Karlsruhe Institute of Technology

Applying Modern Processor Features to L4 Microkernels

Bachelor's Thesis
submitted by

cand. inform. Martin Ludwig Gurres

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Dipl.-Inform. Thorsten Gröninger

17. Juni 2025 – 17. Oktober 2025

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, October 17, 2025

Abstract

Optimizing time performance of interprocess communication (IPC) facilities on microkernels (μkernels), kernels with minimal functionality, is essential to ensuring competitiveness with monolithic kernels like Linux. With the emergence of recent work like *SkyBridge*, modern processor features have become key research subjects to develop new IPC libraries with that surpass native implementations and therefore improve μkernels. seL4 is a modern representative of μkernels, which we use as a design and implementation platform for user-interrupt (UINTR) support, Intel's recently-introduced extension to send and receive (inter-processor) interrupts directly from user-space. In addition to UINTR we also implement support for the new user-wait extension, e.g. timed pause `TPAUSE`, and design an IPC library to make use of both of these new features on seL4. We find that our new IPC library—uIntercom (uIcom)—provides $1.1 - 5.5\times$ better time performance than either existing seL4 IPC facilities in the cross-core case, while potentially indicating better power efficiency in some metrics.

Contents

Chapter 1

Introduction

Microkernels (µkernels), kernels with minimal functionality, built to increase isolation and limit error propagation, have been a topic of research for nearly 40 years now [1]. Since then, they have been redesigned and improved in every way [2], used as operating system (OS) research subjects [3], and even formally verified [4]. Major contributions to the field were also developed here in Karlsruhe [5]. Recently introduced processor features, such as memory protection keys (MPKs), have just started being investigated for applicability in these minimal systems, sometimes with impressive results [6, 7].

User-interrupts (UINTRs) are a new technology that allows interrupts to be forwarded directly to user-space. As a kernel-bypass mechanism, they might be a potential candidate for improving µkernel interprocess communication (IPC) and scheduling performance, which have been major bottlenecks of µkernels ever since their inception [8]. Another recently introduced technology is the user-wait extension, which allows user-space processes to enter power-saving states while waiting for events [9], which was only possible in kernel mode before. Given the precedent for modern processor features improving IPC performance in addition to two yet-to-be-evaluated new features on Intel's x86/64 platform, we ask whether these can be 1. Integrated into a modern µkernel representative and 2. Used to create a new IPC library that performs better than the native systems from both a time and energy perspective in the cross-core case. In the following chapters we answer these questions.

First, we introduce these features and necessary background in detail in Chapter 2 and give an overview of related work in Chapter 3. After this we present our design for both the integration and the IPC library on our chosen µkernel in Chapter 4, which is then followed by our description of both the implementation process and its details (Chapter 5). We then present our benchmark design and evaluate the collected data in Chapter 6, which we use to finally draw our conclusion and present an outlook for future work in Chapter 7.

Chapter 2

Background

This chapter is a summary of important concepts and necessary background information for later chapters. We first introduce the concept of µkernels in §2.1 to provide an insight into some core concepts as well as their history. Afterwards, we take a look at basic terms in IPC and provide some examples in §2.2. As UINTRs relies on interrupts as an underlying mechanism, §2.3 briefly introduces regular interrupts, which will be expanded upon in §2.4, in which we discuss the new UINTR feature, how it works, and how the user-wait extension could provide synergizing effects. Finally, we introduce important systems of the seL4 µkernel in §2.5.

2.1 Microkernels

In this section we introduce µkernels. For this, we first introduce the concept of a kernel itself (§2.1.1), which is then used to define the differences between µkernels and monolithic kernels in §2.1.2. To conclude, we give a brief history of the L4 family of µkernels in §2.1.3.

2.1.1 What is a Kernel?

An OS is responsible for abstracting system resources and providing a generally platform-independent, isolated interface for user applications [10]. To this end, most OS's have a core part, called a kernel, that directly controls these hardware resources with privileged instructions. These instructions can only be executed with certain execution privileges, which the kernel—in most cases—reserves for itself in order to keep untrusted user software from interfering with other programs or itself. This mode of increased execution privilege is incidentally called "kernel mode".

In order to perform low-level interactions with hardware or parts of the OS, user software must do so through system calls, which are predefined entry points into the kernel and run in kernel mode (unless the kernel itself is running on a virtual system, in which case it usually has a different privilege level, which is lower than the actual kernel mode). Once the system call has finished executing, control is returned to the user software with the original privileges [11].

2.1.2 Micro- & Monolithic Kernels

Traditional kernels, such as Linux or Windows, have many different modules with various functions such as device drivers, extensive memory management, file system functions, a or multiple schedulers, etc. integrated right into the kernel. This leads to a large interconnected code base, a monolith, running in kernel mode. This is what is traditionally considered a monolithic kernel [8]. Monolithic kernels have large trusted codebases (TCBs), meaning there is a large amount of code that is trusted to be safe and secure. A potential attacker has a large attack surface to exploit, since compromising a single kernel module can compromise the entire kernel. In a similar vein, an erroneous kernel module can bring the entire system down by, for example, writing into a critical section of memory.

In contrast, a µkernel is often the minimal set of system calls and subsystems required to get a system to run user-mode software. Modules that are part of the kernel in monolithic kernels are delegated to processes in user-space (called "system servers"), which can be called by clients via IPC, which is one of the few systems managed by modern µkernels [12]. Moving kernel modules to user-space system servers reduces the attack surface of the actual kernel by deprivileging large chunks of potentially vulnerable code and leads to better fault isolation, since an erroneous server will only compromise itself, while the rest of the system is unaffected. This isolation, however, comes at a cost. Since in monolithic kernels inter-module communication consists of simple function calls and user-module communication requires a single system call, these processes are comparatively speedy when seen in contrast with µkernels, where a simple file system access can take multiple IPC round-trips through various system servers, tightly coupling client performance with that of the IPC design [7].

2.1.3 Historical and Modern L4 Microkernels

The original L4 µkernel itself was born out of frustration with the state of first-generation µkernels, which were promised to be fast, secure, and lightweight, but in practice were slow, boiled-down versions of existing monolithic kernels, with the slowness of IPC being a major limiting factor for their performance [8]. L4 started as a from-scratch redesign of the IPC subsystem of Liedtke's L3 µkernel,

with a focus on new ideas that would lead to speed improvements. By reducing the amount of invoked system calls by merging send & receive, novel optimizations for message passing, like passing small messages via registers, and a new design for thread control blocks (TCBs) led to a 20x speed advantage compared to the Mach µkernel at that time [13]. These were later expanded upon and spun off into the new L4 µkernel, with a truly minimal amount of kernel systems, like a new form of address space management via system calls for sharing, granting and revoking pages and interrupt forwarding via IPC messages [2].

Fiasco was the first L4 µkernel to be implemented in a higher language and was designed for and used in real-time systems, but continued the belief that µkernels were inherently platform specific and had to be implemented in assembly in order to have competitive IPC speeds [8] by doubling the latency compared to the original L4 [14]. However, this was disproven by Liedtke and his students in Karlsruhe with *L4Ka::Hazelnut*, which reimplemented L4 mostly in C++ and still retained comparable IPC speeds [15] along with *L4Ka::Pistachio*, which was developed in cooperation with UNSW/NICTA and introduced a split of L4's application programming interface (API) and application binary interface (ABI). *L4Ka::Pistachio* was therefore able to be the first easily ported multi-architecture version of L4 [5], with ports mostly only needing to modify 10% of the kernel code [16].

µkernels were a popular target for formal verification of OS's [17], an effort aimed at increasing security and reliability, which culminated in seL4, a from-scratch reimplementation of L4 with security and verification in mind [18]. seL4 was the first general purpose OS to be formally verified [4]. While the initial attempts assumed correctness of the compiler, hardware, assembly and boot code, more recent work targets not just the kernel itself, but the entire core seL4 platform [19] and is still the subject of current research, such as adding time protection to kernels to avoid timing attacks [3].

2.2 Interprocess Communication

IPC is the general term for communication methods between different correspondents on either the same machine or sometimes even remote machines. As such, it encompasses many different types, the most important of which we introduce in §§2.2.2 to 2.2.4. However, before dealing with these communication pathways, we first introduce categories for IPC implementations in §2.2.1, which we can later use to categorize both existing and our own IPC mechanisms.

2.2.1 IPC Categories

We categorize IPC into three different categories, based on required behavior for successful message delivery:

1. Synchronous, meaning both the sender and receiver need to be waiting on the IPC object at the same time

2. Asynchronous, meaning the sender and receiver do not need to be waiting on the IPC object at the same time, but instead only need to call the IPC object after another

3. Asynchronous-Preemptive, meaning only the sender needs to call the IPC object and the receiver is *preempted* to receive and/or process the message

Calls to synchronous, asynchronous, and asynchronous-preemptive IPC objects can be either blocking or non-blocking. However, for synchronous IPC, at least one of the IPC participants needs to be using a blocking call for successful delivery.

Examples for these categories in conventional OS's are: 1. `SendMessage` of the Win32 library, which waits until the message has been processed [20], 2. `Pipes` on Linux, where sent data is buffered by the kernel until retrieved by a receiver [21] and 3. operating system signal (OSS) on Linux, where pending signals are processed upon reentering user-space [22].

2.2.2 Message Passing

Our first IPC type is message passing, which is a mechanism to communicate and synchronize actions between correspondents and, in general, provides two operations: 1. `Send` to send a message and 2. `Recv` to receive a message [23, sec. 3.6]. The communication link used by the mechanism varies from implementation to implementation, but can range from network packets to shared memory. Message passing mechanisms are usually either synchronous or asynchronous, with an asynchronous mechanism also needing to specify a *buffering* policy—either bounded or unbounded buffers—to determine how a "large amount" of messages is dealt with, sometimes by discarding messages if the buffer is full [23, sec. 3.6.3].

2.2.3 Signals

Similar to message passing, signaling mechanism also use the `Send` and `Recv` operations and serve as event notifications, usually combined with a flag or data word to determine the type. The signaling mechanism usually only stores one

pending signal per signal type, as seen with Linux signals [22], or seL4's Notification, which we cover in §2.5.3. Signals are usually either asynchronous or asynchronous-preemptive and sometimes used as the underlying mechanism to implement message passing, where they are combined with a shared buffer and a pending signal signifies pending data in the buffer.

2.2.4 Remote Procedure Calls

Remote procedure calls (RPCs) are an abstraction on top of bidirectional message-passing IPC between a client and a server that types messages from the client to the server as procedure arguments and messages from the server back to the client as return values [23, sec. 3.8.2]. This allows the client to call procedures on a remote server, which allows it to

- Outsource computation and/or
- React to events from outside sources (other clients, for example) by having a shared state without holding shared memory

An example for a system using RPCs is a remote file-system, with calls relating to normal file access such as `read`, `write`, `open`, `delete` instead being remote calls [23, sec. 3.8.2]. Return values would either be file data or status codes, depending on the operation.

"Remote" in this case simply means "not in this address space", meaning that communication between threads with separate address spaces on the same machine that is structured like a procedure call is an example of a RPC. This definition is common for client–system server communication on μkernels [13], however, if system servers are *passive* instead of *active*, meaning they do not have their own executing thread but simply consist of an address space, this form of communication is instead referred to as protected procedure calls (PPCs) [24].

2.3 Interrupts

In this section we introduce the concept of an interrupt (§2.3.1), briefly touch on the hardware mechanism for delivering interrupts (§2.3.2), continue with how interrupts are tied to exceptions on Intel's x86/64 platform (§2.3.3), and how user-space programs usually interact with interrupts (§2.3.4).

2.3.1 Mechanism

Interrupts on x86/64 are, at their basic level, signals sent to the processor to notify it that *something* has happened, combined with a number, which is called an

interrupt vector (IV), to determine how to process the interrupt [25]. The first 32 IVs are reserved for exceptions, while the remaining 224 are called user-defined interrupts (UDIs), and have no architecture-defined causes. Every IV is also assigned a priority, which determines if an interrupt service routine (ISR) may be suspended and interrupted by the arrival of an interrupt with an IV of a higher priority.

Interrupts can be classified either as external (hardware) interrupts or software interrupts [25, sec. 7.3]. Hardware interrupts are caused by external sources to notify the system of certain events that need to be handled. Since interrupts could only be received by privileged software until the introduction of UINTR, the notified system was usually the OS and contained drivers, or the interrupt was forwarded to user-space software with OS-specific mechanisms, some of which we introduce in §2.3.4. Common sources for hardware interrupts are finished-work notifications from input/output (I/O) devices, like a disk drive, and periodic timer interrupts configured by the OS [26]. Software interrupts, on the other hand, are classified into the following categories:

- Exceptions, some details of which we discuss in §2.3.3.
- Interprocessor interrupts (IPIs), interrupts sent from one processor to another.
- Self-interrupts, caused by executing instructions such as `INTR n`.

On a task-level an interrupt is handled by 1. suspending the current task, 2. executing the ISR, 3. restoring the suspended task. This means an interrupt is transparent to the executed software. However, care must be taken when designing ISRs as, if the interrupts arrive faster than they can be handled, the system may end up in a live-lock, constantly servicing interrupts while not completing any other work [26].

2.3.2 Advanced Programmable Interrupt Controller

Interrupts are managed in hardware by the local advanced programmable interrupt controller (APIC), which is a per-core piece of hardware that receives interrupts from processor pins, internal sources, or the IOAPIC [25, ch. 12]. Local APICs also have a unique ID, which can be used to address a core when sending IPIs. The IOAPIC is an external piece of hardware that receives external interrupts from I/O devices and system sources and then forwards these interrupts to the local APICs. APICs, as reflected in their name, are highly configurable and support features such as *posted-interrupt processing*, which allows physical interrupts to be rerouted to virtual interrupts, which can be used to direct interrupts directly into virtual machines [27, sec. 31.6].

2.3.3 Exceptions

Exceptions can be classified into three categories [25]:

- Faults, a usually correctable exception that occurs during the execution of an instruction. The offending instruction is restarted after fault handling.
- Traps, an exception that occurs after the successful completion of an instruction. The next instruction is started after trap handling.
- Aborts, severe exceptions. Handlers need to shut down the offending application or even system.

When an exception occurs, a software interrupt with the associated IV is generated. Depending on the exception, additional data may be pushed to the stack to be consulted by the handler. A common fault exception is a page fault that occurs when a process accesses an address on a page currently not in memory [25].

2.3.4 Interrupt Handling

On the Linux monolithic kernel, interrupts are usually handled in kernel-space, but for memory-controllable devices, user-space drivers can also make use of userspace I/O (UIO) [28] to handle and acknowledge interrupts. UIO can be used to move kernel drivers to user-space, but some devices may still require a small kernel module to control device functions. Interrupts with UIO are controlled by blocking `read()` or `select()` calls on the device-specific files to get notified of pending interrupts, and `write()`, which is usually used to enable or disable interrupts

Alternatively, some drivers, such as the Intel's high precision event timer driver, forward interrupt events as OSS [29] to then be received in user-space. However, interrupt processing itself is still done in the kernel driver.

In contrast to this, µkernels simply forward interrupts to user-space as IPC messages and forego further processing in the kernel [26]. Device drivers either wait for the next interrupt IPC message, or register an asynchronous routine, which is called upon receiving an interrupt. The user-space tasks need to perform interrupt processing and then reset the hardware themselves, by either calling appropriate system calls or performing writes on memory-mapped devices.

2.4 Modern Processor Features

We introduce the background and current state of support for Intel's UINTR feature on their x86/64 platform in §2.4.1, after which we go into detail on how a

thread would receive (§2.4.2) and send (§2.4.3) interrupts with this new feature enabled. We end our focus on UINTR by specifically highlighting some of its limitations in §2.4.5. Finally, we introduce the user-wait extension in §2.4.6.

2.4.1 UINTR Feature Background

UINTR is a relatively new feature of Intel's x86/64 platform to allow forwarding of regular interrupts, which would normally trap into an interrupt handler in kernel mode, to an interrupt handler in user-space. While the theoretical potential of such a feature for faster networking and high-speed devices was already being discussed in the early 2000s [30], its first potential for implementation was the planned Risc-V "N"extension [31], which was later withdrawn due to a lack of support for the current design [32]. Despite further support from some of the embedded systems community [33], the feature has yet to (re-)appear on other architectures. Intel is therefore the first, and, to this day only, manufacturer to support the UINTR feature, which was first introduced on Sapphire Rapids server processors and later on Sierra Forest, Grand Ridge, Arrow Lake and Lunar Lake processors [34].

In addition to allowing regular interrupts to be forwarded to user-space handlers, Intel's UINTR implementation additionally provides user interprocessor interrupts (UIPIs), which allow user processes to send software interrupts with an additional parameter to each other [9]. Since its original introduction, there has been one minor revision that was introduced with processors after Sapphire Rapids. No OS has built-in support for the UINTR feature, with the official proposed Linux patch-set from Intel being fully abandoned in April 2024 [27] after the Linux kernel mailing list seemingly lost interest after 2022 [35].

2.4.2 Receiving User-Interrupt Notifications

In essence, UINTR allows user-space processes to receive interrupts directly in user-space instead of using OS-specific mechanisms like UIO [25]. By allowing this, UINTR has the potential to reduce the inherent latency of first handling the interrupt in kernel-space and afterwards forwarding them to the user process [35].

Once enabled by setting a bit in the `CR4` register, potential recipients of user-interrupt notifications (UINs), which is what the delivery of a normal interrupt to user-space is called [25, sec. 8.1], need to set three model-specific registers (MSRs). These are 1. `IA32_UINTR_HANDLER`, 2. `IA32_UINTR_MISC`, and 3. `IA32_UINTR_PD`, as well as the optional `IA32_UINTR_STACKADJUST` MSR, each of which set a specific aspect used during user-interrupt delivery (UID).

IA32_UINTR_MISC

The IA32_UINTR_MISC MSR sets multiple values. For one, it contains a bit for storing the user-interrupt flag (UIF) flag which is a flag that en- or disables UID. It is of note that the flag is only actually stored in this bit in the MSR's XSAVE-region after the UINTR state is been XSAVES-ed, not in the actual MSR [25, sec. 8.3.2]. The UIF is controlled by three new user-level instructions [25, sec. 8.6]:

1. STUI, which sets the UIF.
2. CLUI, which clears the UIF.
3. TESTUI, which returns the current value of the UIF.

In addition to this, it also contains the user-interrupts notification vector (UINV), which is used to determine the IV that triggers UIN identification, which, if successful, eventually leads to UID. Finally, it also contains the UITTSZ, which we will explain in §2.4.3.

IA32_UINTR_PD and the UPID

This MSR sets the address of the user posted-interrupt descriptor (UPID), which is the structure used by the processor to track a thread's current UINTR state in the PIR field, with the remaining fields only being used by sending agents.

The UPID consists of the ON (outstanding notification) and SN (suppress notification) fields, which determine if a user-interrupt is a) pending and/or b) suppressed. They are used by sending agents to determine if they should send an IPI. ON is set automatically by SENDUIPI, while SN is free to be set by software. The UPID's memory layout can be observed in Figure 2.1.

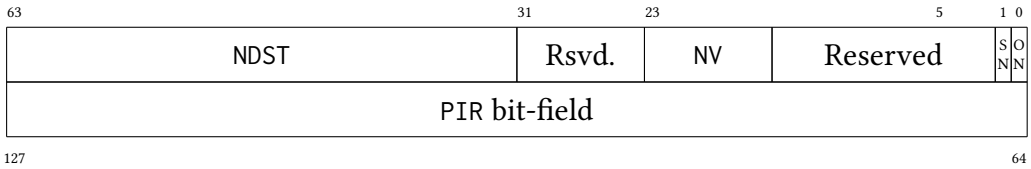


Figure 2.1: UPID memory layout

In addition to these two bits, there is also the NV (notification vector) field, which sets the IV used by the IPI when SENDUIPI is executed while targeting this UPID, as well as the NDST (notification destination), which is the APIC ID of the destination core. Ideally, the NV is equal to the UINV set by the core targeted

by ND, otherwise the interrupt is received by the kernel instead of the targeted user-space thread. The remaining field is the PIR bit-field, which has a bit for every possible user-interrupt vector (UIV) v with $v \in [0; 63]$. Bit v is set if UIV v is requesting service [25]. Upon UIN processing, the PIR is OR-ed into the user-interrupt request register (UIRR), which lives in the IA32_UINTR_RR MSR, the most significant bit of which is the first UIV processed during UID.

IA32_UINTR_HANDLER

This sets the user-interrupt handler (UIHandler), which is the linear address of the routine the rip is set to upon successful UID. The only requirement for this address is that it is canonical[1] and usually points to an ENDBR64 instructions [25]. The UIHandler is given two arguments upon execution: 1. The UINTR frame, consisting of the RIP, RSP, and RFLAGS of the interrupted thread; and 2. The UIV, which multiplexes a single UINV into multiple UIVs, thereby allowing multiple sources to signal the same thread with the same IV and still be distinguished by the receiver. UIVs are necessary because a thread can only have a single IV set as its UINV. Upon completion, the UIHandler should call UIRET, which is a new instruction that restores the state stored in the UINTR frame [25, sec. 8.6] and sets the UIF [25, sec. 8.3.1]. Intel introduced an extension to UINTR called "Flexible Updates of UIF by UIRET", which instead makes UIRET load RFLAGS[1] into UIF, allowing threads to manage UIF from the UIHandler [25, sec. 8.7]. This enables receivers to mitigate potential live-locks from over-eager senders, which was impossible before this extension, as we again mention in §2.4.5.

IA32_UINTR_STACKADJUST

This MSR sets the user-interrupt stack adjustment (UIStackadjust), a value which determines how the stack is adjusted when handling a user-interrupt. This is needed to prevent clobbering of the stack's *red zone*, a region behind the stack pointer considered "reserved" and not allowed to be modified by interrupt handlers in some ABIs [37]. UIStackadjust[0] is used to determine whether the stack address is calculated by subtracting UIStackadjust from the current stack pointer, or simply set to the value of UIStackadjust. Since the resulting stack pointer is then forcibly 16-byte aligned, UIStackadjust[0] is automatically discarded when processing a user-interrupt [25, sec. 8.4.2].

The step-by-step process of successful UID is available in Listing 2.1. It is of note that a user-interrupt recognition (UIR) is only triggered if UIRR $\neq 0$. Therefore,

[1] A canonical address needs to have its unimplemented bits set to either all zeros or all ones. On the current x86/64 platform, these are all bits from index 63–48 [36, sec. 3.3.7.1].

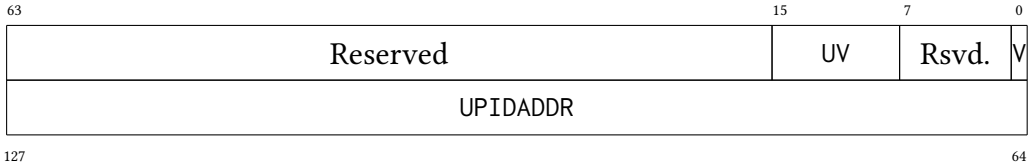


Figure 2.2: UITT memory layout

successful UID can only occur if the PIR field has any set bits before a UIN is processed, which can prove to be a challenge when trying to receive interrupts from external devices [38].

2.4.3 Sending User Interprocessor Interrupts

In addition to allowing the reception of interrupts in user-space, UINTR also introduced the SENDUIPI instruction, which allows threads running in user-space to send IPIs to other user-space threads if correctly configured. Like with the setup of UINs, sending UIPIs also requires setting two MSRs: 1. IA32_UINTR_TT and 2. IA32_UINTR_MISC.

IA32_UINTR_TT and the UITT

Similar to the IA32_UINTR_PD MSR, the IA32_UINTR_TT MSR sets the address of the sender-specific data-structure, the user-interrupt target table (UITT). The UITT, as the name implies, is a table consisting of one or more user-interrupt target table entries (UITTes), each of which describe a combination of a receiver, by containing the address of a UPID in UPIDADDR, with a UIV, which is set in the UV field. Additionally, there is also the V (valid) field, which determines whether an entry is valid. A UITTe's memory layout can be examined in Figure 2.2. Of note is that the length of UV is 8 bits, which sets its possible value range to $[0; 255]$, however, bits 15:14 must be set to zero, lowering the permitted range back to $[0; 63]$, therefore not conflicting with the UPID's PIR bit-field as seen in §2.4.2. In addition to the UITT's address, IA32_UINTR_TT also contains a bit to activate the SENDUIPI instruction. When SENDUIPI n is executed, the nth entry in the UITT is indexed into and checked for a set V bit. If set, the UPID from UPIDADDR is then retrieved and the UVth bit in the UPID's PIR bit-field is set. If neither the UPID's ON or SN bits are set, the executing core then sends an IPI with IV NV to the APIC with ID NDST [9, chpt. 4, SENDUIPI]. An IPI sent via this mechanism is called a UIPI.

```
[Previous Events and Conditions]
- All (receiver) MSRs are set to valid values
- UPID.PIR[UIV] = 1, UPID.PIR[!=UIV] = 0
- UIF = 1
- An interrupt of vector IV = UINV
        was sent to the core with the UPID installed

[Start User-Interrupt Notification Identification]
- Local APIC is acknowledged
- Local APIC interrupt is dismissed
[Start User-Interrupt Notification Processing]
- UPID.ON = 0
- UIRR |= UPID.PIR
- UPID.PIR = 0
[User-Interrupt is Recognized]
[Start User-Interrupt Delivery]
- Processor is woken up from
        user-entered power saving states
- IF UISTACKADJUST[0] == 1:
    - RSP = UISTACKADJUST
- ELSE:
    - RSP = RSP - UISTACKADJUST
- RSP = RSP & ~0xF
- User-interrupt frame (Old RSP, RIP, RFLAGS)
        is pushed to stack
- UIV is pushed to stack
- UIRR[UIV] = 0
- UIF = 0
- RIP = UIHandler
- [End]
```

Listing 2.1: UID, with some minor details omitted

IA32_UINTR_MISC (continued)

As previously mentioned, the IA32_UINTR_MISC MSR also contains the UITTSZ field. UITTSZ + 1 determines the size of the UITT if SENDUIPI is activated in IA32_UINTR_TT. It is used when executing the SENDUIPI n instruction, where, if $n >$ UITTSZ, execution is aborted and a general protection fault (GP fault) is raised.

