Quantifying the Latency and Possible Throughput of External Interrupts on Cyber-Physical Systems

ABSTRACT

An important characteristic of cyber-physical systems is their capability to respond, in-time, to events from their physical environment. However, to the best of our knowledge there exists no benchmark for assessing and comparing the interrupt handling performance of different software stacks. Hence, we present a flexible evaluation method for measuring the interrupt latency and throughput on ARMv8-A based platforms. We define and validate seven test-cases that stress individual parts of the overall process and combine them to three benchmark functions that provoke the minimal and maximal interrupt latency, and maximal interrupt throughput.

DATA AVAILABILITY STATEMENT

A snapshot of the exact version of the prototyping platform [] that was used to conduct the measurements described in this paper has been made available through Zenodo []. The snapshot also includes the captured, raw STM trace data and the matching processing scripts to produce the figures included in this paper. The latest version of the platform can be obtained from [].

1 INTRODUCTION

Cyber-physical systems (CPSs) are characterized by the fact that a computer system works together with a physical environment, or rather controls it. A specific characteristic of such control systems is their necessity to provide short reaction times on events in the physical world, to guarantee a good control quality. However, to properly design a CPS, the reaction times not only need to be fast, but also predictable [6]. Both properties are likewise essential for modern systems, such as tele-operated-driving [11], and classic systems, such as the control of internal combustion engines [5].

An important, but often neglected aspect of the achievable reaction time is the interrupt handling performance in both dimensions the interrupt handling latency and throughput capabilities of a system. Especially the effect of the utilized software stack has not yet been comprehensively assessed, even though the development of key-components of the CPS software stack, such as operating systems, virtualization solutions, and communication stacks, could benefit from the insight of such an assessment. Furthermore, such a systematic evaluation of the interrupt performance would also be of interest for system engineers, to evaluate architecture concepts or acquired software stacks for their suitability in a particularly latency-sensitive or throughput-hungry use-case. Previous approaches in this area require a very detailed knowledge of the utilized hardware and are therefore not easy to apply [10].

We present a flexible interrupt performance measurement method that can be applied to a wide variety of hardware platforms and software systems. As we see an increasing share of ARM based systems used in CPSs [8], we have chosen the ARMv8 hardware platform and its on-chip tracing technologies as basis for our measurement method. Based on the measurement method, we make a first attempt to specify benchmark functions that provoke the minimal and maximal interrupt latency, as well as the maximal interrupt throughput achievable with a system under consideration.

To deduce appropriate benchmark functions, we designed seven independent test-cases, specifically tailored to stress test individual parts of the ARM interrupt handling process. We evaluated those test-case on a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board [12], by comparing the interrupt performance of two different software stacks. A minimal bare-metal application and a full FreeRTOS real-time operating system based stack. Out of the seven test-cases, we formed 10 distinctive combinations and assessed them as potential benchmark functions. Based on this assessment, we propose three benchmark functions to trigger the minimal and maximal interrupt latency and the maximal interrupt throughput with a given software stack.

Our measurements confirmed that the chosen software stack indeed has a considerable effect on the interrupt handling performance of a CPS. Hence, we envision that the proposed evaluation method could be used to benchmark different software solutions against each other, to create a profound basis for further optimizations of the interrupt handling path in CPS software architectures.

The rest of this paper is structured as follows: Section 2 describes the interrupt handling process on ARMv8-A platforms with a Generic Interrupt Controller (GIC) version 2, Section 3 continues with a presentation of the envisioned evaluation method consisting out of a measurement setup and procedure. Section 4 discusses the proposed test-cases and benchmarks along with the measured latency and throughput values. Section 5 concludes the paper.

2 INTERRUPT HANDLING PROCEDURE ON ARMv8-A PLATFORMS

Müller and Paul [9] define an interrupt as an event that causes a change in the execution flow of a program sequence other than a branch instruction. Its handling process starts with the activation through a stimulus and ends with the completion of the interrupt service routine (ISR), which is called in consequence and processes the stimulus. Until the ISR is executed several steps are undergone in hardware to cope for example with simultaneously arriving interrupt requests (IRQs) and masking of certain requests. In the following, we explain this process for ARMv8-A platforms, as specified in the GIC architecture specification version 2 [1]. In Section 4 this information is used to design suitable test cases.

