FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Luis Gerhorst

Analysis of Interrupt Handling Overhead in the Linux Kernel

Bachelorarbeit im Fach Informatik

3. Dezember 2018

Please cite as:
Luis Gerhorst, “Analysis of Interrupt Handling Overhead in the Linux Kernel”,
Bachelor’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU),
Dept. of Computer Science, December 2018.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Analysis of Interrupt Handling Overhead in the Linux Kernel

Bachelorarbeit im Fach Informatik

vorgelegt von

Luis Gerhorst

geb. am 1. Februar 1997
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: **Benedict Herzog, M.Sc.**
Bernhard Heinloth, M.Sc.
Stefan Reif, M.Sc.
Timo Hönig, Dr.-Ing.

Betreuender Hochschullehrer: **Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**

Beginn der Arbeit: **2. Juli 2018**
Abgabe der Arbeit: **3. Dezember 2018**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Luis Gerhorst)
Erlangen, 3. Dezember 2018

ABSTRACT

Modern operating systems have to be responsive to asynchronous events, regularly happening, for example, when computers interact with the user or the network. Interrupts are a basic mechanism used to signal such an event to the processor. Handling them with predictably low latency, is therefore required, to keep the system responsive. Today's computers, including both power efficient mobile devices (e.g., Android phones and Internet of things gadgets) as well as large-scale distributed systems, require complex hard- and software. Many of these systems employ Linux, a general purpose operating system that is freely available and open source. Since precise models to statically analyze the performance of those complex systems and their subsystems do not exist, measurement based approaches are required to ensure responsiveness. In Linux, interrupt handling is implemented as part of the kernel's interrupt subsystem. I developed the *INTerrupt Subsystem Performance Evaluation and Comparison Tool* (INTSPECT), which measures the interrupt handling overhead of the Linux kernel at runtime. A portable kernel component is developed, to measure the runtime delay of different interrupt handling mechanisms with high accuracy. To evaluate the performance of the kernel, INTSPECT is deployed for both an ARM embedded platform as well as an Intel x86 server computer. Using experiments executed on the hardware, we reveal interdependencies between user space workloads and the interrupt handling latency. Components of the kernel that interfere with low latency handling are identified, and similarities and differences between the two platforms are highlighted. For example, my results reveal, that the latency introduced by the *softirq* or *tasklet* interrupt handling mechanism, is at least five times lower than the latency introduced by the *workqueue* mechanism. In conclusion, using INTSPECT, system designers and driver developers can gain valuable insight into the performance of the Linux kernel's interrupt subsystem and finally use this information, to improve the responsiveness of the system at a whole.

KURZFASSUNG

Moderne Betriebssysteme müssen schnell auf asynchrone Ereignisse reagieren können, welche im besonderen regelmäßig durch den Nutzer oder das Netzwerk ausgelöst werden. Interrupts sind ein grundlegender Mechanismus, um ein solches Ereignisse an den Prozessor weiterzuleiten. Sie müssen daher mit vorhersagbar niedriger Latenz behandelt werden, damit das System schnell reagieren kann ist. Mobile Geräte (z.B. Android-Handys und IoT-Geräte) als auch große verteilte Systeme, benötigen komplexe Hard- und Software. Linux, als frei verfügbares und quelloffenes Betriebssystem, wird in diesem Bereich großflächig eingesetzt. Da präzise Modelle zur statischen Analyse der Leistungsfähigkeit dieser Systeme und ihrer Subsysteme fehlen, werden messbasierte Ansätze benötigt, um die Reaktionsfähigkeit zu bestimmen und schließlich zu verbessern. Das entwickelte Tool, INTSPECT, erlaubt es, die zusätzliche Latenz der Interruptbehandlung im Linux-Kernel zur Laufzeit zu bestimmen. Hierzu misst ein portables Kernelmodul die Latenz der unterschiedlichen Mechanismen, die zur Behandlung von Interrupts verwendet werden. INTSPECT wird sowohl auf einem eingebetteten Gerät mit ARM Prozessor, als auch einem Intel x86 Server installiert. Anhand von Experimenten auf der echten Hardware, werden Zusammenhänge zwischen der Anzahl der aktiven Benutzerprozesse und der Interruptverarbeitungslatenz aufgedeckt. Es wird analysiert, welche Komponenten des Kernels die schnelle Interruptbearbeitung beeinflussen und wie sich die beiden Systeme in diesem Zusammenhang vergleichen lassen. Meine Ergebnisse zeigen beispielsweise, dass die Interruptbehandlungslatenz beim *Softirq*- und *Tasklet*-Mechanismus, fünfmal kürzer ist als die Latenz, die beim *Workqueue*-Mechanismus auftritt. Insgesamt gewährt INTSPECT sowohl Betriebssystem- als auch Treiberentwicklern wertvolle Einblicke in das Interrupt Subsystem des Linux-Kernels.

CONTENTS

1 INTRODUCTION

Linux has become one of the most popular operating systems and has been ported to a variety of different hardware platforms and processor architectures. This has been primarily due to the rise of mobile and embedded devices, for which Android phones and Internet of things (IoT) products are two of the most prominent examples. However, also the majority of today's Servers and High Performance Computers is powered by Linux. For these types of devices, a freely available and customizable operating system, is the prevalent choice for most manufacturers. All of these devices, even very small ones, still contain a variety of components aside from their central processing unit (CPU), that implement communication with external hardware and the user.

A tap on the touch display of a smartphone, for example, is expected to cause the associated reaction from the device, which usually includes changes in the displayed content, immediately. To accomplish this, the processor has to execute code that implements the action requested by the user. If the delay until the processor starts carrying out the requested task, is too large, the system can easily become unresponsive and therefore be frustrating for customers to use. User interface guidelines regularly cite 200 ms as a delay, that should not be exceeded when a reaction to user input is pending [1]. Maintaining this may initially not seem problematic, as even today's smartphone processors offer clock rates in the GHz range, which can be translated into about one million instructions executed every millisecond. However, both communication using wireless networks (e.g., WLAN or Bluetooth), as well as accesses to persistent storage media, such as flash memory, require the processor to interact with the respective hardware component. Accessing a single website may involve numerous processor-device interactions, both on the client, as well as the server side.

1.1 Motivation

The example from the previous paragraph shows, that a fast reaction to an asynchronous event, is crucial for modern processors, as even basic user input may prompt numerous interactions between the hardware components. If the code used to implement the interaction is not highly efficient, this can easily cause noticeable delays for the user.

However, while a "laggy" user interface may be annoying, the consequences of a delayed reaction to an asynchronous event can be more dramatic in industrial applications. If the processor controlling the movements of a robotic arm, for example, does not react to input from its sensors in time, the machine could be damaged or even human lives could be put at risk. For this purpose, specialized *real-time* systems, which guarantee a timely reaction to signaled events, exist. However, general purpose operating systems, like Linux, are still regularly employed when a delayed reaction to events

is not catastrophic, but still causes extraordinary damage. This, for example, includes stock trading applications, but also the SpaceX Falcon 9 rocket [2].

As physical limits, such as the propagation delay of an electronic signal, as well as the hardware delay, can not be overcome, the low level system software must ensure that external events, signaled to the processor using *interrupts*, are reacted to in time. Operating systems, in particular, have to allow for efficient hardware drivers and responsive applications, by offering low-overhead abstractions of the raw hardware. Handling interrupts with predictably low latency, can, for example, reduce the risk for overload situations (e.g., congestion on network links [3]), but also increase the throughput of bulk I/O operations [4].

As the single-core processor speed is becoming a bottleneck, applications have to be adapted to benefit from multi-core processors and networks with high throughput [5, 6]. The coordination between threads running on different processor cores, as well as communication over the network, can both require interrupts. Operating systems that allow for efficient handling of interrupts, thus support the development of applications, that scale well for future hardware.

Efficient interrupt handling is also important in embedded applications, as these devices regularly rely on battery power, and therefore have to maximize their sleep time in order to stay functional [7]. The execution time of an interrupt handler, being the code run by the processor when it reacts to a signaled event, directly affects the energy demand of these systems [8]. Here, operating system software has to allow for sufficient performance, while minimizing the amount of energy consumed.

1.2 Problem Statement

To make use of an external component, the processor regularly has to assign it a tasks and await its respective completion. If this interaction is implemented in a naive way, the CPU would repeatedly poll the devices for updates, hence wasting a noticeable amount of resources. Instead, by allowing devices to interrupt the processor in hardware, whenever updates are available, CPUs can eliminate the polling overhead and react to events asynchronously. When an external component triggers an interrupt, the CPU suspends the execution of the currently running task in favor of a previously installed piece of code, that handles the signaled event. After having processed the event, the CPU transparently returns to the regular control flow.

In order to simplify interrupt handling code, avoid race conditions, and eliminate the risk for stack overflows, interrupts are usually executed with run-to-completion semantics: while an interrupt is handled, the interruption mechanism is disabled, introducing a delay in the handling of interrupts that arrive in the meantime. For this reason, interrupt handlers aim to be as short as possible, which is potentially in conflict with the complexity of the external event. To still allow the handling of complex events in an easy manner, the *prologue-epilogue model* was developed [9]. The prologue, executed in the context of the interrupt handler, is executed while interrupts are disabled. Only taking care of the actions that have to happen immediately, such as picking up new data from a device, the execution time is reduced to a minimum. To carry out more complex, non-critical tasks, such as further processing of the received data, the prologue requests an epilogue, which runs in a execution context of lower priority with interrupts enabled. When new interrupts arrive during the execution of an epilogue, their prologues are executed immediately without delay. However, epilogues requested in the interrupting prologues, are just enqueued and only executed when the currently running epilogue finishes. In conclusion, the prologue-epilogue model allows complex operations to be carried out in response to an interrupt, while keeping the chance of interrupts not being handled in time or getting lost, small.

In Linux, the prologue-epilogue model is implemented by splitting the interrupt handling code into *top* and *bottom half*. While the top half is executed immediately when the interrupt occurs, the execution of the bottom half is delayed, and carried out in another execution context. Multiple mechanisms exist, to request further handling of an interrupt. The specific implementation required to enqueue, prioritize and finally execute the bottom halves, determines the functional properties and overhead associated with the mechanism. To allow for a qualified decision when choosing the appropriate bottom half mechanism, and to make well-directed improvements to the kernel code base, the time delay between enqueuing and dispatching is of special interest.

To determine the properties of a software system, static analysis is usually preferred, as it allows for more general statements, than experiments carried out in a particular environment. However, modern processors are increasingly complex, and regularly lack a specification of their timing behavior, as exposing their internal structure (e.g., cache architecture) to competitors is avoided by manufacturers. In addition, since Linux is a general purpose operating system, it has to support a large number of application areas, which inevitably leads to complex code. This makes a complete static analysis of the system to date impossible. However, system designers still have to face the challenge of designing low-latency software for an unpredictable environment. Measurement based approaches, which record the performance of the system at runtime, are needed to fill this gap. These systems allow users to determine the timing behavior of their specific setup, and help system designers identify causes for bad performance in real-world scenarios.

1.3 Approach

The goal of this work is to measure the overhead in time imposed by the use of interrupt bottom halves in the Linux Kernel, namely softirqs, tasklets, and workqueues [10, 11]. For this purpose, I developed the *INTerrupt Subsystem Performance Evaluation and Comparison Tool* (INTSPECT), a tool that measures the runtime overhead of different bottom half mechanisms using a portable kernel component named INTSIGHT. A typical problem when measuring time delays is insufficient accuracy. To accurately measure latencies and minimize interference into the evaluated code path, the processors clock cycle counter register is used. The register allows for time measurements with only few assembler instructions, making the accesses cheap and accurate [12]. By inserting trigger code directly into the kernel binary, we can minimize overhead while maintaining maximum measurement flexibility. In addition, this allows INTSPECT to trace which kernel components are invoked in the observed section, therefore identifying causes for delayed bottom halves.

1.4 Related Work

Rothberg [10] and Wilcox [13] present the implementation of the Linux kernel's interrupt subsystem, as well as the kernel interfaces, that allow for delayed execution of tasks. This includes the bottom half mechanisms also presented in the background chapter of this thesis. However, the main subject of this work, being the measurement of the overhead imposed by the mechanisms, is covered by neither Rothberg nor Wilcox.

Regnier, Lima, and Barreto [14] evaluate the extent to which real-time extensions to the Linux kernel, namely Preempt-RT and Xenamai[1], provide deterministic guarantees when reacting to external events. They measure the interrupt latency, defined as the time delay between an interrupt request (IRQ) and the execution of the corresponding interrupt service routine (ISR), as well as the

[1] https://xenomai.org/

delay until a (real-time) thread, woken up in the ISR, is scheduled. The activation delay of a thread woken up inside an ISR is related to the overhead imposed by the usage of workqueues for interrupt handling, since this mechanisms relies on threads for the execution of requested tasks. However, this work focuses specifically on bottom halves and performs its experiments on a kernel with no real-time extensions installed. In addition, the analysis made for softirqs and tasklets extends the article, since those mechanisms may further delay the activation of a thread awaiting an event.

Calandrino, Leontyev, Block, Devi, and Anderson [15], as well as Cerqueira and Brandenburg [16], evaluate real-time scheduling algorithms. However, the kernels current default scheduler, that is the Completely Fair Scheduler (CFS) which also is employed in this work, is not evaluated [17]. The performance of the scheduler is relevant for workqueues, since this mechanism relies on threads to execute the bottom halves.

Abeni, Goel, Krasic, Snow, and Walpole [18] identified the frequency of the periodic timer interrupt, which is also recognized in the evaluation of this thesis, and non-preemptive sections as the main sources of latency in the Linux kernel. Vicente and Matias [19] identify the processor cache to be a significant source of OS Jitter, which INTSPECT accounts for as described in Section 4.1.3.

As network latency is reduced to microsecond scale, the overhead imposed by interrupt handling in the Linux kernel becomes a bottleneck for networked systems [20]. INTSPECT allows system designers to determine and improve the performance of the interrupt subsystem, which is crucial for applications that rely on low latency.

1.5 Publication

This thesis contains research results of the following paper, which haven been published in a peer-reviewed conference:

[21] Benedict Herzog, Luis Gerhorst, Bernhard Heinloth, Stefan Reif, Timo Hönig, and Wolfgang Schröder-Preikschat. “INTSPECT: Interrupt Latencies in the Linux Kernel.” In: *Proceedings of the 8th Brazilian Symposium on Computing Systems Engineering (SBESC’18)*. 2018

In [21], I contributed the INTSPECT implementation for the ARM hardware platform and assisted Bernhard Heinloth in porting it to Intel x86. The paper motivates the usage of epilogues for interrupt handling, presents the implementation of INTSPECT, and analyzes the performance of the three bottom half mechanisms using the tool. All three topics are also subject of this work, but are presented with recent findings and background knowledge added. In addition, the feedback received at the conference is taken into account. For example, the actions taken to ensure high measurement accuracy are subject of Section 2.4, Section 3.2, and Section 5.2. In particular, Section 4.1.3 explains, why subtracting the recording overhead from the results is not feasible for INTSPECT. Following this thesis, we plan to publish INTSPECT[2], as well as the associated kernel module[3] under an open source license. The former includes the raw results used in the evaluation of this thesis.

