreams can be programmed using JTAG and software can communicate using UART, Ethernet, PCI-Express, and various other interfaces. However, the visibility of the internal state is limited compared to simulation. If the status of a signal or register is vital for debugging or evaluation, a software or hardware interface has to be setup in advance. However, both approaches have some drawbacks: accessing a software interface over shared resources can influence execution behavior, while attaching a hardware debugger is tedious process and the available read-out bandwidth is limited.

In this paper, we present an approach for a non-intrusive monitoring system for large manycore prototypes. The design allows to capture a large number of hardware- and software-based metrics of several different types. Special effort has been spent to prevent inference with the prototyped system to maintain the quality of the acquired metrics. In the following sections, the concept of the proposed monitoring system is introduced, including both the hardware architecture and the corresponding software stack. We present an implementation in an FPGA-based manycore prototype in section 4 and provide an evaluation of performance and resource usage in section 5.

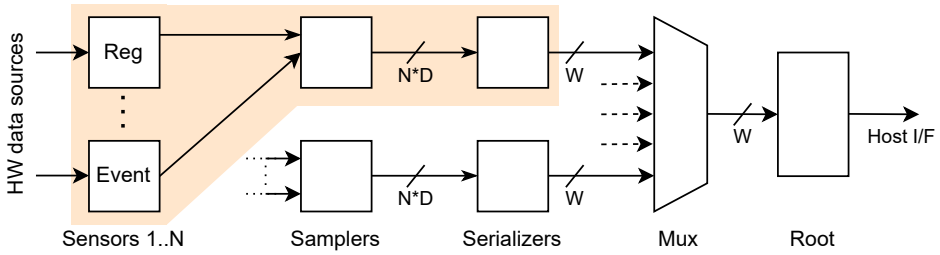

Figure 1: Overview of the proposed hardware architecture with one sensor group highlighted, N denotes the number of sensors in a group, D is the value width of a sensor, W is the data bus width of the mux tree

2 RELATED WORK

Runtime verification is a broad field focusing on the dynamic analysis of computing systems to ensure correct behavior. Starting with a formal specification of the expected system behavior, a set of monitoring instrumentation can be synthesized. The execution traces created by these monitored during runtime are then compared against the specification to identify deviations from the expected behavior [12]. Creating the execution traces in software involves a certain overhead which can alter the system behavior, therefore creating these traces in hardware has been examined [13]. The approach by Solet et. al. greatly reduces the software overhead, although not completely avoiding it, and requires the hardware to be reprogrammable to include the hardware-based monitoring instrumentation. Instead of processing the generated trace data offline, Hoppe et. al. propose to process this data directly within the system to ensure the security of the monitored processors [6]. Runtime verification methods commonly aim to create a complete trace of all significant events, which becomes a great challange for large-scale and distributed systems [4]. In contrast to verification using a formal specification, the proposed system aims to be a tool for debugging and design space exploration, with absolutely no impact on the behavior of the running system. Additionally, our approach does not require a specification to verify against and instead the captured data is visualized to help the developer identify incorrect behavior.

Debuggers are tools to help a system developer to detect and analyze errors. Both hardware and software debuggers exist, however they take fundamentally different approaches and are built differently. Software debuggers either modify the software, which can affect the behavior of the system, or they rely on special hardware support, which may not be available or may also affect behavior [14]. Hardware debuggers have a different scope by capturing wires within a design and therefore give access to otherwise hidden internal signals. However, the number of signals that can be detected is limited and they usually need to be carefully adapted to the analysis of a particular problem. Lee et. al. propose a framework for minimally intrusive online monitoring of embedded processors. Hardware observability interfaces are used to capture specific predefined events from the CPU and peripherals. They are connected using a dedicated bus system together with an additional processor, which runs the observation software. This system allows very detailed non-intrusive analysis and is self-contained, but by using a dedicated processor within the hardware design, it does not scale well to manycore systems.