The GIC architecture specification differentiates among four types of interrupts: peripheral, software-generated, virtual, and maintenance interrupts. In the course of this paper we focus solely on measuring interrupts triggered by external stimuli, the peripheral interrupts. They can be configured as edge-triggered or level-sensitive. This means that the corresponding interrupt is recognized either once on a rising edge in the input event signal, or continuously as long as the signal has a certain strength.

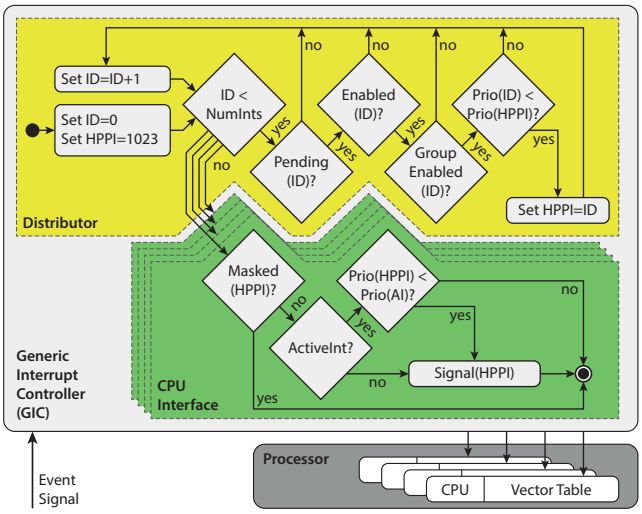

Figure 1: Repeatedly, per CPU interface executed selection and signaling process of the Generic Interrupt Controller (GIC) for handling triggered interrupt requests (IRQs).

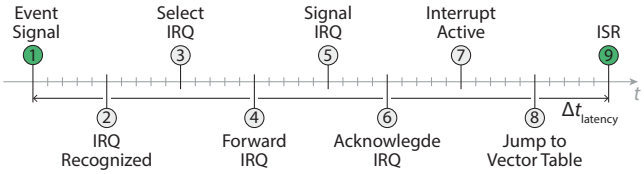

Figure 2: Overall steps of the interrupt handling process on ARMv8 platforms with a Generic Interrupt Controller (GIC) version 2 and an ARMv8-A architecture profile core.

The GIC supervises the overall interrupt routing and management process up to the point that the ISR is called. Figure 1 shows the GIC architecture and the signaling path for the selected measurement hardware, the Xilinx UltraScale+ MPSoC (ZynqMP). The GIC manages all incoming event signals of the system and consists out of a central Distributor and a CPU interface per processor core. The Distributor manages the trigger-type of each interrupt, organizes their prioritization, and forwards requests to the responsible CPU interface. The CPU interfaces perform the priority masking and preemption handling for their associated core.

The timeline of the various steps in the interrupt handling process are illustrated in Fig. 2. The handling process of a certain IRQ begins with the arrival of an event signal at the Distributor (step 1). In case the signal matches the configured trigger type, the actual handling process is triggered through the recognition of a new IRQ (step 2), which eventually leads to the execution of the ISR (step 9).

After being recognized (step 2), the Distributor may select and forward the IRQ to the responsible CPU interface according to the process depicted in the upper half of Fig. 1. This selection process (step 3) is executed repeatedly and potentially in parallel for each CPU interface. When the next highest priority pending interrupt (HPPI) is identified the Distributor forwards the request to the currently inspected CPU interface (step 4). The CPU interface

