Appears in: Proceedings of the 2018 Brazilian Symposium on Computing Systems Engineering (SBESC '18)
Salvador, Brazil, 6–9 November 2018

INTSPECT: Interrupt Latencies in the Linux Kernel

Benedict Herzog[1], Luis Gerhorst[2], Bernhard Heinloth[1], Stefan Reif[1],
Timo Hönig[1], and Wolfgang Schröder-Preikschat[1]

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
[1]{benedict.herzog,heinloth,reif,thoenig,wosch}@cs.fau.de [2]luis.gerhorst@fau.de

***Abstract*—Interrupt handling with predictably low latency is a must for systems to respond to external events. System designers of tiny embedded computers to large-scale distributed systems face the challenge of ever-increasing hardware and software complexity. In the absence of precise timing models, measurement-based approaches are required to achieve predictably low latency.**

In this paper, we present INTSPECT, a tool that systematically evaluates the interrupt latency, at run-time, in the Linux operating system kernel. We apply INTSPECT on two distinct platforms (i.e., ARM and Intel) to measure interrupt latencies, identify jitter causes, and reveal interdependencies between interrupt handlers and user-space workloads. Our tool thus provides valuable insight on interrupt timings and interferences.

***Index Terms*—Linux, Interrupts, Latency, Measurements**

I. INTRODUCTION

For operating systems, interrupt handling is a core functionality which needs to ensure that the processing of interrupt requests occurs in a timely manner to react quickly to (external) events. In particular, when systems are interacting with the physical world (i.e., cyber-physical systems [1]) asynchronous events (i.e., interrupts) are the fundamental signaling mechanism. As interrupt latencies determine the minimum response time of the overall system, operating systems must react to pending interrupt requests as fast as possible. As physical limits (i.e., signal propagation delay) cannot be overcome, it is the responsibility of the low-level system software (i.e., interrupt handling routines of the operating system) to efficiently process interrupt requests and handle them according to their individual needs and priorities.

Predictable and low interrupt latencies are important for several reasons. First, the general performance and responsiveness of a system is bound to the execution time of the interrupt handling routines at operating system level [2]. Thus, it is a first-level prerequisite of operating systems to handle interrupt requests as quickly as possible in order to implement responsive system components at higher abstraction levels (i.e., hardware drivers and applications). Conversely, this means that computing systems cannot operate up to their full potential if interrupt handling at operating-system level does not perform well for a given workload. Second, the timely processing of pending interrupt requests reduces the risk of overload situations (i.e., in case of contention [3]). Thus, efficient handling of hardware interrupts is required to mitigate effects of contending system activities that operate on shared resources. In particular, congestion on network links or bulk I/O operations are examples where low latency translates into improved performance and leads to high throughput [4].

With the current technology trend [5], predictable and low interrupt latencies become even more important as single-core processor speed is becoming Achilles' heel of the systems, for example, in networked multi-core systems. While network throughput is scaling up and higher data rates are achieved with every new generation of network chips [6], the single-core performance of the CPUs is stagnating [7]. This further puts the focus on the "last mile": interrupts caused by external events (i.e., arrival of network packets) must be handled in the best possible manner to improve the systems' responsiveness. Although the data path can be shortened by remote direct memory access [8] and bypassing techniques at operating-system level [9], it remains crucial to provide low latencies for the control path which executes on the processor [10]. Low interrupt latencies are also essential for embedded devices which operate on limited resources. In particular, battery-powered devices must consider the duration of duty cycles to manage power demand and maximize their sleep time [11]. The execution time of the interrupt handler becomes a critical factor as its execution time directly effects the energy demand of the system [12]. Therefore, programmers must consider the implementation of interrupt handling routines that provide sufficient performance but demand as little energy as possible.

Today, even complex control systems use commodity hardware components that cannot provide hard real-time guarantees. For example, the SpaceX Falcon 9 rocket employs Linux on commodity x86 processors [13]. This fact shows that Linux provides sufficiently reliable interrupt handling on commodity hardware, even though it is difficult to estimate interrupt latencies for various reasons. First, the underlying hardware components do not provide timing guarantees, for example, as to non-deterministic activation of processors' system management mode. Second, the multilayered interrupt subsystem of Linux is inherently non-deterministic by itself. Third, varying system loads induce interferences at kernel and user level which influence interrupt latencies during run-time.

The increasing complexity of today's processors makes it inherently difficult to provide accurate numbers on the expected interrupt latencies of a system. To system designers, this poses the challenge of designing low-latency software for potentially critical subsystems in kernel or user space without any profound performance indicator for the run-time behavior of their program code. Further, the unavailability of timing

specifications and reliable data sheets makes it impossible to build precise timing models. Instead, it is necessary to apply accurate timing measurements to identify system properties, such as the interrupt latency, that help at dimensioning the system for its intended use [14].

In this paper, we present INTSPECT, a tool that applies time measurements to determine interrupt latencies of the Linux kernel automatically, precisely, and with lowest run-time interference. The contributions of the paper are threefold: First, we analyze the interrupt latencies of the Linux operating system kernel on two different hardware platforms (i.e., ARM and Intel) to provide numbers on the base latency of Linux with no background noise, and also for scenarios with varying degrees of interference from user and kernel space activities. Second, we present the implementation of INTSPECT which analyzes interrupt latencies and reveals culprits of unwanted delays within the Linux kernel. INTSPECT traces the timing characteristics of individual interrupts and provides information on the root cause for delays within the interrupt handling routines of Linux. Third, we present an in-depth discussion of the interrupt handling of a recent Linux kernel (Linux v4.9) and compare it to previous research ([15], [16]).