3 NON-INTRUSIVE MONITORING SYSTEM

The overall goal of our non-intrusive monitoring system is to capture performance metrics from different data sources in the system prototype and forward this data to an external system for further processing and analysis. Several different sources of performance metrics are available across a SoC prototype, including single-bit events like interrupt lines, fixed-width registers like predefined performance counters, occupation of system busses in general or metrics calculated in software. A single SoC can contain thousands of potential interesting data sources. When such a SoC is scaled up to create an MPSoC or replicated to compose a manycore, the number quickly grows by orders of magnitude, scaling proportionally with the size of the prototyped architecture. Therefore, a monitoring system needs to be scalable, but also flexible enough to adapt to heterogeneous and irregular architectures.

To reduce errors during measurement, the proposed monitoring architecture is designed to share as little resources as possible with the system being prototyped, which is referred to as the design under test (DUT). Hence, it is conceived as an independent hardware design, from capturing the metrics up to sending the data stream out. Besides data provided actively by software, all sensors operate passively and can thus obtain data without impacting the execution of the prototype system. It is desirable that the resource overhead and the timing requirements are kept low, that the majority of resources and timing budget is still available for the DUT. Care has been taken to minimize the timing impact on the target design when integrating the non-intrusive monitoring system, since every data path in the monitoring system can be split using a specialized component if necessary. This component can even be replicated multiple times if necessary to reduce timing and routing complexity further, using a small amount of logic in return. The structure of the hardware design is shown in figure 1 and consists of the following basic components:

- **Sensors:** A sensor captures raw data from the system prototype and converts it into a consistent format for further processing. Different types of sensors need to be considered for different types of data sources (see section 3.1).
- **Sampler:** Data from each sensor is being sampled either at a fixed time interval or based on an external trigger condition. The fixed time interval is usually shared with other sensors in

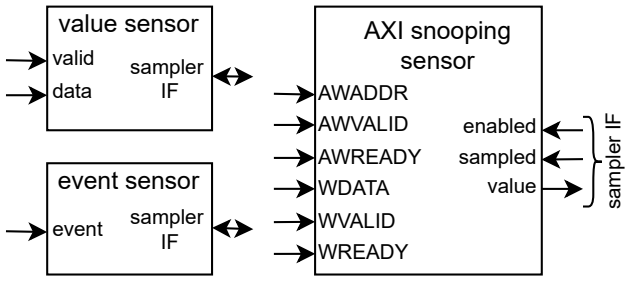

Figure 2: Interface of the value, event and AXI bus snooping sensors

a group, but can be reconfigured individually if a given source has to be monitored more or less frequently. An internal counter determines the time of sampling and can be shared with other samplers.

- **Serializer:** Since the monitoring data has to be converted into a data stream, the serialization component converts the sampled sensor data of arbitrary width to words that match fixed internal bus width. Data is padded to multiples of the internal bus width and prefixed with an identifier number. The serializer node can be configured to omit sensor values from the data stream that have not changed since the last sample. This saves bandwidth, but requires additional hardware resources to compare the values and has to be enabled at design time.
- **Mux-based interconnect tree:** The flow of monitoring data is a N-to-1 communication pattern, as sensors are physically distributed throughout the whole design and their data needs to arrive at one single root node. This results in a tree structure with the sensors as leaves and the root node representing the destination of the data stream. At each fork, a fully buffered data multiplexer is inserted to merge incoming branches into a single data stream.
- **Root node:** The root node receives a single data stream containing monitoring data from all sensors. It is responsible for packetizing, generating a checksum and forwarding this data to an external interface, e.g. Ethernet.

3.1 Types of sensors

Sensor modules are responsible for gathering information from various sources within the prototyped system and converting it into a common representation. Potential data sources include CPU performance counters, bus usage and transfer error indicators, status registers of hardware accelerators and Network-on-Chip (NoC) usage statistics. Depending on what data should be monitored, different sensor types can be used. Our proposed architecture includes a library of different sensor types for both VHDL and Verilog designs, covering most common applications.

