In situ Latency Monitoring for Heterogeneous Real-time Systems

Martin Geier, Tobias Burghart, Martin Hackl and Samarjit Chakraborty
Chair of Real-Time Computer Systems
Technical University of Munich
geier/burghart/hackl/chakraborty@rcs.ei.tum.de

***Abstract*—With the increasing complexity of both application software and the underlying hardware platforms found in current Real-Time Systems (RTSs), static timing analysis methods are struggling to capture the wide variety of delays. In case of real-time control systems, it thus is common practice in industry to measure system latencies over extended periods of time to ensure compliance with predefined control deadlines that specify the maximum permissible delay between the arrival of a sensor value and the transmission of an actuation signal. As such monitoring is commonly implemented using code instrumentation, it not only requires modification of the RTS-under-test, but also is unable to capture delays caused by I/O peripherals or other components that contribute to the end-to-end (i.e., input-to-output) latencies.**
In this paper, we propose a latency monitoring methodology for current heterogeneous RTSs that combine a fixed-function System-on-Chip (SoC) with configurable FPGA fabric. The proposed extension enables an RTS based on such a Programmable SoC (pSoC) to perform *In situ Monitoring* (i.e., without software modification or external hardware) of end-to-end latencies including the I/O delays of current interfaces such as Gigabit Ethernet. We present *I/O-to-Fabric Redirecting* (to tap into the I/O paths of a pSoC-based RTS) combined with a *Trigger/Binning Subsystem* (to identify sensor/actuation signals and perform online execution of both latency calculation and histogram generation), for which we propose two implementation options. Our experimental evaluation of the software-driven Trigger/Binning Subsystem shows that our proposed methodology is capable of capturing latency variations over extended periods with sub-microsecond accuracy.

I. Introduction

Many of today's embedded systems operate under predefined latency constraints to ensure a timely reaction to sensor or user inputs. Although widely used, the design, implementation and verification of such Real-Time Systems (RTSs) remains challenging due to a number of reasons – partly depending on the particular application domain. In case of real-time control systems, for instance, latency constraints arise in terms of control deadlines, i.e., the maximum time available to the RTS for input signal acquisition, execution of the control algorithm and transmission of actuation signal [1]. Depending on both the controller's scheduling parameters (i.e., sampling rate and deadline [2]) and the RTS's processing speed, as little as a few ten thousand CPU cycles are available per control iteration.

During this time, however, not only the actual control computation but also various other software tasks have to be executed. This includes both auxiliary applications and time-critical portions of the Operating System (OS) like task scheduler, inter-process communication (IPC) subsystems and Input/Output (I/O) device drivers that contribute to the controller's response time measured from signal input to output. The runtimes of all those involved software tasks are affected not only by their individual computational complexity, but also by context switches caused by preemption [3] and, more specifically, external interrupts. The former are commonly triggered by an internal timer interrupt or explicit preemption points in kernel or user code. External interrupts, on the other hand, are crucial to react to asynchronous events (such as, e.g., data reception or transmission by I/O peripherals) in a timely fashion whilst avoiding the overhead of continuous polling.

As a further complication, the timing behavior of the underlying Commercial Off-The-Shelf (COTS) hardware platforms has become more and more unpredictable over the last decades due to various measures aimed at increasing performance and flexibility. Even before the widespread adoption of multi-core systems, stateful CPU features such as pipelining, caches, out-of-order and speculative execution started to complicate timing analysis of software workloads as a single instruction could no longer be mapped to a constant number of CPU cycles [4], [5]. All above features, however, amplify the impact of external interrupts that asynchronously – and thus unpredictably – modify program flow and resulting execution times [6], [7].

Apart from those effects arising within an individual core, additional uncertainty stems from resources such as memory subsystems, interconnects and I/O peripherals that are shared amongst multiple bus masters. This holds true for both single-core systems (due to additional bus masters) and multi-core systems where two or more cores share at least one level of cache hierarchy [8]. Although the behavior of the CPU cores (and the resulting interference on shared resources) might still be under control of – or at least visible to – the developer to some extend, that of externally triggered bus masters (e.g., the DMA engine of an Ethernet MAC) is mostly beyond reach.