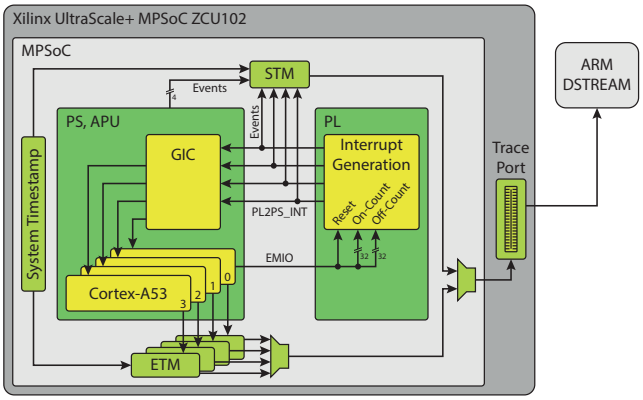

Figure 3: Chosen measurement setup, with four PL2PS interrupts generated by the programmable logic (PL) according to the configuration signaled by the processing system (PS) via its extended multiplexed I/O interface (EMIO). The generated interrupts, executed instructions, and global timestamps are recorded through the system trace macrocell (STM) and embedded trace macrocell (ETM). The captured trace is read via an ARM DSTREAM unit.

filters incoming requests according its configuration, the lower half of Fig. 1 depicts this filtering process. As a result the CPU interface may signal a pending IRQ to its associated core (step 5).

Subsequent to the signaling of a new IRQ through the CPU interface the handling process continues in the core. In case of the ARMv8-A architecture [4], the core acknowledges the signaled IRQ by reading its id (step 6), which marks the IRQ as *active* within the GIC (step 7). Meanwhile, the processing core jumps to the vector table entry indicated by the id of the current IRQ (step 8), which finally leads to the execution of the ISR (step 9). Individual software stacks might add additional steps before the ISR finally starts.

An important factor in the described IRQ handling process is the priority of the interrupt id associated with a specific IRQ where a lower number indicates a higher priority. Besides the regular interrupt requests described above, the GIC architecture version 2 also supports the prioritized and more secure fast interrupt requests (FIQs), however, these lay out of the focus of this paper.

Besides the configured priorities and masks, the GIC internally also tracks already signaled and currently serviced IRQs through an internal state-machine, such that multiple IRQs of the same type can be signaled and handled simultaneously.

3 PROPOSED EVALUATION METHOD

With respect to the interrupt performance of a certain platform and software stack two properties are most relevant and need to be measured: the IRQ *latency* and *throughput*. Considering the overall interrupt handling process depicted in Fig. 2, we define the *latency* of an IRQ as the time between a newly signaled event (step 1) and the moment when the corresponding ISR is called (step 9). In case a software stack adds additional processing steps to the process, an appropriate alternative end-point shall be identified. As *throughput* we understand the number of completed ISRs per time unit.

Depending on the hardware platform and software stack under analysis the above figures may be influenced in variable proportions by software and hardware. This is especially true for the IRQ latency. However, from the system design point of view reliable bounds for the overall latency, between the signaling and processing of an interrupt, are more important than knowing the actual proportion. Thus, we measure the whole interrupt handling process end-to-end.

Nonetheless, it is important that the measurements do not include any intangible factors. Hence, we base our measurements on the hardware tracing functionality provided by the ARM CoreSight technologies [2]. They allow to precisely measure and correlate processor internal and external events in a common time domain.

3.1 Measurement Setup

Our measurements utilize a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board [12], with an ARM DSTREAM [3] hardware trace unit, and the prototyping platform [] . We have chosen the ZynqMP, as it features a seamlessly integrated programmable logic (PL) and versatile hardware tracing capabilities. Figure 3 illustrates the hardware side of the chosen setup.

The ZynqMP is divided into the PL and the processing system (PS). The PS provides two processor clusters, the application processing unit (APU) and real-time processing unit (RPU). The APU consists out of four ARM Cortex-A53 cores and the RPU out of two ARM Cortex-R5 cores, both equipped with their own interrupt handling infrastructure. In the following we will solely focus on the APU and its interrupt handling, as it most closely matches our definition of a commercial off-the-shelf (COTS) hardware platform.