The simplest case is a sensor for a hardware register, which directly represents the value to be monitored, e.g. a performance counter which is part of a CPU. This value is being passed through to the sampler component directly. Another basic sensor type is an event counter, which counts single-bit events within each sample

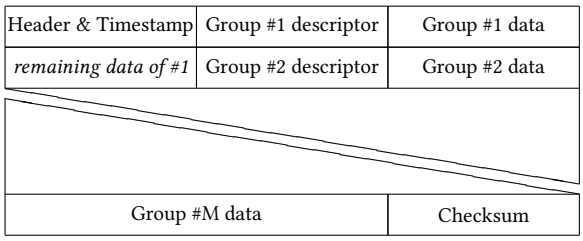

Figure 3: Data packet format of a full sample from M groups at the root node, assuming two sensors per group

interval. It is commonly used to count interrupts, bus transfers, cache hits/misses etc. Combining these two sensor types leads to a conditional counter: when a certain condition is met, a value is being captured and optionally accumulated. This behavior is useful for values which are only present at certain points in time, like the duration of specific hardware operations. Specialized conditional counters for common bus systems are also provided: reads and writes to AXI and AHB busses can be snooped on certain addresses to monitor memory contents without additional accesses. Figure 2 shows the register-based value sensor, the event sensor to count single-bit events and the AXI bus snooping sensor, including the interface to the sampling component.

A generic memory-mapped register file is also provided to monitor software-provided values. Using this interface violates the non-intrusive design paradigm, but has been proven to be quite convenient as a high-performance data logging path. If no predefined sensor matches a specific use-case, a new sensor type can easily be defined.

Multiple physically close sensors are grouped and assigned a specific identifier. Depending on the prototyped architecture, such a group can represent a single CPU, one special hardware component or a cluster of related processing elements.

3.2 Tree-based dataflow for monitoring data

In a large-scale prototype, data sources of sensors are distributed throughout the entire hardware design. We propose to use a hierarchically structured interconnect tree, which can be freely scaled to match the prototyped design. All intermediate components are fully buffered to avoid long paths, improving timing and allowing timing optimization to focus on the actual DUT. The data-flow is primarily from leaves (sensors) to the root node with a configurable width. However, a reverse path is also provided for configuration of the sampling component, i.e. setting a specific sample interval or disabling a sensor entirely without rebuilding the bitstream. As the sensor and serializer nodes are rarely reconfigured, the reverse path is a single-bit serial interface.

On each fork, the muxing component (*mux*) handles merging the incoming branches of datsa. Figure 4 shows an example of the dataflow tree, based on the mux as central building block. The mux is designed to receive a full set of monitored sensors from one branch and proceed to the next after all sensor values have been forwarded. Therefore, a continuous data stream containing all groups from the first input, followed by all groups from the remaining inputs, is the output of the mux component, as shown in

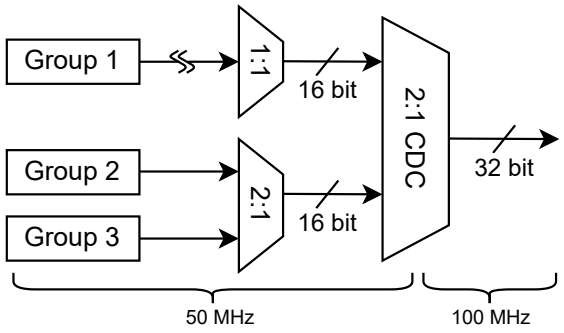

Figure 4: Example of a mux tree including a 1:1 mux and a clock domain crossing (CDC) mux at the root

figure 3. The packet header including a timestamp and a checksum is appended at the root node. A group descriptor contains a group ID to distinguish sensor with equal IDs but from different groups. Each group descriptor is followed by one record per sensor in that group, each record contains the sensor ID and its current value. The width of the ID and value fields are freely configurable, but should add up to a multiple of the bus width of the interconnect tree.

The mux tree features a backpressure signal generated by the output module. This signal is buffered within each mux to avoid a long-distance wire in the design. To support data transfer in every clock cycle, the mux component is able to buffer one data word in case of backpressure. Multiple mux instances can thus be connected in series to adapt to the layout of sensor nodes in the prototyped design.