Whilst not all of these factors apply to all architectures, the resulting unpredictable timing behavior of complex hardware platforms, the large number of potential execution paths and the vast input data sets render *(static) timing analysis* infeasible in practice [8], [9], [10]. Hence, it is common practice in industry to *measure execution times* over extended periods [5]. This approach, however, inherently only yields weak *maximum observed* execution times in contrast to upper timing bounds from static analysis. Due to limited observability, the former only are underestimates of the actual Worst-Case Execution Time (WCET) whilst the latter represent safe upper bounds. Despite this, measurement-based methods are widely deployed in safety-critical domains such as automotive and avionics, as the resulting *high watermarks* (capturing the longest observed execution time) not only approach the actual WCET over continuous system operation, but can also be used to drive probabilistic timing analysis [11], [9]. In addition, real-time operating systems (e.g., Linux with PREEMPT_RT) come

with extensive tracing facilities to gather statistics on scheduling events, execution times and various other tracepoints during runtime [10]. As those software-based methods only rely on code instrumentation (and an accurate time source within the system), they can be integrated on any architecture with relatively low runtime overhead. Their coverage, however, is inherently limited to event chains entirely visible to the CPU(s) of the system, which excludes delays caused by components such as I/O peripherals, interconnects or memories that contribute to the *end-to-end latencies* of the overall RTS.

In this paper, we propose a latency monitoring methodology for current heterogeneous RTSs that combine a fixed-function System-on-Chip (SoC) with configurable FPGA fabric. The proposed approach/extension enables an RTS based on such a Programmable SoC (pSoC) to perform *real end-to-end latency monitoring* ranging from arrival of a sensor value up to the transmission of an actuation signal at the respective boundaries of the system. To this end, we present a combination of *I/O-to-Fabric Redirecting* and an either software- or hardware-driven *Trigger/Binning Subsystem* as generic methods to tap into the otherwise unreachable I/O paths of a pSoC-based RTS, identify sensor and actuation signals and determine the actual end-to-end latencies entirely *in situ*, i.e., both without external equipment and during regular, unmodified system operation.
With I/O-to-Fabric Redirecting, the external interface signals of most hardcore I/O peripherals (located in the pSoC's fixed-function regions) can be redirected to the configurable FPGA fabric. On the two major pSoC architectures (i.e., Xilinx Zynq and Altera Arria), this holds true for both point-to-point and shared bus interfaces ranging from simple UART via I2C and CAN to fully-fledged Gigabit Ethernet – with USB being the only exception due to dedicated I/O pins. Once the interface signals in question are within the FPGA fabric, they are not only routed to the respective external PHYs, but also tapped for additional analysis by our Trigger/Binning Subsystem, for which we propose two implementation options that differ in flexibility, supported interface speed and development effort. Whilst the first uses COTS IP cores to analyze and timestamp the interface signals and perform the required latency binning in software, the latter relies on interface-specific trigger blocks driving a generic, reusable timestamping and binning engine (all implemented as custom IP cores). In case the interfaces of interest are implemented in the FPGA fabric, their signals can directly be captured. For the – continuously increasing – number of hardcore I/O peripherals, however, prior I/O-to-Fabric Redirecting is required for subsequent histogram generation.
We utilize our proposed methodology to analyze the input-to-output latency of a UART forwarding application (executing as a userspace process on Linux patched with PREEMPT_RT) on a Xilinx Zynq pSoC. Although initial experiments with the hardware-driven Trigger/Binning Subsystem show promising results for more complex interfaces, an in-depth discussion is beyond the scope of this paper due to space restrictions. Using the first implementation option, we show that our approach is able to identify latency variations caused by additional load on CPUs and interconnects within the RTS with sub-microsecond resolution. Furthermore, the online execution of both latency calculation and histogram generation enables end-to-end latency monitoring over extended periods. As our proposed solution resides entirely in the FPGA fabric and can operate without runtime configuration, it requires neither additional external hardware nor software modification of the RTS-under-test. Due to its standalone and modular design, it easily can be integrated with current pSoCs and flexibly adapted to most I/O triggers, which opens up many possibilities for future work.
In summary, the main contributions of this paper are

- *End-to-end Triggering* for most hardcore I/O peripherals of current pSoCs using *I/O-to-Fabric Redirecting* ranging from sensor input to transmission of actuation signal and
- high accuracy long-term *In situ Latency Monitoring* using a flexible software-driven *Trigger/Binning Subsystem* that relies on COTS IP cores for I/O interfacing.