1.6 Overview

This thesis is structured in six chapters. The following Chapter 2 introduces background knowledge about the interrupt handling subsystem of the Linux kernel. In particular, the kernel’s different bottom half mechanisms are introduced. Thereafter, the chapter briefly presents the existing

[2] https://gitlab.cs.fau.de/i4/intspect
[3] https://gitlab.cs.fau.de/i4/intsight

timekeeping interfaces considered when implementing INTSPECT. Chapter 3 deduces the basic design of INTSPECT, a tool that measures the runtime overhead introduced by each bottom half mechanism, from the functional and non-functional properties desired for it. This includes accuracy and reproducibility of the measurements, as well as the support of multiple test setups. The implementation of the tool, which includes the kernel component INTSIGHT, is presented in Chapter 4. The chapter describes the interacting components, as well as the interfaces they offer. Thereafter, Chapter 5 presents the results obtained using INTSPECT on an ARM-based embedded platform and an Intel x86-based server computer. It compares the runtime overhead imposed by the usage of softirqs, tasklets, and workqueues for interrupt handling, and uncovers causes for delays. Using INTSPECT's features, I evaluate how changes in the environment, such as system load or the frequency of the test interrupt, influence the results. Finally, Chapter 6 concludes this thesis.

2 BACKGROUND

This chapter introduces concepts and kernel interfaces relevant for understanding this work. Based hereon, the following chapters present the design, implementation, and evaluation of INTSPECT, a tool, that measures the runtime overhead of interrupt handling in Linux. First, Section 2.1 introduces the basic idea and implementation of hardware interrupts. Thereafter, Section 2.2 motivates the prologue/epilogue model, and Section 2.3 presents its implementations in Linux. Finally, since implementing INTSPECT requires a method for measuring the execution time overhead of the different implementations, Section 2.4 discusses timekeeping interfaces offered by the kernel.

2.1 Interrupt Handling in the Linux Kernel

Computers regularly have to react to events that prompt an immediate reaction, but whose time of occurrence cannot be predicted. A key press, for example, is expected to instantly make the corresponding letter appear on screen. Similarly, when the computer is connected to the internet, incoming data, for example, an instant message, must be delivered to the user with minimal delay. In addition, a variety of unpredictable events, not visible from the outside, do exist. Internally, the CPU accomplishes many of its tasks in cooperation with other hardware components. It regularly outsources tasks to other devices, for example, reading data from a connected hard drive, and processes the data only when it is readily available.

To implement these interactions, the CPU could regularly poll connected devices for updates [10, p. 2]. However, when devices are queried too infrequently, the delay until events are recognized, may be intolerably large. In contrast, too frequent polling wastes CPU cycles since most components are likely to not have new events pending most of the time.

Interrupts solve this problem by giving devices a way to asynchronously notify the CPU in hardware whenever they require attention [22, p. 258]. As soon as a device signals an event, the CPU core suspends execution of the current thread and transfers control to a previously installed handler function, reacting to the specific event [10, p. 9]. Figure 2.1 illustrates this series of events, and will be further extended in the following sections. When the interrupt handler returns, the execution of the previously executing program is continued transparently. As a consequence, regular programs may be interrupted at almost any point in time, since interrupt handling takes precedence over user threads. Only the kernel can *mask* interrupts to temporarily postpone the reaction to pending IRQs [22, p. 273]. This happens, especially, while executing an interrupt handler. Until finished, the triggering interrupt is masked, making the handler run to completion: only after the current event is completely handled, processing of the next event may start [10, p. 3]. Interrupt handlers usually have to interact with other hardware devices in order to accomplish their tasks, for example, fetching newly available data from an external storage device [22, p. 269]. Since this

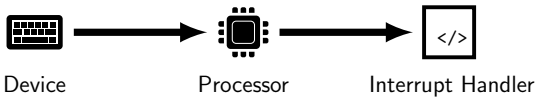

Figure 2.1 – Devices trigger interrupts using dedicated connections to the processor. In response to the IRQ, the processor executes a previously installed interrupt handler function. The figure is based on [21, Figure 2].

interaction is by design not possible in user space, the processor automatically switches into kernel space before transferring control to the handler function.

Based on the aforementioned features of interrupt handlers, there are three use cases for them in operating systems [23, Chapter 4]. First, the automatic change in privilege level on occurrence of an interrupt is employed to implement the synchronous switch from user to kernel space in a safe manner, regularly happening, for example, when an application issues a system call. Second, unexpected runtime errors, for example, page faults or illegal memory accesses, which make the continuation of the current control flow impossible, also trigger dedicated interrupts. In these cases, the handler function maintains system integrity, for example, by loading the memory page to be accessed from a hard drive or by terminating the erroneous process. Finally, external devices use interrupts to signal asynchronous events, which require an immediate response, as elaborated in the previous paragraph.

Each interrupt is identified by a number. External hardware requests specific interrupts over dedicated connections to the CPU, called *interrupt lines*. Following an IRQ, the handler function to be executed is looked up in the *interrupt vector table* using the interrupt number as identifier. Linux offers driver developers abstractions, that allow the installation of ISRs in a platform-independent way. This system even allows multiple drivers to share one interrupt line [22, pp. 259,278]. In that case, the kernel will eventually have to poll multiple ISRs in order to properly react to the IRQ [11].

Since the majority of tasks performed by a computer directly or indirectly involve interrupts, their efficient implementation is an important feature of general-purpose operating systems. Linux, for example, uses interrupts for user input from keyboards or touchscreens, communication over Ethernet or wireless networks, but also general system tasks, like the process scheduler or power management [11].

2.2 The Prologue/Epilogue Model

The occurrence of an IRQ triggers a CPU core to suspend execution of the current thread and jump to a previously installed interrupt handler function instead. While executing the handler, interrupts are masked and their handling is thus delayed until the running handler completes. This greatly simplifies handler code, because it does not have to be reentrant. Furthermore, allowing an interrupt to preempt its own handler over and over again, would open the door to infinite recursion, and thus, on systems with finite memory, cause stack overflows [10, p. 3]. For these reasons, delaying the IRQ until the currently running handler finishes, is preferred, even if hardware limitations may cause the loss of interrupts when multiple requests arrive during the execution of a handler. In addition, delaying interrupts for too long can cause the triggering device to misbehave due to hardware limitations, such as limited buffer sizes inside of components which receive data from the outside world. In order to reduce the risk of interrupt loss and misbehaving devices due to interrupts not being handled in time, it is necessary to keep handler functions as short as possible. This stands in conflict with the amount of computing effort prompted by some interrupts [22, pp. 270, 275]. For

Figure 2.2 – Event series on occurrence of an interrupt when the prologue/epilogue model is used. The interrupt handler from Figure 2.1 is split into prologue and epilogue. The figure is based on [21, Figure 2].

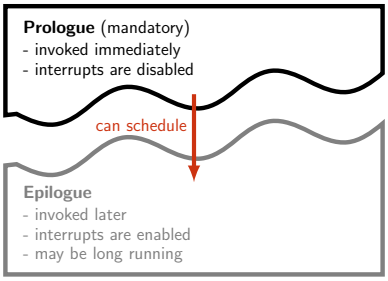

Figure 2.3 – Dividing interrupt handlers into prologue and epilogue. The prologue is invoked immediately on occurrence of the IRQ, and interrupts are masked while its executing. For long-lived tasks, that are not time critical, the prologue can schedule an epilogue, which is executed when interrupts are enabled again. The figure is based on [21, Figure 1].

example, fetched data from a hardware device may have to be decoded and cause complex changes in the system state, which can be time-consuming. However, since task like this neither have to be executed immediately nor do interrupts usually have to be disabled while they are accomplished, such postprocessing may as well be delayed and carried out in a different execution context. The prologue/epilogue model offers the developer an easy way to accomplish this [9]. When an interrupt prompts time-consuming tasks, the interrupt handling code is split into two parts [22, p. 275]: the prologue, also called top half in Linux, is executed immediately on occurrence of the IRQ while the triggering interrupt is still masked. The epilogue, however, named bottom half in Linux, is delayed and executed when the processor has again switched to an execution context that allows for interrupts. Figure 2.3 illustrates the division between prologue and epilogue and Figure 2.2 integrates the model into the figure from the previous section. All time-critical parts are placed in the top half, whereas for the non-critical parts an epilogue is requested. When the top half is finished, interrupts are enabled again and requested epilogues are executed. It shall be noted, that, since interrupts are enabled while an epilogue executes, a prologue may preempt it immediately on occurrence of an IRQ. If this prologue requests an epilogue itself, it is likewise only enqueued into a data structure. Then, when the prologue returns, the CPU first continues the execution of the original epilogue. Only after the original epilogue has finished, the execution of the newly requested one is started.

This subdivision of interrupt handling into time-critical and non-critical parts is a concept common to many operating systems. On UNIX systems, such as Linux, the separate parts are usually referred to as top and bottom half [10]. Other systems however offer similar libraries: Windows offers *deferred procedure calls* [24] and OSEK *interrupt service routines*, which defer work from an interrupt into regular execution context [25].

Since this work focuses on Linux, the mechanisms the kernel offers to defer work are presented in the following. Section 2.3 only discusses the functional and expected non-functional properties

of each mechanism. The measured performance of the different mechanisms will be examined extensively in Chapter 5.

2.3 Epilogues in the Linux Kernel

The previous section introduced the prologue/epilogue model in general. This section discusses the mechanisms offered by Linux to defer work from an interrupt handler into another execution context, that is, request an epilogue from within a prologue. For this purpose, three mechanisms exist. Softirqs form the least flexible mechanism, but their implementation is straightforward and, as Chapter 5 shows, they are also the most efficient. Tasklets are layered on top of softirqs, making them less efficient, but usable for driver developers who create loadable kernel modules. In contrast to these two mechanisms, workqueues are the only one to execute tasks in process context. However, in return, they introduce greater runtime overhead.

2.3.1 Softirqs

Softirqs are an elementary mechanism that allows the kernel to defer work from an uninterruptible to an interruptible execution context [10, pp. 9, 10]. They are employed for very frequent tasks associated with networking, timers, and the block layer, but are also used to implement tasklets [11], which are the subject of the following subsection. The handler functions for pending softirqs are executed whenever the kernel returns from an interrupt or switches from user to kernel space.The different softirqs are specified at compile time in an `enum`, thus it is not possible to allocate new softirqs at runtime. Also, as of kernel version 4.9, the implementation allows for at most 32 different softirqs, however, only ten slots are currently in use. Before one can request the execution of a softirq, which is referred to as a number, it first has to be *opened*, that is, a handler function has to be associated with it. After having registered the handler, one can raise the softirq whenever appropriate which marks it as pending. The order, in which pending softirqs are executed, is determined by their position in the enumerator.In their main execution context, that is, after a hardware interrupt, softirq handlers are executed with run-to-completion semantics.They may thus only be preempted by newly arriving hardware interrupts. The code that executes pending softirqs does not provide any serialization when a specific softirq is invoked, consequently, one handler may run on multiple CPU cores at the same time.It is the responsibility of the handler developer to ensure proper synchronization using spinlocks, when data structures that are not CPU-local are used.Thus, using CPU-local data is preferred inside softirq handlers. Since the kernel executes softirqs on the CPU core they were raised on, this practice in addition maximizes cache locality.

There is no guarantee that the control flow executing the softirq handler has an associated process context, thus handlers are not allowed to sleep [21]. Also, since individual handlers run to completion when executed after a hardware interrupt, the handler should not consume huge amounts of CPU time. Although hardware interrupts can still preempt it, all other tasks are usually delayed until the handler completes. But even when the execution time of individual handlers is kept short, user and kernel tasks may still be delayed indefinitely when there are many pending softirqs or new ones are requested continuously. To prevent this, every time a handler completes the total time spent executing softirqs is checked. When a limit of 2 ms is exceeded, the iteration over the pending softirqs is not continued. Instead, a dedicated CPU-local thread, called `ksoftirqd/n`, where *n* is the CPU core identifier, is activated.Being manged by the scheduler, it works off pending softirqs while not suppressing other tasks.

In conclusion, the main motivation for the usage of softirqs is the ability to move relatively CPU-intensive tasks out of hard interrupt handlers. However, since all softirqs have to be allocated at compile time, this epilogue mechanism is not usable for kernel modules loaded at runtime. This limitation is met by tasklets, which are described in the following section.

2.3.2 Tasklets

Technically, softirqs are simply indexes into the centrally allocated softirq table, which maps numbers to handler functions. In contrast, tasklets are pointers to individual `structs` containing similar information, but with the important distinction that these data structures can be allocated anywhere inside the kernel [11]. This makes them particularly valuable to developers of loadable kernel modules, from which the allocation of new softirqs is simply not possible. Also, there is virtually no limit on the number of different tasklets existing in a system at a given time. Thus, there is no reason to discourage the allocation of new tasklets as there is for softirqs.

An allocated tasklet `struct`, which mainly consists of the handler function and an argument passed to it, can be raised given the pointer to it [10]. *Raising* a tasklet enqueues it into a CPU-local linked list and marks a specific softirq, dedicated to tasklet execution, as pending. The next time pending softirqs are executed, the handler for the tasklet softirq iterates over the CPU-local list, into which raised tasklets were enqueued, and calls their associated handlers. As a consequence, the execution context of tasklets is the same as for softirqs: no process is associated with the calling control flow, thus tasklet handlers are not allowed to perform operations that would to block the current thread. To improve cache locality, tasklets are, just like softirqs, always executed on the CPU core on which they were raised [22, p. 276].

In fact, there are two separate but almost identical implementations for tasklets inside the kernel.One for tasks of regular priority, and one for high priority ones. Although they share many definitions, the CPU-local queues for example are separate. The main difference between the high/normal priority implementation is, that the softirq that executes high-priority tasklets is located right at the beginning of the softirq vector and thus takes precedence over other potentially pending softirqs. The softirq that works off tasklets enqueued with regular priority, however, is located after the softirqs dedicated to network and block operations.

The main difference between tasklets and softirqs is the way they are allocated. Still, there is also a small difference in the way handler functions are executed. Before calling the handler for a tasklet, a lock on the tasklet `struct` has to be acquired. When the lock is already taken, the tasklet is re-enqueued for the executing CPU core and the softirq dedicated to working off the tasklets is marked pending again. Finally, the iteration over the remaining tasklets that were in the queue is continued.The main consequence for users of tasklets is, that one tasklet can run on at most one CPU core at any given point in time.This simplifies handler code because potential concurrency hazards are avoided efficiently by the tasklet implementation.

In conclusion, tasklets are a bottom half mechanism designed for widespread use inside the kernel. They are easier to use than softirqs and there is no upper limit to the number of different tasklets allocated in a system. However, the context in which the handler functions is executed is almost the same as for softirqs. When the handler function has to perform operations that sleep, both mechanisms cannot be used. This limitation can be lifted by using workqueues instead, which are the third mechanism offered by the kernel for deferring work from interrupt handlers.

2.3.3 Workqueues

Workqueues can be used to defer work out of interrupt context, that requires operations which may sleep or block [10, 11, 21]. During the interrupt, *work items*, which encapsulate the tasks to be performed, are enqueued into *workqueues*. Dedicated *worker threads*, started by the kernel, later dequeue these items and call the handler functions associated with them. When the called handler function has to sleep or requires large amounts of CPU time, the scheduler can kick in and transfer control to another process. This prevents starvation of kernel and user threads due to long-running bottom halves and thus improves system responsiveness.