A special case is a single-input mux: there is no need for arbitration, however all input and output signals are still buffered. Such a mux can be inserted into the dataflow to improve timing of the monitoring system on large designs by reducing the critical path length and allowing for flexible placement of the intermediate mux registers by the design tools.

Lastly, a mux can also be configured as a data width converter including potential clock domain crossing. This counters the fact that the root datapath limits the bandwidth of the whole tree structure by increasing both clock and data width by a power of two. For example, a 8:1 clock domain crossing (CDC) mux with a 16-bit input at 50 MHz can output a stream of 64-bit at 100MHz, having a data rate of 800 MiB/s on both sides. However, this comes at a higher hardware costs for FIFOs on each input and should only be considered if the full monitoring system cannot run at the desired data rate of the host interface.

3.3 Output module

The output module is at the root of the dataflow tree and receives the completed stream of monitoring from all attached nodes. This stream is prepared and transmitted to an external system for further analysis and storage. Several types of output modules can be implemented, depending on the I/O interfaces of the target system. As most FPGA platforms provide an Ethernet interface, the standard output module is comprised of an hardware TCP/IP stack to send monitoring data encapsulated in UDP packets. Figure 5 illustrates the structure of the hardware TCP/IP and Ethernet stack,

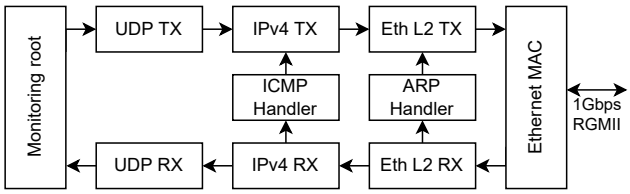

Figure 5: Architecture of the hardware Ethernet stack including ARP and ICMP modules

including ARP and ICMP handlers for standard IPv4 compatibility. The Ethernet output module packetizes the data stream, adds checksums for error detection, and forwards it to the UDP module, which sends the data to a configurable IP address. Other output modules can be implemented to use different interfaces. In any case, the output module can also apply backpressure to the dataflow tree to throttle incoming monitoring data if the I/O interface bandwidth is insufficient. Section 5 provides both an estimate and real-world measurements of the required bandwidth.

3.4 Software stack for evaluation

To generate valuable insights on the execution of the monitored system, we have to store and preprocess the collected data. A commercial-of-the-shelf (COTS) PC system can be used to receive monitoring data and store it for further processing. We provide a software stack to handle incoming monitoring data via Ethernet including checksum verification which stores each sample in a JSON format. This data can then be further processed according the user's needs. Depending on the output module in use, it can be necessary to reassemble packetized data which is done as one of the first steps of the receiving process.

For example, we provide a setup using a time-series database and a web application. Incoming monitoring data is stored in the database according to the hardware timestamp, allowing for offline processing of the captured data after execution. The web application can display live data during execution, but also allows to explore previous runs based on data stored in the time-series database.

4 PROTOTYPE IMPLEMENTATION

The proposed monitoring system has been integrated in a tiled manycore system on a multi-FPGA platform, based on four Xilinx Virtex-7 XC7V2000T FPGAs. The manycore prototype is a tiled 4x4 architecture, connected using a meshed NoC. Each tile contains five LEON 3 processors and 8 MiB of local memory. Peripherals include a network interface to access the NoC with a level 2 cache to improve read performance from the remote, shared DDR memory. Both the tiles and the NoC run at 50 MHz, while the DDR memory is clocked with 200 MHz. Debugger access is available for each tile separately including serial redirection. However, getting runtime performance data outside of the system is still a tricky task: the serial console performance is very limited and collecting metrics in software uses a considerable amount of CPU resources, while sending the results to the I/O interface creates additional NoC load.