The paper is structured as follows. Section II elaborates on the interrupt handling of the Linux kernel. Section III discusses implementation details of the Linux kernel jointly with considerations on design and implementation of INTSPECT. In Section IV we present and discuss the evaluation results for different evaluation scenarios. Related work is presented in Section V and Section VI concludes the paper.

II. Interrupts in Linux

Computer systems continuously need to handle all types of synchronous and asynchronous events such as changing the level of privilege, handling run-time errors, and processing external hardware events. The usual mean for signaling an event to the processor is to raise an interrupt, which suspends the normal execution in order to react to the signaled event.

The processing of an event is done in an uninterruptible execution context, which prevents the delivery of further interrupts, until the currently served interrupt is completely processed. This difference in execution contexts implies that code, executed in a normal execution context, can be temporary superseded by an interrupt at any point of execution, whereas code running in an interrupt context has run-to-completion semantics and delays all further execution of applications and subsequently raised interrupts.

Interrupts are related to asynchronous events (e.g., overlapping data processing and input/output), which potentially need a prompt reaction to guarantee a responsive system, for example, signaling an incoming message in a communication channel. Additionally, interrupts may induce significant processing efforts depending on the causing event, for example, for decoding and processing fetched data from a hardware device. Running a long time in the interrupt context to handle an event, however, delays the acceptance of further interrupts and may lead to interrupts not handled in-time or not at all, which is unacceptable in the general case.

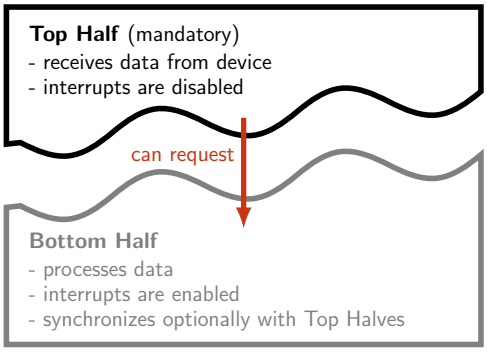

Fig. 1. Top and bottom half in Linux. A top half is executed immediately after the arrival of an interrupt with interrupts disabled. A top half can request a bottom half, which runs with interrupts enabled (except for synchronization).

A common approach to meet the problem of long-running interrupt handlers is to split the handler into two sections, which are executed at different points in time. In the first section, actions are executed immediately upon arrival of an interrupt in the interrupt context, whereas in the second section actions are deferred and executed later in an interruptible execution context. Besides the advantage of improving the system responsiveness by minimizing the time the system is uninterruptible, this structure allows, depending on the mechanisms used for the deferred sections, the usage of additional primitives like waiting for other events. Such primitives are often utilized by complex event handlers (e.g., device drivers).

The subdivision of interrupts into two sections is known by different names such as the prologue/epilogue model [17], the concept of top and bottom halves in UNIX systems (e.g., BSD [18] or Linux [19], [20]), and *deferred procedure calls* in Windows [21]. OSEK has *interrupt service routines* that *activate tasks* to defer work [22]. De facto, all operating systems provide mechanisms to defer work from the interrupt context into interruptible execution contexts. Because this paper focuses on the Linux operating system kernel, the interrupt handling mechanisms in Linux are explained in more detail in the following. Linux calls sections executed in the interrupt context *top halves* and deferred sections *bottom halves* as depicted in Figure 1. A top half is executed immediately after the arrival of an interrupt with interrupts disabled and the top half can request a bottom half, which runs with interrupts enabled. In case a bottom half needs to synchronize with top halves, the bottom half can temporarily disable interrupts.

A. Multi-level Interrupt Handling

Figure 2 summarizes the process of interrupt handling in Linux from start to finish. Initially, an asynchronous event triggers an interrupt, which is delivered to the processor. Before the interrupt can be handled, additional latency potentially occurs due to one of the following reasons. First, an active interrupt (i.e., a running top half) already allocates the

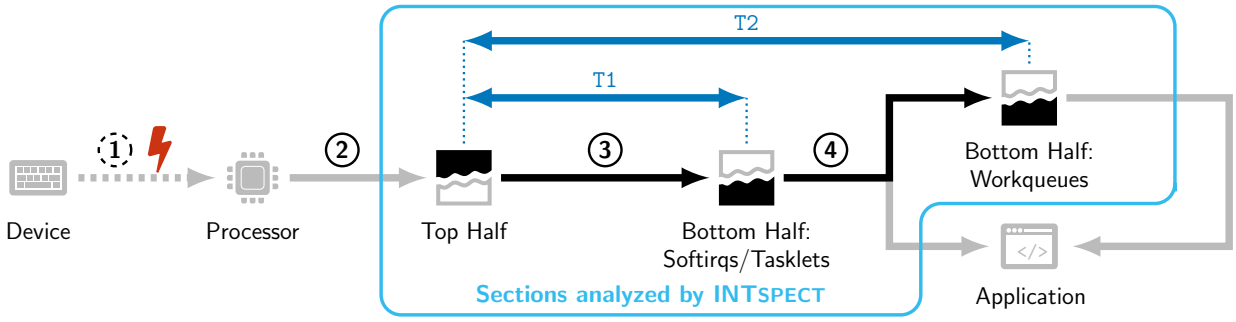

Fig. 2. General process of interrupt handling in Linux. An external event (e.g., a device) triggers an interrupt, which is delivered to the processor. The processor selects the corresponding interrupt vector (i.e., top half) and subsequently executes potentially requested bottom halves.

processor. Second, the operating system has interrupts disabled for synchronization purposes. Third, the hardware needs to finish the currently executed instruction before an interrupt can be handled (①).