The kernel threads that work off the items enqueued are organized in *worker pools*. Since the load put on a workqueue varies depending on the number and type of the enqueued tasks, the size of the worker pools has to be adapted dynamically when system resources shall not be wasted. First, in order to save CPU time, worker threads whose associated workqueue runs empty enter an idle state. When new work items are enqueued, they are woken up again and thus can continue dequeuing work items and executing their handlers with minimal delay. Second, in order to not waste memory for processes which spend most of their time doing nothing, worker threads that have been idle for a long time are destroyed. However, the implementation guarantees that at least one worker exists for each pool at any point in time. In reverse, when the number of worker threads is deemed too small, that is, the CPU is not maxed out although pending work items exist, the implementation spawns new threads on demand. This can happen, for example, when the handler functions to which the workers transfer control perform operations which make the associated process become idle.

The ability to sleep or block inside the deferred task substantially simplifies the usage by making the development of handler functions less error prone. However, the implementation of workqueues is much more complicated than the implementations of tasklets and softirqs. Connected to this, the overhead to enqueue work items is increased. In particular, waking up idle threads is implemented by the scheduler, which is the reason why the runtime of the enqueuing operation is in general not predictable. However, with the added complexity also comes added flexibility. Both tasklets and softirqs are always executed on the CPU core on which they were triggered. While cache locality can be maximized this way, it can become a problem when the executed work items require large amounts of CPU time, thus blocking the core they are bound to. *Unbound* workqueues solve this problem. Work items enqueued into such queues may be executed on any CPU core. In addition, similar to tasklets, a *bound* workqueue can be of either normal or high priority, determining the priority of the worker threads used to dispatch the queued work items.

In conclusion, workqueues are the mechanisms used to defer work from interrupt handlers in the majority of cases. Users do not have to consider whether the task performed may run for a long time or whether it performs operations that block the executing control flow. This greatly simplifies the usage in comparison to softirqs and tasklets.

2.3.4 Execution Priority Summary

Figure 2.4 illustrates the events following an interrupt [11, 21], further extending Figure 2.2. First, the kernel executes the top half which potentially requests the execution of a softirq, tasklet or work item. When the top half finishes, interrupts are unmasked and pending softirqs or tasklets are executed immediately. Thereafter, the kernel restores the previous execution context, thereby potentially continuing the execution of an application executed beforehand. Workqueue items are executed as soon as the scheduler is conducted and dispatches the associated worker threads. While softirqs and tasklets usually run to completion, work items put into workqueues are executed by dedicated kernel threads, making them preemptible by user-space applications.

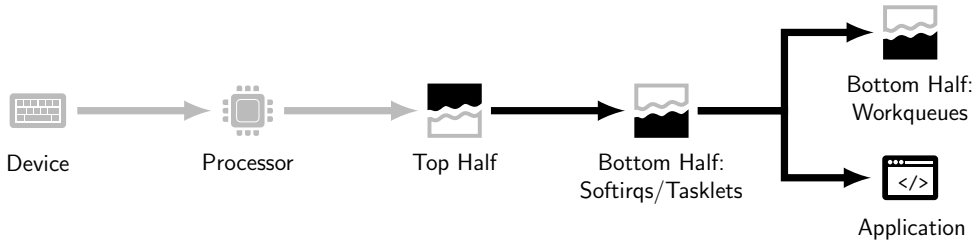

Figure 2.4 – Execution priority of the mechanisms implementing the prologue/epilogue model in the Linux Kernel. Pending softirqs and tasklets are executed before the scheduler is conducted. Workqueues, in contrast, use dedicated worker threads that compete with application threads for runtime. The figure is based on [21, Figure 2].

2.4 Time Measurement Interfaces in the Linux Kernel

The goal of this work is to measure the latency between a prologue and the corresponding epilogue. To do this, a way to efficiently record the point in time, at which a certain piece of code was executed, is needed. From these records, the time delay, and thus the overhead of certain operations can be derived. Linux offers multiple interfaces to retrieve timestamps, each having specific functional and non-functional properties [11]. The following sections present these interfaces, focusing on the features desired when implementing INTSPECT.

2.4.1 Time of the Day

The function `getnstimeofday()` stores the current time of the day into a provided memory location. The storage format allows for nanosecond accuracy, the actual accuracy, however, is dependent on the hardware platform. Since the timestamp represents the current wall time, it does not always increase monotonically and thus is not suited for latency measurement. The returned timestamp is calculated by applying a variable offset, whose value is determined, for example, using communication with NTP-servers, to the monotonically increasing system clock source.

A call to `getnstimeofday()` is not guaranteed to complete in constant time. Since interrupts may cause invalid readouts, they may have to be repeated internally until successful. As part of the experiments carried out for Section 5.2, it was discovered, that reading out this timestamp very frequently can cause its speed to stagnate, making time “move slower” for the caller. Thus, the usage of this interface for latency measurement is highly discouraged by the author or this work.

2.4.2 Kernel Time

The function `ktime_get()` returns the current value of the monotonically increasing system clock source, converted to nanoseconds. The fundamental source and precision of the returned timestamp is thus the same as for the `nstimeofday`-interface. However, no offset is applied to the value returned and thus it is in principle suited for latency measurement. But as for `nstimeofday`, the time it takes to read out the timestamp is not predictable, since interrupts may prompt the implementation to repeat certain operations internally.

The kernel interface `ktime_get_mono_fast_ns()`, just like `ktime_get()`, accesses the monotonically increasing system clock source, but can be used inside of non-maskable interrupts (NMIs). It is also faster than `ktime_get()`. In return, however, the retrieved timestamp is not guaranteed to always increase monotonically when accessed inside of NMIs or on multi-core CPUs.

In conclusion, because the returned timestamps reflect the time elapsed since system boot, `ktime` is in principle well suited for latency measurement. However, whether the readout time is acceptable is to be determined in Chapter 5. `ktime_mono_fast` likely allows for a faster readout and thus also poses an option if `ktime` is too slow, and, when the conditions under which `ktime_mono_fast` does not increase monotonically are guaranteed to not occur on the runtime system.

2.5 Summary

This chapter presented existing concepts related to interrupt handling, and implementations thereof in the Linux kernel, building the basis for understanding the following chapters. Interrupts are needed to efficiently implement the communication with external devices, but in order to prevent them from being lost or delayed, handlers have to be kept short. The prologue/epilogue model makes the outsourcing of work from interrupt handlers into an interruptible execution context easy. The kernel has three implementations that enable this. First, softirqs are very efficient but their usage is restricted to few major kernel components. Second, tasklets are designed for widespread use, for example, in loadable kernel modules, and writing tasklet handler function is simplified because they may not execute on multiple CPU cores simultaneously. Finally, workqueues are used when the outsourced work requires a process context to execute.

In addition to this, the final section presented the timekeeping interfaces existing in the kernel. Of those, `ktime_get()` and `ktime_get_mono_fast_ns()` may be suited for measuring the runtime overhead of interrupt handling. Thus, they will be evaluated, together with other custom timekeeping mechanisms, in Section 5.2.

3 DESIGN

This chapter presents the design of INTSPECT, a tool that measures the runtime overhead of interrupt handling in Linux with high accuracy. This overhead is defined as the additional delay imposed by the usage of the prologue/epilogue model. Measuring it as precisely as possible is the main goal of INTSPECT. However the tool is also portable between different kernel versions and hardware architectures, and its results are easy to validate and reproduce. Section 3.1 motivates INTSPECT's functional features and proposes designs that implement them. Thereafter, Section 3.2 focuses its non-functional properties, namely making *precise* measurements and being portable.

3.1 Functional Properties

Basic Procedure

The runtime overhead of interrupt handling in Linux, being the additional latency introduced when splitting the handler into top and bottom half, is composed of the following parts:

- **Request Delay:** In order to have the kernel invoke the bottom half at a later time, the top half has to call functions to request the invocation of the bottom half. In the case of tasklets, for example, this includes enqueuing the tasklet `struct` into a linked list.

- **Invocation Delay:** Ideally, a requested bottom half would execute as soon as the top half returns. However, in practice, the kernel may first perform some high-priority system tasks, and, in the case of workqueues, even allow user processes to run.

Both the request, as well as the invocation delay contribute to the total time that elapses until the interrupt is completely handled, consequently, both are measured by INTSPECT. In addition, INTSPECT discovers events and conditions that lead to increased latencies. To accomplish these goals, the following basic design is used: At the beginning of each *measurement run*, an interrupt is triggered. The top and bottom halves of the interrupt handler both record *checkpoints*, each consisting of the current time, and an identifier for the point of execution, into an ordered collection. In addition, further checkpoints are recorded inside the code executing between top and bottom half, as well as the code requesting the bottom half from within the top half. By examining which checkpoints are encountered in which order, the conditions that contribute to the total delay are discovered. INTSPECT trades off flexibility for maximized measurement precision, by fixing the points in code where checkpoints are recorded at compile time.

Accounting for Jitter

On modern computers, the time a non-trivial operation takes usually varies from invocation to invocation. This is due to the complex state maintained by modern CPUs, which can only be in part accessed by the hosted application. The execution of a specific code path can be detected by placing a checkpoint in the relevant section. In contrast, caches or data structures used to predict the next branch, are in general inaccessible, but still impact performance. Since their state may be updated on almost every CPU cycle, associating one specific execution time with an operation is impossible on most systems. To account for this, the *measurement runs* described in the previous paragraph are repeated multiple times, forming a *benchmark*. This allows INTSPECT to identify the typical execution time, but also increases its chance of encountering special conditions, that lead to especially large or small latencies.

Supporting Multiple Measurement Methods

Section 2.4 concludes, that multiple interfaces offered by the kernel may be suited for latency measurements. In addition to those, other hardware methods may be available for which no generalized interface exist (i.e. custom measurement devices). To allow for a convenient decision on the measurement method best suited for a specific purpose, INTSPECT supports recording timestamps from multiple different sources in a single checkpoint. However, the more measurement methods are enabled, the longer the recording takes and thus the greater is the interference into the evaluated program. To still allow for measurements with minimal interference and high accuracy, this feature can be disabled at compile time.

Gathering Raw Data at Runtime

Maintaining the hardware required to measure the handling overhead for a specific system on a chip (SoC) can be a time consuming and costly task. Consequently, it is desirable to draw as many conclusions from a setup as possible, which involves analysis of the generated results. Decoupling the generation of raw data from its post-processing, is thus beneficial, since it allows users to draw new conclusions from an experiment, even if the test setup is no longer available. INTSPECT thus exports the raw measurements in a simple, efficient, and platform-independent format, and thereby postpones most processing until the data is analyzed. In addition to the recorded checkpoints, additional data about the runtime system is gathered in order to validate the system state and identify possible sources of anomalies. However, this collection of metadata does not affect the accuracy of the measurements.

Reproducibility

Making the data that justifies the findings presented in ones work available to third parties is best practice, since it allows them to validate the results. The raw data of an experiment, however, depends on many factors possibly distributed across different components of the host system. Not knowing the initial state of the system on which an experiment was performed, makes a validation of the conclusions drawn very hard or even impossible. Thus, INTSPECT allows for easy restoration of a test setup, by including all configs, source files, and disk images, required to rebuild the system, with the final results.

Changing the Test Environment

Once the overhead of interrupt handling in a specific setup is measured, it is also feasible to determine to what extent the results depend on the used environment. This includes static properties, for example, the hardware architecture or kernel version and configuration used, but also dynamic factors, such as the current system load. To analyze the latter, INTSPECT provides a mechanism, which generates defined load situations and runs the experiments under these defined conditions. For the former, no generalized solution is available, however, making INTSPECT modular and well-designed allows it to be easily ported to new hardware architectures and kernel versions. This property is further elaborated in Section 3.2.2.

3.2 Non-Functional Properties

Besides the functional features proposed in the previous section, INTSPECT puts the focus on two non-functional features discussed in the following. First, Section 3.2.1 analyzes, how the runtime overhead of recording checkpoints, and the resolution of recorded timestamps, affects accuracy. Second, Section 3.2.2 discusses, how a modular design can help in the supporting of multiple test environments.

3.2.1 Accuracy

INTSPECT measures the execution time an operation requires, by executing code that records the current point in time at the operation's initiation and completion. In addition, to further analyze how exactly the operation was performed, further records may take place while it is executing. Both the enclosing measurements, as well as those, within the operation, introduce runtime overhead that distorts the measured latencies. In principle, knowing the exact time a recording takes, would allow us to deduce the actual execution time of the operation, by subtracting the calculated overhead from the measured delay. However, since caches and branch prediction make the runtime of complex tasks unpredictable, recording must be as simple and fast as possible, keeping the interference predictably low. Choosing a measurement method that allows for consistently fast retrieval of timestamps, is thus desirable. Also, the memory, into which timestamps and associated metadata are recorded, must be readily available for write access. Especially on systems that employ paging for memory management, guaranteeing this poses a challenge.

Timestamps should not only be fast to retrieve, but should also accurately represent the point in time at which they are recorded. If the resolution of the underlying clock source is too low, analysis of very fast operations becomes impossible. Linux's interrupt subsystem has to be highly efficient if frequent system tasks involving interrupts, shall not be delayed unnecessarily. Thus, it is necessary for INTSPECT to utilize a very fine-grained clock source.

In summary, the measurement interface, should be as precise and fast to read out as possible. However, the recorded timestamps do not need to have any meaning other than their difference representing the time elapsed between readouts. For example, they are not required to correspond to the current wall time or time since system boot. Many CPUs allow access to registers that count the elapsing processor cycles. This metric may be well suited for execution time measurements, because its atomic unit, being one processor cycle, usually is the shortest period of time the CPU can devote to a task. In addition, when the cycle counters are registers located inside the CPU, their retrieval time is likely fast and thus easy to predict. However, the number of processor cycles a task consumes may not always correspond to the actual time spent on it. When other hardware components, for example, memory or hard disks, are involved, the CPU may lower its frequency or

even sleep in order to save energy while it waits for these devices. This would result in mistakenly lower latencies attributed to such tasks and therefore must be prevented.

3.2.2 Portability

In order to evaluate the results, it is best to provide data for a variety of different test environments. To support these, INTSPECT must be portable with respect to both the hardware architecture as well as the kernel version into which it is integrated. To make it platform independent, the architecture specific implementation has to be kept small and be isolated from the generic parts of the code. A modular and clean design also helps in supporting multiple kernel version, by making the required modifications easy to overlook and maintain.

4 IMPLEMENTATION

This chapter focuses on the implementation of INTSPECT. To measure the runtime overhead of softirqs, tasklets, and workqueues as precisely as possible, the tool employs a kernel module called INTSIGHT, whose implementation is described in Section 4.1. Thereafter, Section 4.2 describes the `sysfs` interface offered by the module. INTSPECT, focused on in Section 4.3, uses this interface to initiate benchmarks and retrieve measurement results from within user space.