The in-built tracing capabilities of the ZynqMP allows to record events, such as taken branches in the executed code, recognized external interrupt stimulation, and special software driven events on a common timeline that is given by global system-wide timestamps. The overall recording process is conducted by the system trace macrocell (STM) and embedded trace macrocells (ETMs) in hardware, and thus does not alter the behavior of the executed code, neither in a temporal nor semantic sense. Software driven events, however, are recorded by writing into special registers and thus marginally influence the timing of the executed code.

In addition to the in built hardware features, we deploy the custom interrupt generation block into the PL. The block simultaneously stimulates the APU and the STM with four signal lines, to capture each interrupt signal within the system trace. The stimulation scheme can be individually defined by setting two 32 bit counters. One controls the duration of the logical-high phase (On-Count) and the other that of the logical-low phase (Off-Count). Given our PL clock configuration, both counters allow to specify a duration between 0 s and 17.1798 s.

In parallel to the STM trace, the ETMs capture, among other things, indirect and direct branch instructions executed by the four cores in the APU. As mentioned before, all STM and ETM trace events are marked with global timestamps. We configure the ZynqMP to trigger the timestamp counter every 4 ns.

3.2 Measurement Procedure

Given the above measurement setup we propose two configurations for the interrupt generation block and two time measurements based on the timestamps of the captured trace events. One for throughput and one for latency measurements. Within both measurement types, we utilize core 0 of the APU as measurement core and the other cores of the APU for stimulation purposes, where needed. All measurements are conducted with two different software stacks: *(i)* bare-metal and *(ii)* FreeRTOS based systems.

Throughput. In case of a throughput evaluation, we configure the interrupt generation block to continuously signal the PL2PS interrupts for 9.75 s and then wait for another 250 ms on a rotating basis. We call each repetition of this pattern a stimulation phase. Core 0 is configured to handle the signaled private peripheral interrupt (PPI) on a level-sensitive basis and the corresponding ISR does nothing despite emitting a STM software event, *i.e.*, executing a 8 bit store instruction. Hence, the time spent in each ISR and with that the possible reduction of the maximum throughput is negligible.

For each throughput measurement we capture a 120 s trace and evaluate the contained stimulation phases. The throughput (μ) in each stimulation phase ($i \in [0, 19]$) is obtained from the traced measurement samples by counting the ISR generated STM-software-events between the rising-edge STM-hardware-event of the $(i)^{\text{th}}$ and $(i{+}1)^{\text{th}}$ stimulation phase and dividing it with the length of one logical-high-phase. The set of throughput values considered for the evaluation in Section 4 is then given by $M = \{\, \mu(i) \mid i \in [0, 19] \,\}$.

Latency. The latency evaluation is conducted with an alternating scheme of a 1 ms PL2PS interrupt trigger and a 4 ms pause. The interrupt generation block is configured accordingly. Again we refer to each repetition of this scheme as a stimulation phase. In contrary to the throughput measurement, however, core 0 is configured to handle the signaled PPI on a rising-edge basis. Thus, every stimulation phase provokes only one interrupt. The corresponding ISR is the same as for the throughput measurements.

The results for the latency measurements are obtained by evaluating 30 s trace captures. The interrupt latency $\Delta t_{\mathsf{latency}}(i)$ induced by each stimulation phase $i \in [0, 2399]$ is given by $\Delta t_{\mathsf{latency}} = B - A$, with A representing the point in time where the interrupt was stimulated and B the point where the corresponding ISR was started. Both points can be obtained from the captured trace. A is given by the timestamp of the STM hardware event associated with the rising-edge of the PL2PS interrupt signal. B, on the other hand, has to be determined and defined for every analyzed software stack individually. In the course of this paper we utilize the timestamp of a STM event generated within the interrupt handler of our benchmark application that runs on top of the evaluated software stacks.

Similar to the throughput values, the set of latency values considered in Section 4 is given by $X = \{\, \Delta t_{\mathsf{latency}}(i) \mid i \in [0, 2399] \,\}$.