As soon as the hardware is able to serve an interrupt, it saves the current execution context (e.g., registers), selects the interrupt vector and executes the interrupt service routine. However, the time for context saving and interrupt vector selection introduces additional latency (②). The execution of the interrupt service routine corresponds to the execution of the top half. The execution time of the top half depends on the actual purpose of the top half. Usually, such tasks run for a relatively short time, for example, they fetch data from a hardware buffer and defer the actual processing into a bottom half. After the termination of a top half, the operating system resumes control over the processor and executes potentially requested bottom halves (③).

Linux provides different mechanisms to request bottom halves from top halves, which differ in their properties (e.g., latency, flexibility) and designated use cases. The mechanisms are in particular: *Softirqs*, *Tasklets*, and *Workqueues* [19]. The first bottom half mechanism that is executed directly after top halves are Softirqs. The execution of Softirqs includes the execution of Tasklets, since the Tasklet implementation bases on two Softirqs. As they constitute bottom halves, the execution of Softirqs and Tasklets can be interrupted by other interrupts, which may additionally request further bottom halves. After the execution of all Softirqs and Tasklets, the process scheduler resumes control and, depending on its decision, either kernel threads or user threads are dispatched (④). This scheduling decision includes worker pools that constitute the third bottom half mechanism: they execute jobs from Workqueues.

B. Softirqs

Software interrupt requests (Softirqs) are the most basic type of deferring work from a top half to a bottom half in Linux. Softirqs must be specified at compile-time of the kernel in an enumerator, which makes it impossible to dynamically register additional Softirqs during run-time. Furthermore, the specification order defines the execution order and is fixed at compile-time. Each Softirq is associated with a function that constitutes the event handler.

Softirqs are executed immediately after the return of a top half, and before a switch from the kernel space to the user space is conducted (e.g., after a system call). Although a Softirq can not make use of wait primitives (e.g. sleeping until an event occurs) and runs to completion, it can be interrupted at any point by a top half and is resumed after the top half has terminated. Thus, Softirqs are suitable for handlers which do not need to sleep. In addition, Softirqs may be simultaneously executed on each processor, which allows for processor-local optimizations and fast accesses of data through caches. The advantages, that is, deterministic execution order and cache locality are the reasons why Softirqs are used by the kernel to execute frequently occurring tasks, for example, handling network traffic and input/output operations. However, the limited flexibility (e.g., specification at compile-time, number of different Softirqs) restricts the use cases of Softirqs (e.g. network and input/output operations). Moreover, operations which may sleep or need to wait for other occurring or finishing events can not be used within a Softirq. Hence, for such operations, other mechanisms are needed. One alternative to Softirqs are Tasklets, which pose a mechanism to dynamically register new bottom half handlers.

C. Tasklets

The implementation of Tasklets bases on Softirqs, and provides a more flexible way of requesting bottom halves. New types of Tasklets can be dynamically registered at run-time and are thus suitable for loadable kernel modules. Similar to Softirqs, each Tasklet is associated with a function that constitutes an event handler. New Tasklet executions can be requested from the top half by enqueueing them to a linked list of pending Tasklets. Two dedicated Softirqs iterate over lists of Tasklets and invoke the respective event handlers.

The usage of a Softirq for the implementation simultaneously determines the point of execution of Tasklets, that is, the execution of the Softirq as described in Section II-B. In fact, there are two Softirqs designated for Tasklet execution, one for high-priority Tasklets, running prior to any other Softirq, and one for normal Tasklets, running after Softirqs designated for network and block input/output operations. Although, the way of registering a new type of Tasklet is more flexible and

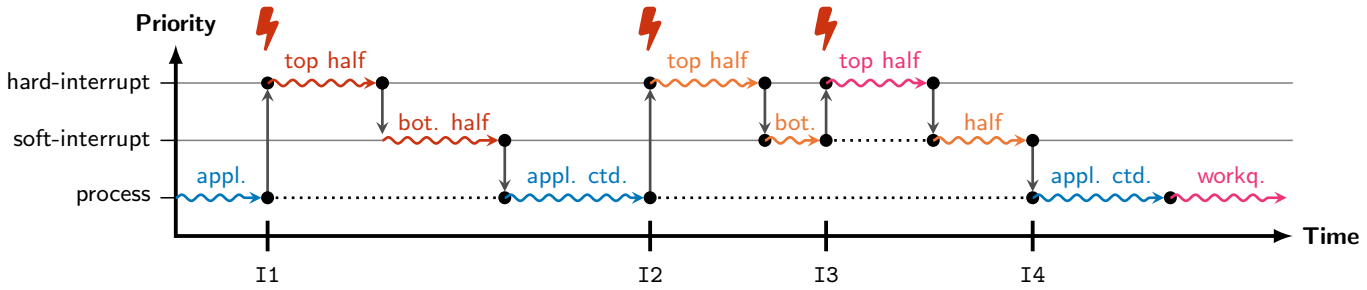

Fig. 3. Logical priority model of Linux interrupt handlers. A top half can interrupt the execution of applications (I1, I2) and bottom handlers (I3). After the execution of all top halves and bottom halves with soft-interrupt priority, the execution of applications and work items is continued (I4).

thus is suitable for dynamic kernel objects, the utilization of Softirqs prohibits the usage of sleeping and waiting primitives.

