SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Page Fault Forwarding via User-Interrupts

Anton Ge

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Page Fault Forwarding via User-Interrupts

Weiterleitung von Seitenfehlern mittels User-Interrupts

Author:	Anton Ge
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisors:	Dr. David Schall, Ilya Meignan--Masson
Submission Date:	28. August 2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 28. August 2025 Anton Ge

Acknowledgments

I would like to sincerely thank my advisors, Dr. David Schall and Ilya Meignan–Mason, for their invaluable guidance, time, and support throughout the course of this thesis. Their insights and feedback were essential in shaping both the direction of my research and the clarity of my work.

Finally, I acknowledge the resources and facilities provided by the Technical University of Munich, which made this work possible.

Abstract

Modern applications increasingly rely on user-space memory management mechanisms to handle performance-critical workloads. While Linux's Userfaultfd provides user-space page fault handling, its reliance on polling, blocking, and wake-up operations incurs significant overhead. This thesis investigates the integration of Intel User-Interrupts into Userfaultfd to enable direct, low-latency page fault forwarding between threads without kernel intervention in the delivery path.

This thesis presents a design and implementation of UINTR-based page fault forwarding and extends the gem5 simulator to prototype and evaluate this mechanism. The approach eliminates wake-up operations, reduces context switching, and avoids the need for dedicated polling threads. Evaluation results on synthetic benchmarks and graph traversal workloads demonstrate a consistent reduction in cycles spent in the handler and outside it, with an overall speedup of approximately 1.6 times compared to stock Userfaultfd. These findings confirm that UINTR enables more efficient user-space fault handling and frees computational resources for application work.

The contributions of this thesis include the design, Linux kernel prototype, and gem5 simulation infrastructure, which can be used for further research and experimentation. By combining hardware-assisted interrupts with user-space memory management, this work shows a way to reduce latency and improve scalability for application-managed memory.

Contents

1 Introduction

Modern data-intensive applications, such as databases [MGR24] and graph processing frameworks [Hei+18], increasingly rely on large in-memory datasets to meet performance requirements. Efficient memory management is therefore critical, as applications frequently encounter page faults when accessing data that is not currently mapped in physical memory. Traditional operating system mechanisms introduce significant latency when handling these faults, which limits the performance of high-throughput systems.

Recent research has explored user-space page fault handling to reduce kernel overhead. uMMAP-IO [Riv+19] uses the `SIGSEGV` signal, UMap [Pen+22] relies on Linux's Userfaultfd (UFFD), and LightSwap [Zho+23] performs user-space swapping to handle page faults using lightweight threads. These works demonstrate that handling page faults in user-space can reduce overhead from the kernel and improve performance, but software-only approaches still incur latency from kernel notifications and context switches. This motivates the use of hardware-assisted forwarding mechanisms, which can minimise fault-handling delays.

Modern Intel CPUs provide a feature called User-Interrupt (UINTR) [Cor24], which enables low-latency notifications between user-space threads. The xUI architecture [Ayd+25] extends this concept to allow hardware interrupts to be delivered to user space and, with this, provide a mechanism for low-latency, hardware-assisted event delivery. This thesis makes use of xUI to implement direct forwarding of page faults to registered user-space threads, providing an efficient mechanism for high-performance applications.

While software-based mechanisms in user-space have reduced kernel involvement, no system combines hardware-assisted interrupt delivery with page fault forwarding at the application level. This thesis addresses this research gap by designing and implementing a mechanism that uses xUI's architecture to forward page faults directly to user-space threads. The main problem can be stated as: existing systems either rely on software-only methods, which incur significant latency [Riv+19], or provide hardware forwarding only for other types of events [Ayd+25]. The goal of this work is to enable scalable page fault forwarding at low latency using xUI and improve performance in memory-intensive applications.

The proposed mechanism builds on xUI's interrupt forwarding and integrates Userfaultfd [The22] with UINTR to enable user-space handling of selected page faults.

Applications register memory regions and a corresponding UINTR handler. When a page fault occurs, the system determines whether it belongs to a monitored region. Monitored faults are forwarded directly to user-space, while all others are handled by the kernel. This design allows hardware-assisted page fault forwarding with low latency, minimal kernel involvement, and backward compatibility.

The mechanism is implemented in a gem5 simulation of an out-of-order CPU and extends xUI. A custom Memory-Mapped I/O (MMIO) device synchronises monitored memory regions between the kernel and simulator, and the UINTR delivery is extended to include the faulting address and error code to enable handlers to resolve faults entirely in user-space. This allows efficient page fault handling without modifying the standard kernel path or requiring polling threads.

The evaluation assesses the proposed page fault forwarding mechanism in a gem5 simulation. Two benchmarks are used: a microbenchmark that measures per-page fault latency, and a Breadth-First Search (BFS) workload with irregular memory access to capture overall performance impact. In both cases, UINTR-based forwarding is compared to stock Userfaultfd. These experiments demonstrate the latency reduction and efficiency gains achieved by forwarding page faults directly to user-space handlers.