The rest of this paper is organized as follows. Related work is presented in Sec. II. Next, we introduce both traditional and the targeted heterogeneous hardware platforms with their various sources of latencies (Sec. III). Based thereon, Sec. IV presents our proposed methodology for in situ latency monitoring using I/O-to-Fabric Redirecting (Sec. IV-A) and then describes both Trigger/Binning Subsystem (Sec. IV-B) and optional Stimulus Generation Unit (Sec. IV-C). Sec. V shows an experimental evaluation of both modules analyzing the end-to-end latency of a Linux-based RTS. We finally conclude our work in Sec. VI.

II. Related Work

Although purely software-based methods are widely available and used to generate single watermarks, entire histograms and – within processing and memory limits of the RTS-under-test – finite event traces, their coverage is limited to events visible to the CPU(s). This, e.g., holds true for evolved and versatile tracing subsystems found in Linux, such as ftrace that has its roots in work improving the kernel's real-time behavior [12] and supports many trace types ranging from internal execution flow (with countless in-kernel functions available as trigger sources) and scheduling latencies [10] to (hardware-related) delays caused due to the CPU(s) temporarily operating in System Management Mode [13]. As the latencies added by system components such as I/O peripherals and their interaction with intermediate subsystems are beyond reach of those methods, their applicability for end-to-end latency monitoring is limited.

One particular class of latencies close to the boundary of hardware and software has received special attention due to its impact on both scheduling and I/O delays. The time required by a CPU (and any upstream interrupt controllers) to react to an interrupt request (IRQ) sent by an internal or external peripheral defines the system's *IRQ latency*. Together with the time required for subsequent scheduling and context switching, it yields the total *wakeup latency* measured from an IRQ until the associated software task resumes its execution. As this latency is present for both internal (such as timer) and external interrupts (of, e.g., I/O peripherals), it not only affects tasks executing periodically, but also those communicating with the external environment. On Linux, it is commonly approximated using the `cyclictest` tool that measures the offset between cyclically programmed (and thus well-known) timer interrupts and reception of the subsequent timeout signals. Although the tool thus allows OS developers to evaluate the performance of the kernel without an external interrupt source, it not only is unable to capture any delays on the outer I/O paths, but also creates significant load on the CPU(s) itself and therefore can not be used in conjunction with actual real-time applications.

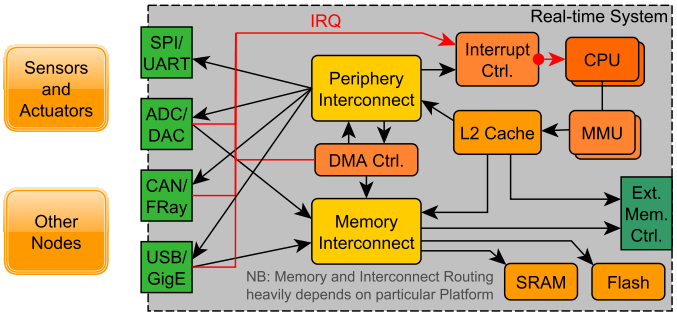

Fig. 1. Typical real-time control system: Physical devices (left) and RTS with generic memory architecture (center) and I/O interfaces (at left RTS boundary)

Whilst the latter issue to some extent can be avoided using ftrace [10], measuring the delays caused by I/O peripherals and downstream subsystems inherently requires tapping the I/O signals at the system boundaries for accurate timestamping. [14] proposes a method for incoming sensor signals that relies on connecting an internal capture/compare timer to an existing data ready signal of the external device (i.e., an SPI ADC). By configuring the counter for free-running mode with edge-triggered capture, the latency between the ADC asserting its ready signal and the execution of the associated interrupt handler (or any subsequent task) can be measured and stored. This requires not only an external device that produces a suitable (i.e., edge-based single-bit) trigger signal, but also software adaptions to the RTS-under-test to manage the timer.

Two fully hardware-driven methods, however, enable latency monitoring without continuous software interaction. OSADL's standalone *Latency Box* [15] supports long-term latency monitoring with online histogram generation on up to two independent channels. Whilst it is commonly used to validate new releases of the PREEMPT_RT patch for Linux on many different architectures, it not only is limited to single-bit, edge-based interfaces, but also – being an external device – is unsuitable for wide deployment along most RTSs out in the field. On the other hand, vendors of Field Programmable Gate Arrays (FPGAs) and pSoCs complement their hardware devices with a selection of *trace/debug IP cores* (e.g., Xilinx ILA and Altera SignalTap) that can be instantiated within the configurable FPGA fabric. Those softcores effectively add a logic analyzer to the system that can be connected to any signal within the fabric and stores the corresponding signal traces in internal memory, where they can be downloaded for offline analysis. Whilst this method is able to timestamp more complex interfaces signals due to programmable trigger conditions, its trace-based nature disallows long-term monitoring.