Figure 4.1 illustrates the components interacting on the test system. Every time a benchmark is executed, the test system is first set up into a defined initial state (i.e., by flashing disk images containing the kernel and Linux distribution to it). A separate computer controlling the benchmark then boots the system and transfers control to local INTSPECT, which, if requested, generates artificial load by spawning applications. INTSPECT also initializes the kernel module INTSIGHT over `sysfs`. During the benchmark, the kernel repeatedly executes a specified experiment in isolation. When done, the generated data is made available to the user space. INTSPECT retrieves the results and transfers them to the separate system, where they are stored and analyzed.

4.1 Kernel Space

The goal of the INTSIGHT kernel component is to precisely measure the runtime overhead of Linux's interrupt bottom half mechanisms, which are presented in Section 2.3, with minimal interference, and expose the gathered raw data to the user space.

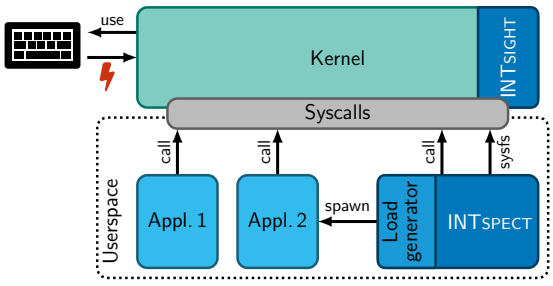

Figure 4.1 – Architecture of the INTSPECT tool. The portable kernel component INTSIGHT measures the performance of the interrupt subsystem with high accuracy. INTSPECT retrieves the raw results over `sysfs` and spawn additional applications to generate system load during a benchmark. The figure is based on [21, Figure 4].

Section 4.1.1 describes, how the kernel executes a benchmark initiated from user space. Section 4.1.2 focuses on the supported timekeeping interfaces, and Section 4.1.3 describes the procedure of recording a checkpoint, whose implementation is crucial for attaining our goal of minimal interference.

4.1.1 Benchmarking Procedure

```
1: allocate checkpoint buffer
2: for all measurement runs do
3:     trigger interrupt
4:     wait fixed time interval
5: end for
6: expose checkpoint records over sysfs
```

Algorithm 4.1 – Operations executed by the coordinating thread during a benchmark. Before the measurements, a buffer into which Algorithm 4.2 records checkpoints is allocated. Then, a configurable number of interrupts is triggered, each causing the execution of the handlers installed at initialization. Finally, the results are exposed to INTSPECT over `sysfs`.

At initialization, the INTSIGHT kernel component allocates both a dedicated interrupt service routine (ISR) as well as one softirq, tasklet, and workqueue item used for testing. Then, when INTSPECT triggers a benchmark, the procedure shown in Algorithm 4.1 is executed. First, the memory into which checkpoints are recorded is allocated and made readily available. To make the access time as predictable as possible, the checkpoints are recorded into a continuous array in a compact format. To account for jitter, a benchmark consists of a configurable number of measurement runs, executed in sequence.

At the beginning of each measurement run, the coordinating thread triggers an interrupt using a platform-dependent mechanism. The specific mechanisms used on ARM and x86, are described in Section 5.1.1 and Section 5.7.1 respectively. When the interrupt is set up to occur, the triggering thread waits for a fixed time interval. Sometime in this period, the interrupt occurs, and in response, the kernel calls the previously allocated ISR. It records a checkpoint, marking the start of the experiment, and then schedules a softirq, tasklet or work item for execution. If enabled, INTSIGHT records a configurable number of intermediate checkpoints, to track which code is executed during the measurement run. This may include the code path ultimately leading to the execution of the requested bottom half, but also unrelated events, such as intermediate interrupts. The experiment ends, when the requested bottom half is invoked and records the final checkpoint into the preallocated buffer. When the triggering thread finishes its waiting period, the next measurement run is started by triggering another interrupt. Finally, after the specified number of repetitions, the raw results are converted into a platform-independent format suited for exporting, which is documented in the following Section 4.2.

INTSIGHT can record intermediate checkpoints to trace the executed code virtually anywhere inside the kernel. For exploration, INTSIGHT in particular uses the *tracepoint* feature present in the kernel. Tracepoints, usually implemented as a call to the macro `TRACE_EVENT()`, mark performance-relevant parts on the kernel and are also employed by other tools (e.g., `perf`) [26]. By hooking into the macro, INTSIGHT can easily record checkpoints at various places, which allows for an exhaustive tracing of the executed code. However, the more checkpoints are recorded, the greater is the runtime overhead introduced by INTSIGHT. Therefore, this feature was only used for exploration and disabled, whenever measurements aiming for accuracy, were performed.

The triggering thread can either use active or passive waiting after having triggered the interrupt. The mechanism used, determines whether the thread is runnable on occurrence of the IRQ. If passive waiting is employed and if there are no other runnable threads, the CPU will sleep and be woken up again by the IRQ. The workqueue implementation uses kernel threads to execute the requested bottom halves. If the triggering thread employs active waiting, it will compete with the worker threads for execution time, and thus interference with the dispatching of work items. To easily test both scenarios, the kernel module provides a runtime option for the waiting mechanism used.

4.1.2 Time Measurement

INTSIGHT supports various sources for the timestamps used to determine the delay between successive checkpoints. The implementation supports all timekeeping interfaces described in Section 2.4. In addition, hardware-specific interfaces, for example, cycle counters, are implemented to achieve increased accuracy. The timestamp sources accessed when recording a checkpoint, are fixed at compile time to minimize runtime overhead. However, the more clock sources are configured, the longer the recording of a checkpoint takes and thus the higher is the interference introduced into the measured latencies. Recording multiple timestamps also increases the amount of memory required to store them, which may become a bottleneck if large data sets are generated on systems where memory is scarce. Writing measurement to a hard disk has to be avoided during a benchmark, since accessing such devices in general introduces unpredictable latencies.

4.1.3 Checkpoint Recording

```
1 #include <linux/intsight.h>
2
3 __always_inline void intsight_checkpoint(const char *name);
```

Listing 4.1 – Synopsis of the kernel interface offered by INTSIGHT for recording checkpoints. Including the header file makes the inlined function `intsight_checkpoint()` available. When used, the caller passes a string describing the recording location. The implementation of the function is outlined in Algorithm 4.2.

INTSIGHT provides an interface, shown in Listing 4.1, that allows users to record checkpoints at arbitrary locations inside the kernel. The recording code is inserted directly into the evaluated sections, and thus has to be highly efficient in order to not distort the measured latencies. The longer a recording takes, the greater is the execution time wrongfully attributed to the tested code. In addition to running as fast as possible, the injected code must also not put unnecessary pressure on memory and caches, since this may also affect the performance of the original kernel code.

Algorithm 4.2 shows the implementation of the function used to record a single checkpoint. If the preallocated buffer, into which all data is recorded, is not filled up yet, the recording can take place. First, each enabled timestamp type is retrieved and stored. Subsequently, the checkpoint name, identifying the point in the source code, where the checkpoint is located, is stored. String literals are used for this purpose, since they are statically allocated and thus do not require copying of the string contents at runtime [27, p. 62]. Also, they enable the compiler with optimizations that would not be possible for `char`-Pointers into dynamically allocated memory. All stores either access the stack or the preallocated buffer, which makes the performance of the accesses more predictable by improving cache-locality. In contrast, for example, accessing dynamically allocated memory (e.g., calling `kmalloc`) while recording a checkpoint, would result in unpredictable timing behavior.

```
1: if buffer would not overflow then
2:    for all enabled clock sources do
3:       retrieve timestamp
4:       store timestamp
5:    end for
6:    store checkpoint name
7: end if
```

Algorithm 4.2 – Implementation of the function `intsight_checkpoint()`, which records a checkpoint of a given name. Checking whether there is still space available in the preallocated buffer is a matter of comparing and incrementing a single pointer. The enabled clock sources are fixed at compile time, thus no looping is required at runtime.

Instead of keeping the injected code as short as possible, the time it takes to execute it could also be measured and subtracted from the final latencies. However, this approach is deemed unsuitable for INTSIGHT, since the injected code is already very compact. While the recording of a second timestamp would allow INTSPECT to calculate the interference caused by the recording of checkpoint names, and the checking for available space, the total indirect interference, for example, due to cache-pressure and hindering of code optimizations, would increase. Also, storing another timestamp would use up space that can otherwise be used for additional measurement runs. Especially on systems where memory is scarce, this is a benefit of the chosen approach.

4.2 Interaction

```
cd /sys/kernel/debug/intsight
echo > init

# Set the parameters
echo softirq > bottom_handler
echo 1000    > reps

# Execute the benchmark
echo > prepare_trigger
echo > do_trigger
echo > postprocess_trigger

# Optional: Inspect the results
head csv_results/pmccntr
cat  reps # -> 1000

# Save the results and paramerters
cp -vrf . ~/my-insight
```

Listing 4.2 – Example shell script demonstrating the usage of INTSIGHT's `sysfs` interface. After having set the benchmark parameters, the measurement runs are executed when writing into `do_trigger`. After the postprocessing, the results can be inspected directly or copied for further analysis using INTSPECT's tools.

The kernel component INTSIGHT provides a `sysfs` interface accessible from user space. INTSPECT uses this interface to communicate a given benchmark configuration to the kernel, trigger its

execution, and finally retrieve the generated data. An example shell script demonstrating the usage of the interface is shown in Listing 4.2. In addition, Table 4.1 documents the virtual files offered by INTSIGHT. Before executing a benchmark, INTSPECT first has to communicate the parameters to INTSIGHT. This, for example, includes whether softirqs, tasklets, or workqueues should be benchmarked, and how many measurement runs should be performed. Since the files used for setting benchmark parameters can also be read out, one can easily obtain a complete description of a benchmark after the execution, by copying the complete `intsight` directory.

The interface gives INTSPECT fine-grained control over the operations performed by the kernel. The allocation of the checkpoint buffer, the postprocessing, and the actual benchmark execution, are all triggered using separate writes to `prepare_trigger`, `postprocess_trigger`, and `do_trigger` respectively. This separation allows INTSPECT, for example, to collect compact `ftrace` records of a benchmark run, when a detailed analysis of the executed code path is desired [29].

Table 4.1 – `sysfs` interface offered by INTSIGHT in the folder `/sys/kernel/debug/intsight`. Allows user space applications to set benchmark parameters, trigger their execution, and retrieve the results. If not noted otherwise, a file is suited for both read and write access. Write-only files accept any input.

File Name	Type	Description
init	Write-only	Initialize INTSIGHT, creates the other files listed here
bottom_handler	*softirq*, *tasklet*, or *workqueue*	Bottom half mechanism to be benchmarked
reps	Integer	Number of measurement runs to perform
delay_ms	Integer	Delay between measurement runs
delay_type	*udelay* or *usleep_range*	Respectively use active / passive waiting between measurement runs
checkpoint_capacity	Integer	Maximum number of checkpoints recorded per measurement run
prepare_trigger	Write-only	Prepare benchmark using the current parameters
do_trigger	Write-only	Execute the benchmark, blocks the writing thread for at least *reps* × *delay_ms* milliseconds
postprocess_trigger	Write-only	Expose the results, creating the *csv_results* folder
csv_results/name	Read-only CSV	One line per measurement run, each line contains the checkpoint names in the encountered order for this run
*csv_results/**	Read-only CSVs	The recorded timestamps matching the checkpoint names in *csv_results/name*
vmalloc_checkpoint_matrix	Read-only Boolean	Determine whether the checkpoint buffer was small enough to be allocated using `kmalloc()`, or whether `vmalloc()` was require [28]

During the benchmark, the records are stored into a memory buffer optimized for fast write access. Only when the benchmark is complete, the data is converted to a platform-independent text-based format. This may only be a matter of encoding integers as strings, but can also include time unit conversion, for example, when the kernel time is recorded (i.e., `ktime_to_ns()` has to be called). The converted values are exposed as CSV-files with one line per measurement run, created in the `csv_results` directory. The file `csv_results/name`, is always created, and contains the checkpoint names in the order in which they were encountered in the specific benchmark (with a newline whenever a new measurement run begins). The matching timestamps are exposed by separate files in the same directory (e.g., called `pmccntr` or `tsc`) if their recording was configured at compile time. The author notes, that when working with `sysfs`, creating one file per value is usually preferred since this avoids parsing incompatibilities [30]. However, this approach was found to be unsuited for the results, since INTSIGHT can easily create hundreds of thousands of values during a single benchmark, resulting in a unacceptable readout delay. To allow for a large number of measurement runs even on systems where memory is scarce, which, for example, includes the ARM embedded setup used through Chapter 5, INTSIGHT must avoid storing the entire contents of the CSV-files in memory. This achieved using the kernel's `seq_file`-Interface to convert the contents of the checkpoint buffer to CSV format every time the files are accessed [31].

4.3 User Space

The task of the INTSPECT user space framework is to control the kernel component and analyze the gathered information. It is split into the tool running on the test system, described in Section 4.3.1, which controls the kernel component and gathers benchmark results, and the analysis software executing on a separate system, described in Section 4.3.2.

4.3.1 Test System

The part of INTSPECT executing on the benchmarked system, has to be chosen carefully in order to not destroy the well-defined environment set up for the benchmarks. To not introduce unwanted interferences, the executed application is kept small.

In our base environment, a minimal number of user space applications execute on the test system during the benchmark. INTSPECT provides a load generator, which can be used to spawn applications before the benchmark to simulate defined load scenarios. The benchmark parameters are communicated to the kernel and the benchmark is triggered. When the benchmark has finished, the triggering INTSPECT process is unblocked and saves the generated data to disk. In addition to the results, information about the state of the running system is gathered before and after the experiment. This includes, for example, the kernel configuration used and the number of interrupts occurring on the system.

4.3.2 Analysis System

While the software running on the benchmarked system has to be chosen carefully to not interference with the measurements, the software used during analysis, which usually happens on a separate system, does not have to satisfy this property. It can be optimized to allow for maximum flexibility and convenience. Outsourcing as much processing as possible from the test system to the analysis system, makes it easy to guarantee that no running application interferes with the measurements.

The developed analysis framework allows users to easily determine the distribution of interrupt handling latencies. In addition, the recorded kernel tracepoints allow the discovery of system tasks that interfere with the fast invocation of the bottom half, and conditions, that lengthen the time it takes to request it from within the top half. For example, an intermediate interrupt is unrelated to the executing code, but still increases the delay until the bottom half is invoked. Aside from analyzing the generated data, the separate system also stores a complete description of the benchmarking environment, consisting, for example, of binary images, source code, and configurations files. This eases the reproduction of generated results, as well as the validation of conclusions drawn from them.

5 EVALUATION

This chapter presents an exhaustive analysis of the runtime overhead of the Linux kernel's different bottom half mechanisms, measured on an ARM-based embedded platform and an Intel x86-based computer. Causes for the measured latencies are uncovered, and it is analyzed how changes in the environment influence the interrupt handling delay.

Section 5.1 describes the ARM evaluation setup, in which the experiments are executed, and Section 5.2 evaluates the different timekeeping interfaces available. Based hereon, the processor cycle counter is chosen as the metric best suited for the analysis. Section 5.3 presents the runtime overhead of each interrupt deferral mechanism in a baseline scenario, and Section 5.4 analyzes which kernel components are causes for delays in this environment. The remaining three sections evaluate, how changes in the test setup influence the results. Section 5.5 uncovers the influence of the test interrupt frequency on the latencies, and Section 5.6 analyzes, how user space applications pressuring the CPU, influence the delays. Finally, Section 5.7 presents, how the results obtained in the ARM baseline scenario, compare to the results obtained on an Intel x86-based server computer.