In contrast to Softirqs, the Tasklet infrastructure prevents simultaneous execution of Tasklets on different processors, hence implicitly synchronizes data accesses. If a processor detects that a requested Tasklet is already running on another processor, it re-adds the Tasklet to the list of requested Tasklets and schedules another list iteration.

D. Workqueues

The Workqueues implementation of the Linux kernel is the most flexible and commonly used bottom half mechanism. An interrupt handler can enqueue a handler, encapsulated in *work items*, in Workqueues, which are later executed by a *worker* (i.e., a kernel thread). To this end, the kernel provides worker pools consisting of one or more kernel threads, which dynamically adapt their level of concurrency to the current load. Each worker dequeues work items from the Workqueues and executes them. When a Workqueue runs empty, the corresponding workers enter an idle state until new work items arrive. Although the level of concurrency (i.e., the number of kernel threads) is dynamically adaptable, the kernel guarantees the presence of at least one worker thread.

Similar to Tasklets, work items are classified as high or normal priority, allowing programmers to prioritize specific work items. In contrast to Tasklets and Softirqs, however, work items can sleep and wait for other events, because they are executed by kernel threads, which are scheduled like application threads by the normal process scheduler. For example, this allows drivers to access sophisticated features including concurrency and synchronization. Moreover, work items that execute for extended amounts of time may be preempted by the scheduler, thus allowing user threads to make progress and preventing starvation of user applications. However, the latency from creating a work item in a top half until the execution of the work item is in general unpredictable, due to the dynamic character of Workqueues, that is, a dynamic number of work items, Workqueues, and worker threads.

E. Example

Figure 3 illustrates the execution of an exemplary sequence of tasks and interrupt handling functions. At the beginning, an application is executed, which is interrupted at point I1. The interrupt handling process begins with the execution of the corresponding top half, which runs at a higher priority (hard-interrupt level). In this example, it requests a bottom half (i.e., Softirq or Tasklet) on the soft-interrupt priority level[1], which is subsequently executed. After the termination of the bottom half, the application continues. At points I2 and I3, two additional interrupts occur. First, the top half of I2 requests another bottom half (Softirq or Tasklet), which is interrupted by I3. Second, the top half of I3 enqueues an item into a Workqueue. After termination of the top half of I3, the bottom half of I2 continues. As soon as the bottom half of I2 is finished (at point I4), the process scheduler decides whether the application is continued, or the worker pool dequeues work items from the Workqueue, as the worker threads run at the same priority level as applications.

This description of the interrupt handling process in Linux outlines that the overall latency from an asynchronous event to the completion of all handlers depends on multiple factors. Therefore, we have developed INTSPECT to quantify such latencies. Thereby, INTSPECT focuses on the analysis of latencies that are controllable in software—that is, between the execution of a top half and a requested bottom half. The results provide system designers valuable information about the latency characteristics of the different bottom half mechanisms. More specifically, we measure the latencies T1 for Softirqs and Tasklets and T2 for Workqueues as depicted in Figure 2. This information enables programmers to trade off functional properties of a bottom half mechanism (e.g., the ability to sleep) with non-functional properties (e.g., latency).

III. IMPLEMENTATION

INTSPECT, the *interrupt subsystem performance evaluation and comparison tool*, is applicable for all bottom half mechanisms in Linux—Softirqs, Tasklets, and Workqueues. Figure 4 summarizes the architecture of our latency evaluation and analysis tool. It encompasses a user-land analysis framework, and a kernel module, INTSIGHT. Thereby, the task of the kernel component is to *gather* latency information with high precision, high accuracy, low overhead, and low interference.

[1] *Soft-interrupt* denotes the priority level of Softirqs and Tasklets, whereas *Softirq* is the name of the specific bottom half mechanism.

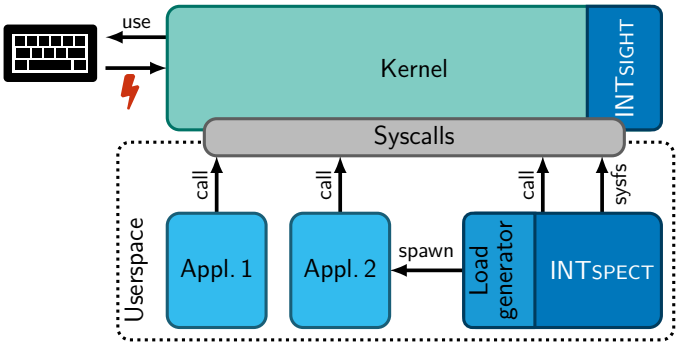

Fig. 4. Architecture of the INTSPECT tool. The kernel module conducts latency measurements (INTSIGHT) and provides insights through the `sysfs` interface to user-space applications. The user-space parts retrieves this information and potentially generates system load for measurements (INTSPECT).