In comparison, our proposed methodology is capable of *end-to-end* latency monitoring (as it taps into the I/O paths at the boundaries of the RTS) and requires *no software modifications* (as it operates independently of the system's CPUs). Furthermore, its histogram-based nature enables *long-term monitoring* whilst the two implementation options of the Trigger/Binning Subsystem allow to balance development effort, flexibility, resource overhead and level of support for capturing complex interface signals (beyond single-bit, edge-sensitive triggers). As I/O-to-Fabric Redirecting can be deployed on both major pSoC architectures, it enables *in situ* latency monitoring out in the field for an increasing number of heterogeneous RTSs.

III. RTS Architectures and Latencies

In a typical real-time control system (Fig. 1), a multitude of components contributes to the performance-critical delay between sensing and actuation. In this paper, we focus on the overall input-to-output latency measured from arrival of a sensor value to transmission of an actuation signal at the boundary of the RTS (Sec. III-A), excluding potential sensing, communication and actuation delays. The device-under-test is a heterogeneous RTS based on Xilinx' Zynq pSoC (Sec. III-B) as one of the two major pSoC architectures currently available.

A. I/O, Interconnect and Software Latency Sources

Once an incoming sensor signal has traversed its communication media and, if required, an external PHY (both not shown in Fig. 1), it arrives at the particular I/O interface of the RTS. The associated integrated I/O peripherals can be divided into two major groups based on their type of internal interface. *Slave-only devices* such as many traditional serial (bus) controllers (for, e.g., SPI/I2C/UART or CAN/FlexRay) are managed by the CPU(s) for both configuration and data transfer. *Master devices* such as high-speed ADCs, USB or Ethernet controllers, however, are capable of performing Direct Memory Access (DMA) transfers to and from various on- and off-chip memories without any CPU intervention or knowledge. This not only is reflected in the interconnect architecture (by linking master devices to relevant memories by means of, e.g., a memory interconnect), but also creates additional, even harder to predict latencies as interconnects and memories are no longer under exclusive use by the CPU(s). In contrast to a simple slave device (which requires a CPU to fill/free its internal data buffers), an I/O peripheral with DMA capabilities performs the necessary memory transactions independently. Whilst this behavior is beneficial for both throughput and energy efficiency of the system, it may cause congestion on interconnects and memories and thus subsequently delay CPU transactions. Independent of the type of transfer, storing the incoming sensor signal thus causes an *input store latency*.
In addition to their internal bus interface(s), most I/O devices are equipped with one or more IRQ lines to inform the CPU(s) of pending work. Once asserted and accepted by the interrupt controller, the later causes the CPU assigned to the respective device to perform a context switch and execute the corresponding interrupt handler. The time taken between assertion and execution defines the *IRQ latency*. In case of a control application waiting for the sensor signal, it – in combination with the time taken for subsequent scheduling and a further context switch – also contributes to the *wakeup latency* covering the delay between IRQ and application execution.
Once the data has reached its particular receive buffer in memory and the application has been woken up, the latter commonly reads the input signal, performs the required processing steps and computes an actuation signal. The application's *execution time* contains not only the actual processing delays but also memory access latencies for both instruction fetching and memory transfers of input, intermediate and output data.
Once the actuation signal has been computed, it is placed in an output buffer and control is handed back to the OS (task suspension via, e.g., a syscall), resulting in an *output enqueue latency* (caused by context switch and output device driver).

Finally, the actuation data is transferred to the I/O device (using CPU or DMA) to leave the RTS after an *output latency*.

It should be noted that the resulting input-to-output latency (comprising above delays) can be increased by both additional load (e.g., other interrupt handlers and higher-priority tasks) on the CPU(s) and congestion on the downstream devices (e.g., interconnects and memories) shared with other masters.