To show how an existing system can benefit from our approach, we mapped the proposed non-intrusive monitoring system to this

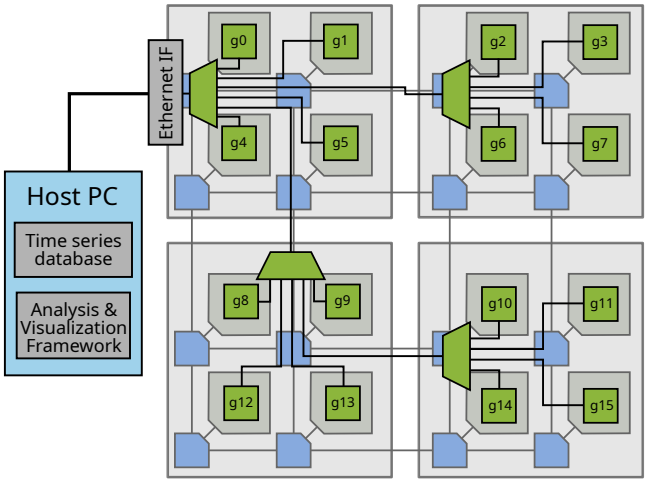

Figure 6: Proposed monitoring system mapped to a 4x4 tiled manycore architecture on a quad FPGA prototyping platform with 16 sensor groups connected over FPGA boundaries

manycore prototype. Figure 6 shows the 16 tiles of the manycore prototype with one group of sensors per tile (g0 to g15), as each tile generates similar monitoring data locally. Since a multi-FPGA platform is used, the design is partitioned into four quadrants, resulting in four tiles per FPGA. Monitoring is first gathered from all four tiles in one quadrant before being sent over high-speed serial inter-FPGA links. Only one of the FPGA features a Ethernet interface, thus the inter-FPGA links are used to connect the remaining FPGAs to the root node. To ease timing, a single-input mux is placed immediately before and after each inter-FPGA SERDES driver.

To show the broad range of applications, different sources of metrics are being monitored, including:

- Eight assignable performance counters per CPU, tracking I/D-cache misses, instruction counts, branch prediction and interrupt statistics.
- NoC transfers by type and buffer levels of the attached router.
- Usage of the tile-local AHB bus by type.
- Power consumption and temperature as estimated by the integrated power emulation system [9].

CPU and NoC performance counters as well as power and temperature are value-based sensors, as these values are already present in registers. For bus usage and interrupt statistics, event-counting sensors are used. In total, 60 sensors are monitored per tile, summing up to 960 sensors for the whole system.

Two implementations of the root node are provided: one using an Ethernet connection and another using a proprietary interface of the FPGA prototyping platform. The Ethernet root node makes it very easy to connect the processing host, but it is limited to 1 gigabit bandwidth. The MMI64 system is a custom bus system used within proFPGA [5] ASIC prototyping systems, which is used to program the FPGAs but also for debugging access to the tiles during runtime. It supports several transports among which PCIe 2.0 x8 offers the highest performance. This transport can be used if the Ethernet connection is available, however it does not offer significantly better performance due to the internal architecture of the MMI64 bus system.

Both the root node's interface to the interconnect tree and the one to the host PC can be bottlenecks in the system. Section 5 provides an estimate on the achievable sampling rate, assuming the host interface does not limit further. Apart from maximum bandwidth, the number of packets per second can be a limitation, especially for configurations with a high sample rate. Therefore both the Ethernet and MMI64 root nodes combine monitoring data of two consecutive sample points if possible to lower the packet rate and maximize bandwidth.

On the receiving host, a combination of InfluxDB [7] and Grafana [8] is used to store incoming monitoring data and handle visualization during and after execution. In addition to the time-series database, live monitoring data is streamed to the Grafana instance directly during execution to display it as soon as it is received from the DUT. This allows us to review the system's behavior directly during runtime and verify the correctness of prototype runs quickly. Detailed post-execution analysis can be done either using the provided web interface or via any other compatible database client on the same granularity as originally produced. Monitoring data, both captured live and stored in the database, can be saved in JSON format using the included tools for further analysis.

5 EVALUATION

We evaluated the maximum possible sample interval of our approach in theory and verified this result using the case study introduced in section 4. In addition, an overview of the resource consumption of our proposed system in this configuration is compared to the manycore prototype. We also measured the impact of the data reduction algorithm when running a parallel benchmark suite on the manycore system. Finally, some specific use-cases of our proposed system for debugging are highlighted.