2.4.4 UINTR-XState

The extended state (XState) is a set of state components that originally was limited to just the x87 FPU state [36, sec. 10.5]. It is thread-specific and needs to be saved and restored during context-switches, for which the instructions XSAVE and XRSTOR were implemented [36, chpt. 13]. XSAVE and XRSTOR are not privileged and save the XState to memory given by an argument [9, sec. 6.1, XSAVE]. Eventually, XSAVES and XRSTORS were also introduced, which are privileged instructions that save and restore additional XState components that should not be accessed or modified from user-space [9, sec. 6.1, XSAVES]. XSAVES-managed state components are both set and enumerated differently than the regular XSAVE components.

UINTR also has an XSAVES-managed state component that can be saved and restored. It encompasses every MSR mentioned above, as well as the UIF flag. Notably, executing XSAVES on the UINTR state component modifies the IA32_UINTR_MISC MSR and clears UINV [36, sec. 13.5.11]. This means that XSAVES is destructive for the UINTR state and executing two consecutive XSAVES will overwrite the saved state with the cleared state, deleting UINV. Implementations therefore need to either keep track of UINVs or ensure that every XSAVES is *always* followed by XRSTORS before the next call.

A potential reason for this truly baffling behavior is the issue of erroneous UID, which might occur of the UINTR MSRs are not cleared when another thread is switched in. The theoretically still valid MSR values lead to UIN processing if the memory address of IA32_UINTR_PD is somehow still valid and ON or SN or any reserved bits are not coincidentally set when a sender, unaware of the receiving threads scheduling, executes SENDUIPI. UIN processing then triggers UID, which sets RIP to the previously-set UIHandler address, which then tries to execute potentially random memory, causing errors. Clearing UINV prevents this, as the IPI is instead delivered to the kernel. However, this only solves the issue for kernels that use XSAVES, which is not guaranteed (see §5.2.3).

2.4.5 Limitations of UINTR

UINTR cannot be fully controlled by user-space. In order to receive UINs, a thread needs to set at least three MSRs, while a sender only needs to set two. Both the sender and receiver need to have the receiver's UPID mapped to their respective address spaces. Furthermore, the sender also needs to have their UITT mapped into their own address space. However, since all memory accesses during the execution of `SENDUIPI` and UIN processing are performed with supervisor privileges [9, 25], these mappings do not not need to be user-accessible. In fact, we believe user access to these data structures is dangerous, the reasons for which we explain in the remainder of this section.

We conclude that some level of kernel-control over the UINTR feature is necessary, for the following reasons:

1. Setting up UIN completely hides any events on the selected IV from the kernel. The kernel should not be circumvented for UINTR setup, as a user-process could then simply "steal" critical IVs, such as the regular timer interrupt, and therefore cripple the OS's abilities, which is in this case the ability to preempt and reschedule tasks.

2. The UPID contains a notification vector field, which determines the IV used by processes when executing the `SENDUIPI` instruction. Since this field is decoupled from the MSR that determines which interrupts are forwarded to user-space, free control over this field could be used by nefarious processes to send any IV they want, potentially messing with OS-systems or introducing live-locks.

3. Freely controlling the UITT can be used to create false entries pointing to user-owned memory. Since every UITTe contains a UPID address to consult when executing `SENDUIPI`, free control over UITTes would lead to the same situation as seen with free control over the UPID.

In addition to these concerns, UINTR should only be used between trusted correspondents if the "Flexible Updates of UIF by `UIRET`" extension is unavailable, due to the concern for potential live-locks caused by malicious senders of UIPIs. Despite exceptions triggering interrupts, UINTR does not support handling of exceptions in user-space, although there have been discussions about potential use-cases of such a feature [39, 40].

2.4.6 User-Wait Extension

The user-wait extension was introduced around the same time as UINTR, with it first appearing on Intel Xeon processors together with UINTR on Sapphire

Rapids [36]. It expands the already existing wait instructions `MWAIT` and `MONITOR`—which allow the processor to enter an implementation-dependent optimized state upon executing `MWAIT` until an event or store occurs on the address range previously specified by `MONITOR` [9, sec. 4.3, MWAIT]—with user-mode equivalents `UMWAIT` and `UMONITOR`. `MWAIT` is allowed to switch to any C-State, while `UMWAIT` is limited to just `C0.1` and `C0.2`. We will call these optimized states "sleep states" after this point.

In contrast to the privileged counterpart, `UMWAIT` has an immediate argument, which sets the time stamp counter (TSC) timestamp after which the processor exits the sleep state even if no event occurs. This means that programs wanting to use `UMWAIT` first need to read the TSC, then add their desired deadline offset, then execute `UMWAIT`. In addition to this argument, there is an additional deadline, usually set by the OS, called the OS deadline. It determines the maximum time in `TSC` units that `UMWAIT` is allowed to wait for. Note that the OS deadline is an offset instead of a timestamp. The OS deadline is set with a new MSR, called `IA32_UMWAIT_CONTROL`, which also controls the maximum sleep state `UMWAIT` is allowed to enter [25, sec. 4.3, UMWAIT]. The extension also adds the `TPAUSE` instruction, which acts like `UMWAIT` but without a monitored address range. `TPAUSE` therefore only waits for the deadlines, while `UMWAIT` can also exit the sleep states earlier due to memory events [9, sec. 4.3, TPAUSE]. The user-mode sleep states are also exited upon receiving an interrupt, with Intel's manuals specifically mentioning that user-interrupts also cause this behavior [25, sec. 8.4.2].

Due to issues with how `MONITOR` and `UMONITOR` reclaim addresses, processors that repeatedly execute these instructions may suffer from performance loss or an inability to enter sleep states on some affected architectures [41].

2.5 Introduction to seL4

This section introduces some technical details of the non-mixed-criticality system (MCS) seL4 µkernel on x86/64. As part of this, we first introduce the limited number of system calls in §2.5.1, describe what seL4's capability-based access-control entails and how it is implemented in §2.5.2. After this, we focus on existing IPC paths for seL4 in §2.5.3 and finally explain existing methods for interrupt handling in §2.5.4.

2.5.1 System Calls

seL4, being an L4 µkernel, implements only three basic system calls [42, sec. 2.2]:

1. `Yield`, which returns control to the kernel and invokes the scheduler. This is the only system call that does not require a capability to invoke.

2. `Send`, which sends data via a capability and performs capability-specific actions.

3. `Recv`, which receives data via a capability.

While these calls are enough to provide all required functionality, seL4 further implements six more system calls [42, sec. 2.2], which are mostly simple variants of the previously mentioned calls. For our purposes, we only focus the following three:

- `Call`, which combines `Send` and `Recv` into a single system call.
- `NBSend`, which is `Send`, but does not ensure delivery and returns immediately if the message could not be delivered.
- `NBRecv`, which polls for a new message and returns immediately if none are available. Otherwise it acts like `Recv`.

`Call` specifically implements some additional functionality, which will be further elaborated in §2.5.3 after capabilities have been introduced in the following sections. The `Send` and `Recv` system calls and variants use thread-specific message registers to pass data and are invoked with user-space stubs. The signature for the `Send`-stub is `void seL4_Send(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)`, where `dest` is the invoked capability and `msgInfo` is a struct that holds a `label` and the `message_length`, among other fields [42, sec. 4.1]. The amount of message registers used is determined by the `message_length` field.

2.5.2 Capabilities

seL4 uses capability-based access-control to hardware features. Capabilities are kernel-controlled objects that can be invoked, created, shared, and revoked [42, sec. 2.1]. Some capabilities have a guard or badge value, which is a simple word of varying length associated with the capability. In addition to this, rights fields are also held by some capabilities, which allow or disallow certain methods to be invoked. After introducing a capability of a certain name, we will refer to a [NAME] capability simply as [NAME].

Addressing Capabilities

Since capabilities are kernel-controlled objects, user-space tasks cannot interact with them directly. Therefore, tasks need a way to address them. These addresses are called capablitiy pointers (CPtrs) and are derived from a guarded page-table constructed by nested CNode capabilities. A CNode holds a list of capabilities,

which can again contain another CNode, which holds another list of capabilities. Every thread has a root CNode, which determines the set of all capabilities a thread can invoke, which is called the capability space (CSpace). It is used as the starting point when resolving CPtrs when no other root CNode is given for the invocation. A CNode capability c has a length l_c, a guard word w_c and a guard length g_c, which are used during the translation process.

The translation process also uses a *depth* d, which determines the bit position at which to start translating, starting from the least significant bit. The default value for d is the *machine word size* s, meaning the translation process starts with the sth least significant bit, which is the most significant bit. To resolve a CPtr for a given root CNode, the first g_{root} bits of the CPtr are checked against w_{root}, after which the next l_{root} bits are used to determine the index into the CNode's list. If the reached capability is another CNode and $d > g_{root} + l_{root}$, the process is repeated with the next CNode's $g_{next}, l_{next}, w_{next}$ until, for the set C of traversed CNodes, either

$$t = d - \sum_{i \in C} g_i + l_i = 0$$

and a capability is found and returned, a *guard mismatch* occurs, or a non-CNode capability is reached while $t \neq 0$, which is called a *depth mismatch* [4, secs. 3.3-3.4]. A visual example of the translation process can be seen in Figure 2.3.

Creating Capabilities

In order to create a new capability, a thread must invoke a capability-creating method. The initial task is given several capabilities for that purpose. The most common of these is the `Retype` method of the Untyped capability, which controls a portion of untyped physical memory. `Retype` can be used to create most capabilities, including memory-managing capabilities to frames or other memory-management structures.

As of seL4 13.0.0, capabilities on x86/64 have 128 bits of storage, some of which are reserved for shared fields, that can be used for rights managed and state-keeping. If a capability requires further memory, for example because it is managing a physical structure (e.g. paging structures), it is designated as a *physical* capability with an object of a certain size and consumes additional memory upon creation, which it needs to track by storing a pointer in the aforementioned storage bits. Physical capabilities need to be created via Untyped's `Retype` and consume part of the Untyped's memory.

Other capabilities, like the root-owned IRQControl capability, are used as rights-managers and can only be used to create new sub-capabilities to specific entities in the managed space. The sub-capabilities cannot have any memory

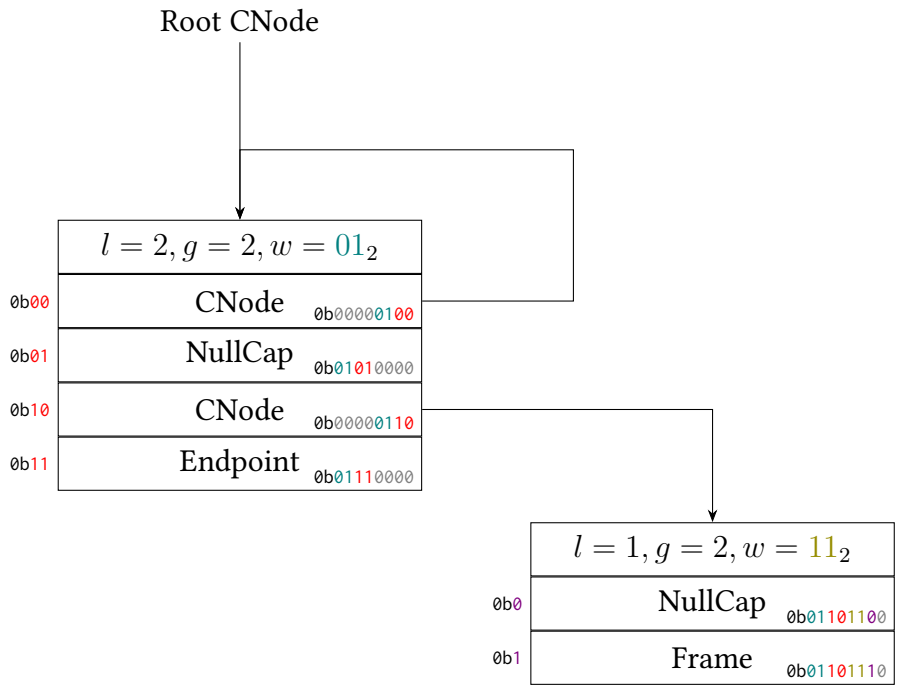


Figure 2.3: An example capability guarded page-table with word size $s = 8$. Capabilities have their simplest CPtr in their box on the right and the index for their containing list on the left. Values in gray are ignored. It is assumed that the proper *depth* is used for CNode addressing. Since CPtr translation follows loops, given the example that the CNode 0b00000100 has the same guard word and guard size as the root CNode, Endpoint can also be addressed with 0b01000111.

associated with them, since the managing capabilities do not have associated memory to hand out and cannot be created via Untyped's `Retype`.

Invoking Capability Methods

To invoke a capability, a thread has to set its message registers, which are housed in a special thread-local memory region, to contain the capability's CPtr, method ID, and further arguments. After which it invokes the `Call` system call and control is handed over to the kernel. The kernel then retrieves the actual capability object and performs the method-specific actions, checking if the thread passed a correct capability with the appropriate rights along with other sanity checks. Return values are then passed back via the thread's message registers and can then be decoded in user-space.

Deriving Capabilities

Many capabilities can be *derived*, meaning a new capability is created from the original capability, either with different rights, a different badge or guard, which is called *minting* or simply *copied*. If the capability is a physical capability, derived capabilities refer to the same object. This is how shared memory can be implemented in seL4 [42, sec. 7.3], since a Frame capability can only be `Mapped` once. The Frame is then derived and either placed directly in the recipient's CSpace, or transferred via capability transfer, which we explain in §2.5.3, and then the derived capability is mapped into the recipient's virtual address space (VSpace). Or the sender can map the capability directly into the recipient's VSpace if they have access to their PML4 capability, which is the VSpace root. Internally, seL4 keeps track of the derivation tree [42, sec. 3.1.5], so that when a capability is revoked or deleted, the derived capabilities (as well as the original capability in case of `Delete`) and the referenced object are deleted properly.

2.5.3 IPC Capabilities

IPC on seL4 is handled similarly to capability invocation. In fact, IPC is handled via capability invocation [42, chpts. 4, 5]. seL4 offers a capability for message passing and signals each, called Endpoint and Notification capability respectively. Endpoints are discussed in §2.5.3, while Notifications are touched on in §2.5.3.

Endpoints

In seL4 an Endpoint represents the right to send or receive messages to and from the specific endpoint represented by the capability. An Endpoint has four different rights [42, sec. 3.1.4]:

- *Send*, which allows the holder to send data via the Endpoint
- *Receive*, which allows the holder to receive data via the Endpoint
- *Grant*, which allows the holder of the Endpoint to transfer capabilities
- *GrantReply*, which allows the holder of the Endpoint to transfer a Reply capability.

A Reply is a special kind of capability that is granted to the receiver of an Endpoint message and can only be used once. It is only used when the sender invokes `Call` with the Endpoint, which blocks the sender until the receiver sends a message on the received Reply. If an Endpoint does not have the *Grant* or *GrantReply* rights, the calling thread is simply suspended and needs to be manually restarted [4, sec. 4.2.4].

An Endpoint with *Grant* can also transfer other capabilities. For this, the receiver simply sets the CNode slot to save the new capability to and calls `Recv`. The sender places the CPtr in the designated array and calls `Send`. After a successful rendezvous, the receiver owns a copy of the original capability. For sake of scope we will not describe capability unwrapping, which is the mechanism used when sending more than one capability [4, sec. 4.2.2].

Since seL4 has no kernel-housed message buffer, both the sender and a receiver need to be waiting on the Endpoint at the same time. This means either the receiver must already be waiting with `Recv` when the sender uses `(NB)Send`, or the sender must already be waiting for the receiver with `Send` for a message to be successfully delivered. `NBSend` fails quietly if the message could not be delivered. In addition to the normal communication pathway, Endpoints also supports the *fast-path*, which is a highly optimized path through the kernel [43]. The fast-path is invoked if the following conditions hold true: 1. Endpoint was invoked with either `Call` or `ReplyRecv` 2. No thread with a higher priority is waiting to be scheduled 3. The message consists only of regular data and fits into the message registers. We categorize Endpoint messages as synchronous, with both blocking and non-blocking variants for `Send` and `Recv` [42, sec. 4.2].

Notifications

A Notification represents an asynchronous signaling mechanism [42, chpt. 5]. The transmitted signal is the badge value of the Notification, which is saved

by the kernel until retrieved. Additionally, if a thread is already `Waiting` on the Notification, the first waiting thread is unblocked. There are three invocations for Notifications [42, sec. 5.3]:

- `Signal`, which sends a signal with the badge value
- `Wait`, which waits until a signal is received
- `Poll`, which maps directly to `NBRecv` and checks if a signal is pending and simply returns if none exist.

A single Notification can also be bound to a thread's TCB, which allows the thread to also receive a signal any time it calls `Recv` on an Endpoint. It is up to user-space to determine whether the message was a signal to the bound Notification or a message on the Endpoint. We categorize Notification messages as asynchronous, with both blocking and non-blocking variants for the receiver and only non-blocking `Signal` for the sender.

2.5.4 Interrupt Handling

seL4 specifically uses two capabilities to control access to interrupt request lines (IRQs). The IRQControl capability is handed to the initial task and can then be used to create new IRQHandler capabilities. An IRQHandler can control legacy interrupts, IOAPIC interrupts, or MSI interrupts [42]. Which type of IRQHandler is created depends on the specific invocation used on IRQControl, which each type using a different method and parameters. Once an IRQHandler capability is created, `SetNotifcation` can be used to register a Notification to the IRQ, which can then be used to receive interrupts by calling `Wait` or `Poll`. Therefore, forwarding an interrupt to user-space is done via an asynchronous IPC mechanism and is not preemptive. Interrupts are acknowledged with IRQHandler's `Ack`, after which the next interrupt can be received with `Wait`.

Chapter 3

Related Work

In this chapter, we first provide a comprehensive overview of analyses on Intel's UINTR feature (§3.1) to gauge its viability for use in µkernels as well as shed some light on potential limitations. The following section explores further work on applications of UINTR, most of which are user-level preemption. Lastly, we explore work related to IPC on µkernels as a potential sources for comparison and inspiration for our design (§3.3).

3.1 User-Level-Interrupts

This section takes a look at recent work on UINTR to provide a better understanding of its benefits and limitations. We first dive into some introductory work (§3.1.1), look at a security analysis (§3.1.2) and lastly into technical analyses of Intel's UINTR (§3.1.3).

3.1.1 Introductory Work

One of the first scientific works to use UINTR was an effort to replace the polling-based receive-mechanism of *NewMadeleine*, an event-driven communication library that allows asynchronous communication, with notifications based on UIPIs. For this, Goedefroit [44] uses Intel's Linux patch to compare UIPIs and OSS and found that UIPIs have a lower and more consistent delivery latency than OSS, with a roughly 3× advantage. Goedefroit also finds that UIPIs is not unaffected by the non-uniform memory access (NUMA) layout, with UIPIs between distant cores being ~1.5× slower, however, OSS are also affected by this, as mentioned in §3.3.4. In NewMadeleine, Goedefroit prepares both a notification mechanism for a shared-memory IPC system based on OSSs and UIPIs and shows that the UIPI driver has a slightly higher latency than a busy-waiting variant, but performs bet-

ter than OSS. These results are reflected in the new driver's throughput, which is slightly lower than busy-waiting but still massively outperforms OSS. In essence, Goedefroit demonstrates that UINTR can provide tangible performance benefits compared to OSS while reducing the overhead ratio compared to busy-waiting. Goedefroit also submitted a pull request for a bug in Intel's kernel patch related to the alt-stack feature.

3.1.2 Security Aspects

The first security analysis of UINTR provides insight into the characteristics of UINTR and IPI virtualization (IPIv), both features were introduced with Sapphire Rapids. Rauscher and Gruss [45] confirm the comparatively low latency of UIPI and use these characteristics to construct a covert channel, a keystroke detection mechanism and a website fingerprinting mechanism, all of which work under virtualization due to the new IPIv feature, which allows virtual machines (VMs) to send IPIs without supervisor intervention. Rauscher and Gruss highlight both the potential benefits and security risks of these new features, if used without mitigations. UINTR also need to be considered when implementing sandboxing mechanisms, as shown with Erebor [46].

3.1.3 Technical Analyses

Using reverse-engineering and fine-grained benchmarks, Aydogmus et al. [47] analyzed the detailed performance characteristics of UIPIs from the sender, receiver, and round-trip perspective. They find that receiving UIPIs flushes the instruction pipeline, which leads to a loss of throughput, according to them, unnecessary latency. Aydogmus et al. propose extended user-interrupts (xUI), with 4 aspects: 1. Tracked interrupts, 2. hardware safepoints, 3. kernel-bypass timers, 4. interrupt forwarding, all of which they implement and evaluate on simulated hardware. The most interesting to us are tracked interrupts, which promise to reduce UIPI latency by using draining instead of flushing together with branch mispredictions to dynamically inject UIPI micro-ops into the instruction stream at a potentially earlier point in time. Hardware safepoints would automatically enable or disable UINTR for applications instead of explicitly requiring the use of `SETUI` and `CLUI` instructions, while interrupt forwarding extends the UIN mechanism already provided by UINTR with dedicated support for multiple UDIs using two new 256 bit fields [47]. Kernel-bypass timers are also of interest to Intel, who have proposed their own user-timer system using UINTR which might be introduced with Clearwater Forest processors [34], potentially to mitigate the need for manual solutions such as the ones found in Skyloft [38].

The most recent technical analysis characterizes UINTR with a focus on virtualization. Kone et al. [48] look at UINTR's general capabilities, trade-offs for potential software wanting to use UINTR and offer a unique perspective on the performance characteristics in a virtualized environment, both with IPIv enabled and disabled. For this, Kone et al. build a new set of benchmarks to compare UIPIs to OSS, with a custom function to read the TSC, `readtsc()`. In line with other previous work, Kone et al. find an increased delivery latency (~1.3×) depending on the physical placement of the sender and receiver threads for both native and IPIv-enabled systems, as well as a severe (up to ~2.6×) delivery latency degradation under IPIv compared to a native system. However, they still prove that UINTR is a viable alternative to OSS in any case, as every UINTR operation outperforms its OSS counterpart, especially the sending operation, which is up to ~25× faster and scales exponentially better under contention.

As a potential use-case for UINTRs, Kone et al. develop a user-level scheduler, *Christine*, which will be further discussed as part of section (§3.2.1).

3.2 Applications of UINTR

Following the technical aspects of Intel's UINTR, we introduce literature that focuses on analyzing UINTR in different use-cases. The most common use-case is user-level preemption to implement user-level schedulers. Work focusing on this is discussed in §3.2.1, while §3.2.2 touches on work focusing on other applications.

3.2.1 User-level Preemption with UINTR

An early work using UINTR is a user-level threading library for cloud applications by Li et al., called *LibPreemptible* [49]. Li et al. use UINTR to construct a user-level timer – their implementation of which is called *LibUtimer* – that provides regular interrupts to threads by having a dedicated timer thread poll on thread-based deadline set in memory. Once a deadline is reached, *LibUtimer* sends a UIPI to the offending thread, which triggers a context switch via the interrupt handler. *LibPreemptible* can implement various scheduling policies and dynamically change time-slice quanta. This results in a flexible scheduler that can achieve better tail-latency and higher throughput than the state-of-the-art scheduling system of the time, Shinjuku [50].

Shortly after, Fried et al. present *Junction* [51], a kernel-bypass system for the cloud. Similar to *LibPreemptible*, *Junction* also uses UIPIs from a separate scheduler core to preempt its user-level threads and provide an equal workload distribution. They find that UIPIs reduce the timeslicing overhead by ~2× when

compared to OSS, which allows for higher time slice granularity, which in turn provides a potential avenue for reducing tail latency in µsecond scale workloads, as demonstrated with *LibPreemptible*. In addition Fried et al. find that, for saving the extended processor state in the interrupt handler, the `XSAVEC` instruction is as fast as `XSAVEOPT`, while being easier to use correctly.

SkyLoft [38] is a user-level scheduling framework, which can support multiple applications, instead of just threads within the same application. Its use of UINTR differs to previous work by being the first published work to enable UINs for hardware timers. As native support for user-space timers and proper hardware interrupt notifications are still in development [34, 52], this is not entirely trivial and requires manually setting a bit in the `PIR` field of the UPID, so the hardware interrupt actually triggers UID. Jia et al. solve this by sending a self-UIPI while UINs are suppressed in the interrupt-handler and at setup, setting the bit without causing UID until the next hardware timer interrupt arrives. Even with this additional overhead, Jia et al. find that hardware timer notification are still faster than dedicated user-level timer cores sending UIPIs.

While UINTR is not the main focus of their publication, Lin et al. [53] use UINTRs in combination with MPKs to create a user-space core scheduler, called *Vessel*, which uses a user-level privileged mode to separate address spaces between threads in user-space, called a *uProcess*. Lin et al. use UINTRs to preempt different *uProcesses*, which then transition into the user-space privileged mode via protected call gates. Once the core is in the privileged state, it switches its address space, scheduling a new *uProcess Vessel* is used to co-locate different types of applications to the same core, while providing an increased throughput for both latency-critical and best-effort applications when compared to contemporary solutions, with UINTRs being a key technology for these results.

Guo et al. [54] compare OSSs to UINTR as a preemption mechanism in two different user-level schedulers, one based on *Caladan*, a kernel-bypass system, and another based on the Go runtime. They compare UINTR-based and OSS-based preemption to compiler instrumentation, with which a compiler inserts regular yield checks for cooperative scheduling, and find that the decreased overhead with UINTR is inconsequential for application performance with larger time quanta, while being at least on-par with compiler instrumentation for a 10µs quantum. OSS, as also shown in previous work, perform less favourably. However, the schedulers' average preemption cost with UINTR is significantly lower than that of compiler-instrumentation. Guo et al. find UINTR "are not a panacea", but have some useful applications, such as µsecond scale preemption. In addition, they find that, compared to OSSs, using UINTRs leads to fewer L1 cache misses and branch mispredictions. In regard to context switches, Guo et al. find that saving the AVX-512 registers with `XSAVEC` incurs a ~10-30× overhead compared to saving the registers one-by-one.