Complementary, the user-space framework *controls* the kernel component and *processes* all acquired information.

The kernel component, INTSIGHT, aims for minimal runtime interference and maximum portability. INTSIGHT provides a top half and a bottom half for each mechanism. Thereby, the top half contains code to start the latency measurement and code to request the respective bottom half mechanism. The actual timing measurements utilize hardware performance counters since they offer fine-grained timing information with minimal interference. Furthermore, the measurement overhead is minimal—on modern hardware architectures (in particular, ARM and x86), cycle-counter based timestamps can be obtained in a single processor instruction and thus additional overhead is avoided. To complete the latency measurement, each bottom half in INTSIGHT contains code to obtain another time-stamp, which ends the latency measurement, and stores the data in a pre-allocated buffer. To minimize interference, further data processing is deferred until the completion of an experiment. Since interrupt latency is subject to unpredictable jitter, INTSIGHT automatically repeats the measurement for a configurable number of iterations. The required buffer to store raw measurement results is allocated prior to the start of an experiment, to minimize interference from memory management. Furthermore, INTSIGHT ensures that a suitable buffer for the entire experiment is kept in memory during the complete measurement run.

INTSIGHT uses the *tracepoint* feature of the Linux kernel to annotate each latency measurement. This tracing mechanism is already integrated into the Linux kernel and monitors performance-relevant code paths. For INTSIGHT, it allows to reconstruct the actual control-flow path for each interrupt. This feature identifies which further interrupt requests occur during measurements, for instance, caused by hardware devices.

In the architecture, the kernel module has the task of *gathering* the latency information. It provides all acquired information for further processing via a `sysfs` interface. This modularization improves the portability of INTSPECT.

The user-space framework has three tasks: First, it sets the system into specific *system states*, which enable an evaluation of the influence of system-level parameters on the interrupt latencies. Second, it notifies the kernel component to start measurements. Third, it collects and analyzes all information provided by the kernel component. The basic system state in our evaluation is the *base line* state with minimal system load and noise. This system state allows an evaluation with minimal interference. Additionally, INTSPECT features a *load generator* that executes benchmark workloads. We can thus evaluate the influence of specific workloads on interrupt latencies.

After collecting raw result data from the kernel component, INTSPECT analyzes the interrupt latencies (e.g. the maximum latency). Additionally, it utilizes the control-flow annotations obtained from kernel tracepoints to correlate the interrupt latency to other system activities. Thus, our tool can identify root causes of interrupt handling jitter. This information is typically unknown to system designers because it depends on complex interaction between hardware and software components, and is therefore vital for system designers to improve timing predictability. We have published the source code of INTSPECT under an open source license[2].

IV. Evaluation

We have deployed and evaluated the INTSPECT tool on two different hardware platforms: an ARM-based embedded platform and an Intel x86 platform. The implementation on these two platforms demonstrates that our approach is applicable for a broad range of hardware architectures, from tiny embedded devices to powerful server computers.

A. ARM Hardware Platform

The ARM-based embedded platform consists of the Atmel SAMA5D3 Xplained[3] board hosting an ARM Cortex-A5 processor, which runs at 528 MHz. The Cortex A5 provides a cycle counter, which allows cycle-accurate measurements with low overhead (read of the `PMCCNTR` register), allowing for measurements with high accuracy and minimal overhead.

All measurement runs consist of 50 000 repetitions and the results are presented as a histogram (bin size of 1 µs), and a logarithmic y-axis to visualize the complete range of values. In order to evaluate the variance, both the 5 %- and 95 %-quantiles are marked with dashed red lines. Hence, 90 % of all values (i.e., 45 000) lie in between.

Figure 5 shows the latency distribution between a top half and the corresponding bottom half for each of the three bottom half mechanisms, on the ARM-based hardware platform. The measurements represent a base line system state, where no other tasks were active. Furthermore, measurements affected by another interrupt are excluded, but shown in an extra column labeled *IRQ*. Similarly, values exceeding 45 µs are also excluded for clarity purposes, but shown in the column labeled *OOB* (out-of-bounds). In this case, only three values out of 50 000 exceed 45 µs. The latencies for Softirqs and Tasklets are similar with slightly shorter latencies for Softirqs (median for Softirqs is 4.0 µs and for Tasklets 4.7 µs). This small difference

[2] `https://gitlab.cs.fau.de/i4/intspect`
[3] Part Number: ATSAMA5D3-XPLD, Microprocessor: SAMA5D36

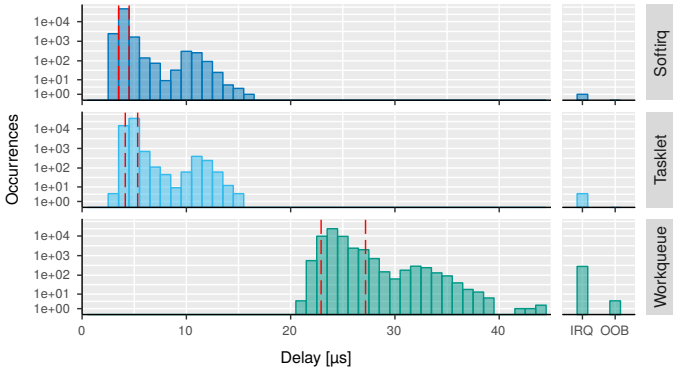

Fig. 5. Latency measurements for all three bottom half mechanisms without system load on the ARM-based hardware platform. Measurements affected by other interrupts are excluded, but shown in the IRQ column. Outliers are shown in the OOB column.