3.3 Precision and Limitations

In our measurement setup, we configure the PL, trace port, and timestamp generation clock to oscillate at 250 MHz. Hence, two consecutive timestamp ticks lay 4 ns apart from each other. Since each sampled event in the ETM and STM is assigned a timestamp, our

measurement precision corresponds exactly to the system timestamp resolution, *i.e.*, 4 ns. This is an order of magnitude smaller than the interrupt latency measured in a previous study for the same hardware platform [] and a quarter of the measured minimal interrupt latency of an ARM real-time core [10].

Even though state of the art oscilloscopes provide a sampling rate of up to 20 GSa/s [7], which would correspond to a measuring precision of 0.05 ns, the actual measurement precision in case of interrupt latency measurements might be considerably lower. The reason for this is that the oscilloscope can only measure external signals of a processor. Thus, a in-depth knowledge of the internal structure of the hardware platform and executed instructions in the software stack during a measurement is required to counter this circumstance and utilize the full precision of the oscilloscope. This makes it less suited for the evaluation of different hardware platforms and software stacks. The Coresight based measurement setup, on the other hand, supports a flexible placement of the measurement points within and outside of the processor and does not require any specific knowledge about the hardware or software.

Besides the measurement precision and flexibility, we also need to ensure that the presented measurement setup is sane and triggered interrupts can actually be recognized by the processor. According to the ZynqMP technical reference manual [13, p. 312], a signal pulse that shall trigger a PL2PS interrupt needs to be at least 40 ns wide to guarantee that it is recognized as such. Hence, the presented stimulation scenarios for the two measurement procedures ensure that all triggered interrupts can be recognized.

The disadvantage of the presented measurement approach, however, is that it is only applicable for ARM based platforms. Given ARM's 40% share of the semiconductor market for IP designs [8], we believe this is an acceptable drawback. An additional limitation is that valid measurements can only be obtained for the interrupt with the highest priority among the active ones, but this limitation applies to measurement setups of any kind.

4 CONSTRUCTING A BENCHMARK

In order to create a benchmark for comparing the interrupt latency and throughput across platforms and software stacks we have designed seven test-cases specifically tailored to stress the ARMv8-A interrupt handling process. To judge the suitability of the individual test-cases for an overall benchmark, we evaluated all of them with two software stacks running on top of the ZynqMP. As software stacks, we utilize a bare-metal system that executes the ISRs and otherwise busy-waits in an endless loop, and a FreeRTOS based system with its own scheduler and timer tick that executes the same ISR routine. In Section 4.1 we evaluate the impact of each test-case by comparing the performance obtained with the test-case with the baseline performance of the two systems.

Given the results of the individual test-cases we compose three benchmarks out of them and show their suitability by applying them to the same system configurations.

4.1 Evaluated Test-Cases

Given the interrupt handling process in Section 2, we conclude that the time spent in the process can be influenced by: the core, caches, memory, and GIC. We have designed seven test-cases that aim to reveal the influence of different configuration settings related to the aforementioned components onto the temporal behavior of the interrupt handling process. However, we do exclude the core from our considerations by only measuring interrupts with the highest priority. An overview on the proposed test-cases and their targeted components is given in Table 1. The remainder of this section elaborates on the intended influence of the listened test-cases on the interrupt handling process. The measurements for all test-cases follow the scheme presented in Section 3, unless indicated otherwise. Depending on the goal of each test-case they are either applied only for latency measurements or both latency and throughput measurements. The reason for that is that we aim at measuring the best and worst possible interrupt latency, but only the maximal possible interrupt throughput. The results of all evaluation runs are summarized in Figs. 4 and 5. The presented results are based on 848–6000 measurement samples per latency measurement and 11–12 samples per throughput measurement.

T1: Baseline. T1 is intended to provide a reference point to compare the other test-cases to and rate their impact. Hence, T1 assess the interrupt latency and throughput of a system in the most isolated way, with only one core and interrupt enabled and caches disabled. Hence, T1 only enables the extended multiplexed I/O interface (EMIO) pin driven interrupt and routes it to core 0. As ISR the default handler, described in Section 3.2, is used. It is evaluated for its latency and throughput performance.