The maximum sample interval of the system depends on several factors: the number of enabled sensors, the width of data and sensor ID values and the bandwidth of the root node. To give a theoretical estimate for the maximum sample interval, a system with N_{grp} sensor groups having N_{sens} sensors is assumed. Each sensor is configured with a w_{id} bytes identifier and a w_{data} bytes value, while the interconnect tree has a width of w_{bus} bits and runs at the frequency f_{bus}. The header size w_{hdr} and the size of each group descriptor, equal to a sensor, is also included in the following formula:

$$\Delta t = \frac{w_{\text{hdr}} + N_{\text{grp}} \cdot (1 + N_{\text{sens}}) \cdot (w_{\text{id}} + w_{\text{data}})}{w_{\text{bus}} \cdot f_{\text{bus}}}$$

Our reference implementation uses 32-bit sensor values and 16 groups with 60 sensors each. This results in a theoretical maximum sample rate of 58.6 µs when running the design at 50 MHz with a 16-bit data bus. The theoretical maximum can still limited by the interface to to the host interface, which needs to support the bandwidth of 100 MiB/s in this case. Measurements on the FPGA prototype using gigabit Ethernet show an average bandwidth of 99.4 MiB/s for this configuration, missing about one single sample per second as the bandwidth does not fully suffice.

Table 1: Resource usage of the full system and individual components, portion of the full prototype design in brackets

Component	FF	LUT	BRAM
Sensor group (60 sensors)	1507 (0.4%)	707 (0.1%)	0
6:1 mux	82 (0.01%)	82 (0.01%)	0
Ethernet root module	4066 (0.9%)	2350 (0.4%)	6 (0.7%)
MMI64 root module	1651 (0.4%)	1249 (0.2%)	6 (0.7%)
Full monitoring system (single FPGA w/ MMI64)	7858 (1.9%)	4428 (0.7%)	6 (0.7%)
Quadrant of full design (1 FPGA, 4 tiles, 20 CPUs)	*423665*	*652652*	*879*

Table 1 lists the FPGA resource usage of the monitoring system including individual components. It focuses on one quadrant of the prototype as shown in figure 6 on one FPGA, containing four tiles with 5 CPUs each. As it contains the root node, a 6:1 mux is used to merge all four local sensor groups with incoming data from the other design qudrants.

The data reduction feature of the sampling components can greatly reduce the required bandwidth. The NAS parallel benchmark suite [2] contains a set of benchmarks for distributed systems which are also suitable for manycore prototypes. For example, running the BT test (block triagonal solver) allows data reduction of monitored data by 85.7% when monitoring 60 sensors per tile as described in section 4. This is due to algorithms used in the BT test, which contain only short communication phases and most time is spent in the computational kernel. Therefore sensors for NoC traffic, temperature and peripheral operations are rarely changed and transmitted. This is however dependent on the application and monitored values, as for other benchmarks data reduction rates of 15% have been measured. The maximum theoretical bandwidth should be supported in any case, otherwise samples could be dropped. The data reduction feature can improve the user experience if a shared interface like the MMI64 system is used, as latency for programming and debugging tools is reduced when the full bandwidth is used at all time.

We want to stress that the main benefit of this work, apart from the low resource requirements and easy applicability, is the debugging flexibility this system opens up. The software stack includes an easy to use, yet powerful tool to visualize the acquired measurements and quickly inspect the run-time behavior of the DUT, even already during execution. The monitoring system helped to successfully detect several bugs throughout hardware modules, operating system and applications and allowed to locate the actual error sources. As an example, we noticed an irregular deviation in the power & temperature emulation system [9]. Unaware of the actual cause of this issue, the emulation system itself was investigated first. However, the monitoring system indicated that the issue only occurs close to the border between two FPGAs of the prototyping system. We could therefore quickly identify that the inter-FPGA SERDES was configured incorrectly. A second example was a rare issue in the scheduling system of our operating system. Specific benchmarks were showing degraded performance on some specific tiles on our manycore system. The monitoring system immediately made clear that for some time periods, tasks were not scheduled to some of the processors. The related scheduling bug was quickly solved. Apart from debugging purposes, the non-intrusive monitoring system has also been used to extract training data for neural networks to identify suspicious side-channel communication.