5.1 ARM Evaluation Setup

This section describes the test setup used in Section 5.2 through Section 5.6. As the experiments do not expect the specific setup to have major influence on the results, I try to eliminate sources of interference from each used component in the following. This includes the relevant hardware of the SAMA5D3 Xplained board, presented in Section 5.1.1, which hosts the ARM Cortex-A5 processor, described in Section 5.1.2. The base setup uses the recent long-term support (LTS) version 4.9, of the kernel, whose configuration and features are described in Section 5.1.3. Finally, Section 5.1.4 focuses on the role of the Linux distribution employed in the experiments.

5.1.1 Board

The SAMA5D3 Xplained board, shown in Figure 5.1, is designed for developers of power-efficient embedded applications [32]. It features an ARM Cortex-A5 processor, focused on in the following section, together with 2 GB of DDR2 memory [33, p. 14]. The debug unit's serial line, internally implemented using interrupts and direct memory access (DMA) [32, pp. 1377–1401], is used to control the device and retrieve benchmark results from it.

The board features various general-purpose input/output (GPIO) pins easily accessible to the user. This enables us to implement the triggering of interrupts by connecting two GPIO pins using a wire (idea taken from [34]). Using existing kernel libraries, a thread can trigger an interrupt by outputting a signal on one pin, which causes a incoming signal on the other pin.

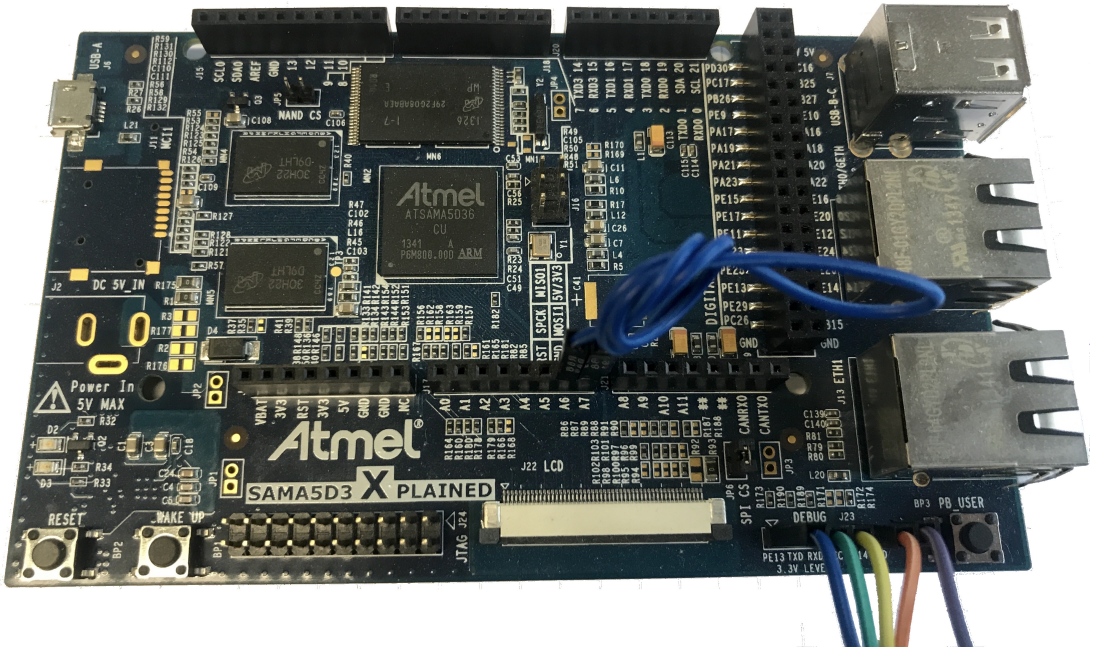

Figure 5.1 – The Atmel SAMA5D3 Xplained board, used to execute the experiments, features the ATSAMA5D36 MPU and 2 GB of DDR2 memory. The GPIO pins A6 and A7 are connected using a wire, and the debug port is connected to a computer used to monitor the benchmarks.

5.1.2 Processor

The ATSAMA5D36 is a single-core ARM Cortex-A5-based microprocessor unit (MPU), that runs at 528 MHz[4]. It has an in-order pipeline with dynamic branch prediction, plus an instruction and a data cache, each 32 kB in size. In order to save energy, the processor can enter *standby mode* when unused, and be woken up again, for example, when an interrupt occurs [32, pp. 40,111].

The Cortex-A5 processor includes a cycle counter as part of its performance monitoring unit [12]. The cycle counter fulfills the requirements for precise measurement as described in Section 3.2.1, which are in particular, that it can be retrieved quickly, using a single instruction which accesses the `PMCCNTR` register, and has a high resolution since it counts individual processor cycles. However, as of version 4.9, no interface for the counter exists in the kernel [11], thus, functions that allow zero-overhead retrieval of the register are implemented as part of INTSIGHT. A problem for using the `PMCCNTR` as time metric emerges, when the processor enters the aforementioned standby mode during a measurement. In standby mode, the cycle counter does not increment, and thus this time span is excluded from the measured delay. INTSIGHT can detect when the processor enters standby mode using its tracepoint feature, and affected measurements could thus be filtered out. However, in the experiments described in this chapter, the processor does not enter standby mode between executing the top and bottom halves of an interrupt handler.

[4]The boards datasheet [32, p. 2] states, that the processor runs with *up to* 536 MHz, however, both our experiments as well as the kernel log indicate that it runs at 528 MHz.

5.1.3 Kernel

Unless explicitly stated otherwise, the benchmarks are executed on a Linux kernel of version 4.9, maintained by Microchip Technology for the AT91SAM SoC family, which includes the used board [11]. Version 4.9 features the Completely Fair Scheduler (CFS) as well as the most current workqueue implementation as of September 2018, named Concurrency Managed Workqueue.

The kernel is built using the standard configuration provided for the board[5], customized to eliminate sources of interference and speed up the compilation time. To not have the CPU frequency change during an experiment, making the cycle counter unusable as a time metric, CPU frequency scaling (`CONFIG_CPU_FREQ`) is disabled. No forced kernel preemption (`CONFIG_PREEMPT_NONE`) is configured, thus, no user space application can preempt INTSIGHT during the system call performed by INTSPECT. Networking (`CONFIG_NET`) is disabled entirely to eliminate it as a source of interference. Communication with the device is instead accomplished using the serial line mentioned in Section 5.1.1. To speed up the compilation time, unused features, including loadable kernel modules and power management, are also disabled. The exact configuration used to build the kernel for a specific benchmark, is included with the results.

5.1.4 Linux Distribution

As this work focuses on the kernel, the distribution running is not relevant for most experiments. However, for example, when the kernel is preemptible or when workqueues are benchmarked, user space applications can interfere. Thus, the Linux distribution used in our setup is described in the following.

The Yocto Project[6] develops a framework for creating custom Linux distributions for embedded systems. The reference distribution, called Poky, is used in our setup with small modifications. The framework encapsulates related modifications or extensions into *layers*. A layer can, for example, include additional software with the distribution or add support for another SoC. This allows users to create complete distributions tailored to their needs. In our setup, two layers are applied to the Poky reference distribution. First, the official layer[7] provided by Microchip Technology, adds support for the SAMA5D3 Xplained board. Second, binaries required by INTSPECT are injected using a custom layer. Thus, to make INTSPECT support a new hardware platform, one must simply adapt the latter layer to their custom distribution.

It is feasible to keep the extent to which the benchmarking results depend on the used distribution as small as possible. This is accomplished by disabling forced kernel preemption. The system call performed by INTSPECT, in which INTSIGHT executes the benchmark, thus runs to completion unless the thread voluntarily goes to sleep. In the case of workqueues, this is required to give the worker threads a chance of getting scheduled. Thus, a runtime option, determining whether active or passive waiting is employed, after having triggered the interrupt, exists in INTSIGHT. In addition, user space applications scheduled during the experiment, are detected by INTSIGHTs tracepoint feature.

[5] https://github.com/linux4sam/meta-atmel/blob/4169efdd21b0941df934935da5aadb9a8301d9ae/recipes-kernel/linux/linux-at91-4.9/sama5/defconfig

[6] https://www.yoctoproject.org/

[7] https://github.com/linux4sam/meta-atmel

5.2 Measurement Accuracy

This section analyzes the resolution as well as the readout delay of each timekeeping interface available to INTSIGHT. Both influence the accuracy of measurements, as concluded by Section 3.2.1. Based on the results, the PMCCNTR is chosen as the timekeeping interface best-suited for the analysis of interrupt handling overhead in the following sections.

In this chapter, each plotted measurement series, also referred to as *benchmark*, consists of 50 000 measurements, referred to as *measurement runs*. Every measurement run is carried out as described in Section 4.1.1, usually consisting of the triggering of a single interrupt, followed by a waiting period in which the top and bottom half execute. If not noted otherwise, 50 measurement runs are performed every second. The thread triggering the interrupts goes to sleep while the top and bottom halves execute, making the CPU idle if no other thread is runnable. The results are displayed as a histogram. To better visualize the minimum and maximum latency encountered, and allow for a better distinction between latencies that occur rarely, and those that occur never, a logarithmic y-axis is used. Since the benchmarked tasks do not involve interrupts (memory swapping is disabled), measurement runs where an interrupt occurs in the critical section are filtered out, and displayed in a separate column labeled IRQ. In some benchmarks, measurement runs exceeded the scale of the x-axis. To not disturb the display of the majority of results, those cases are also shown in a separate column labeled OOB (out-of-bounds).

Figure 5.2 shows the reported time delay between two successive checkpoints, measured using the four different timekeeping interfaces. Since no work is performed between the checkpoints, this delay would ideally be zero. However, both the code recording the two checkpoints, presented in Section 4.1.3, as well as the enabled timekeeping interface, introduce overhead. The plot displays, how the measured overhead varies depending on the timekeeping interface employed. While the overhead for `ktime`, `ktime_mono_fast`, and `nstimeofday`, which all rely on the system clock source, is comparable, the PMCCNTR can be read out much more quickly with the majority of measurements completing within 60 ns. The largest encountered delay is well below 1 µs for the PMCCNTR, while the other interfaces report up to 2 µs elapsing between the checkpoints.

By plotting the individual nanosecond latencies as dots on a one-dimensional axis above each histogram, we can visualize the resolution offered by the respective timekeeping interface. The scatter plots for `ktime`, `ktime_mono_fast`, and `nstimeofday`, only show a single dot above each bin, which is a cause of the lower resolution offered by these interfaces. An interface offering real nanosecond resolution, would likely report almost every value belonging to a bin at least once, as caches and branch prediction introduce small variances into the used execution time. This would lead to a continuous line being displayed above the histogram. The dots plotted for the interfaces, that rely on the system clock source, however, are overlapping each other, as the latency-values put into a single bin are all the same. This leads to the dashed line plotted above the respective histogram. A look at the raw data confirms, that the latencies reported by the system clock source are always multiples of 60 ns–61 ns apart from each other. The PMCCNTR, however, reports latencies that differ by single CPU cycles (one cycle takes 1.89 ns at 528 MHz).

In order to ensure that the time delay deduced from the elapsed clock cycles corresponds to the actual delay, the PMCCNTR was compared to the latency returned by `ktime`. Over a period of 1 s, the metrics were found to not diverge by more than 400 ns, as long as the CPU did not sleep between the measurements (calling `msleep()` or `usleep_range()` thus must be avoided unless other threads are known to be runnable). INTSPECT can easily validate whether the CPU went idle during an experiment, since INTSIGHT places a checkpoint in the kernel code responsible for putting the CPU asleep when no thread is runnable.

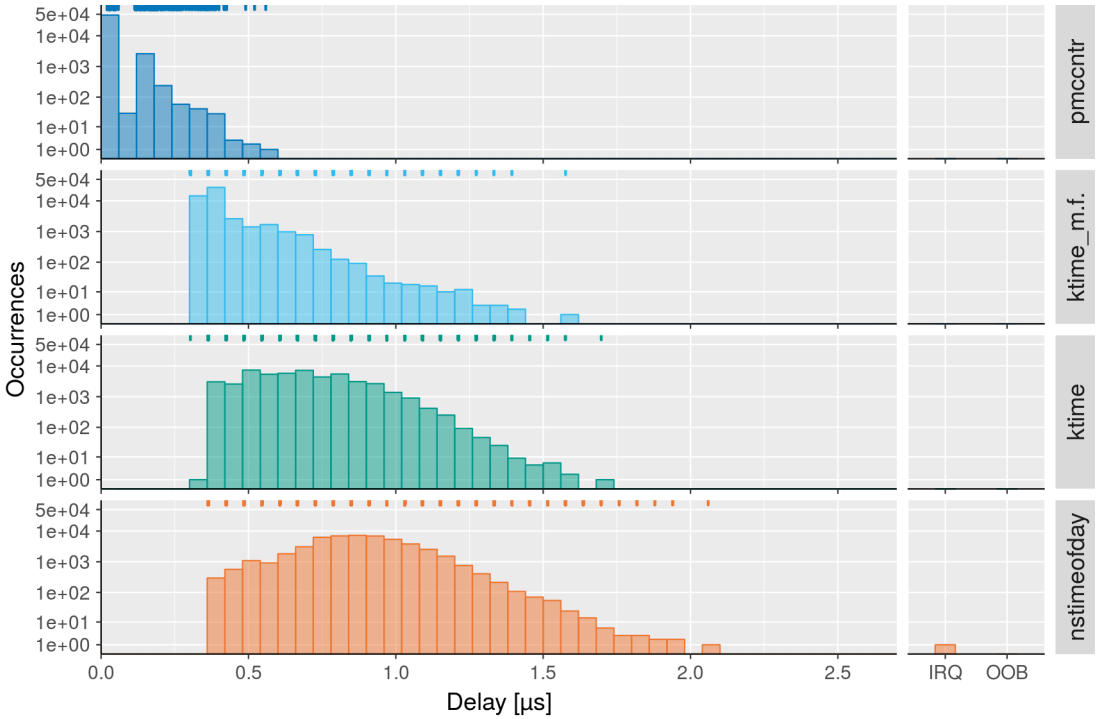

Figure 5.2 – Comparison of the measured time difference between two successive checkpoints, using different timekeeping interfaces available to INTSIGHT: ARM's cycle counter register (pmccntr), `ktime_get_mono_fast_ns()` (ktime_m.f.), `ktime_get()` (ktime), and `getnstimeofday()` (nstimeofday). The histogram (bin size of 60 ns) displays the distribution of measured delays. In addition, individual cases are shown as a one-dimensional scatter plot at the top of each facet, visualizing the resolution of the timekeeping interfaces.