T2: Caches enabled. T2 equals T1, with the exception that all operations are executed with enabled caches. This test is conducted for both latency and throughput measurements.

T3: Caches invalidated. T3 is also based on T1, but the ISR additionally invalidates the data and instruction caches. Due to the fact that this is not feasible in throughput measurements, as new interrupts would arrive independently of the cache invalidation process, we conduct only latency measurements with T3.

T4: Enabled interrupts. T4 aims at stressing the GIC with the highest possible number of enabled interrupts, as the interrupt selection and signaling process, illustrated in Fig. 1, suggests that more checks have to be done the more interrupts are enabled/pending. Hence, this test-case enables up to 180 stressing interrupts, the maximum number supported by the ZynqMP, except those required for driving the measurements itself. All interrupts are routed to and handled by core 0. The EMIO triggered PL-to-PS interrupt is assigned to the highest priority and all other interrupts to the lowest priority.

Core 0 installs an empty ISR that immediately returns after clearing the IRQ in the GIC for all interrupts, except the EMIO pin based PL-to-PS interrupt, which uses the same ISR as T1.

As this test aims at stressing the GIC to reduce its performance, we only evaluate it with respect to the interrupt latency. To be able to identify trends, we evaluated this test-case with 1, 36, 72, 108, 144, and 180 stressing interrupts. However, due to the marginal differences between the results of the different T4 variants and space constraints we only show the results of T4-180, T4 with 180 stressing interrupts, which provoked the highest latency.

T5: Order of priorities. T5 utilizes the same setup as T4 and is also applied to latency measurements only. However, in contrast to T4,

Table 1: Properties of the evaluated test-cases and benchmarks used to compare the interrupt latency (L) and throughput (T).

Description	Targeted Component	Measurements		Enabled Interrupts	Cache Config	Enabled Cores	Benchmarks		
		L	T				L_{min}	L_{max}	T_{max}
T1: Baseline	–	X	X	1	Disabled	1			
T2: Caches enabled	Cache	X	X	1	Enabled	1	X		X
T3: Caches invalidated	Cache	X		1	Invalidated	1			
T4: Enabled interrupts	GIC	X		2–181	Disabled	2		X	
T5: Order of priorities	GIC	X		15	Disabled	2			
T6: Parallel interrupt handling	GIC	X	X	1	Disabled	2, 3, 4		X	X
T7: Random memory accesses	Memory	X		1	Disabled	4			

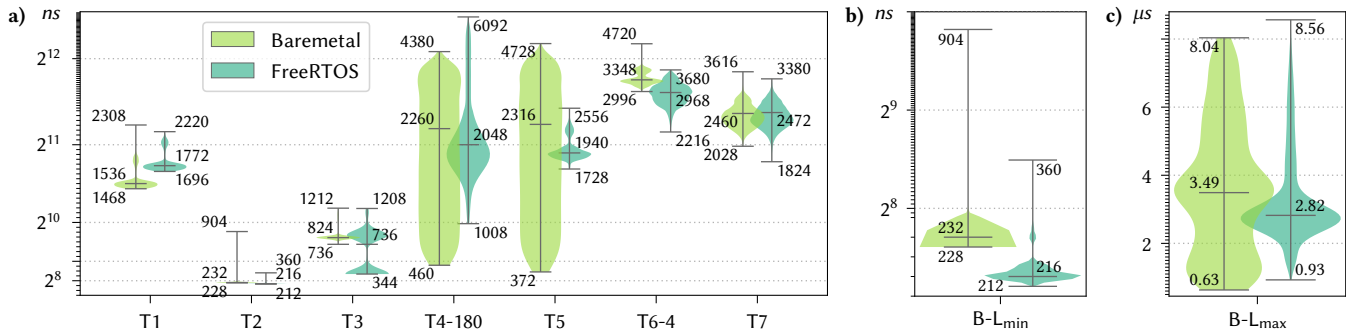

Figure 4: Latency measured with T1–T7 (a) and B-L_{min} and B-L_{max} (b-c). Figure a) uses a symlog scale with a linear threshold of 2 496 ns, Fig. b) uses a symlog scale with a linear threshold of 240 ns, and Fig. c) uses a linear scale.