As a further application of user-level preemption, Huang et al. [55] introduce UINTR to database engines with *PreemptDB* and compare UINTR-based user-level preemption to waiting, hand-crafted, and automatic cooperative scheduling. *PreemptDB* uses preemption to suspend low-priority transactions when new high-priority transactions are queued. While doing so, Huang et al. recognize the need for atomicity in their context switch routine, which can be called outside of interrupt handlers. To achieve this, the context switch routine temporarily disables UID and includes instruction pointer checks to ensure that the currently handled interrupt was not delivered while inside the context switch. *PreemptDB* with UINTR provides lower latency for high-priority tasks while maintaining throughput and requires no specific tuning.

Finally, the user-level scheduler by Kone et al. [48] – *Christine* – is used to compare and document differences between UINTR and OSSs in both native and virtualized systems to document the extent to which UINTR can reduce tail latencies. Similarly to *LibPreemtpible*, Kone et al. use a dedicated timer thread to busy-spin and call their custom `rdtsc()` function. Kone et al. additionally implement worker synchronization to ensure workers are not spending their entire time in the scheduling routine due to an insufficient time quantum. *Christine* predictably has degraded performance in every metric under virtualization, while showing that UINTRs are capable of supporting a more precise, stable and focused scheduler at smaller timer quantum sizes than OSSs.

3.2.2 Other Applications of UINTR

Li et al. [56], similar to Jia et al. use self-UIPIs to receive hardware interrupts over the UINTR mechanism. They use this to modify *SPDK*, a I/O storage software, which can either use a polling thread to check for hardware availability or receive and handle MSI-X interrupts via the kernel. Their modifications result in a two-thread user-level threading framework, which switches between an idle-thread executing `TPAUSE` and the actual worker thread, depending on whether the thread is waiting for the I/O operation interrupt, redirected via the IOMMU to the UINV, or not. *SPDK+* achieves a similar latency to the original polling method, while also achieving slightly better power efficiency than both the original polling and interrupt method. Li et al. predict even higher efficiency for increased core counts.

Goedefroit continued their work on applying UINTR to the BXI network [57]. Most importantly for us, Goedefroit et al. found a way to trigger a UINTR directly from a PCIe device by combining the posted interrupt descriptor (PID) structure of IOMMU interrupt redirection with the UPID. While useful for avoiding unnecessary interrupt-management via self-UIPIs, it requires mapping the UPID/PID-union to user-space, which would allow malicious software to trigger arbitrary

interrupts using SENDUIPI, as we already discussed in §2.4.5. Goedefroit et al. find that UINTR are slower than polling but provide a decent communication/-computation overlap, which makes it a good tool for use in high performance computing applications, in line with Goedefroit's previous findings [44].

Similar to Goedefroit et al., Li et al. [58] found their own way to circumvent the self-UIPI in order to receive hardware interrupts in their UINTR-based storage system *Aeolia*. Instead of having hardware set the required PIR field, they set the PIR field themselves by mapping the UPID into their user-space driver's memory, effectively emulating the mechanism self-UIPIs are used for in previously presented work. *Aeolia* therefore suffers from the similar security issues as Goedefroit et al. when it comes to the triggering of arbitrary interrupts.

3.3 IPC on Microkernels

With the clear potential of UINTR in µkernel IPC highlighted in the previous sections, we look at some other recent work in this field. We take a closer look at the "Bridge"-family of IPC mechanisms, starting with SkyBridge (§3.3.1) and continuing along the line of advancements with UnderBridge (§3.3.2) to the most recent representative, HyBridge (§3.3.3). We end the section with a short look at other IPC-related work (§3.3.4).

3.3.1 SkyBridge

SkyBridge is another new IPC mechanism, this time specifically designed for µkernels. It uses the VMFUNC instruction to bypass the kernel and directly call code from other processes by switching the extended page tables (EPTs) of the client and server when the client wants to invoke a server function. For this, servers register with the kernel to map function entry points to a table, which a registered client can then use to call server functions via a library call. To keep unauthorized processes from executing VMFUNCs, switching EPTs and calling server functions without permission, SkyBridge utilizes binary rewriting when mapping code pages. For actual message passing, SkyBridge uses shared buffers for large messages and registers, complying with the x86/64 calling convention, for small messages. SkyBridge achieves a ~2-10× round-trip-time advantage compared to the default IPC methods [59].

3.3.2 UnderBridge

The successor to SkyBridge, UnderBridge, instead circumvents cross-server IPC overhead entirely by moving system servers back into the kernel. Since the

kernel has a shared state, cross-server IPC skips context switches and kernel-controlled argument validation. UnderBridge can therefore be seen as a new form of kernel-bypass, but communication between servers needs new isolation and protection mechanisms to remain secure. In addition to this, servers can now execute privileged instructions, potentially endangering the entire system if faulty. UnderBridge achieves isolation via MPKs, which creates isolated *execution domains* for each server in kernel mode. Execution domains consist of code segments, stored in the core kernel protection domain with read-only access, and data segments, stored in their own protection domain to which they have full access. Cross-server communication is handled via IPC-gates installed in the code-domain and a shared memory domain for each communication channel, other channels are handled via shared memory pages with the core kernel or client. Since MPKs only support 16 different protection domains, servers are dynamically migrated between user- and kernel-space as needed if the maximum amount of protection domains is reached. Privileged instructions are removed via binary rewriting and trapped by introducing a hypervisor which ensures privileged instructions are only executed by the core kernel. These measures prevent malicious system servers from intentionally modifying or compromising the core kernel. UnderBridge, despite having a worse base-case of 0 cross-server IPC on client-server IPC, scales much better and therefore reaches parity and even surpasses SkyBridge at 1-2 cross-server IPCs per client-server call. This leads to a ~1.5× throughput advantage in the SQLite3 benchmark compared to SkyBridge [6].

3.3.3 HyBridge

HyBridge further enhances UnderBridge by increasing the amount of available isolation domains by combining MPKs with EPTs, calling the combined mechanism extended protection keys (EPKs). By observing that MPKs and EPTs are both thread-specific values, it assigns each of the 512 EPT the 15 available domains with MPKs, leading to 7680 possible memory protection domains. This allows EPKs to massively outperform software-based multiplexing solutions for EPKs on Linux and eliminates one of the major downsides of UnderBridge by allowing HyBridge to move every system server into kernel space instead of just a few. Cross-Server communication is unchanged from UnderBridge, but client-server communication now utilizes a EPT switch via `VMFUNC` instead of a regular context switch to kernel. This allows HyBridge to achieve throughput parity in the SQLite3 benchmark with UnderBridge, while surpassing it in most cases [7].

3.3.4 Other work

We end this section with a draft on IPC benchmarking, which finds, by testing IPC latencies on different OS's in virtualized and native environments, that an observable part of IPC latency is affected by the NUMA layout [60], which will be relevant for later sections and would also affects a new IPC implementation for multi-core systems on µkernels, which uses shared memory and regular IPIs on ARM to implement a inter-core IPC and notification utility on its own µkernel [61].

Chapter 4

Design

This chapter details the design for our implemented system, which is a new IPC facility on L4 μkernels using the new UINTR feature in conjunction with the user-wait extension. Given that previous research on UINTR proved its usefulness as a replacement for OSS on Linux, we intend to see if UINTR may also have a place in the realm of μkernels. Due to the differences between L4 and Linux, we first introduce our design for UINTR on our chosen kernel in §4.1, as we could not simply port Intel's Linux patch.

If this kernel-bypass mechanism can be used to implement most IPC facilities with potentially improved performance compared to already existing facilities, introducing UINTR as a core of modern μkernels design could align the next generation of μkernels to have an even smaller TCB, further reinforcing the tenets mentioned in §2.1.3. With this in mind, we explain our design for our IPC library in §4.2.

4.1 Capability-based User-level Interrupts

We first explain some of our initial thoughts and approach in §4.1.1, after which we illustrate our capability-based designs for UINTR objects in §4.1.2.

4.1.1 Initial Approach

We initially intended user-interrupts to be configured by a server, which receives requests from clients to establish UINTR-based connections between server-clients and client-clients to allow for RPCs. Server-clients would advertise their API via the server, where client-clients would then request access. They would then be handed the appropriate permissions or tokens and the server would set the

client's MSRs and map the appropriate memory to open the communication pathway. Ideally this system would be kernel-agnostic, to allow for easy portability.

Given the impossibility to control MSRs from user-space (at least for now, future plans from Intel lay out plans to control some MSRs from user-space [34], however, none of these are UINTR MSRs), we instead switched to a capability-based approach. This massively decreased portability. However, we were instead able to observe how well UINTR fit into a capability-based design, instead of Intel's file-based approach on Linux.

4.1.2 Capability-based User-level Interrupts

Since seL4 is uses capabilities to control access to hardware features, we decided it would be fitting to design our implementation UINTR support around this. Therefore, we decided to control UINTR objects via capability invocations instead of normal system calls. As there are two physical objects, the UPID and the UITT, we decided to introduce two new capabilities, UINTRNotif and UIPICap to control the UPID and UITT, respectively. Design details for the UINTRNotif are discussed in §4.1.2, while the UIPICap is introduced in §4.1.2.

User-Interrupt Notifications

There are four aspects of a UPID that need to be controlled:

1. The user-interrupt notification vector UINV
2. The handler UIHandler
3. The stack adjust UIStackadjust
4. Mapping of the UPID into both the sender's and receiver's address spaces
5. The user-interrupt vector UIV

These were initially set as follows:

1. Set during capability creation, by deriving from IRQHandler

2,3 `SetHandler`, which sets the UIHandler and UIStackadjust at the same time

4. `InstallUPID`, which maps and installs the UPID
5. Intended to be freely-chosen by the sender

Aside from necessary adjustments that came up during the implementation and will be discussed in §5.2, our design was later expanded to also use a badge value for the UINTRNotif to set the allowed user vectors, which allows the deriving thread to "lock" a UIV. UINTRNotifs with a badge value $b \in [0, 63]$ have a fixed badge value, meaning they cannot be derived if the deriving thread intends to change the badge.

In addition to this, we also split `InstallUPID` into `MapUPID` and `InstallUPID`, since not every thread that wants to map a UPID wants to install it. `MapUPID` maps the UPID into a given VSpace, once per capability. If a UPID needs to be mapped into multiple address spaces, the UINTRNotif needs to be derived first. The new `InstallUPID` now only installs the UPID into a given TCB and sets the appropriate MSRs. This split allowed us to designate two rights states for UINTRNotif:

1. *Map*, which only allows the holder of the capability to map the UPID to an address space

2. *Modify*, which allows the holder of the capability to modify the UPID's fields in addition to mapping, as well as install it in a TCB

These are useful when establishing a UINTR connection, since `SENDUIPI` requires the sender to also have the UPID mapped into their address space. A receiver might want the sender to have free control over the address the UPID is mapped to in the sender's VSpace, while only wanting the receiver to be able to modify it. UINTRNotifs are unable to derive new UINTRNotifs with higher rights for obvious reasons. The steps required to create a new UINTRNotif can be observed in Figure 4.1.

User Interprocessor Interrupts

As mentioned in §2.4, `SENDUIPI` uses the UITT to look up where to send the UIPI to. This leads to the second capability of our design, the UIPICap, which controls a UITT and its entries. For `SENDUIPI` to work, a UITT needs to be mapped into the sender's VSpace and contain valid UITTes. Additionally, the related MSRs must be set. We therefore arrive at our first invocation, `InstallUITT`, which controls two of these aspects: It maps the UITT and sets the related MSRs, which associates the UITT with a TCB.

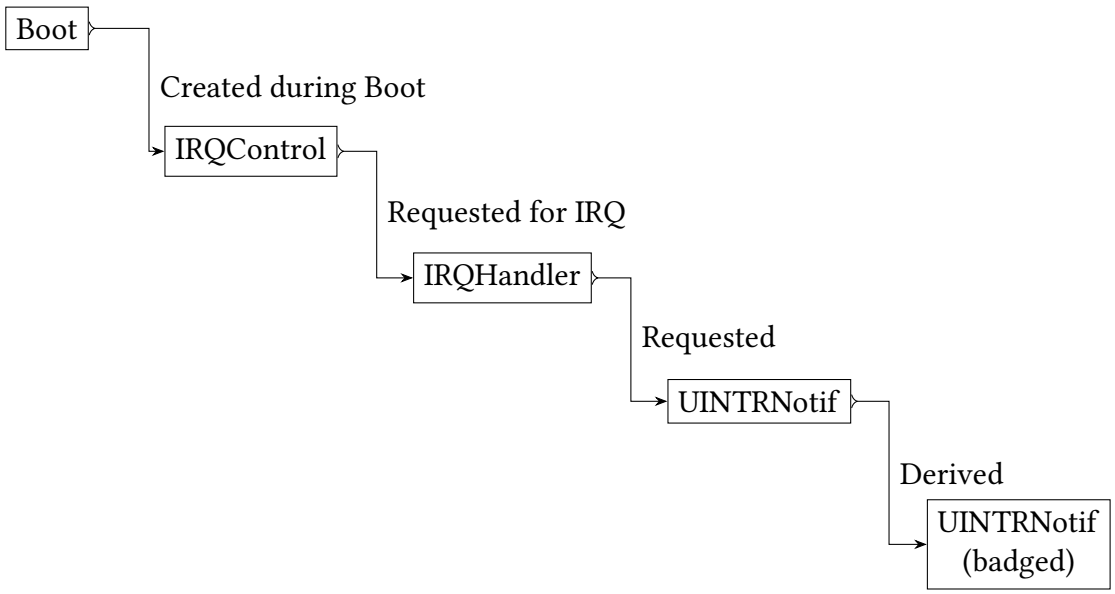


Figure 4.1: Relationships for UINTRNotif and related capabilities for UINTRNotif creation

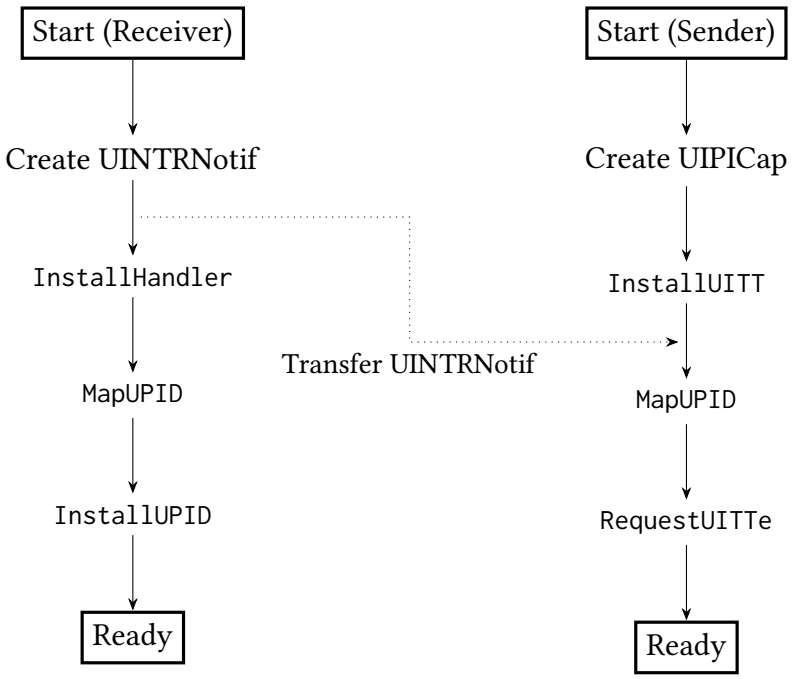


Figure 4.2: Example flow for setting up a connection using our capability design

The second invocation consequently needs to control the third aspect: Creating valid UITTes. A valid UITTe consists of an address to a UPID and a UIV. We control these by using a CPtr to a UINTRNotif and a desired UIV as parameters to our invocation, `RequestUITTe`. The invocation fails if the passed UINTRNotif is not already mapped. As previously mentioned, UINTRNotif can have a locked UIV. If the UIV is locked, the UINTRNotif's UIV is chosen for the entry, instead of the passed parameter. `RequestUITTe` returns the index n of the new UITTe, which can then be used by applications as the argument to `SENDUIPI` n. We provide a flow-chart for the required invocations necessary to be able to send and receive UIPIs in Figure 4.2. We also have a rights field for the UIPICap, which sets the capability to one of two states:

- *Request*, which only allows the holder of the capability to request new UITTes with `RequestUITTe`
- *Map*, which allows the holder of the capability to map and install the UITT in addition to the rights granted by *Request*

Our design allows us to employ three different UINTR usage scenarios:

- UINTR-director: A single thread holds all capabilities. It alone installs UITTs and UPIDs into threads and requests new UITTe on behalf of sending threads. The capabilities are never passed to the controlled threads.
- UINTR-setup thread: A single thread creates all capabilities. It installs the UITTs and UPIDs and passes reduced-rights versions of these to the managed threads. Sender threads are then free to request UITTes, while receiver threads can decide how to share their UINTRNotifs and which UIVs to distribute to which sender.
- Self-Managed: Threads manage the UINTR capabilities themselves. However, receiver threads still need to be handed the appropriate IRQHandler capabilities to create UINTRNotifs.

4.2 IPC Library with UINTR support

This section details the design of our IPC library—uIntercom (uIcom)—which uses UINTR and the user-wait extension to provide every type of IPC mentioned in §2.2, along with some additional functionality. Since our design employs variables to poll on, the reasons of which will be explained in the appropriate sections, we use the user-wait extension to design different types of polling, explained in §4.2.1. For our library's design, we first explain our approach to signals in §4.2.2, after which we go on to message passing in §4.2.3 and finally RPC in §4.2.4.

4.2.1 Wait Types

For our library we would like to offer as much flexibility as possible and have therefore chosen four different wait types:

- `Poll`, which simply polls on the variable
- `TPAUSE`, which uses the new `TPAUSE` instruction while polling
- `UMWAIT`, which uses the new `UMWAIT` instruction on the variable while polling
- `Yield`, which yields the thread while polling

These should be able to be chosen on a per-call basis instead of a per-connection basis, to provide further flexibility.

4.2.2 Signals

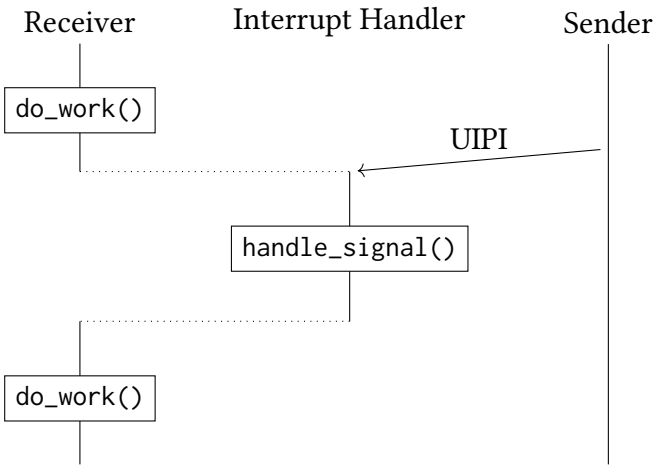
Signals, as defined in §2.2.3, are event notifications combined with a data word. The astute reader might have noticed that user-interrupts are already a full-fledged asynchronous-preemptive signaling mechanism, with the UIV being the data word. Therefore, our design for the asynchronous-preemptive signals consists only of the method `uIcom_Signal()`, which simply sends a user-interrupt in a non-blocking manner. In order to have the receiver handle the preemptive signal, our design lets the receiver associate a handler function with the connection during setup, which is then called when a signal is received.

If the connection is instead marked as **not** asynchronous-preemptive, this mechanism should either be disabled or ignored, with the signal handler instead setting a flag, which can then be polled on. Our non-asynchronous-preemptive signals therefore add two additional methods:

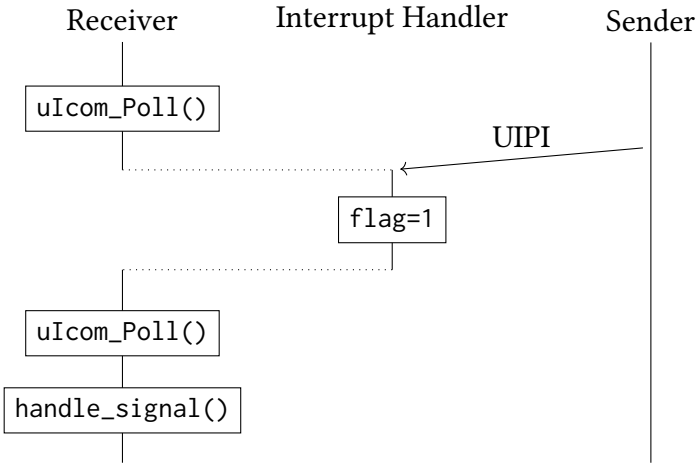
- `uIcom_Poll()`, which checks if the flag is set, possibly resets it and returns the flag's state before the reset
- `uIcom_Wait(wait_type)`, which continuously polls on the flag with a dedicated `wait_type` as listed in §4.2.1

The signal can then be handled once `uIcom_Poll` or `uIcom_Wait` return. A sequence diagram for both the asynchronous-preemptive and asynchronous variants are available in Figure 4.3.

Our design for signals offers near-parity with seL4's Notification capability. While our design has a new asynchronous-preemptive mechanism, which is entirely unexplored on seL4, using the UIV as the signal type limits us to a single bit of data per signal, whereas seL4's Notification can update multiple bits of the notification word by having a badge with multiple set bits [42, sec. 5.2.]. However, we believe this does not limit our design's potential in any significant way.



(a) A thread does work, gets preempted and auto-handles the signal, then continues to do work



(b) A thread calls `uIcom_Poll()` until a signal is received and afterwards handles the signal

Figure 4.3: Sequence diagrams for receiving a signal with uIcom for both the asynchronous-preemptive and manually synchronous paths

4.2.3 Message Passing

As mentioned in §2.2.3, signals can be used in combination with a transport layer to create a message passing system, which is exactly how we designed ours to function. Since other experimental IPC systems on seL4 have used shared memory as a transport layer [6, 7, 59], our design intends to do the same. Therefore, every connection also needs a region of shared memory to pass data.

In terms of methods, we introduce `uIcom_SendNB(data)`, which combines a call to `uIcom_Signal()` with a data copy to the shared region. The receiver can then use `uIcom_RecvNB(data_destination)` or `uIcom_Recv(data_dest, wait_type)` to copy the data to a destination after an underlying `uIcom_Poll()` or `uIcom_Wait()` has succeeded. Alternatively, in case of an asynchronous-preemptive method, the receiver can again register the handler to deal with the data directly in the signal handler.

In order to have a blocking `uIcom_Send()`, we design our `uIcom_Recv(NB)` to also send a non-blocking signal back to the sender. This way our sender can wait in `uIcom_Send()` until the data has been received. Finally, to offer full parity with seL4's IPC capabilities, our design also has simple combinations of `uIcom_Send()` and `uIcom_Recv()` with `uIcom_SendRecv(input, output)` and its counterpart `uIcom_RecvSend(input, output)`. This introduces an issue with UINTR-based connections, they are one-way. To enable two-way connections, our design introduces two-way connections with an in-subconnection and an *out*-subconnection, with `uIcom_Send` and related calls using the *out* and `uIcom_Recv`, related calls, and the signal handler using the *in*-subconnection.

In summary our message passing design consists of our signal design combined with a transport layer for additional data. This decision was made to allow for an easier implementation, allowing us to potentially reuse the signaling components with only minor additions. The decision to have both blocking and non-blocking variants for the `Send` and `Recv` methods was made to have functional parity with seL4's already existing Endpoint message passing system, excluding capability transfer, since that would require entering the kernel. In addition to functional parity, we also want to explore new communication possibilities offered by UINTR, which is why we also thought of a mechanism for asynchronous-preemptive message passing.

4.2.4 Remote Procedure Calls

Since RPC is little more than structured message passing, as already seen in §2.2.4, our design essentially already provides methods for RPC. However, we still introduce two dedicated methods for synchronous RPC:

- `uIcom_Call(input, output)`, which is nothing more than a simple alias of `uIcom_SendRecv(input, output)` that sends procedure arguments to the *out*-subconnections and receives return values from the *in*-subconnection
- `uIcom_Await(handler)`, which first receives procedure arguments, then operates on these using the handler, then sends a reply with return values provided by the handler

In case of asynchronous-preemptive communication, the registered handler is used to transform received parameters into return values inside the signal handler and also trigger a reply with the non-blocking `uIcom_SendNB` on the *out*-subconnection Our design does not include dedicated methods for asynchronous RPC. However, these can be easily constructed by combining the existing `uIcom_RecvNB()` and `uIcom_SendNB` calls.

4.2.5 Final Overview

In total, our design encompasses 9 different methods, some of which provide similar functionality. Each of these methods operates on a connection, combined with function values. A connection consists of two one-way subconnections, resulting in the following fields:

- *out*-subconnection
  - UITTe index for `SENDUIPI`
  - Shared memory
- *in*-subconnection
  - Registered handler for asynchronous-preemptive communication
  - Shared memory
  - A flag to poll on for asynchronous and synchronous methods

A table of every “real” method, aliases, and their existing seL4 equivalents is available in Table 4.1.

Method	List of aliases	Equivalent seL4 methods	Blocking?
uIcom_Send		Endpoint.Send	Y
uIcom_SendNB	uIcom_Signal	Notification.Signal, Endpoint.NBSend	N
uIcom_Recv	uIcom_Wait	Endpoint.Recv, Notification.Wait	Y
uIcom_RecvNB	uIcom_Poll	Notification.Poll, Endpoint.NBRecv	N
uIcom_SendRecv	uIcom_Call	Endpoint.Call	Y
uIcom_RecvSend		Endpoint.RecvSend	Y
uIcom_Await			Y

Table 4.1: Table of overy uIcom method

Chapter 5

Implementation

In this chapter we present our implementation, which we group into three parts. We first worked UINTR into kernel-based virtual machine (KVM) and quick emulator (QEMU) and performed initial testing, which we describe in §5.1. After this, we added support for UINTR on seL4, decribed in §5.2, which we then used to implement our IPC library in §5.3. All of our additions and modifications were made with C, which is the language seL4, KVM and QEMU are written in. While doing so we also made sure UINTR could be turned off, which would allow us to measure the potentially added overhead, which we will examine in Chapter 6.