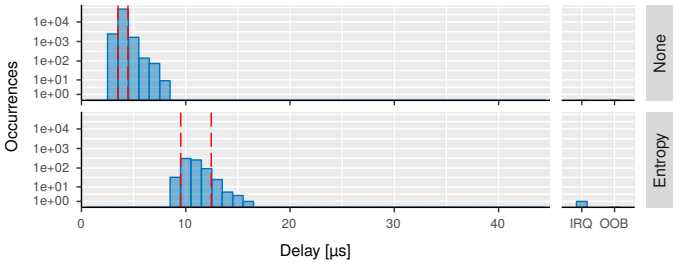

Fig. 6. Further analysis of the values for Softirqs as shown in Figure 5. For parts of the measurements additional latency is introduced, because the interrupt is used to enhance the entropy pool for the random number generator.

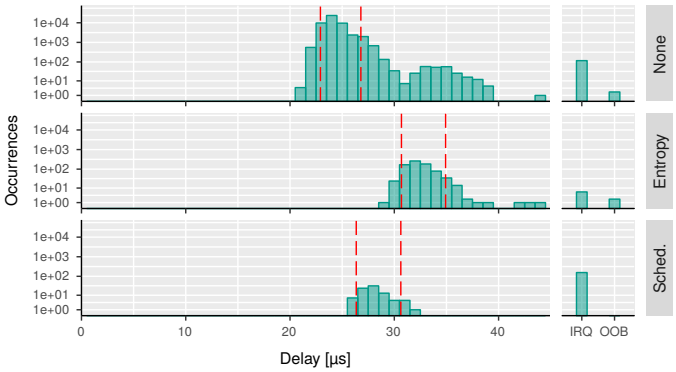

Fig. 7. Further analysis of the values for Workqueues as shown in Figure 5. For parts of the measurements additional latency is introduced, because either the entropy pool is filled or the process scheduler updates run-time statistics.

corresponds to the fact that Tasklets are executed by dedicated Softirqs. Workqueue requests have significantly higher latency (median of 24.1 µs) and jitter, because work items are executed after Softirqs and Tasklets. Furthermore, the process scheduler introduces additional non-deterministic latency.

Figure 6 and Figure 7 show the measurements runs of Softirqs and Workqueues already shown in Figure 5 in further detail. The tracepoint feature of INTSPECT offers more insight about the control flow leading to specific interrupt latencies.

For all three bottom half mechanisms, we have identified the random number generator entropy pool as a source of additional latency. Once per second, the Linux kernel uses interrupt-related information as a source of randomness. The procedure where a specific interrupt contributes to the kernel entropy pool takes place immediately after the return of a top half, hence in such cases a higher latency is measured.

Besides the entropy pool, the process scheduler is another source of latency and jitter for Workqueues, since it manages the worker threads of Workqueues. The rescheduling overhead is one major reason for the higher latency of Workqueues, compared to Softirqs and Tasklets. However, in some cases the process scheduler additionally updates run-time statistics of tasks and thus adds further latency and unpredictability. The row in Figure 7 labeled with *Sched.* illustrates the latency, if the scheduler additionally updates run-time statistics.

The presented examples for Softirqs and Workqueues show that the tracepoint feature of INTSPECT is capable to identify kernel features which influence the latency of interrupt handling mechanisms. This makes INTSPECT an valuable tool for system designers to evaluate and improve interrupt latencies.

Besides the measurements without system load, we have conducted our measurements in additional system states with increased system load. Therefore, we utilize the INTSPECT load generator feature, which allows to spawn an adjustable number of processes to increase the system load during measurements. Figure 8 and Figure 9 show the latency distributions for Softirqs and Workqueues, respectively, for varying degrees of system load. The rows labeled *Zero*, *One*, and *Many* present the base line system state without load, with one active process (full processor utilization), and with 256 active processes, respectively.

In summary, increasing the system load has only a minor influence on the latency distributions for Softirqs, but a noticeable effect for Workqueues. The latencies decrease when one process fully utilizes the processor, compared to the measurement without system load. We suppose that the latency reduction results from the absence of sleep state transitions and caching effects. Due to the decreased latency the probability of interrupts affecting our measurements is also decreased, which results in less values shown in the IRQ column. To investigate this hypothesis, we have repeated our measurements with active waiting between measurements instead of sleep states. The results show the same behavior, which supports our assumption. However, the effect disappears if many processes (i.e., 256 in our measurements) are running in case of Softirqs and Tasklets and even reverses in case of Workqueues. The increased latency for Workqueues presumably results from the increased process scheduler latency. Due to the increased latency values, we also observe an increased number of outliers. In the case of one process, 44 values are greater than 45 µs, which represents less than 0.1 % of all measurements.