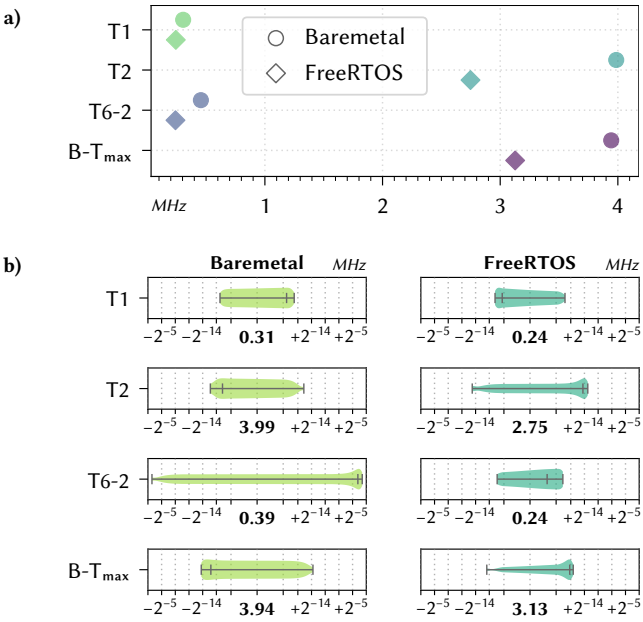

Figure 5: Throughput measured with T1, T2, T6, and B-T_{max}. Figure a) compares the median of all measurements on a linear scale and Fig. b) illustrates the measured throughput ranges on a symlog scale with a linear threshold of 1 Hz, normalized to a 500 kHz range around the highlighted median.

T5 only utilizes as much interrupts as there are priorities, *i.e.*, 15. The measured interrupt remains at priority 0 and the priorities of the other 14 are assigned in an ascending order (*i.e.*, 14 to 1). This design intends to provoke a maximal number of HPPI updates.

T6: Parallel interrupt handling. To test the influence of parallelly handled interrupts on the interrupt handling process, T6 enables up to 4 cores and configures all of them to handle the EMIO pin 0 interrupt. The interrupt is configured as level-sensitive with the highest priority. The PL ensures that this interrupt is signaled continuously and simultaneously as soon as the test is enabled. The ISRs on all cores generate a STM event, which are evaluated for throughput measurements. In case of latency measurements, however, only those STM events produced by core 0 are considered.

We evaluated T6 with 2, 3, and 4 enabled cores. The results showed a clear trend that the more enabled cores the higher the observed latency, but the lower the achieved throughput. Due to space constraints we thus only show the results for T6-4, with 4 enabled cores, in case of the latency considerations and T6-2 in case of the throughput measurements.

T7: Random memory accesses. As pointed out earlier, the shared memory and interconnecting buses of multi-core processor represent a major source of unforeseen delays. Accordingly, T7 is designed to delay memory accesses by overloading the interconnecting bus and memory interface. For this purpose all 4 cores are running random, concurrent memory accesses in form of constants that are written to random locations in a 96 MB large array.

In parallel core 0 executes the standard latency test. Throughput evaluations are not considered with this test-case, as it targets to delay the interrupt handling process.

4.2 Proposed Benchmarks

Analyzing the measured interrupt performances under the different test-cases, shown in Figs. 4 and 5, we conclude that first of all different setups and software stacks indeed considerably influence the interrupt handling performance. All three targeted components, provoke a considerable effect on the interrupt latency and throughput. Particularly noticeable are the differences between the test-cases with enabled (T2, T3) and disabled caches (T1, T4–T7), for both the observed latency and throughput. As well as the effects of stressing the GIC on the measured latency (T4–T6).

T2 produces by far the shortest interrupt latency of 232 ns in average with only a few outliers. Hence, we propose to utilize T2 as benchmark for the minimal achievable latency (B-L_{min}).