5.3 Interrupt Latencies Under Minimal System Load

In the following sections, the runtime overhead of the different bottom half mechanisms is analyzed using the PMCCNTR. Based on the results from the previous section, a bin size of 1 µs is chosen for the histograms. To visualize the extent to which the latency varies between runs, the 5 % and the 95 % quantiles are shown as red dashed lines in the plots, and denoted as $Q_{0.05}$ and $Q_{0.95}$ in the tables. Therefore, the majority (i.e., 90 %) of results lie within that range. The median, as well as the minimum and maximum latency, are denoted using $Q_{0.50}$, min, and max, respectively. As already noted in the previous section, runs where an interrupt occurred between the benchmarked top and bottom half, are shown in the separate IRQ column in the histograms, as the interrupt is usually not related to the executing code. These latencies, are therefore also excluded when calculating $Q_{0.05/0.50/0.95}$, min, and max. However, when bottom halves have to be executed by a thread (i.e., by `ksoftirqd` or a `kworker`), the maximum latency over all runs is of interest, as the thread may only get scheduled, after another thread has been preempted using an interrupt. Consequently, the maximum latency from *all* measurement runs of a benchmark, including those shown under IRQ in the histogram, is denoted as $\max_{IRQ}$ in the corresponding table.

Figure 5.3 shows the overhead introduced when using softirqs, tasklets, and workqueues for interrupt handling under minimal system load. The matching minimum and maximum latency, as

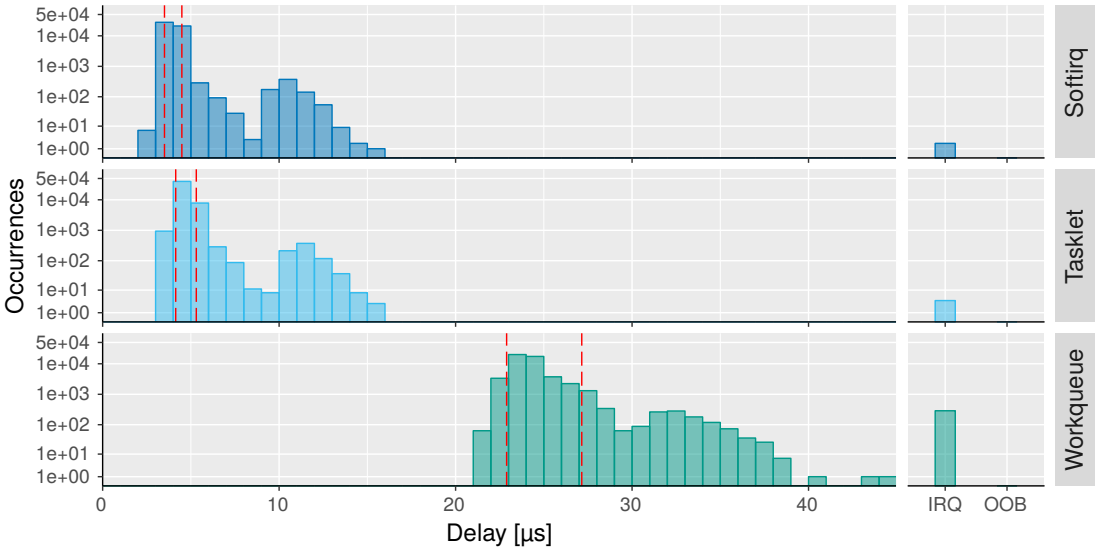

Figure 5.3 – Comparison of the overhead introduced by Linux's three bottom half mechanisms. Softirqs and tasklets are invoked with a comparable delay, with tasklets being only slightly slower. Workqueues in turn, are considerably slower and also have greater variance. In addition, the number of cases where an interrupt occurred in between the top and bottom half, is much higher for workqueues, due to the increased overall latency.

Table 5.1 – Latency for softirqs, tasklets, and workqueues in an environment with minimal interference on the ARM hardware platform. Softirqs and tasklets offer comparable performance, but can still be delayed when an interrupt occurs between the top and the bottom half (max_{IRQ}). The median latency ($Q_{0.50}$) for workqueues is greater by a factor of five.

	min	$Q_{0.05}$	$Q_{0.50}$	$Q_{0.95}$	max	$\max_{IRQ}$
Softirq	2.9 µs	3.5 µs	4.0 µs	4.5 µs	15.6 µs	15.6 µs
Tasklet	3.4 µs	4.1 µs	4.7 µs	5.3 µs	15.9 µs	47.5 µs
Workqueue	21.3 µs	22.9 µs	24.1 µs	27.1 µs	44.2 µs	84.3 µs

well as the quantiles, are shown in Table 5.1. As expected, softirqs offer the best performance, as their invocation is hard-coded into the code path executed before the kernel returns from interrupt context. Tasklets in comparison, are only slightly slower, with the median increasing by less than 1 µs with respect to softirqs (being 4.7 µs instead of 4.0 µs). This small difference can be explained by the fact, that pending tasklets are invoked from within a dedicated softirq. When there are no other tasklets pending except the one used for testing, which is likely the case in the clean room scenario presented in this section, only the code invoking the tasklet from within the softirq introduces additional overhead.

The median for workqueues is much higher when compared to tasklets and softirqs, being 24.1 µs in the presented benchmark run. As indicated by the 5 %- and 95 %-quantiles lying further apart, there is also greater jitter in the results. The number of measurement runs where an interrupt occurred between the top and bottom half, is also increased. I suggest, that this is a result of the increase in latency, which makes it more likely for an interrupt to occur in the observed period. The

increase in the median delay as well as the jitter is related to the usage of threads for implementing workqueues. In order to execute a work item, those worker threads have to be woken up and dispatched. Both are comparably complex operations, which are not required for softirqs and tasklets, since their invocation is performed directly within the code path executed after an interrupt. This explanation is further explored in the following section, which analyzes the causes for the delays presented here, using INTSPECT's tracing capabilities.

5.4 Delay Causes

This section analyzes which kernel components contribute to the runtime overhead of the three bottom half mechanisms in the clean room scenario from the previous section. Using INTSPECT's ability to trace the code executed between the top and bottom half, the scheduler, as well as the kernel's entropy pool, were found to be a source of delay when invoked. Figure 5.4 and Figure 5.5 display the results from the same softirq and workqueue benchmarks as Figure 5.3, but separate the measurements where the scheduler or entropy pool introduced additional latency from the measurements where they did not. A graph analyzing tasklets is omitted, since both the measured latencies as well as the delay causes, only differ insignificantly from softirqs in the baseline scenario.

The kernel regularly uses the exact time and source of an interrupt to enhance its entropy pool, which is used for random number generation (e.g., for `/dev/random`). Because the random numbers generated, have to be cryptographically secure, the raw data fed into the entropy pool has to be handled carefully in order to not be leaked to third parties. Leaking would ultimately allow those to attack the user, by predicting the random numbers generated for him. After having invoked the top half, the kernel checks whether one second has elapsed since the last collection of interrupt entropy [11]. If so, the information about the interrupt is immediately mixed into the entropy pool[8], before any pending bottom half is invoked. For this reason, the collection of randomness for the entropy pool is a major cause of jitter for softirqs, tasklets, and workqueues.

For workqueues, the scheduler was found to be another source of delay. Specifically, whenever it updates runtime statistics for the current thread[9], the invocation of a requested work item was delayed. This update usually happens as part of the call that wakes up the worker thread chosen to execute the work item, but also occurs regularly as part of a timer interrupt, which can also happen between a top and bottom half. The latter explains the large number of measurement runs, where the scheduler introduced additional delay after an intermediate IRQ. Since the timer interrupt can also occur between a top half and a corresponding softirq or tasklet, the scheduler may also introduce delays for these bottom half mechanisms.

While the scheduler as well as the entropy pool explain most of the runs where workqueues were delayed, a part of the slower runs remains unexplained. Based on analysis using the Linux kernel's function tracer (`ftrace`), I suggest the following explanation. The interrupt used for experimenting simply returns to the previous thread, when it finishes. In our case, this is usually the INTSIGHT thread that triggered the interrupt just before, because it is still busy executing trigger-related code or putting itself asleep. The work item requested in the interrupt, however, is only invoked when the corresponding worker thread is scheduled. *When* this happens, is in general unpredictable. It may even be delayed indefinitely, for example, if the triggering INTSIGHT thread uses active waiting (i.e., by calling `udelay()` with forced kernel preemption disabled). In the presented benchmark, however, the worker thread was usually scheduled as soon as the triggering thread was able to finish putting itself asleep. Since the time at which the interrupt preempts the triggering thread (which

[8] Marked by the kernel tracepoint `mix_pool_bytes`.
[9] Marked by the kernel tracepoint `sched_stat_runtime`.

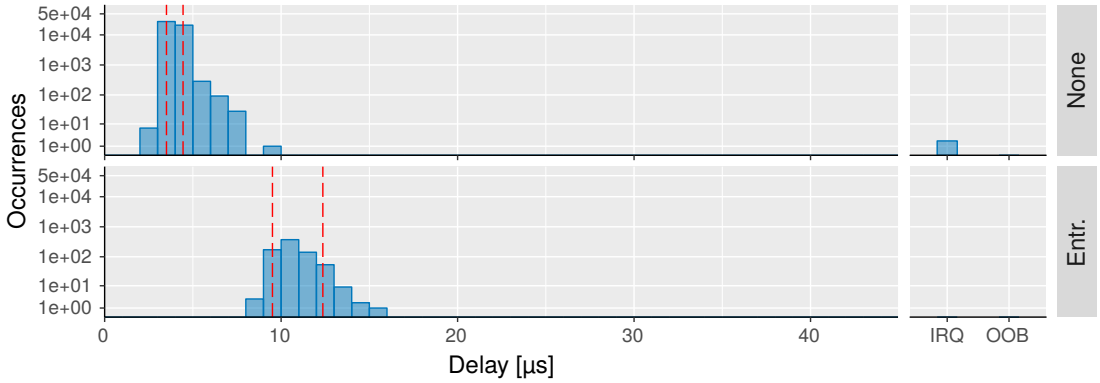

Figure 5.4 – Analysis of the delay causes for softirqs. The kernel regularly uses interrupts as a source of entropy for its random number generator (Entr.). As this happens directly after the top half, the execution of softirqs is delayed in those measurement runs. In the presented benchmark run the scheduler was not observed to be a source of delays for softirqs.

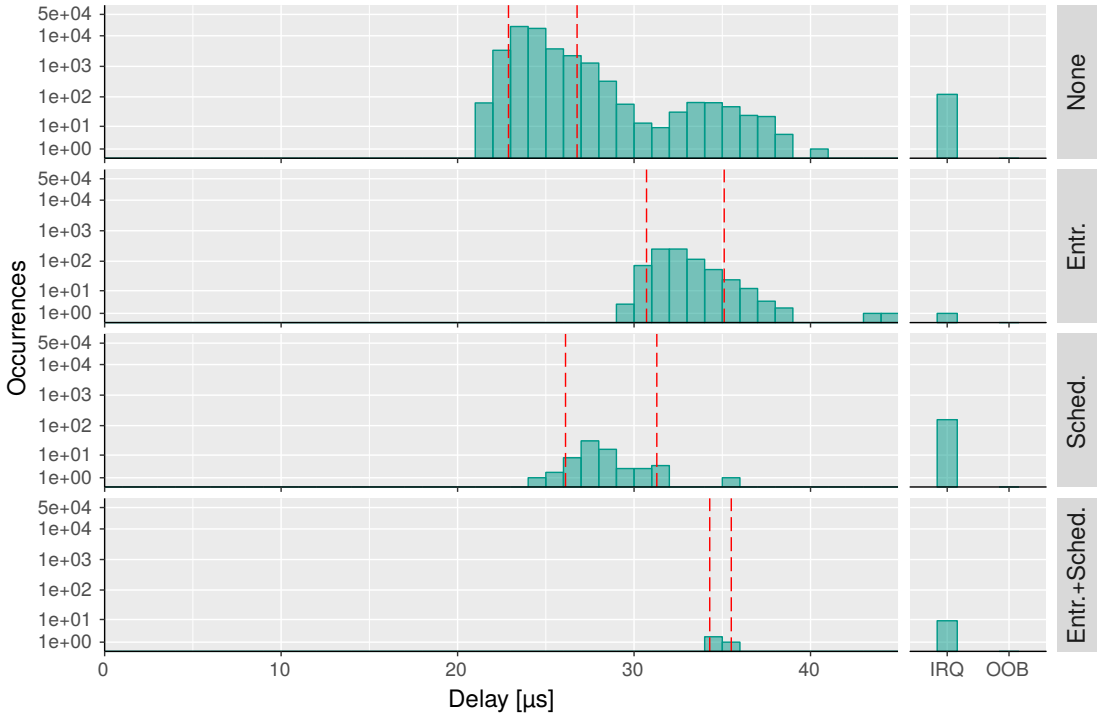

Figure 5.5 – Analysis of the delay causes for workqueues. Both the usage of the interrupt as a source of entropy (Entr.), as well as the updating of runtime statistics by the scheduler (Sched.), may delay the invocation of a work item. When both events (Entr.+Sched.) occur, the median delay is further increased.

determines the time it takes for this thread to invoke the scheduler, after the interrupt handling has finished), is relatively constant, the observed runtime overhead for workqueues is not as large (or small) as it could have been in a different test setup. The delay may be small, if the interrupted thread voluntarily gives up the CPU right after the interrupt, or may be large, if the next timer interrupt (usually happening every 1 ms to 10 ms) has to be awaited. Future work could introduce greater variance in the interrupt timing using dedicated hardware (e.g., one which waits for a varying period of time before redirecting an incoming signal to the CPU).

In conclusion, both the calculation of runtime statistics by the scheduler, as well as the usage of interrupts to enhance the kernel's entropy pool, are major sources of delays in the execution of bottom halves. For softirqs and tasklets, only the entropy pool was observed to infer in the presented ARM setup, while for workqueues, both components introduced delays.

5.5 Influence of Interrupt Frequency

The previous section uncovered, that the usage of interrupt-related information for the kernel's entropy pool, is a source of delays for every bottom half mechanism. On every interrupt, the kernel checks whether one second has elapsed since the last usage of interrupts as an entropy source [11]. Only if this is true, the information about the current interrupt is mixed into the pool.

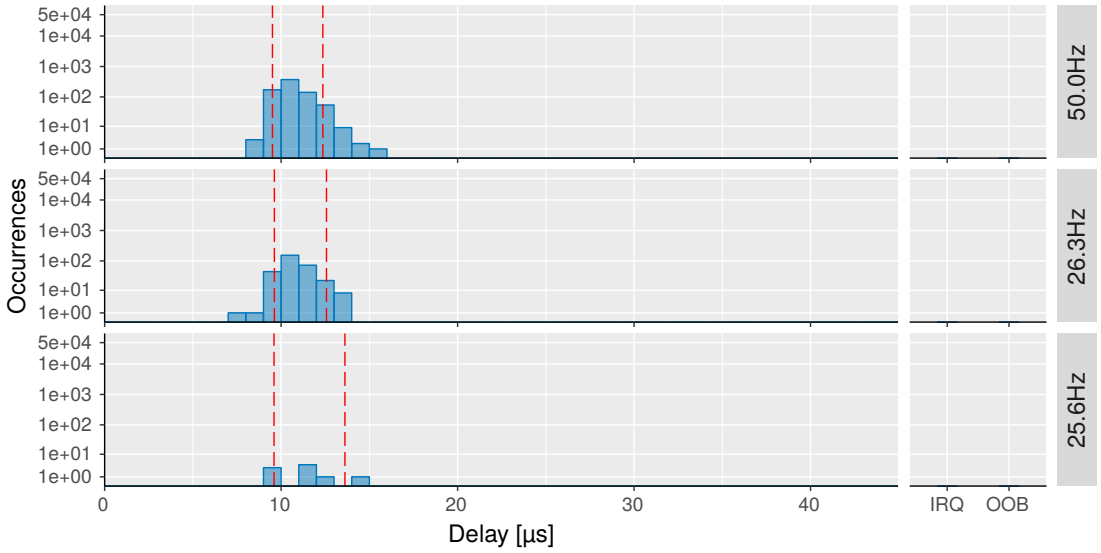

Figure 5.6 – The entropy pool as a delay cause for softirqs, when measurements are performed with a frequency of 50.0 Hz, 26.3 Hz, and 25.6 Hz. The number of measurement runs, where the entropy pool delays the softirq, decreases, as the waiting period between the runs increases.