5.1 User-Interrupts on KVM/QEMU

This section describes our implementation process for getting UINTR to work on QEMU. Our goal was to run a Linux kernel with the Intel patch-set [62] on a VM, to confirm that we could virtualize the UINTR feature and use this virtualized environment for implementing the feature on an L4 μkernel. In order to use UINTR in a virtualized environment, we need to pass the following conditions on to the VM in order to enable and control the new instructions [9, chpt. 4, SENDUIPI]:

- The appropriate `CR4` feature bit needs to be set
- `CPUID` needs to enumerate the UINTR
- The UINTR-specific MSRs need to be passed through

We ported the Intel patch-set to the—at the time—current version of Ubuntu's kernel, v6.14. Despite the patch being developed for mainline kernel v6.0, the porting process was quick and relatively easy, taking only a few hours to complete. We skipped porting changes to `io_uring`, however, this did not cause any

issues. After this we moved on to enabling CR4, MSR and CPUID pass-through on the modified kernel's KVM (§5.1.1) during which it became apparent we also had to add support for UINTR's XState component information pass-through (§5.1.2).

5.1.1 CR4 and CPUID pass-through

As previously mentioned, the first element that needs to be passed to the VM to be able to use and control UINTR is the CR4 bit together with the proper CPUID sections. We immediately ran into the problem that Intel's patch-set relies on a working XSAVES setup. We were able to disable these checks to boot the VM, however, if we wanted a fully-working UINTR implementation we would also need to implement support for this enumeration. There are facilities in KVM that allow for a quick addition of pass-throughs for CR4 bits and CPUID leafs, however, despite having modified KVM, CPUID inside QEMU did not emit support for UINTR or its XSAVE region. Eventually we noticed that we also had to modify QEMU to allow for proper pass-through. Once this was established, we further added the necessary MSRs to the pass-through list and soon after that it was possible to successfully boot a VM with working UINTR instructions, after which we shifted on to re-enabling the XState-related sections of the Intel's UINTR code.

5.1.2 UINTR-XState support

While, as mentioned in §2.4.4, enabling the UINTR-XState is not technically necessary for using user-interrupts, it is still helpful for managing UINTR-enabled threads and should be part of a fully-featured implementation. When enabling the check for the UINTR-XState we encountered two main issues:

- CPUID's XSS enumeration was not properly set
- The size calculation of the XState produced wrong results

XSS enumeration was simply cleared at multiple points in both KVM and QEMU, where we then had to forcibly set the UINTR-XState bit again. In addition to this, the XState save region size calculation both in QEMU and KVM did not properly consider the size of XSAVES-managed XStates when passing the required size to the intercepted CPUID output, with KVM not even passing the enumerated XSS to the calculation function, despite previous commits declaring they "fixed" support for enumeration XSAVES-managed features [63].

With these feature implemented and tested using the Intel patch-set, we were confident in our ability to properly test and implement every aspect of our future modified µkernel's UINTR support. While initially scheduled to be implemented in two or three days, this whole process took nearly twice that amount of time, due to the many issues with KVM and QEMU.

5.2 User-Interrupts on seL4

After establishing that seL4 could, in theory, support UINTR when run on QEMU, we implemented this support in the kernel. In order to test this support, we also had to implement a user-space runtime, for which we used the "sel4test" runtime as a starting point, which is a test suite for seL4 [64]. By regularly switching back to the unmodified test suite we were also able to ensure our implementation did not break the kernel or required functionality in unforeseen ways.

5.2.1 Additional Background

seL4 uses a custom bit-field generator with its own language specification for its structures and capabilities, since C's own bit-field definitions are too under-specified for use in formal verification [65]. These specifications are then turned into accessor functions for the fields, while the structure itself is simply defined a multi-word array[1]. The generator outputs C code, which are then fed into the compiler during compilation. Capabilities are a union type, where every capability has the same size—128 bits for x86/64, as mentioned in §2.5.2—and a shared `capType` field, which is then used to determine the validity of a field access. These generated bit-fields are also used to model MSRs, various registers and other objects, like paging structures. For defining capability invocations seL4 instead uses extensible markup language (XML) files which are then converted to user-space stubs by another tool. These stubs, when called, then marshal the invocation arguments into message registers, call `syscall` with the correct arguments, and also unmarshal an invocation's return values the same way, significantly simplifying the process for creating and modifying invocations. The invocation format defines the name and return type of an invocation and its arguments, documentation, and configuration dependencies. The generated code is also fed into the kernel to provide enums that can be used to determine the current invocation and call the correct implementation.

5.2.2 Initial Steps and UINTR Capabilities

We started implementing kernel support by first activating the required bit in the `CR4` register. This was comparatively simple, as we only had to find the section that already set the `CR4`, expand the `CPUID` output structure to contain the bit that indicates support for UINTR, which is then checked to set the appropriate `CR4` bit. After this we shifted to implementing the desired capabilities.

[1] The accessors follow the pattern `<[union_name]_>[struct_name]_<ptr_>[get/set]_[field_name]()`, where parts inside <> are optional

Our inital UINTR capability was a precursor to UINTRNotif. We implemented `SetHandler` and `InstallUPID`, initially with a fixed UINV and capability creation via Untyped's `Retype`. At this point these methods were little more than wrappers around `wrmsr`. We then added `DebugTrigger`, which triggers UID by writing the desired UIV directly to `IA32_UINTR_RR` for testing purposes without first having to implement the setting and generation of UITTs. Finally, UIStackadjust was then moved from `InstallUPID` to its final location, `InstallHandler`, which also took on the functionality of `SetHandler`. The process of creating a new capability and its invocation was surprisingly straightforward and simple, due to seL4's extensive tooling for this purpose.

5.2.3 Issues Encountered

During testing and implementation of the UINTR capabilities, we often ran into issues of random failures. There were two major types of these, which we discuss in the following sections.

Memory Access Fault

Apart from the "expected" access faults coming from an incorrect implementation of UITTs and UPIDs configuration, we also encountered unexpected memory access faults after exiting a system call or an invocation. Believing this to be an issue with how registers are saved and restored in the UIHandler or a related issue, perhaps some register was clobbered and caused erroneous memory accesses, we added every general purpose register to the list of registers saved by the UIHandler. After the issue kept appearing anyways, we concluded the issue was something else and added guards around every system call and invocation by modifying the appropriate tooling to emit `CLUI` and `STUI` instructions in the user-space stubs, which did solve the issue.

Unsatisfied with our solution, we later revisited this and found that the seL4 kernel sets `RSP` to `0` before calling `sysexit` to avoid leaking information [64], while the user-space code saves and restores the register before and after `syscall`. When combined with the use of UIStackadjust`[0]` $= 0$, which sets the new `RSP` value by subtracting UIStackadjust from the current `RSP`, the CPU issues memory accesses at `RSP` $= -8$ if UID occurs between `sysexit` and the user-space `RSP` restore. We therefore arrive at the first limitation imposed on us by using seL4 as our platform:

UIStackadjust is limited to the jumping behavior and must have a valid address as its value.

While this could be fixed by leaving kernel-space with a non-zero `RSP`—either by setting it to a fixed memory region for this specific case or by saving and

restoring RSP on kernel entry and exit instead of doing so in user-space—any such modifications must be done with great care to not break existing functionality and is outside of the scope of this thesis. Therefore, every user-space application must provide an alt-stack when using UINTRNotif. Our implementation was adjusted to automatically set this bit in the MSR and the user-space invocation stub was changed to disallow selecting whether to use the alt-stack behavior or not.

Stop of User-Interrupt Delivery

We encountered our second recurring issue while running self-UIPI in an infinite loop. Eventually, our system would crash, because the intended user-interrupt was delivered to the kernel instead[2]. Since we had not registered the UINV with the IRQControl, this caused our kernel to fail. However, once we had modified IRQControl to allow the creation of non-IOAPIC IRQHandlers for UDIs and requested the offending UINV, the issue continued in a different form.

Instead of causing a kernel panic, we instead did not receive any more user-interrupts. This was caused by a lack of UIN processing for this interrupt, which in turn resulted in the UPID's ON bit never getting cleared. With a set ON, SENDUIPI would never send an IPI, which would therefore not trigger UIN processing, causing the UPID to become "stuck". We ultimately solved the issue by manually performing UIN processing when returning to user-space by adding a return hook which moves and clears the PIR manually. To make this possible, we also had to add a direct association between a thread's TCB and a UPID, for which we took inspiration from a similar mechanism with Notification capabilities, later expanded to also include the UITT. This finally resolved the issue. Furthermore, we were able to use the association to correctly clear and set MSRs for both the UPID and UITT, to fix crashes caused by "leaking" these MSRs, as well as dynamically updating the UPID's NDST when rescheduling. Usually clearing and setting the MSRs would be done by XSAVES and XRSTORS, however, the seL4's use of lazy XRSTORS makes these unviable without major kernel modifications that would be outside the scope of this thesis[3]. Unfortunately, this manual clear and restoration causes significant overhead, as we will see when we evaluate our results in Chapter 6.

[2]This is to be expected, if a UIPI is sent but the thread is rescheduled before the sent interrupt is recognized, the interrupt ends up in the kernel.

[3]Around the time we finalized our implementation, an RFC for seL4 was released and implemented that added support for eager XRSTOR(S). This will be touched upon in §7.2

5.2.4 Finalizing our Capabilities

Once we had resolved all important issues, we set out to implement UINTRNotif creation via a IRQHandler invocation, as the design in §4.1.2 intends. Unfortunately, we were unable to do this, due to the associated UPID requiring physical memory, which we could only provide with Untyped's `Retype`. While this change does not change the capability's necessity when setting up UINTRNotif, it does change our intended creation path, the final version of which is available in Figure 5.1. After this we set out to add the `SetVector` invocation to UINTRNotif, which sets the UINV used when targeting its UPID with `SENDUIPI`. `SetVector` uses a IRQHandler capability as its argument and extracts its IV, which is then stored in the UPID.

Following this, we added support for OSS-emulating UINTR (OSSeUINTR), which is what we call the behavior caused by not setting the `UINV` in `IA32_UINTR_MISC` but setting all other UPID fields and MSRs correctly. Any UIPI sent to the receiver are not directly received in user-space, but are instead sent to the kernel, essentially allowing applications to send signals via the kernel without executing a system call. We believe this has potential applications, which we again mention in §7.2, after a preliminary analysis in Chapter 6. As our final steps we also implemented UIV locking and fields for our designed rights states, as well as adding support for checks and conditions when deriving the capabilities with these rights. Unfortunately, we were unable to add checks for these rights in our invocation code and, due to time constraints, we leave them for future work.

5.2.5 Summary

In summary, we added capability-based UINTR support to seL4. While doing so, we encountered both expected and unexpected issues, which changed our initial design plans in various ways. The most important aspects of our implementation and differences to our design are:

- Manually saving and restoring the UINTR MSRs
- UIStackadjust is limited to jumping behavior
- UINTRNotif is created from Untyped instead of IRQHandler, which necessitates the `SetVector` invocation
- Added the `DebugTrigger` invocation

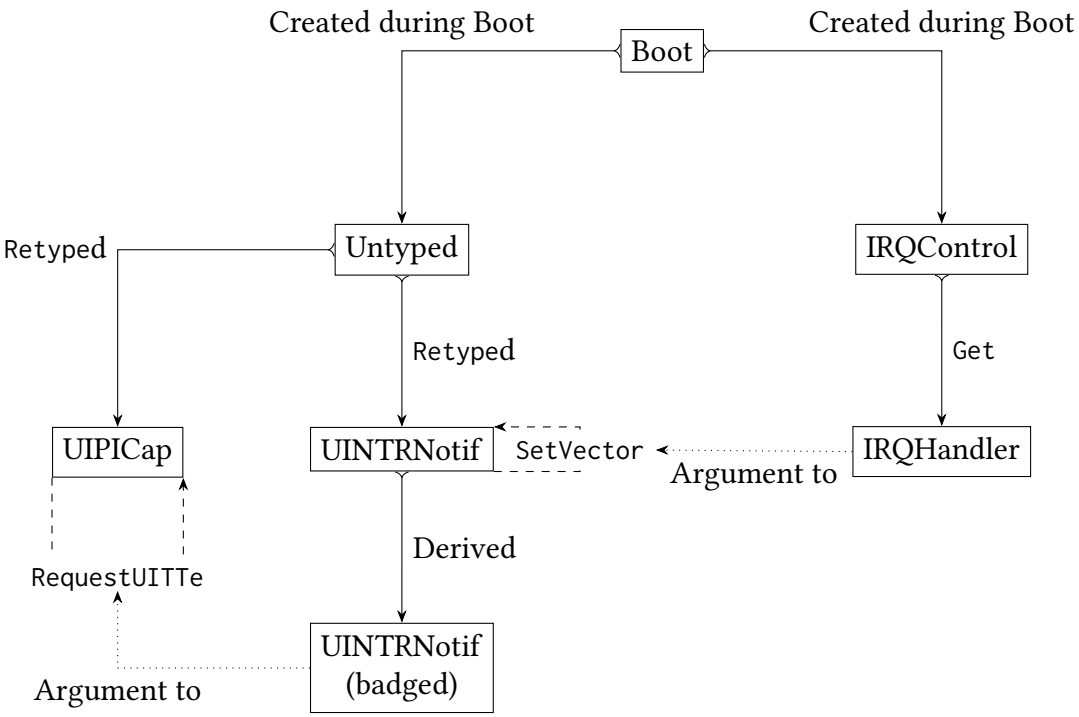


Figure 5.1: Relationship diagram ofUINTR capabilities

5.3 libUIntercom

After implementing the UINTRNotif and UIPICap, we went on to implement our IPC library, uIcom. We begin by describing how we expanded uIcom to support more capabilities (§5.3.1), then describe the connection setup (§5.3.2), how we implemented our UIHandler (§5.3.3), how we implemented `uIcom_Send` and `uIcom_Recv` and their variants (§5.3.4), and finally provide a short summary in §5.3.5.

5.3.1 More than just UINTR

To allow for an evaluation with more comparable results, we also conceptualized uIcom to support more than just our UINTR capabilities. The following is a list of supported capabilities and the reason for their inclusion:

1. UINTRNotif and UIPICap to test both regular UINTR and OSSeUINTR
2. Frame, for shared variable polling, which is a common point of comparison in work on user-interrupts [47, 56, 57]
3. Notification, which is seL4's existing signaling mechanism
4. Endpoints, which supports the *fast-path*, the supposedly fastest communication pathway in seL4

In order to keep code-complexity under control we decided to use these capabilities only as the signaling mechanism and keep data for message passing in shared memory. Unfortunately, while synchronous and asynchronous mechanisms can be constructed from asynchronous-preemptive IPC, this is only one way. Meaning these additional capabilities can only be used to implement uIcom's asynchronous IPC at best and only synchronous IPC at worst, This is simply a limitation of the capabilities which we cannot overcome.

5.3.2 Connection Setup

The setup is a synchronized process between two threads, both of which control how their *in*-subconnection is structured.

Requirements

Ideally, our connection setup would have a unified interface for signaling, message passing, and RPC for ease-of-development. It also needs to perform the following actions: 1. Create the required capabilities, based on a given pathway

type. 2. Setup the required capabilities. 3. Setup shared memory for message passing. 4. Setup both the *in* and *out* subconnections. 5. Test both subconnections. To do this, the setup functions also need access to both a thread's CSpace and VSpace. Our setup function is based on the assumption that both connection partners have access to these, as well as their respective UINTR capabilities.

Implementation

The arguments to our setup are:

1. `tcb`, the calling thread's user-space TCB, which contains both the CSpace and VSpace.
2. `setup_endpoint`, a pre-shared Endpoint for sharing capabilities with.
3. `in_type`, the pathway type of *in* (§5.3.1).
4. `in_cap`, the capability to use for *in* setup (optional).
5. `additional_out_cap`, a capability to use during the *out* setup (optional).
6. `in_handler`, a pointer to a handler function (optional).
7. `mode`, a reply mode, which determines the mechanism by which asynchronous-preemptive receivers process messages (see §5.3.3)
8. `wait_type` the wait type (§4.2.1), which is used when calling `uIcom_Wait` or similar functions.

It starts by setting up *in*, followed by *out*. There is a partner function that first sets up *out* and then *in*, which is also non-blocking, to allow a server to occasionally poll for new connections from clients.

In each subconnection, the receiver's `in_cap` is first created if it does not already exist, then sent to the sender via the `setup_endpoint`, together with a data word to help the sender determine what capability was sent. After this, a page of shared memory is created and mapped, which is also sent to the sender via the `setup_endpoint`. If `in_type` is UINTR or OSSeUINTR, the receiver must provide a pre-installed UINTRNotif `in_cap` and uIcom invokes InstallHandler to install a pointer to uIcom's handler function, which is further discussed in §5.3.3. Every connection has a connection ID, which is simply an index into the thread-local list of connections, and reserved at setup function-entry. It is used for connection identification during `uIcom_Send` and `uIcom_Recv` and also determines the UIV the transferred UINTRNotif is locked to, to allow the receiver to uniquely associate each connection with a UIV in the handler. However, this limits the amount of

possible (UINTR-based) uIcom connections to 64, as well as disallowing the receiver to have any other UINTR-based connections outside of the ones managed by uIcom, to avoid unexpected behavior. Once all the capabilities are transferred and set up, the receiver waits for a confirmation from the sender.

Meanwhile, the sender installs the transferred capabilities into its CSpace and VSpace and, if needed, performs addition setup for the transferred UINTRNotif by mapping it into its VSpace and invoking `RequestUITTe` on its `additional_out_cap`. The sender then saves the returned UITT index for the connection. This also leaks the receiver's connection ID to the sender, which might be a security concern if a similar mechanism were to be employed in a real-life application. Another side-effect of this is that the receiver favors connections that were established later, since UID occurs for the highest UIV first, we leave the implications of this to future work.

Once both subconnections are set up and tested, the setup function returns the connection ID to be used for future operation on the connection.

5.3.3 User-Interrupt and Connection Handlers

uIcom uses a custom UIHandler when operating on UINTR-based connections, a slightly modified version of which is available for examination in Listing 5.1. As we can see from this listing, the `in_handler` from our setup is called when receiving a user-interrupt with any `mode` other than `NOMODE` and has two arguments: `data_in` and `data_out`, which determine the location of the input-data and where the handler function should output its data to. An example of a handler function can be observed in Listing 6.2, where we ignore the `data_in` and `data_out` arguments while collecting TSC values, demonstrating some of the possibilities of our approach. When operating in `NOMODE`, the UIHandler exits with `data_received` set, which can then be polled on in `uIcom_Recv(NB)`.

5.3.4 Sending and Receiving

We quickly realized that it was much easier to implement uIcom when viewing signals as null-messages sent via the message-passing mechanism, instead of viewing the message-passing pathway as an extension to signals. We therefore only had to implement four functions, with a few more or less minimal wrappers. These are:

- `void uIcom_Send(uIcom_id_t id, seL4_Word* data, seL4_Word* data_len)`
- `void uIcom_Recv(uIcom_id_t id, seL4_Word* data, seL4_Word* data_len)`

```
1  void UINTR_HANDLER_ATTRIBUTES
2              uIcom_UIHandler(struct __uintr_frame *ui_frame,
3                              unsigned long long vector)
4  {
5      // Vector determines the uIcom ID
6      uIcom_id_t id = vector;
7      // Fetch the connection using the ID
8      uIcom_con_t* con = &connections[id];
9      // Set the data_received bit to use in Poll/Wait
10     con->in.memory->data_received = 1;
11
12     // If no mode is set, return (used for Poll/Wait, etc)
13     if (con->mode == NOMODE) {
14         return;
15     }
16
17     // It's not NOMODE therefore it's either REPLY or AUTO_HANDLE
18     // Both of which execute the handler
19     if (con->handler != NULL) {
20         con->handler(con->in.memory, con->out.memory);
21     }
22
23     // Mark the connection as ready for new data
24     con->in.memory->data_received = 0;
25     con->in.memory->data_sent = 0;
26
27     // If the mode is REPLY, auto-send a reply with no new data
28     // If data is NULL in uIcom_Send, the data is not copied
29     // and data outputted by con->handler(_,_) will be preserved
30     if (con->mode == REPLY) {
31         uIcom_Send(id, NULL, 0);
32     }
33     return;
34 }
```

Listing 5.1: uIcom's UIHandler

- `void uIcom_SendNB(uIcom_id_t id, seL4_Word* data, seL4_Word* data_len)`
- `seL4_Word uIcom_RecvNB(uIcom_id_t id, seL4_Word* data, seL4_Word* data_len)`

`uIcom_RecvNB` returns a `seL4_Word`, which is `1` if data was received and `0` if not.

uIcom_Send

Sending performs the following actions:

- Transfer data to shared memory if `data` $\neq$ `NULL`
- Send signal
- Wait for receive confirmation (`uIcom_Send` only)

As described in §4.2.3, for our blocking `Send`, we wait for a receive-confirmation signal from the receiving function until we exit. Our receive confirmations are sent via a shared variable (`data_sent`), which we take from the previously shared memory page. It is set before the signal is sent (in case of Notification, Endpoint, and UINTRNotif) and cleared by the receiver once the data has been copied to the output buffer. `data_sent` functions as our signal variable when using Frame as the signaling mechanism and is polled on with the `wait_type` set during setup, which is where we decided to make `wait_type` connection-specific instead of call-specific, as originally planned in §4.2.1.

For Endpoint and Notification, we decided to use the next-closest invocations for blocking and non-blocking purposes. Specifically, we used `Call` and Send for Endpoint and `Signal` for Notification. `Call` was chosen as it is required to invoke seL4's *fast-path*[4] but introduced the need to call `Reply` in the receiver. Since Endpoints represent synchronous IPC we cannot use them to implement asynchronous communication. Therefore, we had to use `Send` in `uIcom_SendNB` instead of `NBSend` in order to be able to send any data at all, due to us choosing to use `NBRecv` for Endpoint in `uIcom_RecvNB`.

uIcom_Recv

`uIcom_Recv` performs the reciprocal actions to `uIcom_Send`, namely:

- Poll or Wait for a signal

[4]It was later revealed that the *fast-path* is only provided for same-core communication, however, we kept the use of `Call` to have different mechanisms for both `uIcom_Send` and `uIcom_SendNB`.

```
1  while (con->memory->data_received == 0) {
2      switch(main_con->wait_type) {
3          case POLL: break;
4          case TPAUSE: {
5              RDTSC(seL4_Word temp);
6              TPAUSE(0, temp + TIME_SLICE);
7              break;
8          }
9          case YIELD: {
10             seL4_Yield();
11         }
12     }
13 }
```

Listing 5.2: `uIcom_Recv`: Waiting for signal variables

- Transfer data from shared memory to output data region if `data` $\neq$ `NULL`
- Send confirm signal

UINTRNotif and Frame use a while loop to poll on their signal variables, which can be seen in Listing 5.2. Since our machine was affected by the `UMWAIT` issue [41], we decided to only implement the `TPAUSE` wait-type instead of both `TPAUSE` and `UMWAIT`. For `TPAUSE`, we decided on a `TIME_SLICE` of $1.25\times$ the TSC units polling takes to receive the value for Frame and an infinite time for UINTRNotif, since user-interrupts cause `TPAUSE` to abort. For Notification we use the `Wait` invocation, while we employ `Recv` for Endpoint. Endpoint also has to invoke a Reply if the sender used `Call`, which is determined by an additional word sent with the Endpoint and set to two different magic values, depending on whether the sender used `uIcom_Send` or `uIcom_SendNB`. After the signal has been received, data is transferred to the output data region and the return signal is sent by setting the `data_sent` variable to `0`.

For the non-blocking `uIcom_RecvNB`, UINTRNotif and Frame signals check their respective signal variables `data_received`—which is set in the UIHandler—and `data_sent`—which is set by the sender—once, Endpoint uses `NBRecv` to check for a pending message, and Notification calls `Poll`. If no signals are pending, the function simply returns `0`. Otherwise, it performs the same actions as `uIcom_Recv` after it received a signal.

```
1 void uIcom_RecvHandleReply(uIcom_id_t id) {
2     uIcom_Recv(id, NULL, NULL);
3     uIcom_Handler(id);
4     uIcom_Send(id, NULL, 0);
5 }
```

Listing 5.3: `uIcom_RecvHandleReply` simply wraps the listed functions

5.3.5 Summary

To summarize, we managed to implement the functionality of our design from §4.2. While slight changes were made, such as `wait_type` instead being implemented as connection-specific instead of call-specific, in total our implementation offers signals, message passing, and RPC using our UINTR capabilities in every designed sub-variant. Additionally, uIcom offers similar functionality for additional capabilities as a point for comparison using only four functions and wrappers, an example of which can be seen in Listing 5.3, to provide an appropriate starting point for our evaluation.

Chapter 6

Evaluation

Our initial goals from Chapter 4 were to implement a new IPC facility using UINTR as its basis and compare it to existing methods on seL4 in various scenarios. At the same time, we want to observe behavior of UINTR on our system, given that previous work only examined UINTR on Linux systems. Our implementation allows us to disable our UINTR-related code by simply disabling the corresponding compile flag, which enables us to also observe and analyze the overhead added by our implementation.

In this chapter, we describe the data we collected and derived metrics (§6.1), how we designed and implemented our benchmarks to collect data (§6.2), and finally discuss the results from our benchmarks (§6.3).

6.1 Methodology

Since it is not a given that user-interrupts might perform better from a time perspective than seL4's native IPC pathways, we also evaluate how they fare from a power-management perspective, which was also still unevaluated in any scenario when we conceptualized this thesis.

In this section, we give an overview over what data we collected, how it was collected and determine the metrics we calculate from said data. Namely, we collected data in regards to time characteristics (§6.1.1), energy characteristics (§6.1.2), efficiency indicators (§6.1.3), and further performance indicators (§6.1.4). Most of this data is only available from kernel-mode. We therefore added another capability to seL4—BenchCap—whose invocations simply measure the requested data with a timestamp and then return said data to user-space.