B. Intel x86 Hardware Platform

Aside from the detailed measurements made on ARM in an embedded environment with minimal interference, we have also repeated our benchmarks on a server computer employing

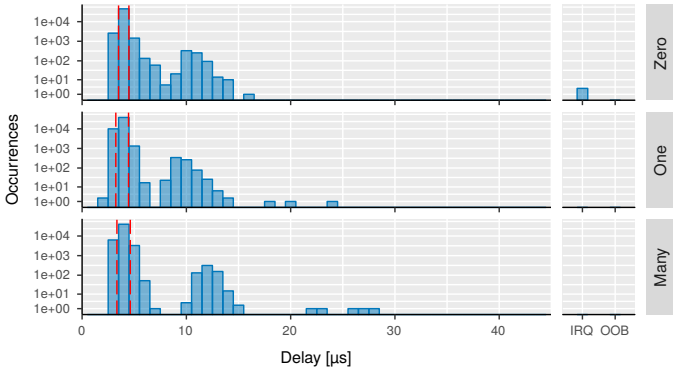

Fig. 8. Latencies for Softirqs with zero, one, and many active processes, respectively. The latencies are mostly constant with slightly shorter latencies, when one process is actively running.

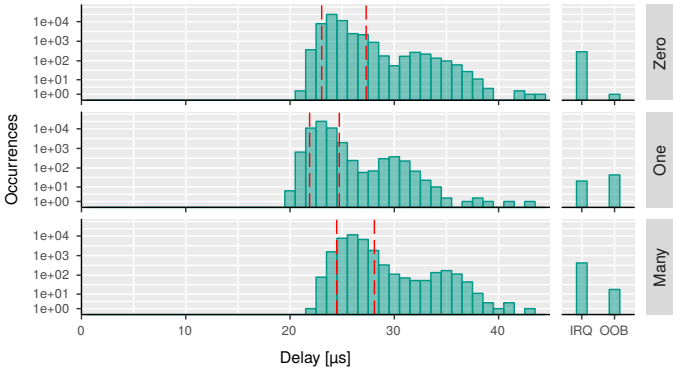

Fig. 9. Latencies for Workqueues with zero, one, and many active processes, respectively. Latencies are shorter if one process is running and longer if many processes are running.

a x86 processor. Implementing a completely different instruction set architecture (CISC vs. RISC), a more than six times higher clock frequency (3.3 GHz vs. 528 MHz), and a different cache model, the platform differs fundamentally from the ARM platform. Also, both the system distribution used and the configuration of the kernel had major differences (i.e., network was enabled, the number of running tasks was higher). The motivation is to examine which observations made for ARM platform are specific to our embedded scenario, and which carry over to a completely different environment.

Figure 10 displays the latency between a top half and the execution of a corresponding bottom half for the x86 hardware platform, and thus corresponds to the results for the ARM-based hardware platform shown in Figure 5. On the faster x86 processor, the delays are much lower, although the speedup is not 6.25, as one would deduce from the increase in clock cycles per seconds. In numbers, the median on x86 for Workqueues is 3.3 times lower, compared to the ARM platform. Similarly to the ARM platform, the latency distributions of Softirqs and Tasklets are very similar (again, Tasklets are slightly slower). Workqueues, in comparison, are slower in average. Throughout the evaluation, the x86 platform shows higher jitter. For instance, Softirqs and Tasklets have a higher worst-case delay than Workqueues in the best case.

In addition to the plain time measurements, as on ARM, we have used the tracepoint feature of INTSPECT to identify jitter causes. Here, both the collection of randomness for the entropy pool and the accounting of task run-time by the scheduler also occur on x86. For Workqueues, however, the accounting of task run-time occurs on every sample in our test series, unlike the ARM platform, where this event is an exception. Another difference to ARM is that the accounting of task run-time also occurs at Softirqs, although it has a less significant overhead, compared to for Workqueues on x86.

V. Related Work

Wilcox [23] and Rothberg [19] present general surveys of the Linux interrupt handling subsystem, and the variety of second-level handler mechanisms. However, both lack an

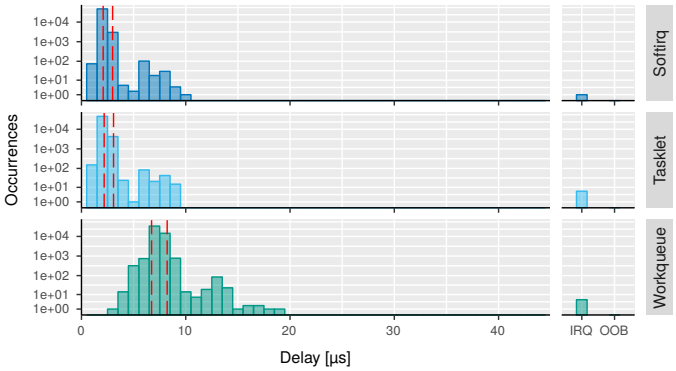

Fig. 10. Latency measurements for all three bottom half mechanisms without system load on the Intel x86 hardware platform. The latencies are lower due to the faster processor, but show higher jitter compared to the ARM platform.

empirical evaluation of latency and jitter associated with the interrupt handling mechanisms.

Regnier et al. [16] compare the interrupt handling times of two popular real-time extensions of Linux, preempt-rt and xenomai. Their evaluation measures the two aspects of interrupt-related latency: First, the interrupt latency is defined as the duration between an interrupt request and the start of the first-level interrupt handler function. Second, the activation latency is the duration between an interrupt request and the moment in time when an high-priority application starts running, which has been waiting for this interrupt. This paper extends the analysis by dissecting the latency of second-level interrupt handler mechanisms.