Figure 5.6 also displays the "Entr."-row from Figure 5.4, but compares it to the data collected in benchmarks, where INTSIGHT only generated test interrupts at a frequency of 25.6 Hz and 26.3 Hz, instead of 50.0 Hz. Increasing the waiting period between the measurements, reduces the number of INTSIGHT interrupts awarded as an entropy source. I suggest, that this is a result of the INTSIGHT interrupt making up a smaller fraction of the total number of interrupts. This makes it less likely for it, to be the first interrupt in the time window where entropy collection is pending.

5.6 Influence of Processor Load

While the previous sections performed experiments in an environment with a minimum number of user space processes active during the benchmarks, this section uses INTSPECT's load generator to spawn applications and create defined load situations. It is analyzed, how this affects the delay between a top half and the different types of bottom halves.

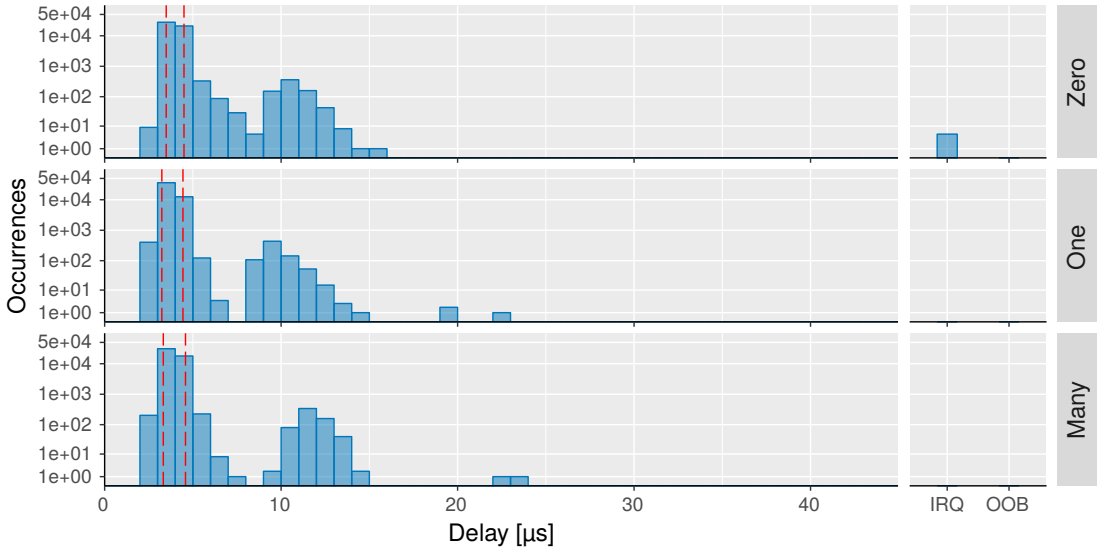

Figure 5.7 – Comparison of softirq delay with zero, one, and many user space processes fully utilizing the CPU during benchmarks. No significant difference is observed, however, when one process is active, the latency is slightly reduced.

Figure 5.7 and Figure 5.8 display, how having zero, one, or many processes active during a benchmark, influences the runtime delay for softirqs and workqueues. A plot for tasklets is omitted, since the results do not differ significantly from the results for softirqs. Table 5.2 and Table 5.3 list the respective minimum and maximum latency, as well as the relevant quantiles for each plot. When no processes are spawned, the CPU usually goes idle between measurement runs. Having one process that aims for full CPU utilization, prevents this, since this thread is scheduled whenever no task of greater priority is available. Spawning many processes (specifically 256) also prevents sleeping, but in addition puts the scheduler under pressure.

Table 5.2 – Softirq latency when zero, one, and many user space processes are active during a benchmark. When processes are active, the median latency is insignificantly smaller. The maximum latency for runs where no IRQ occurred (max), is increased by 6.8 µs and 8.7 µs for one and many processes, respectively.

	min	$Q_{0.05}$	$Q_{0.50}$	$Q_{0.95}$	max	$\max_{IRQ}$
Zero	2.8 µs	3.5 µs	4.0 µs	4.5 µs	15.2 µs	36.2 µs
One	2.5 µs	3.2 µs	3.8 µs	4.4 µs	22.0 µs	22.0 µs
Many	2.6 µs	3.3 µs	3.9 µs	4.6 µs	23.9 µs	23.9 µs

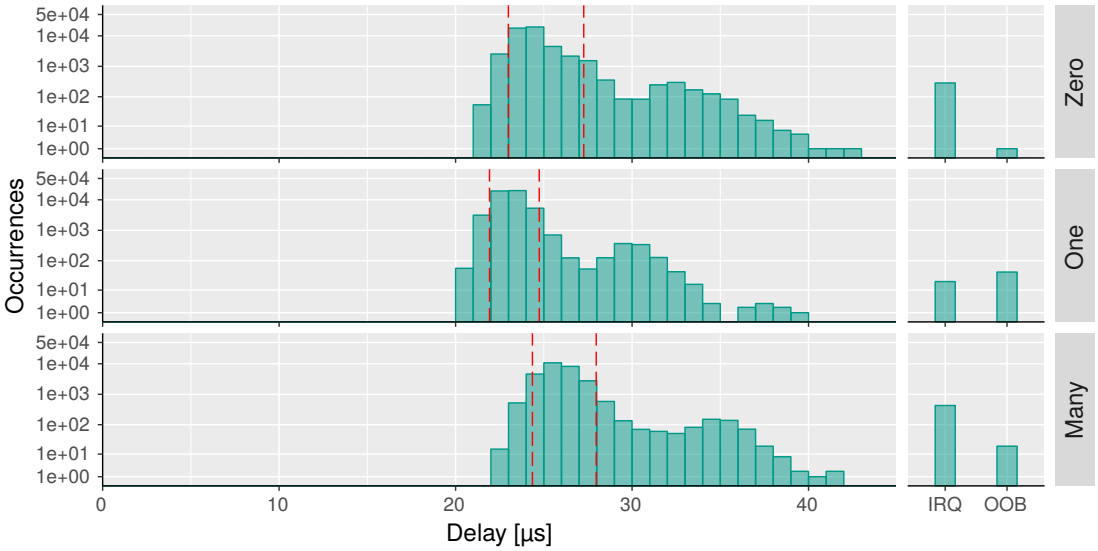

Figure 5.8 – Comparison of workqueue delay with zero, one, and many user space processes fully utilizing the CPU during benchmarks. Having one process active, prevents the CPU from sleeping between measurement runs and reduces the runtime delay, therefore also reducing the chance of being interrupted (IRQ). In turn, having many processes negates the effect. The number of delays exceeding 45 µs (OOB) is increased for both one and many processes.

Table 5.3 – Workqueue latency when zero, one, and many user space processes are active during a benchmark. When processes are active, the maximum latency (max) increases significantly, especially when we also consider the measurements in which an interrupt occurred between the top and bottom half ($\max_{IRQ}$).

	min	$Q_{0.05}$	$Q_{0.50}$	$Q_{0.95}$	max	$\max_{IRQ}$
Zero	21.3 µs	23.0 µs	24.2 µs	27.4 µs	45.9 µs	84.9 µs
One	20.5 µs	21.9 µs	23.1 µs	24.8 µs	196.3 µs	10 073.9 µs
Many	22.6 µs	24.4 µs	25.8 µs	28.5 µs	197.5 µs	19 944.4 µs

Having one process prevent the CPU from sleeping, slightly reduces the median latency ($Q_{0.50}$) for softirqs, tasklets, and workqueues. I suggest, that this is due to caches not being erased between the measurement runs, which happens whenever the CPU goes to sleep. This proposal is supported by the fact, that active waiting between measurement runs inside INTSIGHT, also reduces the median latency for softirqs and tasklets. It is noted, that the processes spawned in this experiment, only pressure the CPU, but not the memory and cache hierarchy.

Spawning many processes for a benchmark has no noticeable influence on the softirq and tasklet latencies, as user space processes can not prevent their invocation before return from the requesting interrupt. For workqueues however, the median, as well as the maximum latency when no intermediate interrupt occurs (max-column in Table 5.3), increases. This is suggested to be due to the scheduler requiring more runtime to manage the added processes. In our benchmarks, the CFS was configured, which is the current default scheduler. In contrast to the older $O(1)$ scheduler, the runtime required to determine which thread executes next, depends on the total number of

threads managed [17]. Future work may analyze, whether using a different scheduler can prevent work items form being delayed, when many processes are active.

Both when one and many processes are active during a benchmark, the number of measurement runs not completing within 45 µs (labeled OOB in Figure 5.8) increases for workqueues. While the latency stays within 200 µs, when runs, that include intermediate interrupts, are filtered out, no limit was found if these runs are also considered. As the recorded maximum latencies (see $\max_{IRQ}$ in Table 5.3) are close to 10 ms and 20 ms, I suggests that the scheduler did not choose the worker thread as the first task to receive a time-slice after the test interrupt, in those runs. As the scheduler interrupt occurs at a frequency of 100 Hz, this would result the latency increasing by multiples of 10 ms.

5.7 Comparison with Intel x86 Hardware Platform

Aside from the detailed measurements made on ARM in an embedded environment with minimal interference, we also repeated our benchmarks on a server computer employing an 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 one used in our previous evaluations. Also, both the system distribution used and the configuration of the kernel had major differences (i.e., network was enabled and the number of running tasks was much higher). The goal is not to provide an extensive analysis for x86, but instead, to get an idea, to which extent the observations made for ARM are specific to our embedded and minimum interference scenario, and to which extent they carry over to a high performance real-world scenario.

5.7.1 Porting INTSPECT

The kernel module employed by INTSPECT, that is, INTSIGHT, was designed to be as modular as possible. Therefore, porting it from the ARM hardware platform, for which is was initially designed, to x86, only required minor changes. First, another mechanisms to trigger interrupts had to be implemented. The used server computer does not offer easy access to GPIO pins like the Atmel SAMA5D3 Xplained board does. However, in comparison to the ARM instruction set architecture (ISA) [35], the x86 ISA [36] includes an instruction to generate an arbitrary interrupt, which is employed by INTSIGHT. Second, as the `PMCCNTR` is an ARM-specific register, an equivalent way to accurately measure execution time is required for x86. The x86 Time Stamp Counter (TSC) is chosen for this purpose. As INTSIGHT is designed to support multiple timekeeping interfaces, adding the TSC required minimal changes to the code base.

5.7.2 Interrupt Latencies in a Real-World Scenario

Figure 5.9 displays the delay between a top half and the execution of a corresponding softirq, tasklet, and work item, requested from within the interrupt on the x86 hardware platform.[10] The associated quantiles, as well as the minimum and maximum latency for each measurement series, is listed in Table 5.4. The figure showing the respective results for ARM is therefore Figure 5.3. The results differ notable, but are also comparable with regard to some aspects. As expected, the delays are much lower on the faster processor, although the speedup is not 6.25 , as one would deduce from the increase in clock cycles per second. In numbers, the median is only 3.3 times lower on x86.

[10] In the shown benchmark, the delay between measurement runs was set to 59 ms.

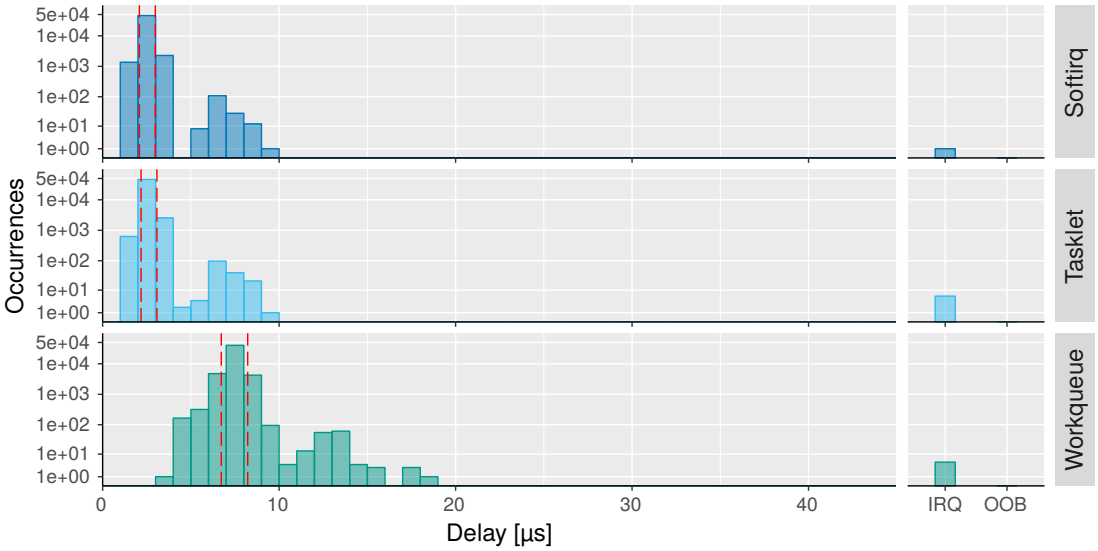

Figure 5.9 – Comparison of the runtime overhead introduced by softirqs, tasklets, and workqueues on x86. Figure 5.3 shows the same benchmarks executed on ARM. When compared, the variance is increased for all bottom half mechanisms on x86, if we account for the higher clock speed of the Intel processor.

Table 5.4 – Latency for softirqs, tasklets, and workqueues in a real-world scenario on the Intel x86 hardware platform. The slowest softirqs and tasklets (max) took longer than the fastest work items (min).

	min	$Q_{0.05}$	$Q_{0.50}$	$Q_{0.95}$	max	$\max_{IRQ}$
Softirq	1.08 µs	2.08 µs	2.19 µs	2.99 µs	9.88 µs	9.88 µs
Tasklet	1.10 µs	2.18 µs	2.36 µs	3.08 µs	9.07 µs	9.07 µs
Workqueue	3.22 µs	6.72 µs	7.28 µs	8.22 µs	18.68 µs	29.45 µs

This is likely to be a cause of the differing instruction set architectures. As on ARM, the speed of softirqs and tasklets is very comparable (again, tasklets are slightly slower, although, because of the increased processor speed, the difference is almost unnoticeable). Workqueues, in comparison, are clearly slower on average. However, instead of being completely off from the other mechanisms, there are some cases where work items got executed with less delay than the slowest softirqs or tasklets.