B. Zynq as pSoC Platform for Heterogeneous RTSs

Xilinx' Zynq pSoCs join a fixed-function hardcore Processing System (PS) with an FPGA-based Programmable Logic (PL) fabric [16]. The PS alone implements a traditional embedded system with two ARM Cortex-A9 CPUs, shared L2 cache and various peripherals (e.g., memory and I/O controllers). Each core deploys features such as branch prediction, out-of-order and speculative execution, dedicated L1 instruction/data (I/D) caches and NEON SIMD execution units to increase performance. Whilst a Memory Management Unit (MMU) inside each core enables both memory protection and address translation, a downstream Snoop Controller ensures coherence between the L1D caches of the two cores. In addition, it links the CPUs to both On-Chip Memory (OCM) and the shared L2 Cache, which subsequently splits the remaining transactions between off-chip DDR memory and a Slave Interconnect.
The latter integrates various I/O peripherals that range from simple slave-only devices (e.g., UARTs and GPIOs) to the status and configuration registers of complex high-speed cores. Such master devices additionally are connected to the Central Interconnect and serve both external storage (e.g., SDIO) and I/O interfaces such as Gigabit Ethernet (GigE). Whilst the former might not directly contribute to the input-to-output latency (as all application data might reside within volatile memories), it might still cause interference due to unrelated DMA traffic. This also holds true for I/O interfaces with integrated DMA capabilities such as GigE – which, in addition, might be used to transfer sensor and/or actuation data. As shown in Fig. 2, the external interface signals of most I/O peripherals in the PS can either be connected to I/O pins dedicated to the PS (MIO) or routed to the PL by means of the EMIO interface.
In addition to all these I/O peripherals, the Slave Interconnect interfaces PS to PL via two general-purpose master ports (M-GP) that enable CPUs and other PS masters to access slave PL resources. In the opposite direction, two GP slave (S-GP) ports, four high performance (HP) ports and one Accelerator Coherency Port (ACP) are available for PL to PS accesses.

Due to their complex internal architectures combining state-of-the-art multi-core CPU subsystems containing various interconnects and memories with freely configurable FPGA fabric, current pSoCs thus come with even more sources of latencies than traditional RTSs (as introduced in Sec. I and Sec. III-A).

IV. Proposed System Architecture for In situ Latency Monitoring of Heterogeneous RTSs

As end-to-end latency monitoring requires tapping a system's outer I/O paths, the number of design steps to integrate our proposed solution varies with the specific setup of RTS and external environment. At first, all signals of interest have to reach the configurable FPGA fabric which – in case any hardcore I/O peripherals within the pSoC's fixed-function regions are used – requires prior I/O-to-Fabric Redirecting (Sec. IV-A).

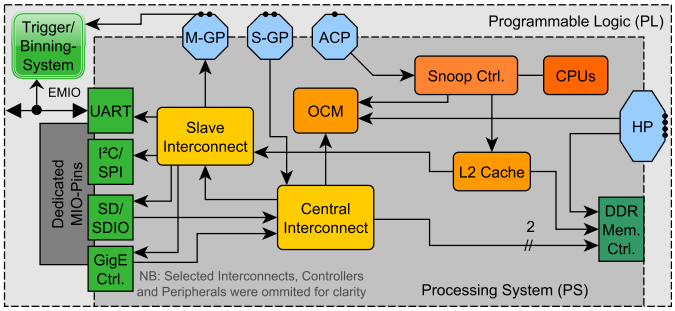

Fig. 2. Zynq pSoC: PL with redirected I/O signals and our Trigger/Binning Subsystem (left), PS I/O peripherals (at left PL/PS boundary) and PS (center)

The Trigger/Binning Subsystem (Sec. IV-B) taps the signals to generate the desired latency histogram (Sec. IV-B1), which – if desired by the system designer – can be both controlled and retrieved by the RTS-under-test during idle periods (Sec. IV-B2). In case external sensors are unavailable, an optional Stimulus Generator (Sec. IV-C) enables standalone system analysis.

A. I/O-to-Fabric Redirecting

The I/O structures of both major pSoC architectures share an explicit split of their I/O pins into two general classes. Whilst a large number of I/Os is available at the configurable FPGA fabric, a comparatively small number of pins is dedicated to the I/O peripherals of the device's fixed-function SoC region.

All (but one) Zynq devices implement a total of 54 dedicated Multiplexed I/O (MIO) pins within the fixed-function PS. The integrated I/O peripherals, however, drive over 300 internal interface signals. As thus only a small fraction can be connected to external devices via the MIO pins, most of the remaining signals can be routed to the PL using the Extended Multiplexed I/O (EMIO) interface at the PS/PL boundary (see Fig. 3). The required signals then can be linked to the regular I/O pins of the PL, of which there are up to 400 depending on the device.

Altera's pSoCs further divide the up to 94 I/O pins assigned to the fixed-function Hard Processor System (HPS) into dedicated and shared/loanable pins. Whilst the former (analogously to Zynq's MIO) only map to HPS peripherals, the latter can also be driven by the FPGA region (in groups of 12 pins each). In addition and as with Zynq's EMIO, the signals of most HPS peripherals are available in the configurable FPGA fabric, too.