Besides interrupt handling, another important factor for the effective latency of interrupt handling is the operating system scheduler. If a user-space application waits for events, handling the interrupt is only the first step of information processing. In the following steps, the application waiting for the information has to wake up, and must be scheduled. To evaluate scheduling latency, Calandrino et al. [24] present LITMUS^{RT}, a testbed to compare real-time schedulers [25].

On the operating-system level, Abeni et al. [26] identify two further factors for the effective system latency: The timer

resolution and non-preemptable sections where control flows are delayed due to concurrency control. INTSPECT helps identifying such jitter causes in interrupt handlers and, thus, selecting suitable mechanisms that tolerate concurrency.

Interrupt latency can have a significant impact on the whole-system performance [27], [28]. Especially for large distributed systems, the performance depends on the *tail latency* [29]. These systems therefore need latency hiding and tolerance techniques [30]. This paper presents a tool to evaluate the interrupt latency, which is crucial in networked systems [31]. In summary, the information gathered with INTSPECT assists system designers to improve performance and predictability.

VI. Conclusion

Operating systems must handle interrupts with predictably low latency in order to be responsive to (external) events. The complexity of modern hardware and software requires system designers to measure interrupt latencies and the interference of system activities at run-time. To this end, this paper has presented INTSPECT, a tool for systematic interrupt latency measurement in Linux. We demonstrate the applicability of our tool by a quantitative evaluation of interrupt-related latencies of a recent Linux kernel (Linux v4.9), on an ARM and an Intel platform. Our analysis reveals that the Workqueue latency benefits from low system load, but increases at high load. In future work, we will evaluate different Linux variants in various load scenarios, further analyze the interdependence between interrupt latencies and the process scheduler, and utilize this information to optimize Linux for predictably low interrupt latency, and consequently, improved system responsiveness.

Acknowledgements

This work was partially supported by the German Research Council (DFG) under grant no. SCHR 603/13-1 ("PAX"), grant no. SFB/TR 89 ("InvasIC"), and grant no. SCHR 603/8-2 ("LAOS").

References

[1] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th Design Automation Conference (DAC'10)*. IEEE, 2010, pp. 731–736.

[2] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer, "Using latency to evaluate interactive system performance," *ACM SIGOPS Operating Systems Review*, vol. 30, no. si, pp. 185–199, 1996.

[3] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX, 1999, pp. 45–58.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI'14)*. USENIX, 2014, pp. 49–65.

[5] N. Hanford, V. Ahuja, M. Farrens, D. Ghosal, M. Balman, E. Pouyoul, and B. Tierney, "Improving network performance on multicore systems: Impact of core affinities on high throughput flows," *Future Generation Computer Systems*, vol. 56, pp. 277–283, 2016.

[6] J. D'Ambrosia, "100 gigabit Ethernet and beyond," *IEEE Communications Magazine*, vol. 48, no. 3, pp. 6–13, 2010.

[7] B. Davari, R. H. Dennard, and G. G. Shahidi, "CMOS scaling for high performance and low power-the next ten years," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 595–606, 1995.

[8] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[9] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'13)*. USENIX, 2013, pp. 44–47.

[10] B. H. Leitao, "Tuning 10Gb network cards on Linux," in *Proceedings of the Linux Symposium (OLS'09)*, 2009, pp. 169–184.

[11] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX, 2008, pp. 323–338.

[12] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS'18)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 24:1–24:25.

[13] H. Leppinen, "Current use of Linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[14] S. Reif, A. Schmidt, T. Hönig, T. Herfet, and W. Schröder-Preikschat, "X-Lap: A systems approach for cross-layer profiling and latency analysis for cyber-physical networks," *ACM SIGBED Review*, vol. 15, no. 3, pp. 19–24, Aug. 2018.

[15] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating system profiling via latency analysis," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX, 2006, pp. 89–102.

[16] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time Linux operating systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 52–63, 2008.

[17] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "On interrupt-transparent synchronization in an embedded object-oriented operating system," in *Proceedings of the 3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*. IEEE, 2000, pp. 270–277.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[19] V. Rothberg, "Interrupt handling in Linux," Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Tech. Rep. CS-2015-07, 2015.

[20] D. Bovet and M. Cesati, *Understanding the Linux kernel*, 3rd ed. O'Reilly Media Inc., 2005.

[21] D. A. Solomon and M. Russinovich, *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[22] OSEK/VDX Group, "Operating system specification 2.2.3," 2005.

[23] M. Wilcox, "I'll do it later: Softirqs, tasklets, bottom halves, task queues,work queues and timers," in *Proceedings of the 3rd linux.conf.au Conference*, 2003, pp. 1–6.

[24] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th International Real-Time Systems Symposium (RTSS'06)*. IEEE, 2006, pp. 111–123.

[25] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS^{RT}," in *Proceedings of the 9th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'13)*, 2013, pp. 19–29.

[26] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of Linux," in *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. IEEE, 2002, pp. 133–142.

[27] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, 2005, pp. 303–312.

[28] E. Vicente and R. M. Jr., "Exploratory study on the Linux OS jitter," in *Proceedings of the 2nd Brazilian Symposium on Computing System Engineering (SBESC'12)*. IEEE, 2012, pp. 19–24.

[29] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[30] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[31] S. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. Ousterhout, "It's time for low latency," in *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS'11)*. USENIX, 2011, pp. 1–5.