We collected our data on a single Intel(R) Xeon(R) Silver 4410Y with 320 GB of main memory. For our evaluation runs we disabled Intel *SpeedStep*, which should ensure our machine runs at its base frequency of 2.00 GHz and constant

CPU	Intel Xeon Silver 4410Y
Cores	12 (24 Threads)
Frequency	2.0 GHz (max Turbo: 3.9 GHz)
L1	90 KB (32 KB instruction + 48 KB data)
L2	2 MB per core
L3	30 MB (shared)
TDP	150 W
DRAM	4× 16 GB DDR5-4800
Motherboard	Supermicro X13SEI-F
CPU Features:	
Hyperthreading	enabled
Speedstep	disabled
Turboboost	disabled
seL4 Configuration Changes:	
Meltdown Mitigations	disabled

Table 6.1: Description of our evaluation system

```
1 #define RDTSC_BENCH(var) \
2     __asm__ __volatile__("mfence\n"); \
3     __asm__ __volatile__("lfence\n"); \
4     var = __builtin_ia32_rdtsc(); \
5     __asm__ __volatile__("lfence\n");
```

Listing 6.1: The macro used to collect TSC values

voltage. Unless otherwise specified, our evaluation threads run on cores #2 and #4. A more detailed description of our system can be seen in Table 6.1.

6.1.1 Measuring Time

In order to determine the one-way delay (OWD) and round trip time (RTT) of various configurations of uIcom, we used `RDTSC` to read the TSC. The TSC is a constantly-incrementing counter and can be used to determine the time between events on a processor. `RDTSC` is non-serializing and can therefore be reordered around previous and following instructions. We added `LFence` and `MFence` instructions around `RDTSC` to prevent these reorderings, as also used by Kone et al. [48] and recommended by Intel[9, sec. 4.1, RDTSC]. We then insert this `RDTSC_FENCE` macro (see Listing 6.1) around uIcom calls to collect TSC values. TSC data collection is further described in §6.2.

6.1.2 Measuring Energy Consumption

Since we also wanted to evaluate the energy consumption characteristics of UINTR and, by extension, uIcom, we needed to find a way to measure these. Modern Intel processors expose running average power limit (RAPL) MSRs, which can be used to determine these characteristics in various domains, as well as set energy limits [25, sec. 16.10.1]. We used BenchCap to collect RAPL data for four different domains:

- `PP0`, which measures power consumption of the cores [25, sec. 16.10.4]
- `PKG`, which measures power consumption of the entire package
- `DRAM`, which measures power consumption for main memory
- `PLATFORM`, which is vendor-specific and measures power consumption every device that is supplied by the integrated power delivery mechanism

This selection allows us to decently analyze the energy characteristics of our implementation. Unfortunately, our test machine is affected by RAPL filtering, which was introduced to mitigate a hardware vulnerability [66]. We were unable to enable unfiltered values and are therefore limited to only the `DRAM` and `PKG` domains for our evaluation. Nonetheless, we use the gathered data to calculate the following metrics: 1. Total power consumption. 2. Average power consumption over execution time
RAPL values are collected at the start and end of benchmark runs.

6.1.3 Measuring Efficiency

In addition to energy data, Intel also provides counters to evaluate power efficiency with. The `IA32_APERF` MSR counts at the current clock frequency when the processor is in the `C0` power state, while `IA32_MPERF` counts at the reference clock frequency when the processor is in the `C0` power state. When combined, these two can be used to calculate the effective frequency of the processor over a given time frame [67, Table 2-2].

There is also the `IA32_PPERF` counter, which increments at the same frequency as `IA32_APERF`, but only if the current cycle is perceived to have contributed to instruction execution [25, sec. 16.4.5.1]. When combined with `IA32_APERF`, this can be used to determine workload scalability as a metric to determine how frequency changes would affect software performance.
`PERF` counter values are collected at the start and end of benchmark runs.

6.1.4 Further Performance Indicators

We also collected data from three of Intel's fixed performance counters [25, Table 21-1]. They behave as follows:

- `CTR1`: Counts the amount of retired instructions
- `CTR2`: Counts the amount of unhalted clock cycles
- `CTR3`: Counts the amount of reference clock cycles

The counters can be used to calculate the ratio of halted to reference clock cycles—an indicator of how much time was spent in the halt state—and the ratio of retired instructions to reference clock cycles, which we call instruction density.
These miscellaneous indicators are collected at the start and end of benchmark runs.

6.2 Benchmarking

Our benchmarks were initially intended to evaluate all three implemented IPC types, each in every variation of IPC category seen in §2.2.1. The initial plans envisioned us to write a simple benchmark for the signal facility, evaluate different combinations for message passing, and also introduce a "real-life" scenario in the form of an in-memory database to evaluate RPC with, as seen in previous work [59]. Unfortunately, our scope proved to be too ambitious, which forced us to delegate evaluation of our RPC facility to future work. We begin this section by describing how our benchmarks were set up in and how we transferred our data in §6.2.1, after which we further describe our benchmark design in §6.2.2

6.2.1 Setup

Since we are evaluating on seL4, we had to write a custom benchmark facility to handle resources and also find a way to transfer data from our test system to our actual system to process collected data. Setup for our benchmarks is handled by the initial thread, which runs our custom benchmarking facility. Every benchmark had a specific combination of (shared) memory, (shared) capabilities, and functions with initial arguments, which we defined statically with auto-generated definitions. Benchmark threads are created, their capabilities and memory assigned and mapped, and finally started. Once a benchmark finishes execution, its memory and capabilities are freed and ready to be re-used in the next benchmark. Since every benchmark thread has its memory pre-installed,

no page faults can occur during the benchmark runs themselves. If they do, the affected thread terminates.

We were unable to find working device drivers for network devices, mass-storage, or file-systems for seL4. While we considered writing these ourselves, we ultimately settled on using serial over LAN (SoL) for data-transfer, due to seL4's standard library already providing a serial driver. However, the standard driver was not compatible with non-initial threads—presumably because of internal assumptions about CSpace layout that we were unable to reconcile—and we had to introduce a proxy-`printf`, which could be called from benchmarking threads to send data to the initial thread, which would then print this data to serial. Transferring data over SoL was also limited to a maximum of around 12.5KB/s, which drastically reduced the amount of data we could collect in a realistic time-frame. In addition to this, we had to send all of our data twice to have a reasonable chance of ensuring our data would not get corrupted due to transmission errors, halving our effective transfer speed. We were able to confirm that our final data showed no signs of corruption after recombining these duplicates.

6.2.2 Benchmark Design

Our benchmark for signaling performance consists of two threads running simple loops, where with our `RDTSC_BENCH` macro before the sender's `uIcom_Signal` call and another after the receiver's `uIcom_Wait` call, allowing us to measure the OWD of `uIcom_Signal`. We synchronize both threads after the receiver is done measuring their TSC, to ensure our results are as close to the actual OWD as possible. For our "Signal" benchmark, we vary the following factors:

- Capabilities used for uIcom
- `uIcom_Poll` vs `uIcom_Wait`
- Wait-types for `uIcom_Wait`
- Thread affinity of the receiver

If the receiver is configured to call `uIcom_Poll` instead, it does so in a loop and the macro is called after `uIcom_Poll` returns successfully.

Our second benchmark concerns message passing performance and again consists of two threads—here called the producer and consumer instead of sender and receiver—that call `uIcom_SendRecv` and `uIcom_RecvReply` in a loop. We measure TSC timestamps before and after both threads call their respective functions, which allows us to measure the RTT of uIcom. Therefore, we name this benchmark "Roundtrip". Since both `uIcom_RecvReply` use the blocking variant of

```
1 void roundtrip_consumer_auto_handler(uIcom_mem_t *d_in,
2                                       uIcom_mem_t *d_out) {
3     RDTSC_BENCH(seL4_Word temp1);
4     consumer_before_time[consumer_run] = temp1;
5     consumer_run += 1;
6     if (consumer_run >= RUNS) {
7         bench_running = 0;
8     }
9 }
```

Listing 6.2: Connection handler for collecting TSC values

`uIcom_Recv`, we additionally evaluate using the asynchronous-preemptive variants of uIcom with UINTR and OSSeUINTR. This allows us to measure the OWD from producer to consumer in addition to the RTT, with the handler function we used for these cases listed in Listing 6.2. "Roundtrip" also allows us to test various combinations of capabilities in uIcom and might highlight potential "ideal" combinations of IPC pathways. We only vary the capabilities used for uIcom in these benchmarks, to keep the amount of collected data reasonable. For asynchronous-preemptive cases, the program calculates consecutive Fibonacci numbers to demonstrate the scenario of a busy thread occasionally serving requests.

Both our "Signal" and "Roundtrip" benchmarks run for 131 072 iterations before ending. With 68 benchmarks we collected more than 200MB of raw data, which took 15 hours to transfer with SoL.

6.3 Results

In this section we present, analyze, and compare our results from the benchmarks described in the previous sections. We begin by first introducing the time-wise performance of uIcom in various aspects (§6.3.1), continue with power performance in §6.3.2, and evaluate further performance indicators in §6.3.3. Finally, we compare uIcom to previous work in §6.3.4, where we take a look at the "Brdige"-family of IPC mechanisms, previously introduced in §3.3. All of the discussed figures are available in Appendix A, while figures for remaining data are in Appendix B, even if not explicitly mentioned.

6.3.1 Time Performance

Before we begin with our comparison of UINTR-based uIcom to other capabilities, we first compare UINTR and OSSeUINTR under varying affinities and wait

types, while using Frame-based uIcom as a point of reference to characterize these different mechanism.

Varying Affinity

In Figure A.1, we present cumulative distributions of UINTR-, OSSeUINTR, and Frame-based uIcom in the "Signal" benchmark. The receiver uses `uIcom_Recv` with the `Poll` wait type to receive signals. In Figure A.1a, we observe that the OWD is offset by up to 100 TSC units with varying core affinity for regular UINTR. Similarly, OSSeUINTR Figure A.1b have a comparable offset in the 0th to 25th percentile, however, the OWD converges around the 75th percentile. It then slowly tails off in the 95th with a much higher tail latency than can be observed for UINTR. We expect this convergence around the tail to be due to the kernel-entry associated latencies introduced by OSSeUINTR's nature. In contrast to this, Frame-based uIcom, as seen in Figure A.1c, shows very similar behavior to UINTR, with the distribution function having a similar shape among its variants but with a much smaller offset for the different affinities when compared to UINTR.

The graph seen in Figure A.1c is a clear indicator for memory layout being a factor for all of these mechanisms, as UINTR accesses memory to deliver UIPIs, however, since UINTR-based appears to have a much wider offset distribution than Frame-based uIcom, we expect that the second factor is core distance increasing the IPI latency, as seen in previous work [60]. However, due to only collecting data on a limited number of core combinations and a lack of data about the actual internal layout and interrupt routing behavior of our test machine's processor, we cannot draw any definitive conclusions.

Varying Wait Types

Figure A.2 shows the same "Signal" benchmarks under varying wait types for fixed affinities, this time as simple histograms instead of cumulative distribution graphs. We begin our analysis with a focus on UINTR, whose histogram can be observed in Figure A.2a and shows a large peak at approximately $1\,400$ TSC units with three minor peaks in the tail for UINTR. We believe these minor peaks are caused by the stepping warm-up behavior of UINTR, as can be observed in Figure A.3a, which shows a filtered scatter plot of every 17th iteration's data for the `Poll` and `TPause` variants of UINTR-based uIcom. Interestingly enough, Frame-based uIcom also shows this stepping behavior during warm-up, which takes roughly $30\,000$ iterations for UINTR and twice that amount for Frame-based uIcom, as can be seen in Figure A.3b. Presumably some element of caching is causing this, as both the UPID, UITT, and the polled variable live in memory,

which might be moved to different shared cache levels. We leave further analysis of this behavior to future work, which could test and potentially influence this behavior with explicit pre-fetching.

While `TPause` and `Poll` show a similar distribution for UINTR, with `TPause` being shifted by around 350 TSC units, `Yield` shows a much more distributed two-peak structure. Contrasting this to Frame-based uIcom's single peak (seen in Figure A.2c) for `Yield`, we presume this difference might come from two different execution paths in the kernel based on the timing of interrupt delivery and processing. We theorize that the first peak is caused by kernel entries that immediately process the caused IPI in the kernel, while the second tapered-out peak might be caused by kernel re-entries while seL4 is restoring the user context. The function that restores the user-context has a section to recognize pending interrupts and re-enters the kernel if needed, which we hypothesize is the reason for the existence of the second peak.

Alternatively, since a less balanced form of the two-peak structure exists for OSSeUINTR, as can be seen in Figure A.2b, perhaps this pattern is caused by the triggering and processing of a different interrupt[1] in the kernel, which perhaps becomes more likely with an increased OWD, shifting the bulk of the OWD from the first peak to the second. We consider this more likely, as the results for `Yield`ing Frame-based uIcom also appear to have a very minor second peak. Since the sender does not enter the kernel in these cases, we can exclude kernel-lock contention as a possible reason for this distribution.

Overhead Analysis

Before we finally compare different capabilities against each other, we perform an overhead analysis of our implementation by disabling UINTR in the seL4 configuration, which we ensured completely removes any of our UINTR-based modifications. Figure A.4 shows the cumulative distributions for Frame, Endpoint, and Notification-based uIcom, in every configuration that entails a kernel-entry while receiving data. As we can see in Figure A.4c for Notification and Figure A.4a for Endpoint, the added overhead for `uIcom_RecvNB`, which in this case means the added overhead for `Poll` and `NBRecv`, respectively, is similar for both capabilities, shifting the respective distributions by roughly 2 500 TSC units.

Given that `uIcom_Wait` with Frame set to `Yield` only has a curve offset of about 1 000 TSC units, as can be seen in Figure A.4b, we expect this overhead to be a result of `uIcom_RecvNB` being called at least two additional times, with the first call being too early to receive the signal or message sent by the sender, and the second kernel-entry usually receiving it. We believe this to be a plausible explanation

[1] For example a regular timer interrupt

for the doubled offset of the cumulative distribution curve when compared to Frame, with the overhead itself being caused by MSR writes that were added to the kernel entry hook to clear the UINTR-MSRs to prevent erroneous UID, as explained in §5.2.3.

Similarly to `uIcom_RecvNB`, both Endpoint and Notification again show the same behavior for `uIcom_Recv`, which calls `Recv` and `Wait` for the respective capabilities. Since `Wait` is nothing more than a convenience wrapper of `Recv` [42, sec. 10.2.1.10], this is expected. The cumulative distribution shows two steep slopes for the UINTR-disabled case, one around 6 500 TSC units and another around 11 500 TSC units. The entire distribution is offset by around 2 000 TSC units this time, with the first slope significantly reduced in favor of the second. Apart from this, the general shape of the distribution stays the same. This represents a shift from a first peak of the paired histogram distribution to the second. As we have already seen this two-peak structure in Figure A.2, we suppose this might simply be the shape for kernel-entry based IPC mechanisms on seL4 and refer to the previously proposed reasons, in a addition to potential kernel-lock contention, as possible reasons for this distribution shift. While certainly interesting, fully characterizing and analyzing the reasons for the two-peak structure of seL4's system calls is left to future work.

Varying Capabilities - Signal

As a final evaluation of the time aspects of our "Signal" benchmark, we present the best cases in terms of OWD for every capability, which were the UINTR-disabled runs for Notification and Endpoints as discussed above, as both cumulative distributions and cumulative sums in Figure A.5.

First, we take a look at Figure A.5a, which shows the cumulative distributions for the OWD of different capabilities for `uIcom_Recv`. As the scale of the graph is rather large, we refer to previous sections for a detailed view of each capability's distributions. It is quite apparent that the best capability in terms of time for uIcom is the Frame capability with `Poll`, as the bulk of its OWD distribution is around 500 TSC units. This is followed by UINTR with `Poll`, at around 1 300 TSC units, and OSSeUINTR at roughly 3 700. Both the blocking variants for Endpoint and Notification perform rather poorly in this case, both following a two-peak structure at around 6 500 and 11 500 TSC units. This is also reflected in the cumulative sum graph for all of these capabilities, seen in Figure A.5c, with no unexpected switch-up in the order from best to worst in terms of OWD.

In contrast to this, the cumulative distribution for `uIcom_RecvNB`, which is available in Figure A.5b, shows two distinct overlaps for UINTR and Notification-based uIcom, and OSSeUINTR and Endpoint-based uIcom. While Notification offers better performance starting at 1 200 up to the 70th percentile, which

can be observed more closely in Figure A.5e, UINTR's cumulative distribution has a much steeper slope that continues at $1\,300$ and ends below $1\,500$ TSC units up until the 99th percentile, while Notifications tail slowly tapers off around the 80th. Similar behavior can be seen with Endpoint and OSSeUINTR, albeit with a more gradual slope for both and only meeting in the 80th percentile, where Endpoint tapers off, with OSSeUINTR also tail also tapering off by the 97th percentile. At this point we would like to remind the reader that Frame is still far ahead with both a better starting point of about 400 TSC units and a steeper slope than even UINTR.

Nonetheless, while both pairs cross in their cumulative distributions, only UINTR surpasses its counterpart when it comes to the average case, as can be seen by the cumulative sum graph observable in Figure A.5d, offering slightly better OWD performance when compared to Notification-based uIcom, with OSSeUINTR being worse than Endpoint in this regard.

Varying Capabilities - Roundtrip

Unfortunately our data for "Roundtrip" on the UINTR-disabled kernel was unusable due to an error in the benchmark itself that we were only able to correct for the UINTR-enabled case within the time-frame. We therefore cannot reference this data for an overhead analysis, refer to the sections above as an approximation of the potentially observable overhead, and move on to comparing the, to us, most interesting RTT of various capability combinations. Fortunately, none of the best cases—from a time perspective—involve Notification or Endpoint-based uIcom and we deduce from Figure A.4, which shows the cumulative distribution for `uIcom_Recv` for every capability, that even in the UINTR-disabled case none of the overhead-affected configurations would be fast enough to be covered here. However, that is no more than an educated guess.

Figure A.6 shows both the cumulative distributions (Figure A.6a) and histograms (Figure A.6b) for RTT data collected using variants of the "Roundtrip" benchmark. All of the shown variants are eiher Frame-based, UINTR-based, or a combination of the two. The UINTR-based uIcom variants keep their distinct histogram shape with three minor peaks, which indicates the three-step warm-up pattern, while the first peak for every variant is less tall than what was seen in Figure A.2.

Frame-UINTR and *UINTR-Frame* follow the same distribution, while both of the asynchronous-preemptive combinations of UINTR-based and UINTR and Frame-based uIcom perform worse than their polling counterparts, which goes against our initial assumption that asynchronous-preemptive cases do not have the polling overhead and therefore would perform marginally better than the synchronous counterparts. Perhaps this is caused by the `uIcom_Recv` already

being near the `uIcom_Send` code-path in `uIcom_RecvReply`, which would allow the processor to pre-fetch some instructions or perform other optimizations, whereas the asynchronous-preemptive case is preempted from its calculation of Fibonacci numbers, where the processor cannot expect to have to call the `uIcom_Send` function in advance.

We believe this to also be a potential performance indicator for the unexamined asynchronous-preemptive RPC case, which would, as it is based on the same combination of functions, also perform worse than in the synchronous case. However, the trade-off of having slightly higher RTT but being able to perform background work is interesting, might be required in some applications, and is worth considering in future work.

6.3.2 Power Performance

Presumably due to RAPL filtering[2], we were only able to observe two of four intended domains, which were `PKG` and `DRAM`. Graphs for the total power consumption and average power consumption over execution time (represented by TSC differences), can be seen in Figure A.7. While the distribution of average power consumption over time

$$e_{avg} = \frac{\Delta Power}{\Delta TSC}$$

does show a pattern of memory-based signaling having a consistently lower values, for all other variants of the "Signal" benchmark there appears to be no clear correlation between any of the varying factors for neither the `PKG` nor the `DRAM` domain, as can be seen in Figures A.7c and A.7d. We therefore do not believe that our collected energy consumption data shows any significant results besides more time taken $\rightarrow$ more power consumed. The same applies to RAPL data collected for the "Roundtrip" series of benchmarks, which we do not fully discuss here but are available in Appendix B. Perhaps these results could display more meaningful correlations if *SpeedStep* and *Turboboost* were enabled, however, we leave the evaluation of this possibility to future work.

As other work has previously achieved a roughly 20% increase in power-efficiency by using `TPAUSE` opposed to "pure" polling in other applications [56], we present two possible explanations for these results:

1. Disabling *SpeedStep* resulted in 24 threads running at the same frequency and power consumption and simply drowned any power-efficiency gains provided by one thread occasionally lowering its power state.

[2]Or perhaps `PLATFORM` is simply missing in addition to `PP0` being filtered

2. Unlikely but possible benchmarking issues, either in design or in implementation

However, we do believe TPAUSE successfully transitions into a sleep state, due to the increase in OWD observable in Figures A.2a and A.2b.

6.3.3 Further Performance Indicators

Below we describe and analyze our results based on further performance indicators, such as the efficiency metrics from §6.1.3 and §6.1.4.

Instruction Density

Instruction density d, whose formula is

$$d = \frac{\Delta\text{Retired Instructions}}{\Delta\text{TSC Values}}$$

can be seen in Figure A.8a for different configurations of the "Signal" benchmarks. There appears to be a clear correlation between the wait type, used capability, and instruction density, with the graph showing that, regardless of used capability, `uIcom_Recv` with `TPause` resulted in a very low instruction density, which is further evidence for an entered sleep state. In contrast to this, benchmark variants that rely on `Poll` have a very high instruction density, as expected. Also of note are the lowered instruction density of variants relying on kernel-entry for waiting, such as any `uIcom_Recv` variant with `Yield` and both variations of uIcom with Endpoint and Notification, which have a much higher instruction density on the UINTR-disabled runs that can be seen in Figure A.8b. We believe this to be an indicator of the high cycle cost per WRMSR and RDMSR instructions that were introduced due to the issues encountered in §5.2.3.

We would expect to also see some reflection of instruction density in the collected data for energy consumption. The absence of such a correlation[3] is presumed to either be the cause of the reasons already mentioned in §6.3.2, or the different instructions that were executed in the low instruction density variants simply consumed more energy. However, we believe the second reason to be unlikely, since we expect the best and worst performing Frame-based variants to execute the same instructions except for TPAUSE. Unless TPAUSE has an absurd power cost, which related work does not mention and in fact does seem to show the opposite results [56], we believe this to, again, simply be an issue with our

[3]The color pattern observable in Figure A.8a does not correspond to any pattern observed in Figure A.7 in any way

RAPL data. At this point it would also be interesting to analyze various types of NOP-like instructions, such PAUSE instead of TPAUSE. We leave this to future work.

Figure A.8 also shows that uIcom_RecvNB has a consistently lower instruction density than uIcom_Recv with Poll. This is expected due to uIcom_RecvNB being called in a loop, which checks the polled variable once, while uIcom_Recv checks the polled variable in a loop directly. For variants with uIcom_RecvNB, the JMP instruction of the Poll-loop is effectively switched out for CALL instruction, which we presume has a higher cycle cost due to higher instruction complexity [9, sec. 3.3, CALL, JMP]. However, further measurements would be required to fully confirm this assumption.

Unhalted Cycles and Effective Frequency

To confirm that disabling *SpeedStep* did indeed ensure our machine runs at its base frequency, we calculate the unhalted cycle ratio u as follows:

$$u = \frac{\Delta\text{Unhalted Cycles}}{\Delta\text{Reference Cycles}}$$

And compare it to the effective frequency derived from IA32_APERF and IA32_MPERF in Figure A.9. If our machine were to run in an unhalted state at the base frequency, we would expect both of these values to be at $100\,\%$ for every benchmark variant. Unfortunately, the effective frequency for Endpoint and Notification-based uIcom is around 80–$95\,\%$, as can be seen in Figure A.9a, which indicates the processor had a lower effective frequency than expected. This can be explained by the ratio of unhalted cycles to reference cycles seen in Figure A.9b, whose entries with lower than $100\,\%$ match the entries from Figure A.9a.

We expect this is due to seL4 running its idle thread when no other thread is currently ready to be scheduled, which simply consists of a HLT instruction, halting the processor until an interrupt arrives. Threads that invoke the Recv and Wait methods of Endpoint and Notifications are unable to be scheduled until a message or signal has arrived. Therefore this anomaly is not an indicator of P-State shifting despite *SpeedStep* begin disabled but instead intended behavior. Similar behavior of effective frequency and the unhalted cycle ratio were observed for the remaining benchmark variants and are available to view in Appendix B.

Workload Scalability

Finally, we analyze the hardware-perceived scalability of uIcom. As mentioned in §6.1.3, workload scalability s is calculated as follows:

$$s = \frac{\Delta\texttt{IA32\_PPERF}}{\Delta\texttt{IA32\_APERF}}$$

And is the ratio of cycles that have contributed to instruction execution to unhalted cycles. It is effectively a measurement of how much of the passed time was spent actually performing work, with the assumption that this is a metric that can be used to assess the scalability of a workload with increased clock speeds.

In the "Signal" benchmark we first examine the workload scalability for the sender, which is visible in Figure A.10a. Is is ordered as follows: Frame → Endpoint and Notification → UINTR→OSSeUINTR, with a higher workload scalability for the `Yield` and `TPause` variants for each. On the receiver side, whose graph is shown in Figure A.10b, the scalability values are also mostly grouped by used capability, with the `Yield` wait type variants instead offer the lowest scalability within the respective variant groups. `TPause` variants boast $5-20\%$ higher scalability than the variant norms, with UINTR with `TPause` having the highest workload scalability of roughly 85%, with the next highest being a plateau of UINTR with `Poll`.

The variants using `uIcom_Poll` appear to have a lower scalability than the blocking `uIcom_Wait` variants. We expect the increase in workload scalability for `TPause` is due to a decrease in `IA32_APERF` counting frequency when entering the sleep state. However, unless `IA32_MPERF` frequency also has a proportional decrease, which it cannot [25, sec. 21.7.2], this should also be reflected in the effective frequency, which, as can be seen in Figure A.9, it is not. We leave the further analysis of the effects of `TPAUSE` on hardware-perceived workload scalability and the discussion of the remaining benchmark variants to future work.

6.3.4 Comparison to Related Work

In this section we compare uIcom to different IPC libraries from previous work in terms of relative and total time performance differences where possible.

SkyBridge

As already mentioned in §3.3.1, SkyBridge uses EPT switching via the `VMFUNC` instruction to bypass the kernel and directly call code from other processes as a form of PPC. By its nature as a PPC mechanism, SkyBridge does not provide a mechanism for the cross-core case [59] and is therefore difficult to compare to uIcom, which focuses exclusively on this. For the same-core case, SkyBridge provides a speed-up of around $3\times$ on seL4, at around 400 TSC units for its RTT, which is around $6-7\times$ better RTT performance than uIcom using UINTR. However, comparing TSC units between systems must be done with caution and may have little practical implications, as these units can have wildly different meanings depending both on clock-speed and underlying architecture.