These architectural features enable us to tap into the outer I/O paths in front of the (H)PS peripherals in case the external devices (such as a PHY or transceiver) are connected to regular FPGA or (on Altera devices only) shared/loanable HPS I/Os. Whilst this can easily be achieved for slower interfaces (e.g., GPIO, UART, SPI, I2C or CAN) by appropriate wiring, it might pose an additional challenge during PCB design in case of high-speed peripherals such as Gigabit Ethernet due to added signal integrity issues. Fig. 3 exemplifies this redirecting for two serial interfaces realized using a PS UART each with the external devices connected to the PL I/Os. Instead of being multiplexed to the MIO pins, the four receive/transmit signals pass through the PL and are tapped by the Trigger/Binning Subsystem. If the relevant peripherals are implemented in the PL, their signals are directly tapped without prior redirecting.

On Xilinx devices our proposed methodology is thus limited to interfaces attached to the regular PL I/Os. On Altera devices, their more flexible I/O structure even allows tapping external

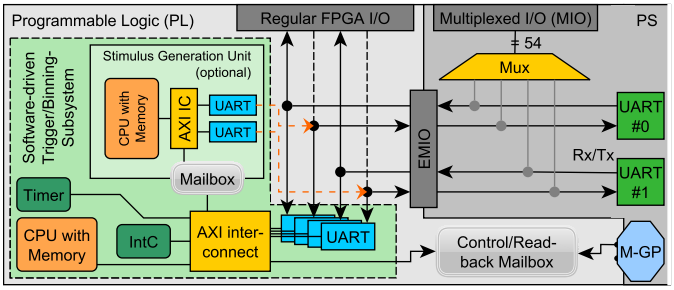

Fig. 3. I/O-to-Fabric Redirecting and internals of Trigger/Binning Subsystem

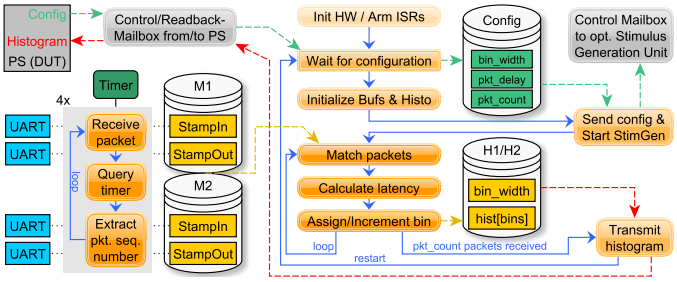

Fig. 4. Control and Data Flow within Firmware of Trigger/Binning Subsystem

devices connected to the shared/loanable pins – only those on the (few) dedicated ones remain out of reach. Although COTS Evaluation Boards commonly use (H)PS I/Os for onboard PHYs/transceivers (e.g., GigE/RS232) and thus are somewhat affected, it generally is advisable to use FPGA I/Os on custom boards anyway – irrespective of this limitation. As a primary benefit, they increase system flexibility in case of changes in the interfacing requirements such as timing adaptions (due to, e.g., replacement of the PHY) or protocol updates (that might require moving the affected peripheral to the FPGA fabric). In addition, using FPGA I/Os for communication interfaces frees (H)PS pins for other peripherals that cannot be redirected. On Zynq, e.g., this holds true for controllers managing external storage devices such as Flash (via QSPI) or SRAM (via SMC).

B. Software-driven Trigger/Binning Subsystem

The now available interface signals are then connected to the Trigger/Binning Subsystem for analysis. As the remaining design steps – in contrast to initial I/O-to-Fabric Redirecting – do not depend on structural details of the two pSoC architectures, we again focus on Xilinx' Zynq for further explanation.

1) Signal Capture and Histogram Generation: Fig. 3 shows our proposed solution in the PL and all its connections to the PS. Inside the Trigger/Binning Subsystem, one COTS IP core per interface of interest converts its signal(s) to AXI4(-Lite) which links to an AXI interconnect. The only master device in the subsystem is a simple COTS Microblaze CPU (without caches) whose firmware can be merged with the bitstream for automated boot after PL initialization. Volatile storage is provided by a dual-port Block RAM for both instructions and data. Its size can dynamically be adjusted to adapt to the memory requirements of the tracing application, which depend on complexity of the analyzed protocol, maximum number of outstanding latency measurements and histogram configuration (bin count and width). A mailbox connects the subsystem to the PS for optional synchronization and readback (Sec. IV-B2) whilst a timer serves as measurement time base. In case the receiving COTS IP cores have limited buffer capabilities, an interrupt controller enables timely reception by the subsystem.