To obtain a suitable benchmark for the maximal latency, we analyzed all combination out of the test-cases T4-36, T4-144, T6-3, and T7. Except for the combination out of T6 and T7, all tested combinations showed a similar performance with only slight differences. An exception to that forms the interrupt latency performance of the combination out of T4-144 and T6 on FreeRTOS, which is considerably more confined than all other observed ranges. The highest latency is achieved with a combination out of T4-36 and T6, however, the combination of T4-36, T6, and T7 is close. Accordingly, we propose to use the combination out of T4-36 and T6 to benchmark the achievable maximal interrupt latency (B-L_{max}).

For the maximal throughput benchmark (B-T_{max}) we evaluated all four variants of the T6 test-case with enabled caches (T2). Interestingly, the enabled caches seem to mitigate the effect of more enabled cores, as all combinations showed a similar throughput. However, the combination out of T6-2 and T2 still performed best. Even though the maximal achieved throughput of the combined test-cases lags a little behind that of T2 alone, in case of the baremetal software stack, it outperforms T2 by far in case of the FreeRTOS based stack. Hence, we propose the combination out of T6-2 and T2 to benchmark the maximal throughput of a system.

5 CONCLUSION AND OUTLOOK

We presented a flexible evaluation method based on the ARM CoreSight technology, which enables the assessment of various software stacks on top of commodity ARMv8-A platforms with respect to their interrupt handling performance. Utilizing the evaluation method, we crafted seven specifically tailored test-cases that were shown to stress the ARM interrupt handling process. Out of these test-cases we deduced three benchmark functions, tailored to provoke the minimal (B-L_{min}) and maximal (B-L_{max}) interrupt latency, and the maximal throughput (B-T_{max}), of a given software stack. We validated the test-cases and benchmark functions by comparing two software stacks (a simple bare-metal and FreeRTOS based environment) and measuring them on top of a Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board [12].

Our measurements showed that different software stacks can have a considerable impact on the interrupt handling performance of a hardware platform. Hence, we hope to draw some attention on the importance of a good software design for CPS, with respect to interrupt processing and the need of a more profound analysis on how interrupt handling processes can be made more predictable with respect to the achievable latency and throughput.

ACKNOWLEDGMENTS

REFERENCES

[1] ARM Ltd. 2013. ARM Generic Interrupt Controller – Arch. ver. 2.0. (IHI 0048B.b).
[2] ARM Ltd. 2013. *CoreSight Technical Introduction.* White paper ARM-EPM-039795.
[3] ARM Ltd. 2015. ARM DS-5 Version 5 ARM DSTREAM User Guide. (DUI 0481K).
[4] ARM Ltd. 2018. ARM Architecture Reference Manual – ARMv8, for ARMv8-A architecture profile. (DDI 0487D.a).
[5] P. Frey. 2010. *Case study: engine control application.* Technical Report. Open Access Repositorium der Universität Ulm.
[6] E. D. Jensen. 2008. Wrong Assumptions and Neglected Areas in Real-Time Systems. In *11th Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing (ISORC).*
[7] Keysight Tech. 2020. InfiniiVision 6000 X Series Oscilloscopes Data Sheet.
[8] D. Manners. 2020. Design IP market grew 5.2% last year. https://www.electronicsweekly.com/news/business/732874-2020-04/. Accessed 06-05-2020.
[9] S. M. Müller and W. J. Paul. 1995. *The Complexity of Simple Computer Architectures.* Springer-Verlag, Chapter 8, 141–178.
[10] NXP Semiconductors. 2018. Measuring Interrupt Latency. Appl. Note AN12078.
[11] T. Tang, F. Chucholowski, and M. Lienkamp. 2014. Teleoperated driving basics and system design. *ATZ worldwide* 116, 2 (Jan. 2014), 16–19.
[12] Xilinx, Inc. 2018. ZCU102 Evaluation Board – User Guide. UG1182 (v1.4).
[13] Xilinx, Inc. 2018. Zynq UltraScale+ Device – TRM. UG1085 (v1.8).