UnderBridge and HyBridge

Both UnderBridge and HyBridge move system servers into the kernel to avoid IPC overhead in cross-server communication and therefore do not offer a new user-space IPC pathway directly. In addition to this, our source for HyBridge does not contain any data for RTT and instead focuses on throughput [7], as their primary metric for comparisons. However, their throughput metrics suggest equivalent or only slightly better performance than UnderBridge, which does contain a table for RTT times [6]. UnderBridge has a RTT performance of around 800 TSC units [6], which is roughly $3\times$ better than uIcom with UINTR.

All in all, comparing uIcom to these existing technologies is tricky, due to differences in testing system, methodology, and benchmarks. Nonetheless, our work appears to be the only one to improve the cross-core case for communication between active IPC partners and can offer potential OWD speed-ups of $1.1 - 5.5\times$ when compared to existing seL4 capabilities, excepting shared memory, with similar results assumed for the RTT case. We anticipate future work will shed more light on the differences and similarities to other technologies by evaluating them and UINTR on similar applications now that a base-line has been established.

Chapter 7

Conclusion

For our final chapter, we first draw our final conclusions in §7.1 and ultimately give suggestions for avenues for future work in §7.2.

7.1 Conclusion

Our initial goal was to design, implement, and evaluate a new IPC library that uses modern processor features on a modern representative of the L4 μkernel family. We chose seL4 as this representative and were able to be, to our knowledge, the first to produce a design for a capability-based integration of UINTR—a kernel-bypass mechanism for sending and receiving interrupts—on any OS. UINTR, in our opinion, suits itself quite well to a capability-based management and posit that this design could be used to add UINTR to other capability-based μkernels, such as *Fiasco*, in the future. To use this new capability-based design in practice, we also designed uIcom, an IPC library that provides functions for every type of IPC, with synchronous, asynchronous, and asynchronous-preemptive variants. Where reasonably applicable, we integrated the new user-wait extension into our design.

We were able to implement both our capability design and the IPC library on a recent version of seL4, albeit with some changes and additions. While implementing the UINTR capabilities, we conceptualized and implemented support for OSSeUINTR on seL4, which allows the kernel to have some control over UIPI delivery, while still bypassing the kernel from a sender's perspective. By modifying our design for uIcom, we were able to implement it to be able to use any of the seL4 IPC-capable capabilities as the underlying mechanism, which allowed for easier comparison of the UINTR mechanism to the existing suite.

In our evaluation, we characterize aspects of uIcom in terms of time and hardware efficiency indicator. We show that the UINTR-based approach is faster than

the existing suite of seL4 IPC capabilities in both the blocking and non-blocking cross-core case and also offered preliminary insights into time characteristics of OSSeUINTR. The overhead added to the pre-existing IPC capabilities by our implementation is significant, however, we believe future work will be able to greatly reduce this, which we will again discuss in §7.2. While we were unfortunately unable to offer any insights into energy efficiency, which we initially believed to be one of the key advantages of UINTR when compared to shared-variable-based polling, data for workload scalability and instruction density from a hardware perspective showed that UINTR is perceived to be much more scalable than its counterparts and also that `TPAUSE` drastically decreases instruction density, albeit doubling the OWD of uIcom. We believe that both metrics might be indicators of potential gains in energy efficiency.

Except for the energy measurements, our results were within the expected frame set by previous work and we believe to have shown multiple new avenues for future research to take in regard to seL4, UINTR, and the user-wait extension, while presenting UINTR as a viable alternative for user-space IPC in seL4, with $1.1 - 5.5\times$ speed-ups for OWD.

While asynchronous-preemptive UINTR with active background work does not perform as well from a RTT perspective as UINTR that is actively being waited for, we believe the existence of a fast user-space asynchronous-preemptive communication pathway can offer many new possibilities for both IPC and software design, with the background work vs. speed trade-off certainly being worth considering.

7.2 Future Work

Finally, we describe avenues for potential future work on seL4, UINTR, the user-wait extension, uIcom and our implementation of capability-based UINTR. We begin with a list of suggestions on how to expand the implemented designs in §7.2.1 and conclude this thesis with suggestions for further evaluations §7.2.2

7.2.1 Expanding uIntercom

We see four immediate avenues for expanding the work presented in this thesis. For one, we did not implement checks for the rights we added to UINTRNotif and UIPICap. Future work could implement these and implement different scenarios in which these rights are used, as well as (a) analyze the design of these rights and (b) potentially expand on them, to then apply these to real world applications. In the spirit of real-life applicability, future work could also try to minimize the overhead added by our implementation of capability-based UINTR. We believe

that, with the recent implementation of an RFC related to XState-management on seL4 to allow for eager XState restoration [68], future work could port or re-implement our design to a newer version of seL4 and examine how these changes affect general latencies for kernel entry, as well as the added overhead with `XSAVES`-managed UINTR when compared to the current implementation. We believe uIcom could be expanded with more wait-types to test different kinds of `NOP`-like instructions, as already mentioned in §6.3.3. Research on more modern platforms might be able to integrate `UMONITOR/UMWAIT` without the potential for performance degradation as well. Perhaps different combinations of these wait-types might provide a substantial difference in energy efficiency, which would have to be analyzed. Finally, while we believe UINTRNotif should in theory already be able to receive hardware interrupts, confirming this support, adding a mechanism for interacting with devices, and exploring new systems for bypassing self-UIPIs are all potential tasks for future work wanting to evaluate whether UINTR could sensibly replace Notification polling for IRQ handling.

7.2.2 Expanded Evaluation

For further evaluation of uIcom, we enumerate the following new paths: First, our RPC facility is not evaluated at all and, while the "Roundtrip" benchmark did test the functions under 0-length messages, the message passing facility is, in our opinion, in further need of evaluation. In both of these cases metrics like total throughput, among others, are a crucial indicators of IPC and server performance, which have yet to be measured or assessed. In addition to this, re-evaluating uIcom with *SpeedStep* and/or *Turboboost* enabled might offer a different perspective into the power efficiency of our implementation, which we were unfortunately unable to infer much about in §6.3.2. Furthermore, UINTR itself is still under-evaluated under varying clock speeds or different combinations of both E- and P-Cores for the different IPC partners, which was not possible on our platform. We believe that all of these factors deserve further examination, both on our existing test system, as well as on other systems to also uncover micro-architectural influences in all metrics.

In conclusion, seL4's two-peak structure for system calls, the stepping warm-up behavior of Frame and UINTRNotif on our machine, the effects of `TPAUSE` and `UMONITOR` on hardware-perceived workload scalability, OSSeUINTR on Linux, and finally expanding our UINTR capabilities to receive hardware interrupts on seL4, are all research avenues that are of high interest to us and we believe make great material for future work.

Bibliography

[1] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger, "The duality of memory and communication in the implementation of a multiprocessor operating system," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 63–76, 1987-11, ISSN: 0163-5980. DOI: 10.1145/37499.37507. Accessed: 2025-06-11. [Online]. Available: https://dl.acm.org/doi/10.1145/37499.37507.

[2] J. Liedtke, "On micro-kernel construction," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95, New York, NY, USA: Association for Computing Machinery, 1995-12, pp. 237–250, ISBN: 978-0-89791-715-5. DOI: 10.1145/224056.224075. Accessed: 2025-05-27. [Online]. Available: https://dl.acm.org/doi/10.1145/224056.224075.

[3] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time Protection: The Missing OS Abstraction," en, in *Proceedings of the Fourteenth EuroSys Conference 2019*, Dresden Germany: ACM, 2019-03, pp. 1–17, ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303976. Accessed: 2025-05-29. [Online]. Available: https://dl.acm.org/doi/10.1145/3302424.3303976.

[4] G. Klein et al., "seL4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, 2010-06, ISSN: 0001-0782. DOI: 10.1145/1743546.1743574. Accessed: 2025-05-14. [Online]. Available: https://dl.acm.org/doi/10.1145/1743546.1743574.

[5] S. A. G. University of Karlsruhe, *L4Ka:Pistachio Microkernel white paper*, 2003-05. [Online]. Available: http://l4ka.org/l4ka/pistachio-whitepaper.pdf.

[6] J. Gu et al., "Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication," en, 2020, pp. 401–417, ISBN: 978-1-939133-14-4. Accessed: 2025-05-03. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/gu.

[7] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "{EPK}: Scalable and Efficient Memory Protection Keys," en, 2022, pp. 609–624. Accessed: 2025-05-03. [On-

line]. Available: https://www.usenix.org/conference/atc22/presentation/gu-jinyu.

[8] J. Liedtke, "Toward real microkernels," *Commun. ACM*, vol. 39, no. 9, pp. 70–77, 1996-09, ISSN: 0001-0782. DOI: 10.1145/234215.234473. Accessed: 2025-05-27. [Online]. Available: https://dl.acm.org/doi/10.1145/234215.234473.

[9] *Intel® 64 and IA-32 Architectures Software Developers Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, en, 2025-03. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671110.

[10] A. S. Tanenbaum and H. Bos, *Modern operating systems*, en, 4. ed. Boston: Prentice Hall, 2015, ISBN: 978-0-13-359162-0 978-1-292-06142-9.

[11] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*, 1.00. Arpaci-Dusseau Books, 2018-08.

[12] G. Heiser, "The seL4 Microkernel – An Introduction," en, 2025-01.

[13] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, ser. SOSP '93, New York, NY, USA: Association for Computing Machinery, 1993-12, pp. 175–188, ISBN: 978-0-89791-632-5. DOI: 10.1145/168619.168633. Accessed: 2025-05-02. [Online]. Available: https://dl.acm.org/doi/10.1145/168619.168633.

[14] M. Hohmuth and H. Hartig, "Pragmatic nonblocking synchronization for real-time systems," en, [Online]. Available: https://os.inf.tu-dresden.de/~hohmuth/prj/usenix2001.pdf.

[15] J. Liedtke et al., *The L4Ka Vision*, 2001-04. [Online]. Available: http://l4ka.org/publications/.

[16] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, New York, NY, USA: Association for Computing Machinery, 2013-11, pp. 133–150, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522720. Accessed: 2025-05-15. [Online]. Available: https://dl.acm.org/doi/10.1145/2517349.2522720.

[17] M. Hohmuth, H. Tews, and S. G. Stephens, "Applying source-code verification to a microkernel: The VFiasco project," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10, New York, NY, USA: Association for Computing Machinery, 2002-07, pp. 165–169, ISBN: 978-1-4503-7806-2. DOI: 10.1145/1133373.1133405. Accessed: 2025-06-02. [Online]. Available: https://dl.acm.org/doi/10.1145/1133373.1133405.

[18] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty, "Running the manual: An approach to high-assurance microkernel development," in *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, ser. Haskell '06, New York, NY, USA: Association for Computing Machinery, 2006-09, pp. 60–71, ISBN: 978-1-59593-489-5. DOI: 10.1145/1159842.1159850. Accessed: 2025-06-02. [Online]. Available: https://dl.acm.org/doi/10.1145/1159842.1159850.

[19] M. Paturel, I. Subasinghe, and G. Heiser, "First steps in verifying the seL4 Core Platform," in *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '23, New York, NY, USA: Association for Computing Machinery, 2023-08, pp. 9–15, ISBN: 979-8-4007-0305-8. DOI: 10.1145/3609510.3609821. Accessed: 2025-06-02. [Online]. Available: https://dl.acm.org/doi/10.1145/3609510.3609821.

[20] jwmsft, *SendMessage function (winuser.h) - Win32 apps*, en-us. Accessed: 2025-09-26. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendmessage.

[21] *Pipe(2) - Linux manual page*. Accessed: 2025-09-26. [Online]. Available: https://www.man7.org/linux/man-pages/man2/pipe.2.html.

[22] *Signal(7) - Linux manual page*. Accessed: 2025-09-26. [Online]. Available: https://www.man7.org/linux/man-pages/man7/signal.7.html.

[23] A. Silberschatz, P. B. Galvin, and G. Gagne, "Operating System Concepts,"

[24] B. Gamsa, O. Krieger, and M. Stumm, "Optimizing IPC Performance for Shared-Memory Multiprocessors," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1, 1994-08, pp. 208–211. DOI: 10.1109/ICPP.1994.144. Accessed: 2025-05-16. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4115718.

[25] *Intel® 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, en, 2025-03. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671447.

[26] P. Mejia-Alvarez, L. E. Leyva-del-Foyo, and A. Diaz-Ramirez, *Interrupt Handling Schemes in Operating Systems* (SpringerBriefs in Computer Science), en. Cham: Springer International Publishing, 2018, ISBN: 978-3-319-94492-0 978-3-319-94493-7. Accessed: 2025-06-11. [Online]. Available: http://link.springer.com/10.1007/978-3-319-94493-7.

[27] *Archive repository · intel/uintr-linux-kernel@0ee776b*. Accessed: 2025-06-04. [Online]. Available: https://github.com/intel/uintr-linux-kernel/commit/0ee776bd38532358159013ed0188693b34c46cf5.

[28] *The Userspace I/O HOWTO — The Linux Kernel documentation*. Accessed: 2025-06-14. [Online]. Available: https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html.

[29] L. Torvalds, *Linux Kernel*, 2025. Accessed: 2025-09-28. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v6.14&id=38fec10eb60d687e30c8c6b5420d86e8149f7557.

[30] M. Parker, "A case for user-level interrupts," *SIGARCH Comput. Archit. News*, vol. 30, no. 3, pp. 17–18, 2002-06, ISSN: 0163-5964. DOI: 10.1145/571666.571675. Accessed: 2025-05-26. [Online]. Available: https://dl.acm.org/doi/10.1145/571666.571675.

[31] K. Asanović, *Presentation on Risc-V Interrupts*, 2016-07.

[32] *Remove N extension chapter for now · riscv/riscv-isa-manual@b6cade0*, en. Accessed: 2025-06-02. [Online]. Available: https://github.com/riscv/riscv-isa-manual/commit/b6cade07034d39e65134a879a5c3369d50e0df0e.

[33] S. Pinto and C. Garlati, "A Must for Securing Embedded Systems," en,

[34] *Intel® Architecture Instruction Set Extensions and Future Features*, 2025-03. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671368.

[35] *[RFC PATCH 00/13] x86 User Interrupts support*. Accessed: 2025-05-26. [Online]. Available: https://lore.kernel.org/lkml/20210913200132.3396598-1-sohil.mehta@intel.com/T/#m0a43e921ae1e8e6aa11b8a51380ef2ff3a87fb4a.

[36] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, en, 2025-06. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671436.

[37] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell, *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*, 2025-12. Accessed: 2025-10-01. [Online]. Available: https://gitlab.com/x86-psABIs/x86-64-ABI.

[38] Y. Jia, K. Tian, Y. You, Y. Chen, and K. Chen, "Skyloft: A General High-Efficient Scheduling Framework in User Space," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24, New York, NY, USA: Association for Computing Machinery, 2024-11, pp. 265–279, ISBN: 979-8-4007-1251-7. DOI: 10.1145/3694715.3695973. Accessed: 2025-04-27. [Online]. Available: https://dl.acm.org/doi/10.1145/3694715.3695973.

[39] C. A. Thekkath and H. M. Levy, "Hardware and software support for efficient exception handling," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 110–119, 1994-11, ISSN: 0163-5980. DOI: 10.1145/381792.195515. Accessed: 2025-05-26. [Online]. Available: https://dl.acm.org/doi/10.1145/381792.195515.

[40] *Re: [RFC PATCH 00/13] x86 User Interrupts support - Chrisma Pakha*. Accessed: 2025-05-18. [Online]. Available: https://lore.kernel.org/all/3d8d8dd7-deb4-f5c4-c7c5-e1d5972c71f4@andrew.cmu.edu/.

[41] *MONITOR and UMONITOR Performance Guidance*, en. Accessed: 2025-09-28. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/monitor-umonitor-performance-guidance.html.

[42] *seL4 Reference Manual Version 13.0.0*. Accessed: 2025-07-13. [Online]. Available: http://sel4.systems/Info/Docs/seL4-manual-13.0.0.pdf.

[43] *IPC | seL4 docs*. Accessed: 2025-10-09. [Online]. Available: https://docs.sel4.systems/Tutorials/ipc.html.

[44] C. Goedefroit, "Interruptions en espace utilisateur pour le réseau BXI," fr, other, Université de bordeaux, 2023-08. Accessed: 2025-05-03. [Online]. Available: https://inria.hal.science/hal-04693787.

[45] F. Rauscher and D. Gruss, "Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24, New York, NY, USA: Association for Computing Machinery, 2024-12, pp. 94–108, ISBN: 979-8-4007-0636-3. DOI: 10.1145/3658644.3690242. Accessed: 2025-04-27. [Online]. Available: https://dl.acm.org/doi/10.1145/3658644.3690242.

[46] C. Zhang et al., "Erebor: A Drop-In Sandbox Solution for Private Data Processing in Untrusted Confidential Virtual Machines," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25, New York, NY, USA: Association for Computing Machinery, 2025-03, pp. 1210–1228, ISBN: 979-8-4007-1196-1. DOI: 10.1145/3689031.3717464. Accessed: 2025-05-21. [Online]. Available: https://dl.acm.org/doi/10.1145/3689031.3717464.

[47] B. Aydogmus et al., "Extended User Interrupts (xUI): Fast and Flexible Notification without Polling," en, in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, Rotterdam Netherlands: ACM, 2025-03, pp. 373–389, ISBN: 979-8-4007-1079-7. DOI: 10.1145/3676641.3716028. Accessed:

2025-04-27. [Online]. Available: https://dl.acm.org/doi/10.1145/3676641.3716028.

[48] Y. Kone, L. Duval, R. Lachaize, P. Felber, D. Hagimont, and A. Tchana, "Understanding Intel User Interrupts," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 9, no. 2, 40:1–40:32, 2025-06. DOI: 10.1145/3727132. Accessed: 2025-06-04. [Online]. Available: https://dl.acm.org/doi/10.1145/3727132.

[49] Y. Li et al., "LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling," en, in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Edinburgh, United Kingdom: IEEE, 2024-03, pp. 922–936, ISBN: 979-8-3503-9313-2. DOI: 10.1109/HPCA57654.2024.00075. Accessed: 2025-04-27. [Online]. Available: https://ieeexplore.ieee.org/document/10476424/.

[50] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 345–360, ISBN: 978-1-931971-49-2. Accessed: 2025-06-10.

[51] J. Fried et al., "Making Kernel Bypass Practical for the Cloud with Junction," en, 2024, pp. 55–73, ISBN: 978-1-939133-39-7. Accessed: 2025-04-27. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/fried.

[52] *Re: [RFC PATCH 00/13] x86 User Interrupts support - Sohil Mehta*. Accessed: 2025-06-04. [Online]. Available: https://lore.kernel.org/lkml/7c1038de-0bae-3b87-d4e4-8a30a910ebdd@intel.com/.

[53] J. Lin, Y. Chen, S. Gao, and Y. Lu, "Fast Core Scheduling with Userspace Process Abstraction," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24, New York, NY, USA: Association for Computing Machinery, 2024-11, pp. 280–295, ISBN: 979-8-4007-1251-7. DOI: 10.1145/3694715.3695976. Accessed: 2025-05-02. [Online]. Available: https://dl.acm.org/doi/10.1145/3694715.3695976.

[54] L. Guo, D. Zuberi, T. Garfinkel, and A. Ousterhout, "The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling," en, 2025-04, pp. 1015–1032, ISBN: 978-1-939133-46-5. Accessed: 2025-04-27. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/guo.

[55] K. Huang, J. Zhou, Z. Zhao, D. Xie, and T. Wang, "Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt?" en, vol. 3, no. 3, 2025-06.

[56] E. Li et al., "SPDK+: Low Latency or High Power Efficiency? We Take Both," in *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '25, New York, NY, USA: Association for Computing Machinery, 2025-07, pp. 17–23, ISBN: 979-8-4007-1947-9. DOI: 10.1145/3736548.3737824. Accessed: 2025-07-10. [Online]. Available: https://doi.org/10.1145/3736548.3737824.

[57] C. Goedefroit, A. Denis, M. Barbe, B. Goglin, and G. Pichon, "Communication Notification through User-Level Interrupts for the BXI Network," en, 2025-09. Accessed: 2025-09-18. [Online]. Available: https://inria.hal.science/hal-05150209.

[58] C. Li et al., "Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25, New York, NY, USA: Association for Computing Machinery, 2025-10, pp. 479–495, ISBN: 979-8-4007-1870-0. DOI: 10.1145/3731569.3764816. Accessed: 2025-10-13. [Online]. Available: https://doi.org/10.1145/3731569.3764816.

[59] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels," in *Proceedings of the Fourteenth EuroSys Conference 2019*, Dresden Germany: ACM, 2019-03-25, pp. 1–15, ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303946. Accessed: 2025-05-14. [Online]. Available: https://dl.acm.org/doi/10.1145/3302424.3303946.

[60] S. Smith et al., "Draft: Have you checked your IPC performance lately?" en,

[61] C. Liu, L. Luo, M. Li, P. Lei, L. Chen, and K. Xiao, "Inter-Core Communication Mechanisms for Microkernel Operating System based on Signal Transmission and Shared Memory," in *2021 7th International Symposium on System and Software Reliability (ISSSR)*, 2021-09, pp. 188–197. DOI: 10.1109/ISSSR53171.2021.00031. Accessed: 2025-05-02. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9626422.

[62] *Intel/uintr-linux-kernel*, 2025-09. Accessed: 2025-10-08. [Online]. Available: https://github.com/intel/uintr-linux-kernel.

[63] *[v8,1/7] KVM: CPUID: Fix IA32_xss support in CPUID(0xd,i) enumeration - Patchwork*. Accessed: 2025-10-09. [Online]. Available: https://patchwork.kernel.org/project/kvm/patch/20191101085222.27997-2-weijiang.yang@intel.com/.

[64] *seL4Test | seL4 docs*. Accessed: 2025-10-07. [Online]. Available: https://docs.sel4.systems/projects/sel4test/.

[65] *The seL4 Bitfield Generator | seL4 docs*. Accessed: 2025-10-07. [Online]. Available: https://docs.sel4.systems/projects/sel4/bfgen.html.

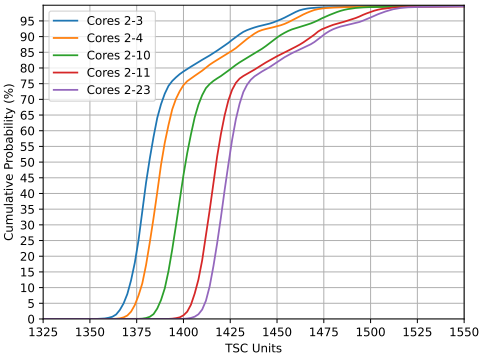
[66] *Running Average Power Limit Energy Reporting CVE-2020-8694,...* Accessed: 2025-10-11. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[67] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers*, en, 2025-06. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671098.

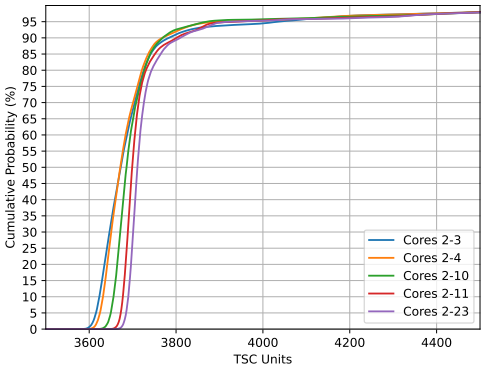
[68] seL4, *Sel4 rfc-18: Fpu context switching*, en. Accessed: 2025-09-27. [Online]. Available: https://github.com/seL4/rfcs/blob/746b98b282c280545cf85d3efae98aab1155e7cf/src/implemented/0180-fpu-switching.md.