The internals of the firmware executed on the Microblaze CPU are shown in Fig. 4. During measurements, the control flow (sketched in solid blue) is as follows. After initialization of interface IP cores and other peripherals, the subsystem waits for configuration data and the start command from the RTS-under-test. Once received, they are used to initialize internal data structures and forwarded to the optional Stimulus Generator. For each of the (in our case four) interfaces, the CPU then monitors the tapped signal and performs protocol-/application-specific packet identification (based on an available sequence number or other unique attributes such as checksums). Once generated, both the identifier and a timestamp are stored in a software FIFO (StampIn) and monitoring continues. As soon as the associated response of the RTS-under-test (StampOut) is found, both entries are removed from the two FIFOs after the resulting latency has been calculated and mapped to its associated bin that, finally, is incremented in the corresponding histogram data structure. Besides, it should be noted that the subsystem could easily be adapted to other setups by replacing the UARTs (used in our evaluation) with IP cores matching the various other supported interfaces such as I2C or CAN.

2) RTS-under-test Synchronization and Readback: In case numerous configurations (such as different scheduling parameters, OS or application versions) of the RTS-under-test should be evaluated, it is crucial to enable automated testing over extended periods. To generate separate latency histograms for each system configuration, at least some basic means of synchronization between RTS-under-test and Trigger/Binning Subsystem is required. Our solution relies on a mailbox IP core to enable the RTS-under-test to not only configure and initiate each measurement cycle, but also to download the captured latency data afterwards. This drastically simplifies the measurement setup, as the system designer no longer has to interact with the subsystem directly. Instead, the entire cycle can now be controlled from the known RTS-under-test.

This enables the system designer to determine the (typically entire) measurement process in advance using a shell script executed on the RTS. During runtime, it not only cycles through the various system configurations (by appropriate setup of the RTS-under-test), but also interacts with the Trigger/Binning Subsystem using two included userspace tools. Whilst one parameterizes the histogram data structure (e.g., bin width in μs) and arms the subsystem, the other fetches the results that then can be stored along with current system configuration and other performance data captured on the RTS-under-test itself.

C. Integrated Stimulus Generation Option

For long-term measurements across numerous system configurations or preliminary evaluations without external sensors, an Integrated Stimulus Generator can be added to drive the inputs of the RTS-under-test. This requires to disconnect the external inputs (dashed black wires in Fig. 3) to avoid conflicts with the outputs (dotted orange) of the Stimulus Generator. As it reuses the COTS IP cores to drive the signals of interest, its internal structure is similar to the Trigger/Binning Subsystem. An additional mailbox enables the latter to forward extended parameterization data (e.g., desired interface utilization and sample count) from the RTS-under-test for automated testing.

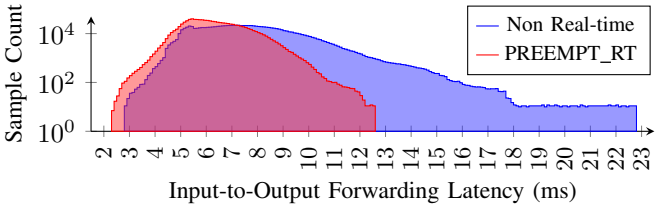

Fig. 5. I/O Latency Distribution with high Scheduling/IPC Load (hackbench): Min/Avg/Max latencies are 2.8/7.6/22.9ms (non RT) and 2.3/6.3/12.7ms (RT)

V. Experimental Evaluation

Our proposed methodology was originally devised to analyze the input-to-output latencies of an RTS that executes a complex userspace application managing the safety-critical communication between an Unmanned Aerial Vehicle and its ground station. The RTS resides on the serial link between onboard flight controller and telemetry downlink and is based on the MicroZed 7020 system-on-module. Apart from a 7Z020 Zynq pSoC with CPUs and DDR RAM operating at 666 / 533 MHz, the RTS integrates 1 GB RAM, various communication interfaces (ranging from UARTs to GigE and HDMI) and JTAG.