6 CONCLUSION

In this paper, we described a monitoring infrastructure for FPGA-based hardware prototypes. It is designed to be scalable for both homogeneous architectures like manycores as well as heterogeneous designs by supporting unbalanced sensor trees with asymmetric bandwidth requirements. Different kinds of value- and event-based sensor types have been discussed, providing an interface to commonly used data sources. Most importantly, the monitoring system can operate fully non-intrusive if required, so that the behavior of the prototyped system is not altered. As the system resides on the same platform as the DUT, it uses some of the available hardware resources and timing budget. However, the resource overhead has been kept as low as possible by using a lightweight interconnect and specialized mux components can be used to improve timing on complex designs. A standard gigabit Ethernet interface, which is present on most of todays FPGA prototyping platforms, is sufficient to greatly increase the observability of the DUT and gain useful insights at runtime. In a case study with a 4x4 tiled manycore architecture, a sample set with 960 measurements could be acuired with a sample rate of up to 59µs. The non-intrusive monitoring system made several errors throughout the prototyped platform, in hardware components, operating system and applications, visible and allowed us to detect the source of the error more quickly. In this paper, we presented the potential of the proposed monitoring infrastructure for future MPSoCs and manycore prototypes and highlighted the advantages for users of FPGA-based systems.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) as part of the *Transregional Collaborative Research Center Invasive Computing [SFB/TR 89]*.

REFERENCES

[1] Ayaz Akram and Lina Sawalha. 2016. x86 computer architecture simulators: A comparative study. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 638–645. https://doi.org/10.1109/ICCD.2016.7753351
[2] David H. Bailey. 2011. *NAS Parallel Benchmarks.* Springer US, Boston, MA, 1254–1259. https://doi.org/10.1007/978-0-387-09766-4_133
[3] Nathan Binkert et al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (8 2011), 1–7. https://doi.org/10.1145/2024716.2024718
[4] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. *Runtime Verification for Decentralised and Distributed Systems.* Springer International Publishing, Cham, 176–210. https://doi.org/10.1007/978-3-319-75632-5_6
[5] Siemens Electronic Design Automation GmbH. 2022. *proFPGA FPGA Prototyping System.* http://www.profpga.com/
[6] Augusto Hoppe, Jürgen Becker, and Fernanda Lima Kastensmidt. 2021. High-speed Hardware Accelerator for Trace Decoding in Real-Time Program Monitoring. In *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*. 1–4. https://doi.org/10.1109/LASCAS51355.2021.9459137
[7] InfluxData Inc. 2022. *InfluxDB Times Series Data Platform.* http://www.influxdata.com/
[8] Raintank Inc. 2022. *Grafana: The open observability platform.* http://www.grafana.com/
[9] Alexandra Listl et al. 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).* https://doi.org/10.1109/IOLTS.2018.8474284

[10] Leonard Masing, Fabian Lesniak, and Jürgen Becker. 2021. A Hybrid Prototyping Framework in a Virtual Platform Centered Design and Verification Flow. *IEEE Embedded Systems Letters* 13, 1 (2021), 1–4. https://doi.org/10.1109/LES.2020.2995084

[11] Olga Melnikova, Irina Hahanova, and Karina Mostovaya. 2009. Using multi-FPGA systems for ASIC prototyping. In *2009 10th International Conference - The Experience of Designing and Application of CAD Systems in Microelectronics*. 237–239.

[12] Cesar Sanchez et al. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods in System Design* 54, 3 (01 11 2019), 279–335. https://doi.org/10.1007/s10703-019-00337-w

[13] Dimitry Solet, Sebastien Pillement, Jean-Luc Béchennec, Mikael Briday, and Sebastien Faucou. 2018. HW-based Architecture for Runtime Verification of Embedded Software on SoPC systems. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 249–256. https://doi.org/10.1109/AHS.2018.8541459

[14] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *2015 IEEE Symposium on Security and Privacy*. 55–69. https://doi.org/10.1109/SP.2015.11