Appendix A

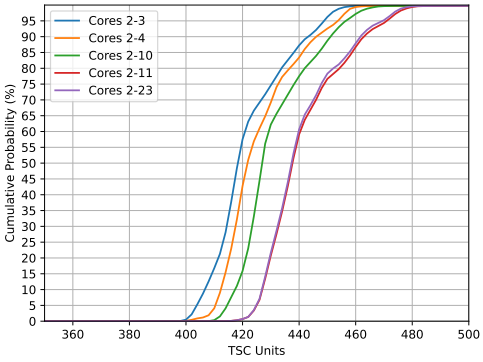
Discussed Data



(a) Results for UINTR

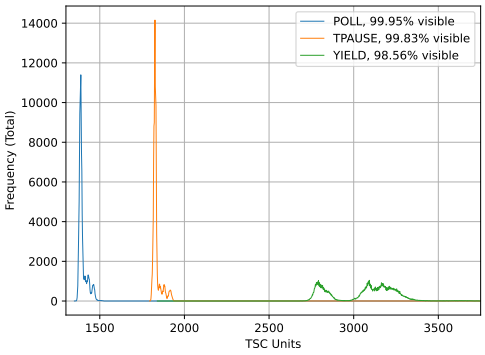


(b) Results for OSSeUINTR

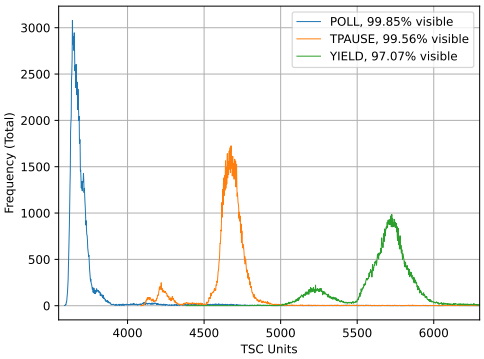


(c) Results for Frame

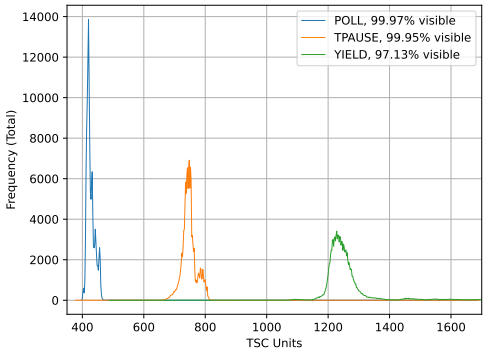
Figure A.1: Cumulative distributions for OWD under varying affinities for UINTRNotif-based uIcom and Frame-based uIcom



(a) Results for UINTR

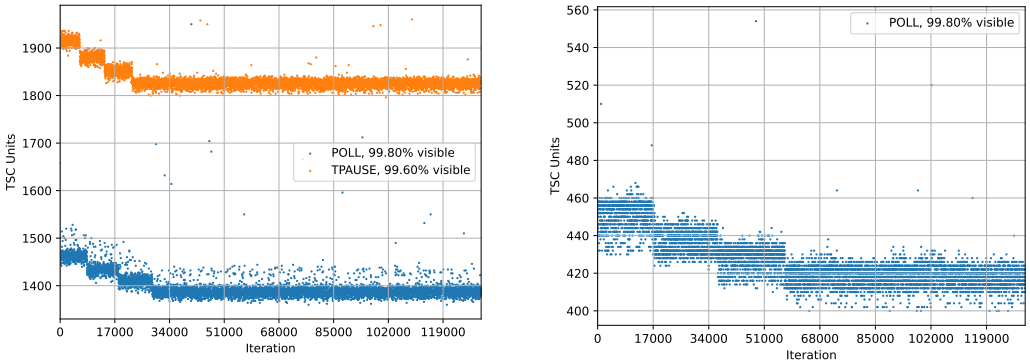


(b) Results for OSSeUINTR



(c) Results for Frame

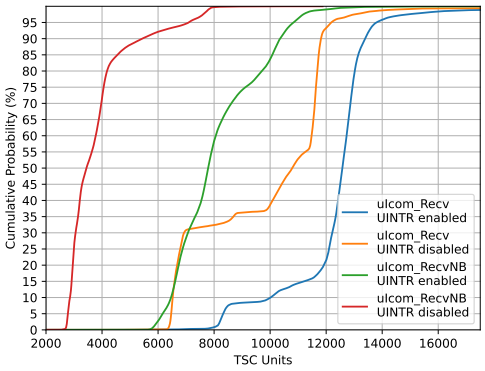
Figure A.2: Histograms for OWD under varying wait-types for UINTRNotif-based uIcom and Frame-based uIcom



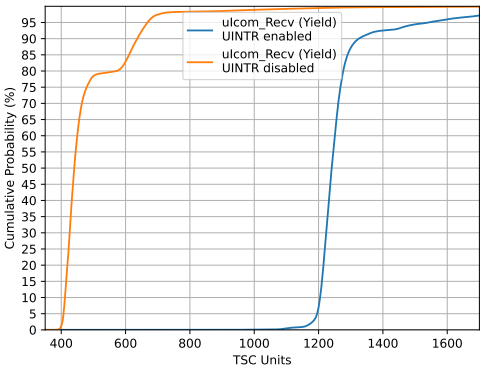
(a) Results for UINTR

(b) Results for Frame

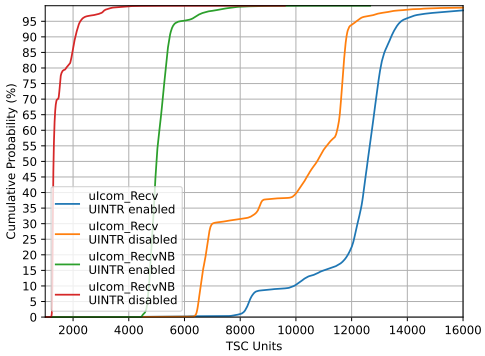
Figure A.3: Scatter plot for UINTR and Frame-based uIcom that shows the stepping warm-up behavior. Filtered to results within shown region, with only every 17th result shown to decrease plot complexity



(a) Overhead for Endpoint

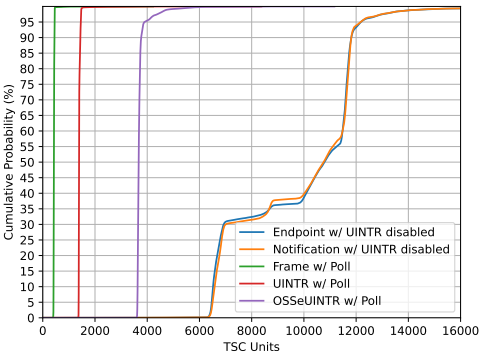


(b) Overhead for Frame

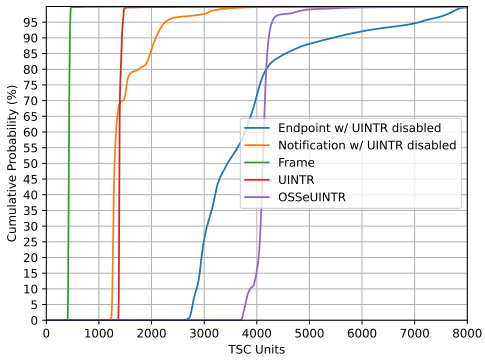


(c) Overhead for Notification

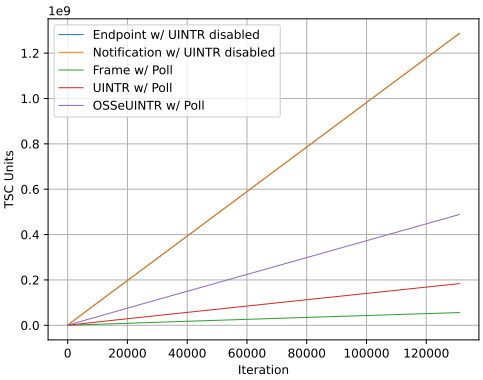
Figure A.4: OWD overhead for every non-UINTR-based uIcom variant that relies on kernel entry. Other variants are unaffected and therefore not shown



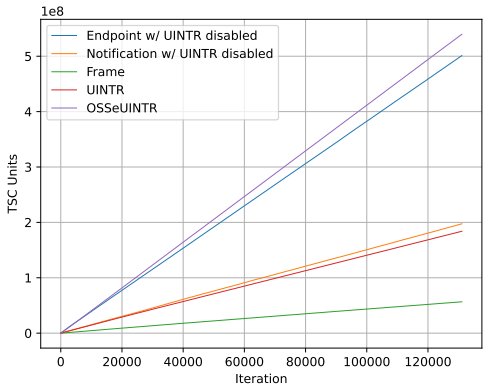
(a) Cumulative distribution for `uIcom_Recv`



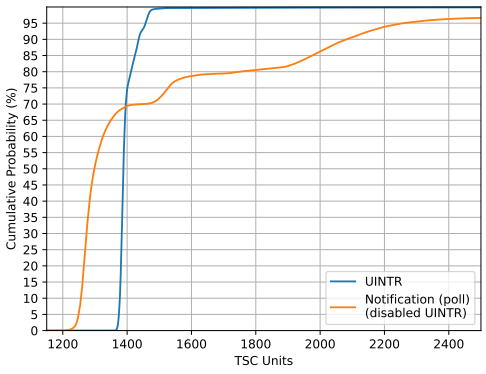
(b) Cumulative distribution for `uIcom_RecvNB`



(c) Cumulative sum for `uIcom_Recv`

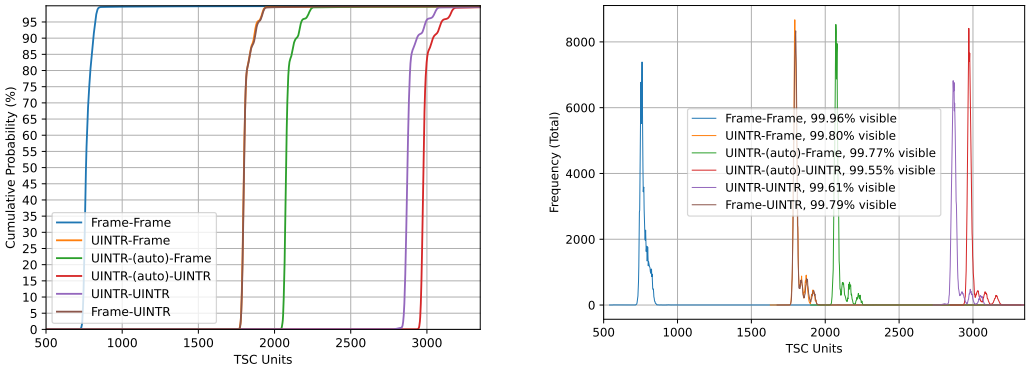


(d) Cumulative sum for `uIcom_RecvNB`



(e) Detailed view of the cumulative distribution for UINTR and Notification for `uIcom_RecvNB`

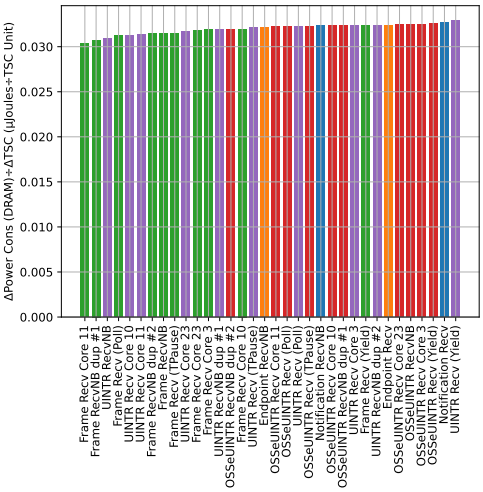
Figure A.5: The best cases for every capability uIcom can be configured with for both `uIcom_RecvNB` and `uIcom_Recv`



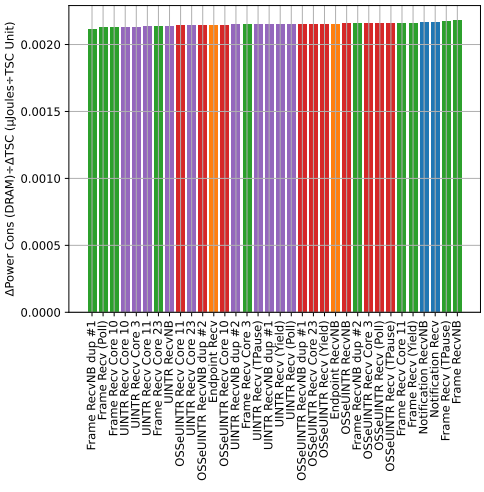
(a) Cumulative distribution

(b) Histogram

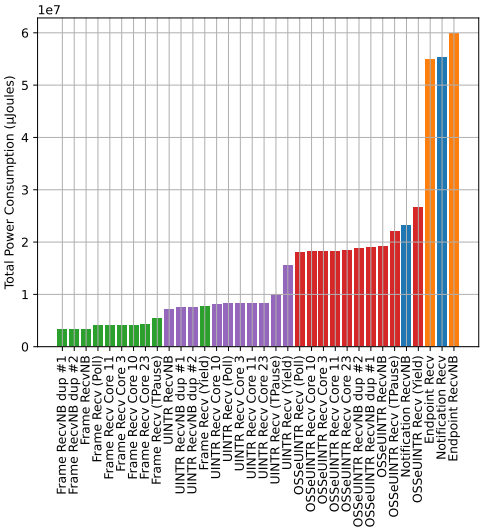
Figure A.6: Views of the fastest RTTs of the “Roundtrip” benchmark



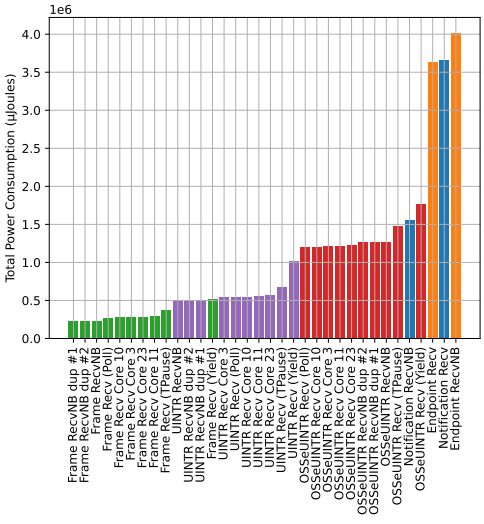
(a) $\frac{\Delta Power}{\Delta TSC}$ (PKG)



(b) $\frac{\Delta Power}{\Delta TSC}$ (DRAM)

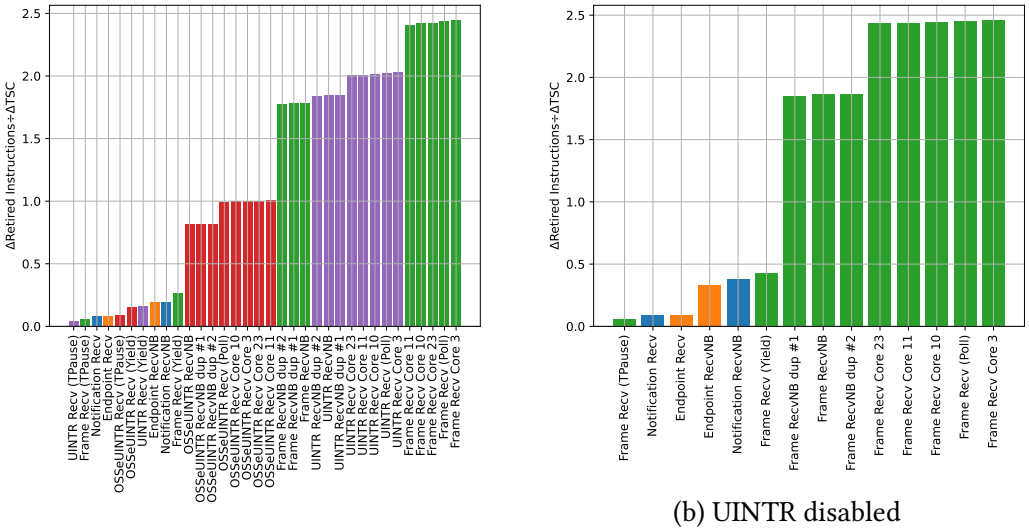


(c) Total Power Consumption (PKG)



(d) Total Power Consumption (DRAM)

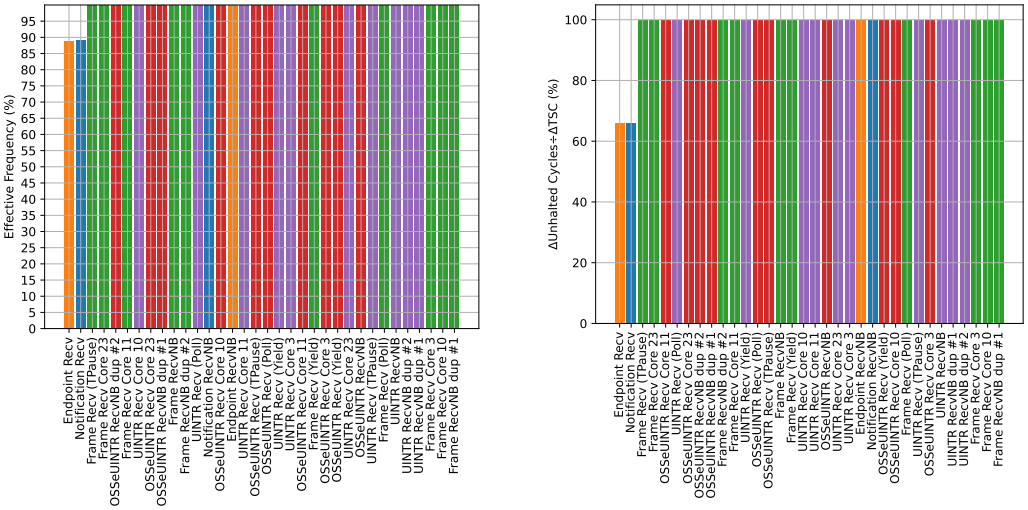
Figure A.7: RAPL data for all the "Signal" benchmarks. Bars are color-coded to group different IPC pathways together



(a) UINTR enabled

(b) UINTR disabled

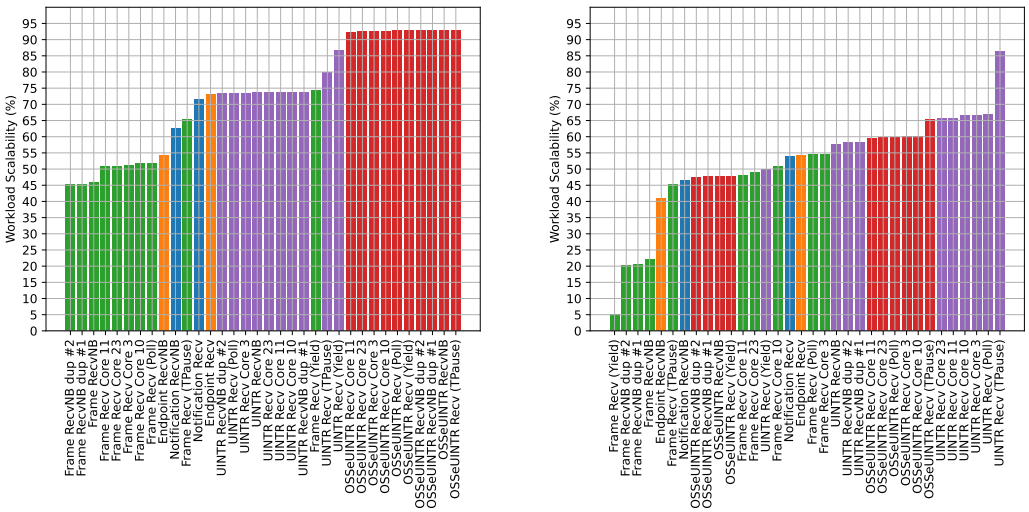
Figure A.8: Instruction density for varying receiver configurations for “Signal”



(a) Effective Frequency

(b) Ratio of Unhalted Cycles to Reference Cycles

Figure A.9: Effective frequency and ratio of unhalted to reference cycles for the Receiver in the “Signal” benchmark

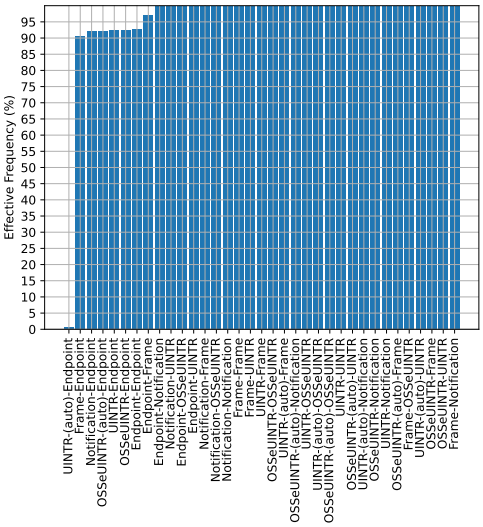


(a) Sender (b) Receiver

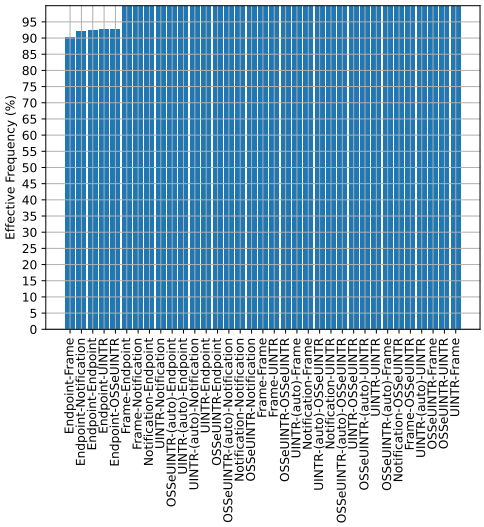
Figure A.10: View of the workload scalability ($\frac{\Delta \text{PPERF}}{\Delta \text{APERF}}$) for every benchmark run

Appendix B

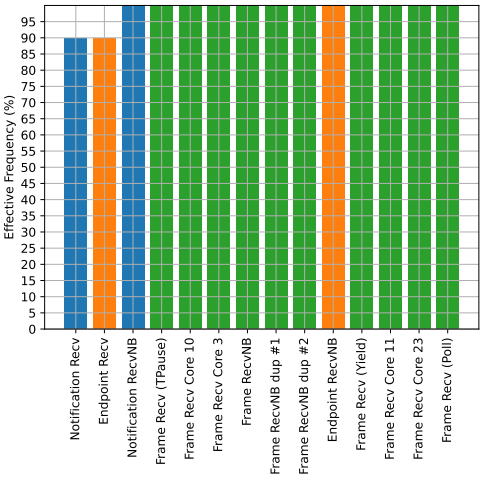
Further Data



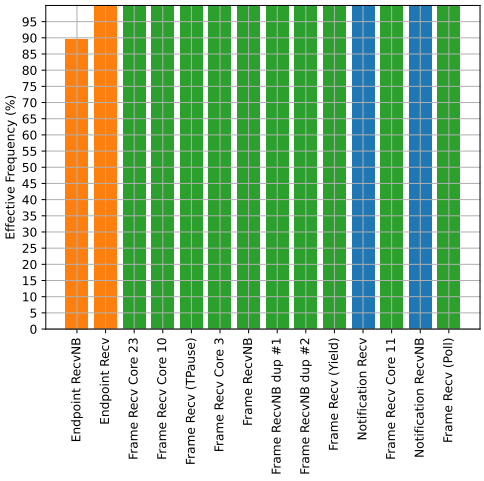
(a) “Roundtrip” Consumer



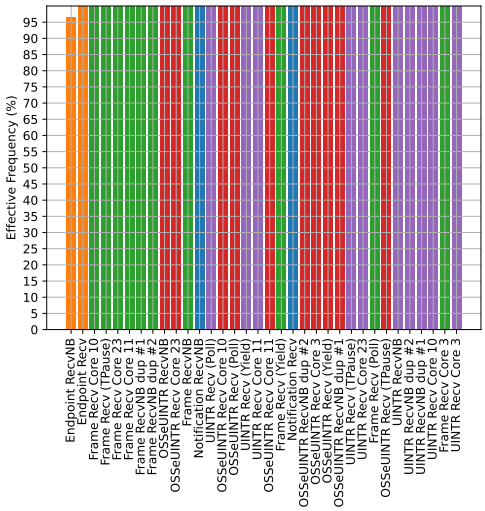
(b) “Roundtrip” Producer



(c) “Signal” Receiver (UINTR disabled)

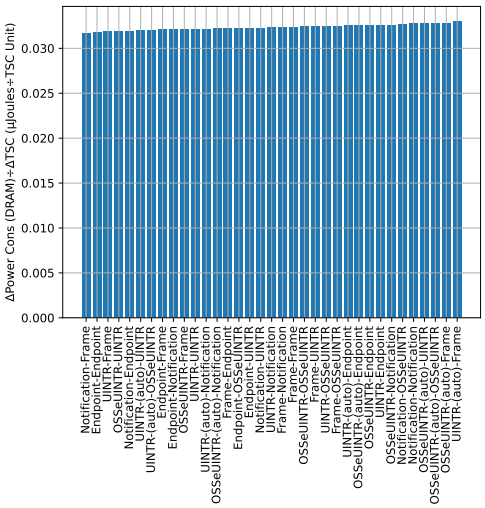


(d) “Signal” Sender (UINTR disabled)

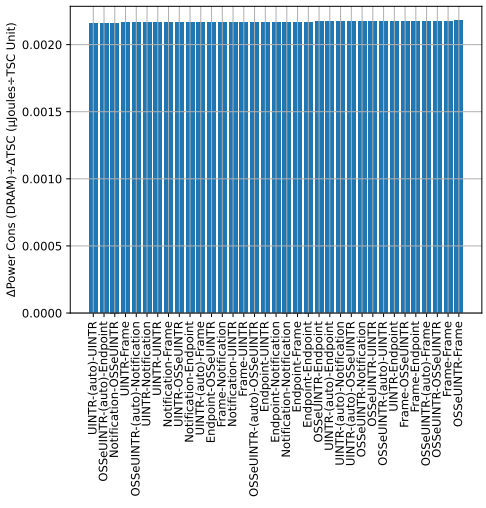


(e) “Signal” Sender

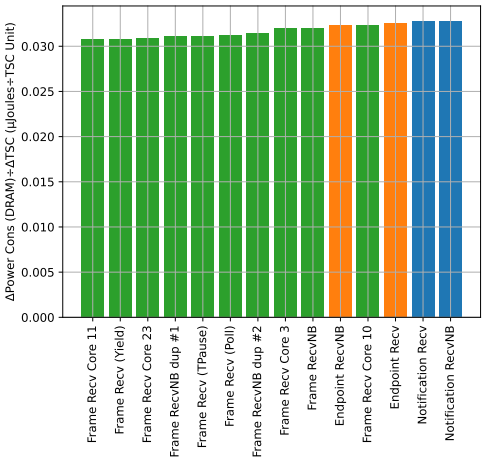
Figure B.1: Remaining data for effective frequency



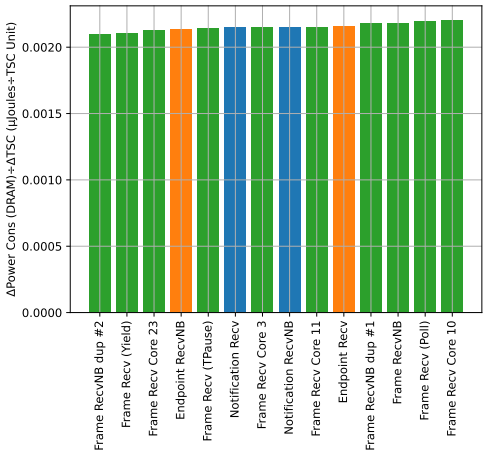
(a) “Roundtrip” (PKG)



(b) “Roundtrip” (DRAM)

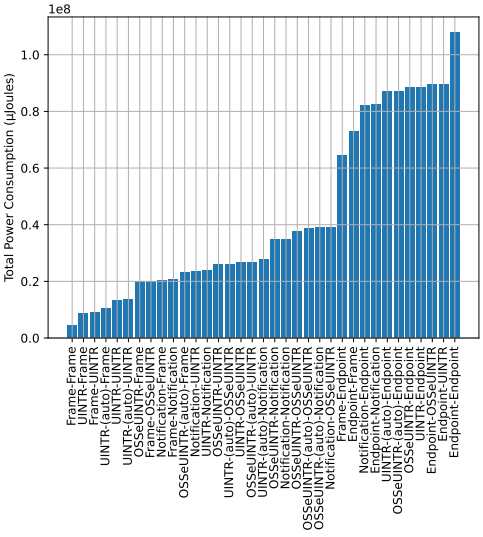


(c) “Signal” (UINTR disabled) (PKG)