For sake of simplicity, the original application is replaced by a UART forwarder. We use our Stimulus Generator to send two serial packet streams (at 115,200 bit/s each) to the RTS-under-test, which, although reduced in size, still contain a sequence number suitable for identification. We evaluate the forwarding latencies using the Linux 4.4.0 kernel from Xilinx' repositories both without any (additional) modifications and with the latest PREEMPT_RT patch applied. To manage the two PS UARTs used in our setup, we rely on the included `xuartps` driver whilst a 100 MHz clock from the PS drives the Trigger/Binning Subsystem. In addition to the actual application (running at highest real-time priority), we execute the `hackbench` tool to simulate extreme load conditions on both CPUs and downstream subsystems (options `-g 4 -f 6`). For selected experiments, independent latency histograms were created using an external logic analyzer to confirm accuracy.

As shown in Fig. 5, the non real-time kernel exhibits a much more widespread latency distribution compared to the PREEMPT_RT version. Although the average latencies are relatively close (7.6 to 6.3 ms), the worst-case values differ by nearly a factor of two. In addition, the non real-time kernel suffers from large buffer backlogs that are responsible for the far right hand side of the histogram (from 18 to 23 ms). The patched kernel benefits from its more fine-grained preemption capabilities that result in lower minimal latencies. These fundamental characteristics were further observed in extensive tests covering several millions of packets per measurement.

It should be noted that idle load conditions result in much more similar latency distributions with nearly identical average values of around 4.9 ms. With the real-time kernel, however, minimum and maximum values are slightly higher ($\approx$ 0.3 ms).

VI. Conclusion

In this paper, we proposed I/O-to-Fabric Redirecting combined with a Trigger/Binning Subsystem as a novel methodology to enable in situ latency monitoring on current heterogeneous RTSs. By tapping the I/O paths of such pSoCs, we are able to capture sensor and actuation signals at the device boundaries to determine actual end-to-end (i.e., input-to-output) latencies crucial to the performance of real-time control systems. On the two major pSoC architectures, this redirecting is supported for most I/O peripherals in both fixed-function SoC and programmable FPGA regions. In contrast to trace-based methods, online latency calculation and histogram generation allow for long-term measurements required for sound analysis.

As the proposed *in situ solution* only requires adaption of the programmable FPGA fabric, both software and external hardware of the heterogeneous RTS-under-test can remain unchanged. Combined with its long-term monitoring capabilities, our approach thus is suitable for straightforward deployment.

Even though the current software-driven Trigger/Binning Subsystem performs well for slower interfaces, high-speed I/O peripherals are out of its reach due to limited processing power. As initial experiments with a hardware-driven version have shown promising results for Gigabit Ethernet, we intend to widen our evaluations towards fully-fledged networked RTSs.

References

[1] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, Jan 1994.

[2] I. Bate, J. McDermid, and P. Nightingale, "Establishing timing requirements for control loops in real-time systems," *Microprocessors and Microsystems*, vol. 27, no. 4, pp. 159 – 169, 2003.

[3] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[4] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[6] K. Sandstrom, C. Eriksson, and G. Fohler, "Handling interrupts with static scheduling in an automotive vehicle control system," in *Fifth International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, 1998.

[7] J. Regehr and U. Duongsaa, "Preventing interrupt overload," in *2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*.

[8] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 34:1–34:25, Jan. 2012.

[9] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla, "Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study," in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*.

[10] C. Emde, "Long-term monitoring of apparent latency in preempt_rt linux real-time systems," in *12th Real-Time Linux Workshop (RTLWS)*.

[11] G. Bernat, R. Davis, N. Merriam, J. Tuffen, A. Gardner, M. Bennett, and D. Armstrong, "Identifying opportunities for worst-case execution time reduction in an avionics system," in *Ada User Journal, Volume 28, Number 3, Sept. 2007*.

[12] Steven Rostedt, "ftrace - Function Tracer (Linux Kernel Documentation)," https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[13] Jon Masters, "[RT] [RFC] simple SMI detector," Linux Kernel / Linux RT Users Mailing Lists; Fri, 23 Jan 2009 17:55:12 -0500; Message-ID: 1232751312.3990.59.camel@perihelion.bos.jonmasters.org, [Online at https://lkml.org/lkml/2009/1/23/330; accessed 2018-04-04].

[14] P. Thomas, "Hardware based latency profiling of interrupt handlers," in *17th Real-Time Linux Workshop (RTLWS)*, [private communication].

[15] OSADL - Open Source Automation Development Lab eG, "Latency Box," https://www.osadl.org/Latency-Box.projects-latency-box.0.html.

[16] Xilinx Inc., "Zynq-7000 All Programmable SoC Technical Reference M." https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, [UG585, Revision 1.12.1].