5.7.3 Delay Causes

In addition to the plain time measurements, as on ARM, the tracepoint feature of INTSPECT was used on x86 to analyze which events caused delays. Figure 5.10 and Figure 5.11 display the softirq and workqueue results from Figure 5.9 in greater detail. They separate measurement runs, where the scheduler or entropy pool introduced delays, from undelayed runs. Figure 5.4 and Figure 5.5, together with Figure 5.3, therefore show the respective results for ARM. Here, it should be noted that both the collection of randomness for the entropy pool, as well as the accounting of task runtime

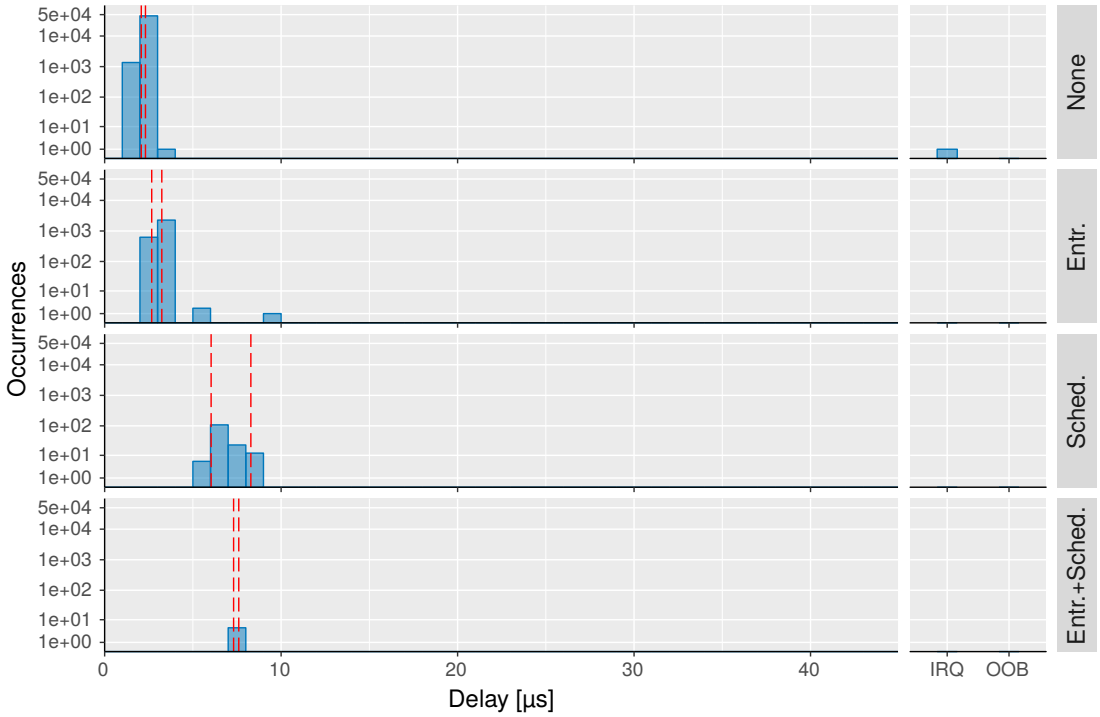

Figure 5.10 – Analysis of the delay causes for softirqs on x86. Figure 5.4 shows the respective results for ARM. Both the kernel's entropy pool (Entr.) as well as the scheduler (Sched.) introduced delays, while on ARM only the entropy pool did.

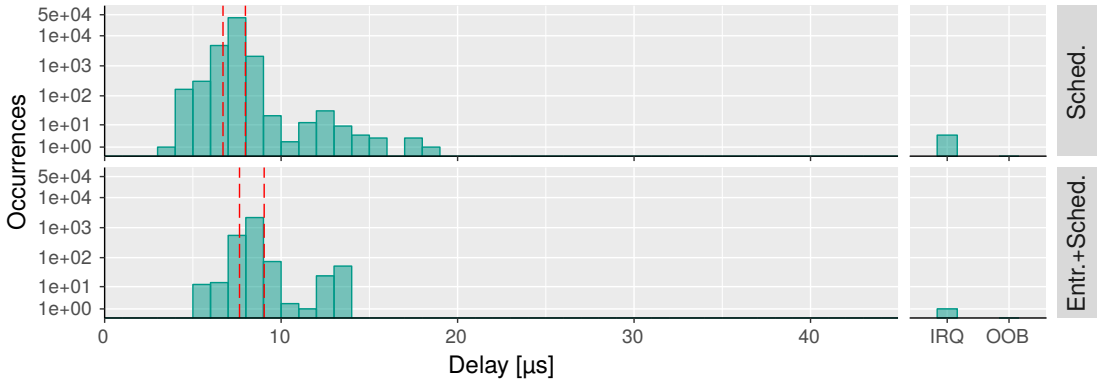

Figure 5.11 – Analysis of the delay causes for workqueues on x86. Figure 5.5 shows the respective results for ARM. The scheduler (Sched.) introduced delays on every observed measurement run, while on ARM, it only did in some cases.

by the scheduler, also occur on x86. For workqueues, however, the accounting of task runtime is no longer an exception, but instead occurs on every sample in our test series. Also, in contrast to ARM, the accounting of task runtime is also a source of delays for softirqs and tasklets, although being by far not as dominant here, as for workqueues on x86.

5.8 Review

This section reviews the results presented in this chapter, highlighting findings that may be of special interest to Linux users and kernel developers. The invocation latencies for softirqs and tasklets, are predictably low on both ARM and x86. In contrast, the delay for workqueues is unpredictable and may be subject to millisecond-scale jitter. Still, the experiments show, that workqueue performance can be comparable to softirq and tasklet performance, if the scheduler is invoked early after the requesting interrupt. Using INTSPECT's ability to trace the kernel code executed between the top and bottom half, we identified the kernel's entropy pool, which is required for random number generation, as a source of jitter in the latencies every mechanisms. On ARM only for workqueues, but on x86 also for softirqs and tasklets, the scheduler was also observed to introduce additional latency when it updates runtime statistics of threads. Depending on the hardware platform and interrupt frequency, the extent to which the scheduler and the entropy delay the bottom halves, varies. On ARM, it was evaluated whether running user space processes influence the performance of the bottom half mechanisms. The latency of softirqs and tasklets was mostly unaffected, with the influence being limited to cache-related effects. Workqueues, however, were found to be delayed more frequently. It is noted, that the absolute delay and variance (measured in microseconds), was smaller on x86, although the used environment was more noisy. Using a faster processor can thus be a way to reduce interrupt handling latency.

In conclusion, our evaluation proves, that INTSPECT is valuable to both users and developers of the kernel. Users can evaluate the responsiveness of a specific setup, and identify user space workloads that negatively influence it. This can help in optimizing the performance of a system, but can also uncover problems, when switching to another hardware platform. Kernel developers can use the results from this evaluation, as well as the information collected by other users of INTSPECT in the future, to identify problematic parts of the kernel code and improve the system at a whole.

6 CONCLUSION

In order to be responsive to external events, operating systems have to handle interrupts with predictable low latency. The complexity of modern hardware and software makes static analysis of system performance infeasible. Instead, measurement based approaches that evaluate the system at runtime are required. For this purpose, we developed INTSPECT, which measures the runtime overhead of Linux's interrupt subsystem at execution-time. The tool relies on a maintainable kernel component called INTSIGHT, which records the time elapsing between the top and bottom halves of interrupt handlers with microsecond accuracy. To uncover the causes for delays, INTSIGHT in addition allows for low-overhead tracing of the executed code path. The tool was integrated into a recent LTS version of the kernel and deployed for both the ARM and Intel x86 hardware platform.

Finally, we evaluated the runtime overhead of the Linux kernel's three bottom half mechanisms, which are softirqs, tasklets, and workqueues, on the two hardware platforms using INTSPECT. The latency between request and dispatching was found to be predictably low for softirqs and tasklets. Only the scheduler as well as the kernel's entropy pool introduced constant delays when invoked. However, system load caused by user space applications did not delay these two bottom half mechanisms. My results reveal, that tasklets, being designed for widespread use, only add insignificant overhead over softirqs. If system load is small, the latency for workqueues is only a factor of microseconds slower and subject to the same jitter causes as the latencies for softirqs and tasklets. However, when user space processes are active, the delay can increase significantly, as there is no mechanisms that prioritizes workqueues over application threads. The largest encountered delay for workqueues under system load was 20 ms, this is already one tenth of the maximum delay tolerable when reacting to user input [1]. Although there exist configuration options in the kernel, that improve its responsiveness, a user interaction, for example, on smartphones, regularly requires remote servers to respond. Here, the kernel is usually not configured for low-latency, which can easily result in a noticeable delay for the user, if multiple workqueue items are involved. In conclusion, workqueue performance can be relatively predictable if the environment does not interfere, but is unpredictable if the system is under load. Softirqs and tasklets offer predictable performance and are unaffected by system load from user space.

Future work may evaluate additional Linux variants and load scenarios. In this context, real time extensions to the kernel may be of special interest. Additional work is also required, to empirically determine the minimum and maximum latency for workqueues and research the interdependence between the bottom half latency and the scheduler. It may also be of interest, which typical user space tasks generate a large number of bottom halves, and therefore put the interrupt subsystem under load. INTSIGHT can be ported to other hardware architectures beside ARM and x86. It may also be adopted to measure the performance of kernel components, which require similar accuracy as the interrupt subsystem.

LIST OF ACRONYMS

CFS Completely Fair Scheduler

CISC Complex Instruction Set Computer

CPU Central Processing Unit

DMA Direct Memory Access

GPIO General-Purpose Input/Output

INTSPECT INTerrupt Subsystem Performance Evaluation and Comparison Tool

I/O Input/Output

IoT Internet of Things

IRQ Interrupt Request

ISA Instruction Set Architecture

ISR Interrupt Service Routine

LTS Long-Term Support

MPU Microprocessor Unit

NMI Non-Maskable Interrupt

RISC Reduced Instruction Set Computer

SoC System on a Chip

TSC Time Stamp Counter

WLAN Wireless Local Area Network

LIST OF FIGURES

LIST OF TABLES

REFERENCES

[1] Hiroshi Nittono. “Event-Related Brain Potentials Corroborate Subjectively Optimal Delay in Computer Response to a User’s Action.” In: *Proceedings of the 7th International Conference on Engineering Psychology and Cognitive Ergonomics (EPCE’07)*. Harris, Don (Ed.), 2007, pp. 575–581.

[2] Hannu Leppinen. “Current Use of Linux in Spacecraft Flight Software.” In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017), pp. 4–13.

[3] Gaurav Banga, Peter Druschel, and Jeffrey 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] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard 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] John D’Ambrosia. “100 Gigabit Ethernet and Beyond.” In: *IEEE Communications Magazine* 48.3 (2010), pp. 6–13.

[6] Bijan Davari, Robert H. Dennard, and Ghavam G. Shahidi. “CMOS Scaling for High Performance and Low Power—The Next Ten Years.” In: *Proceedings of the IEEE* 83.4 (1995), pp. 595–606.

[7] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion 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.

[8] Peter Wägemann, Christian Dietrich, Tobias Distler, Peter Ulbrich, and Wolfgang 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, 24:1–24:25.

[9] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute 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.

[10] Valentin Rothberg. *Interrupt Handling in Linux*. Tech. rep. CS-2015-07. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2015, pp. 1–22.

[11] *Linux Kernel Source Code*. Version 4.9, for Atmel AT91SAM SoC. URL: `https://github.com/linux4sam/linux-at91/tree/44fd1b88e4b068a31480050de0e86b47873e3a19` (visited on 10/19/2018).

[12] *Cortex-A5 Technical Reference Manual*. Atmel. Jan. 2016. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0433c/DDI0433C_cortex_a5_trm.pdf (visited on 10/20/2018).

[13] Matthew 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.

[14] Paul Regnier, George Lima, and Luciano Barreto. "Evaluation of Interrupt Handling Timeliness in Real-time Linux Operating Systems." In: *ACM SIGOPS Operating Systems Review* 42.6 (2008), pp. 52–63.

[15] John Calandrino, Hennadiy Leontyev, Aaron Block, Umamaheswari Devi, and James 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.

[16] Felipe Cerqueira and Björn 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.

[17] Nikita Ishkov. "A Complete Guide to Linux Process Scheduling." MA thesis. University of Tampere, School of Information Sciences, 2015.

[18] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan 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.

[19] Elder Vicente and Rivalino Matias 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.

[20] Stephen Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John 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.

[21] Benedict Herzog, Luis Gerhorst, Bernhard Heinloth, Stefan Reif, Timo Hönig, and Wolfgang Schröder-Preikschat. "INTSPECT: Interrupt Latencies in the Linux Kernel." In: *Proceedings of the 8th Brazilian Symposium on Computing Systems Engineering (SBESC'18)*. 2018.

[22] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd Edition. O'Reilly Media, 2009. ISBN: 0-59600-590-3.

[23] Daniel Lohmann and Volkmar Sieh. *Betriebssysteme*. Lecture at the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Winter Term 2017/18. URL: https://www4.cs.fau.de/Lehre/WS17/V_BS/Vorlesung/#folien (visited on 10/09/2018).

[24] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000. ISBN: 3-86063-630-8.

[25] OSEK/VDX Group. *Operating System Specification 2.2.3*. 2005. URL: http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf (visited on 11/30/2018).

[26] Steven Rostedt. *Using the TRACE_EVENT() Macro (Part 1)*. Mar. 2010. URL: https://lwn.net/Articles/379903/ (visited on 11/26/2018).

[27] *C99 Standard*. ISO/IEC 9899:1999. 1999. URL: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf (visited on 10/25/2018).

[28] Jonathan Corbet. *On the Proper Use of vmalloc()*. Nov. 2003. URL: https://lwn.net/Articles/57800/ (visited on 11/27/2018).

[29] Steven Rostedt. *Debugging the Kernel Using Ftrace - Part 1*. Dec. 2009. URL: https://lwn.net/Articles/365835/ (visited on 11/27/2018).

[30] Neil Brown. *A Critical Look at Sysfs Attribute Values*. Mar. 2010. URL: https://lwn.net/Articles/378884/ (visited on 11/26/2018).

[31] Jonathan Corbet. *Driver Porting: The seq_file Interface*. Feb. 2003. URL: https://lwn.net/Articles/22355/ (visited on 11/27/2018).

[32] *SAMA5D3 Series Datasheet*. Atmel. Feb. 2016. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet.pdf (visited on 06/07/2018).

[33] *SAMA5D3 Xplained User Guide*. Atmel. Nov. 2015. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11269-32-bit-Cortex-A5-Microcontroller-SAMA5D3-Xplained_User-Guide.pdf (visited on 10/20/2018).

[34] GitHub User gkaindl. *Linux GPIO IRQ Latency Test*. URL: https://github.com/gkaindl/linux-gpio-irq-latency-test/tree/07f6c2264d1426975ccc2177bc0c75c1458f5994 (visited on 10/20/2018).

[35] *ARM Architecture Reference Manual*. ARMv7-A and ARMv7-R Edition. ARM. May 2014. URL: https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf (visited on 11/27/2018).

[36] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel. Nov. 2018. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (visited on 11/27/2018).