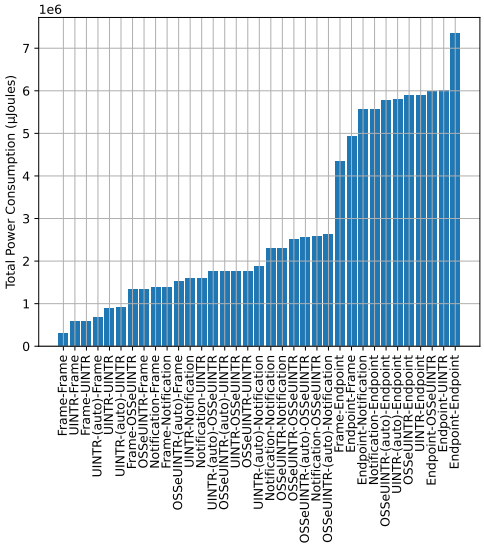


(d) “Signal” (UINTR disabled) (DRAM)

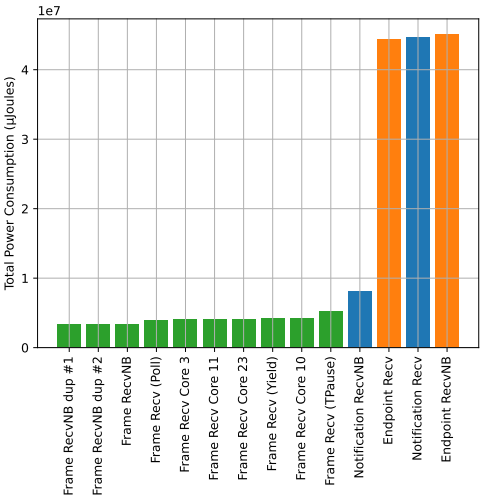
Figure B.2: Remaining data for $\frac{\Delta Power}{\Delta TSC}$



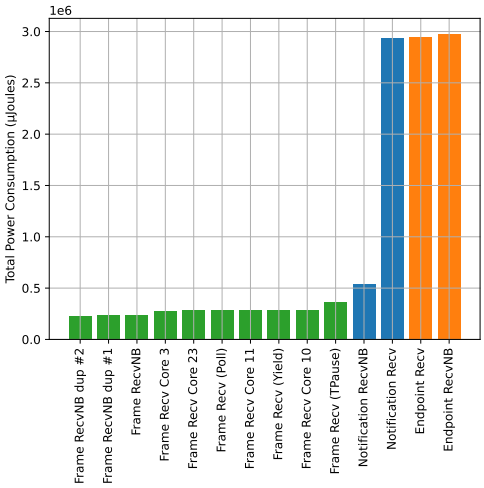
(a) “Roundtrip” (PKG)



(b) “Roundtrip” (DRAM)

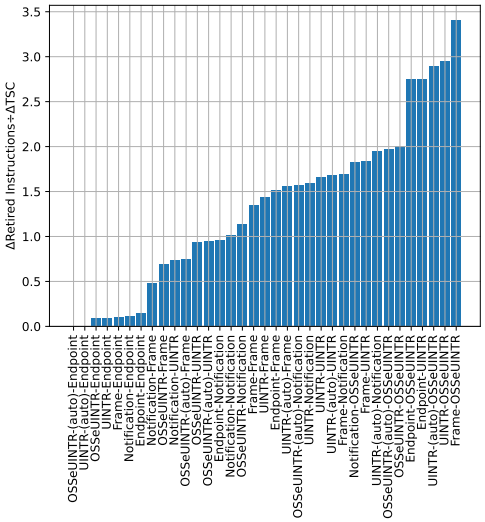


(c) “Signal” (UINTR disabled) (PKG)

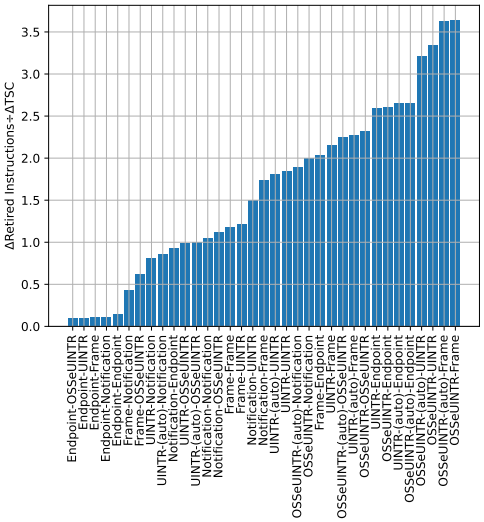


(d) “Signal” (UINTR disabled) (DRAM)

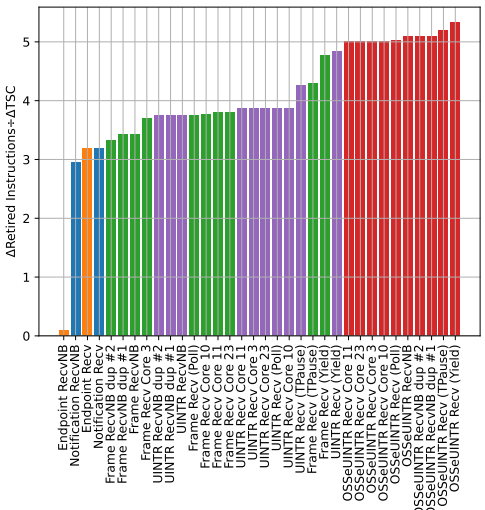
Figure B.3: Remaining data for total power consumption



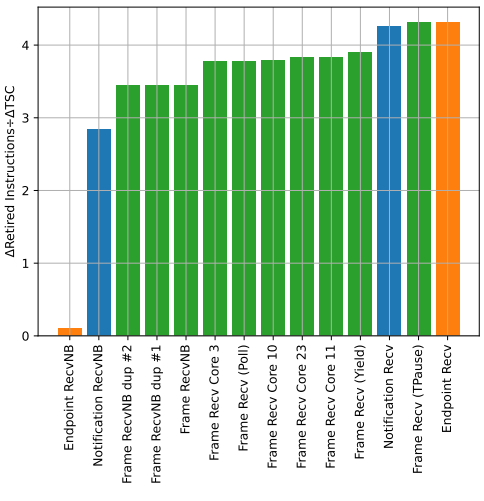
(a) “Roundtrip” Consumer



(b) “Roundtrip” Producer

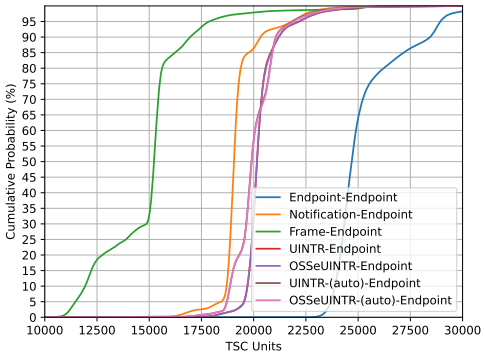


(c) “Signal” Sender

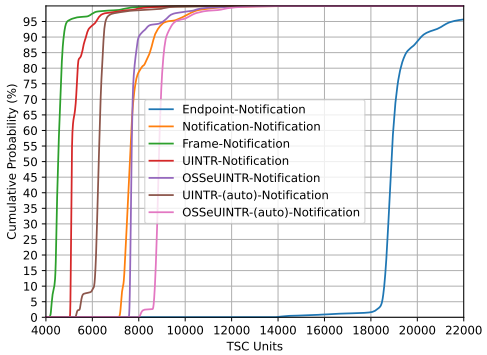


(d) “Signal” Sender (UINTR disabled)

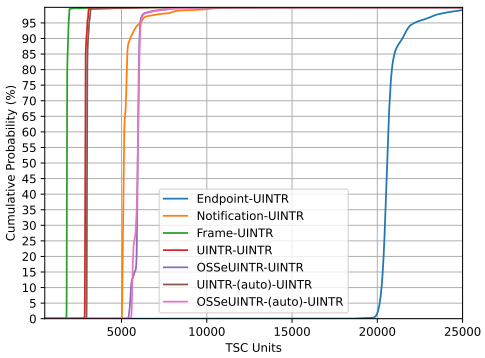
Figure B.4: Remaining data for instruction density



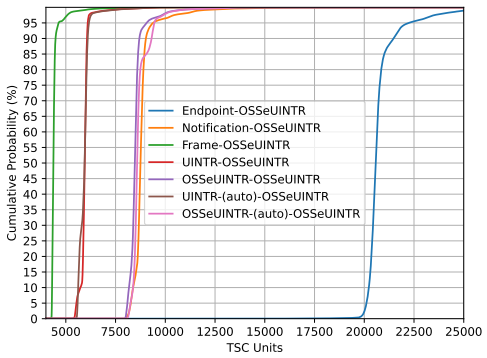
(a) Producer: Endpoint



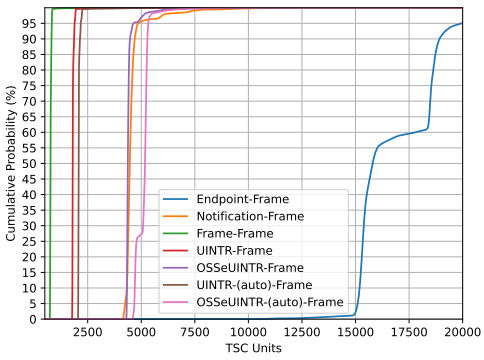
(b) Producer: Notification



(c) Producer: UINTR

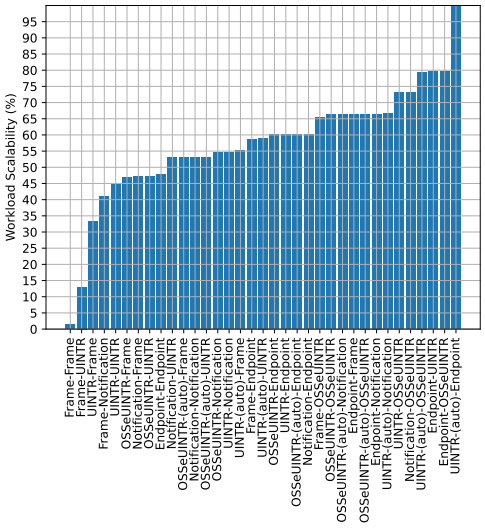


(d) Producer: OSSeUINTR

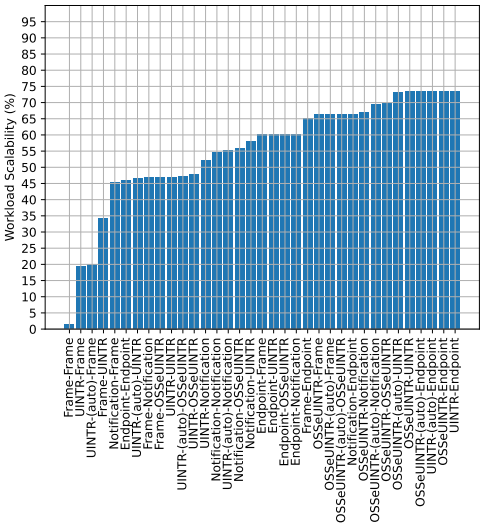


(e) Producer: Frame

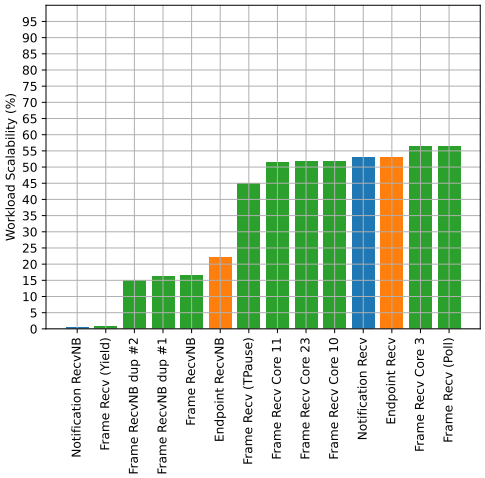
Figure B.5: Remaining data for "Roundtrip" RTTs. Grouped by capability used for producer's *in*-subconnection



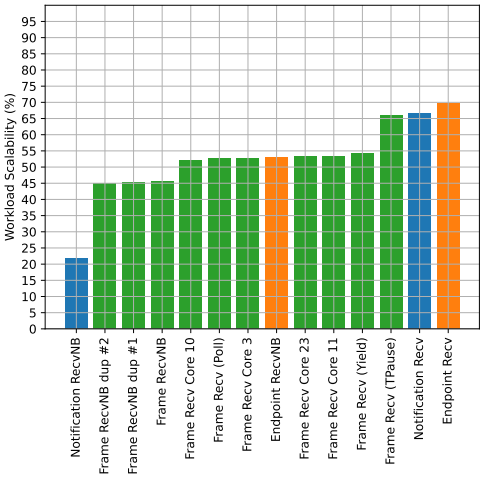
(a) “Roundtrip” Consumer



(b) “Roundtrip” Producer

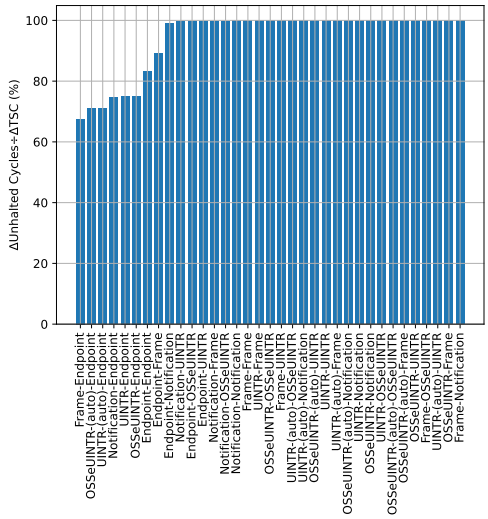


(c) “Signal” Receiver (UINTR disabled)

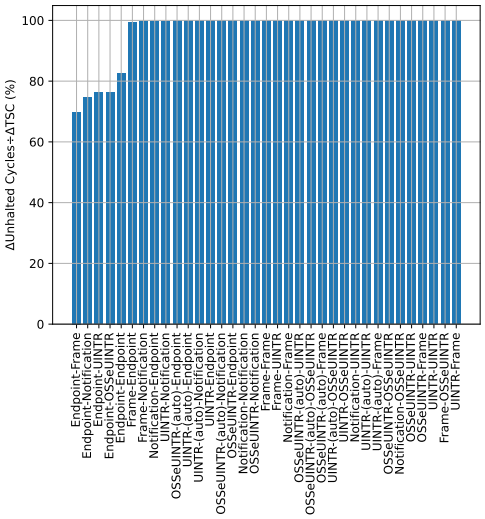


(d) “Signal” Sender(UINTR disabled)

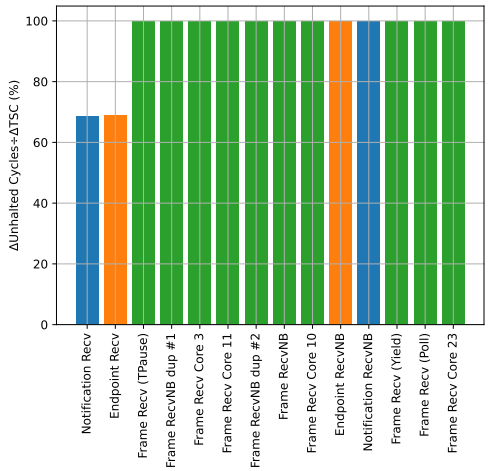
Figure B.6: Remaining data for workload scalability



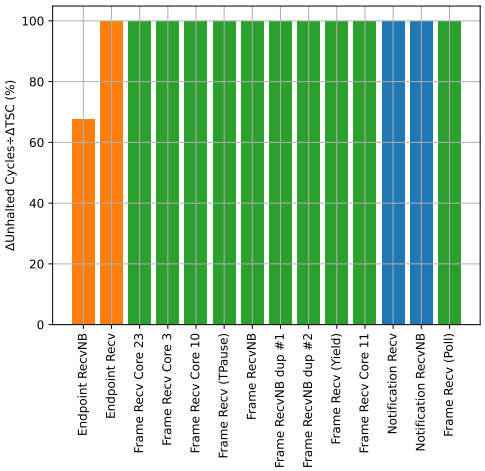
(a) “Roundtrip” Consumer



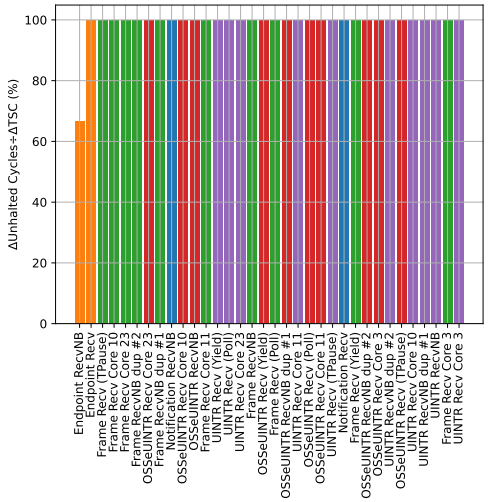
(b) “Roundtrip” Producer



(c) “Signal” Receiver (UINTR disabled)



(d) “Signal” Sender (UINTR disabled)



(e) “Signal” Sender

Figure B.7: Remaining data for ratio of unhalted cycles to reference cycles

Appendix C

Glossary

advanced programmable interrupt controller (APIC) 12, 15, 17

application binary interface (ABI) How functions are called and how their parameters are passed. 9, 16

application programming interface (API) Set of available functions with parameter definitions that can be called by an application. 9

binary rewriting Technique by which a compiled binary is scanned for forbidden instructions and dynamically rewritten at load-time to replace or remove the offending instructions. 34

capability pointer (CPtr) seL4-specific data type, which describes how a specific capability is stored in the thread's CSpace. Implemented as a simple word. 22–26, 41

capability space (CSpace) The set of capabilities available to a thread in seL4. The CSpace root is the CNode capability located in the thread's TCB. 23, 25, 55, 56, 65, 109

extended page table (EPT) Intel's hardware virtualization uses EPTs to map physical addresses of the guest, to physical addresses of the host. Combined with the guest page table this maps guest virtual addresses to host physical addresses. The EPT can be switched using `VMFUNC` by the guest from a list configured by the host. 34, 35, 74, 109

extended protection key (EPK) Mechanism that combines EPTs and MPKs to provide more memory protection domains [7]. 35

extended state (XState) 19, 48, 79

extended user-interrupts (xUI) 30

extensible markup language (XML) A common markup language 49

general protection fault (GP fault) A type of fault, that can be raised a variety of different reasons. Used as a catch-all for access-violations without dedicated faults [25, sec. 7.15 - Interrupt 13]. 19

input/output (I/O) 12, 13, 33, 114

interprocess communication (IPC) v, 5, 7–11, 13, 21, 25, 27, 29, 34–37, 41, 44, 47, 54, 58, 61, 64, 66, 69, 74, 75, 77–79, 96

interprocessor interrupt (IPI) An interrupt that was triggered by a different processor. 12, 15, 17, 19, 30, 36, 51, 67, 68, 110

interrupt request line (IRQ) Hardware mechanism by which devices send interrupts to the processor. Sometimes used to denote system dealing with this mechanism. 27, 79

interrupt service routine (ISR) Routine that is called by a processor on the receipt of an interrupt with the associated IV. 12, 110

interrupt vector (IV) An integer between 0-255, which is associated with an interrupt. The interrupt vector determines how the processor handles the interrupt by being the index into a table used to determine the address of the ISR. 12, 13, 15–17, 20, 52, 110, 112

IOAPIC 12, 27, 51

IOMMU 33

IPI virtualization (IPIv) Intel hardware feature that allows IPIs to be processed by the underlying hardware on virtualized systems. 30, 31

kernel-based virtual machine (KVM) Kernel-based virtualization environment that allows Linux to act as a hypervisor. 47, 48

memory protection key (MPK) Intel x86/64 feature from 2019, introduces memory protection keys to the page table, allowing up to 16 different isolated memory domains on the same page table. Applications can switch between MPKs using 2 new user-space instructions, which can enable or disable read/write access to these memory domains. 5, 32, 35, 109

microkernel (µkernel) A minimal kernel with only basic functionality and a small trusted codebase optimized for speed and/or security. v, 5, 7–9, 11, 13, 21, 29, 34, 36, 37, 47, 48, 77

mixed-criticality system (MCS) An extention of seL4's main kernel which provides *scheduling contexts* and cpu-time-based scheduling 21

model-specific register (MSR) A set of special registers that hold a variety of processor functions (debugging, features, power-management) [25, Chapter 11]. Can only be written to and read by spececial priviledged instructions. 14–17, 19–21, 38, 39, 47–49, 51, 52, 63, 69

monolithic kernel v, 7, 8, 13

MSI 27

MSI-X 33

non-uniform memory access (NUMA) Computer memory design in which different logical processors are connected to different parts of memory, affecting memory access time depending on the physical location of the core. 29, 36

one-way delay (OWD) Time taken for a message to be sent and received. 62, 65–70, 72, 75, 78, 90, 91, 93

operating system (OS) 5, 7–10, 12, 14, 20, 21, 36, 77

operating system signal (OSS) 10, 13, 29–33, 37, 52, 54, 55, 66–70, 74, 77–79, 90, 91, 106, 111

OSS-emulating UINTR (OSSeUINTR) 52, 54, 55, 66–70, 74, 77–79, 90, 91, 106

PCIe 33

posted interrupt descriptor (PID) A structure used for interrupt virtualization. 33

protected procedure call (PPC) Form of RPC on systems with passive servers.In this case the clients thread switches to the server's address space and executes the procedure itself, similar to a library call. 11, 74

quick emulator (QEMU) "A generic and open source machine emulator and virtualizer"-Qemu Website 47–49

remote procedure call (RPC) 11, 37, 41, 44, 45, 54, 60, 64, 71, 79, 111

round trip time (RTT) Time taken for a message to be sent, received, and confirmed. 62, 65, 66, 70, 71, 74, 75, 78, 95, 106

running average power limit (RAPL) Interface for reporting accumulated power consumption of various domains 63, 71, 73, 96

seL4 v, 7, 21–23, 25–27, 38, 42, 44–47, 49–52, 54, 58, 61, 62, 64, 65, 68, 69, 73–75, 77–79, 109, 111, 114

serial over LAN (SoL) 65, 66

thread control block (TCB) Datastructure that holds per-thread control data. 9, 27, 39, 51, 55, 109, 114

time stamp counter (TSC) Register on Intel's x86/64 platform which counts the number of cycles since the last reset. 21, 31, 56, 59, 62, 65–71, 74, 75

trusted codebase (TCB) 8, 37

uIntercom (uIcom) v, 41, 43, 46, 54–57, 60, 62, 63, 65–70, 72–75, 77–79, 90–94

user interprocessor interrupt (UIPI) An interprocessor interrupt that was triggered by the `SENDUIPI` instruction. 14, 17, 20, 29–34, 39, 41, 43, 51, 52, 67, 77, 79, 113

user posted-interrupt descriptor (UPID) Thread-specific descriptor for user-interrupts, used by a receiver to hold a state and senders to determine target and used IV. 15, 17, 20, 32–34, 38, 39, 41, 50–52, 67, 113

user-defined interrupt (UDI) List of interrupt vectors (integers between 32-255) that don't have a architecture defined cause 12, 30, 51

user-interrupt (UINTR) Name of the architectural feature which allows user processes (software operating with CPL=3) to receive and process interrupts and send send user interprocessor interrupts to negotiated targets. v, 5, 7, 12–17, 19, 20, 29–34, 37–39, 41, 44, 47–50, 52–56, 60, 61, 63, 66–70, 72, 74, 75, 77–79, 90–94, 97, 102–108, 111, 112

user-interrupt delivery (UID) Process by which a recognized UINTR is delivered to user-space. Calls the interrupt handler. 14–19, 32, 33, 50, 56, 69, 113

user-interrupt flag (UIF) Flag used to determine if UID is enabled or not. 15, 19, 20

user-interrupt handler (UIHandler) The linear address of the function that's called on a successful user-interrupt notification. 16, 19, 38, 50, 54, 56, 57, 59

user-interrupt notification (UIN) When an ordinary interrupt is sent and the target is configured to receive this interrupt in user-space it's called a user-interrupt notification. 14–17, 19, 20, 30, 32, 51

user-interrupt recognition (UIR) Process by which a user-interrupt is recognized. The following reasons cause a user-interrupt to be recognized [25, sec. 8.4.1]:

- WRMSR to the `IA32_UINTR_RR` MSR
- XRSTORS of the user-interrupt state component.
- User-interrupt delivery
- User-interrupt notification processing
- VMX transitions that load the `IA32_UINTR_RR` MSR.

16

user-interrupt request register (UIRR) Core-specific bitmap that holds information on which UINVs are currently pending processing. 16

user-interrupt stack adjustment (UIStackadjust) Adjustment by which to move the stack during user-interrupt notification processing. Can either be set to be a strict adjustment, moving the stack pointer by x bytes, or an address, moving the stack pointer to x 16, 38, 50, 52

user-interrupt target table (UITT) Sender-managed table of UPID pointers and associated user-interrupt vectors. 17, 19, 20, 38, 39, 41, 50, 51, 56, 67

user-interrupt target table entry (UITTe) Data structure containing a pointer to a UPID, a valid bit and the `UV` field, which contains the UIV used when sending UIPIs. 17, 20, 39, 41, 45

user-interrupt vector (UIV) An Integer between 0-64 used in the user-interrupt target table entries and is forwarded to the user-interrupt handler. 16, 17, 38, 39, 41, 42, 50, 52, 55, 56, 113

user-interrupts notification vector (UINV) The interrupt vector that is forwarded to the user as a user-interrupt notification. 15, 16, 19, 33, 38, 50–52, 113

userspace I/O (UIO) Linux system that allows user-space applications, among other things, to read from and write to device-specific files to handle interrupts from that device. 13, 14

virtual address space (VSpace) The set of virtual memory available to a thread in seL4. The VSpace root is the PML4 capability located in the thread's TCB. 25, 39, 55, 56

virtual machine (VM) 30, 47, 48