The evaluation shows that UINTR-based page fault forwarding significantly reduces latency compared to stock Userfaultfd, both in microbenchmarks and application-level workloads. The mechanism eliminates thread wake-ups, context switches, and extra bookkeeping, which results in a more efficient fault resolution. These improvements translate into measurable runtime gains in real-world scenarios. They demonstrate that hardware-assisted forwarding of page faults can effectively benefit applications with frequent memory accesses.

This thesis makes three main contributions. First, it designs a hardware-assisted page fault forwarding mechanism that forwards selected faults to user-space. Second, it demonstrates the mechanism with both synthetic and real-world workloads, thereby showing reduced page fault latency and runtime improvements. Third, it provides a gem5-based prototype that validates the approach and can serve as a foundation for further research in hardware-assisted fault handling.

2 Background

Before introducing the page fault forwarding mechanism, a few topics need to be covered first for a better understanding of the later chapters. Namely, the hardware feature the forwarding mechanism is based on, the software interface that is coupled with the mechanism and lastly, the paper that enables the forwarding of hardware event notifications using UINTR by enhancing the current hardware design. These topics provide the necessary context for understanding the mechanisms developed to address the limitations described in Chapter 1.

2.1 Userfaultfd

UFFD is an ideal starting point for the page fault forwarding mechanism, as it naturally moves part of the page fault handling routine away from the kernel into user space and cleanly splits the routine into separate notification and resolution parts.

UFFD allows user-space applications to intercept and handle their own page faults for selected memory regions [The22]. This is achieved by explicitly registering a memory range with the kernel through the `Userfaultfd` system call. Typically, applications reserve such a range using `mmap`, either backed by a file or as an anonymous mapping. As shown in Figure 2.1, the mapping is part of the process's user-space address space, but lies outside dynamically growing regions such as the heap or stack [BC05].

When the program accesses an address in the registered region that is not yet backed by a physical page, a page fault occurs. The CPU raises a page-fault exception (`#PF`),

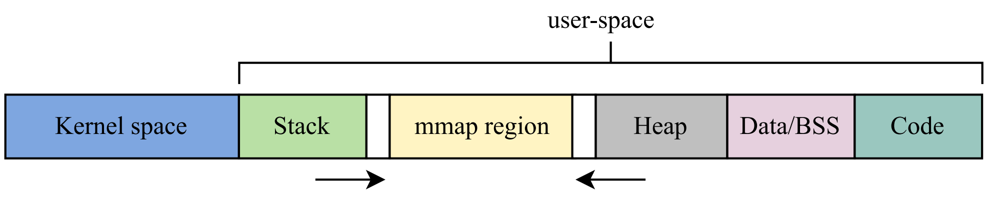

Figure 2.1: Typical process memory layout on Linux. Userfaultfd registers memory from the mmap region.

which traps into the kernel. If the faulting address falls within a UFFD-registered range, the kernel does not perform its normal fault resolution. Instead, it queues a fault notification on the file descriptor associated with the registration.

A separate handler thread typically polls on the file descriptor to receive a faulting notification. The faulting thread is blocked in the kernel until the handler resolves the fault. The kernel exposes faults through the `uffd_msg` struct. The `struct` most notably contains the faulting address and an error code that describes the fault. For example, a missing page is typically represented by the CPU error code 0x4. Another example is a minor fault, signalled via the `UFFD_PAGEFAULT_FLAG_MINOR` flag, which occurs when the page is already present but still requires handling by user-space. Depending on the fault and the specific needs of the application, `ioctl` with one of the following options is called to resolve the page fault:

- `UFFDIO_ZEROPAGE`: Maps a physical page at the faulting address and initialises it to zeros.
- `UFFDIO_COPY`: Allocates a page and copies data supplied by the user into it. Common patterns include reading from a file or another mapping and storing the data temporarily in a buffer.
- `UFFDIO_CONTINUE`: Resolves minor faults by installing a page that already exists in the kernel's page cache. This is typical for file-backed mappings registered with `minor` mode.

After the handler's `ioctl` succeeds, the kernel installs the mapping and unblocks the faulting thread, which resumes at the faulting instruction. While this mechanism gives applications control over page population, they still rely on the kernel to notify the faulting thread. Page fault forwarding aims to replace this kernel-handled notification with UINTR, allowing faults to be delivered directly to user-space. The next section introduces UINTR and its potential for low-latency, asynchronous event handling.

2.2 User-Interrupt

UINTR is a hardware feature that Intel introduced with their Sapphire Rapids server CPUs in 2023. This feature allows interrupts to be delivered to processes running in user-space without involving the kernel apart from the initial setup [Cor24]. Its underlying architecture forms the basis for xUI's interrupt forwarding, which is outlined in Section 2.3, and consequently underpins the mechanisms studied in this thesis. For that reason, this section presents the workflow for sending and receiving UINTRs and its major components involved in the process, as they will be relevant for the page fault

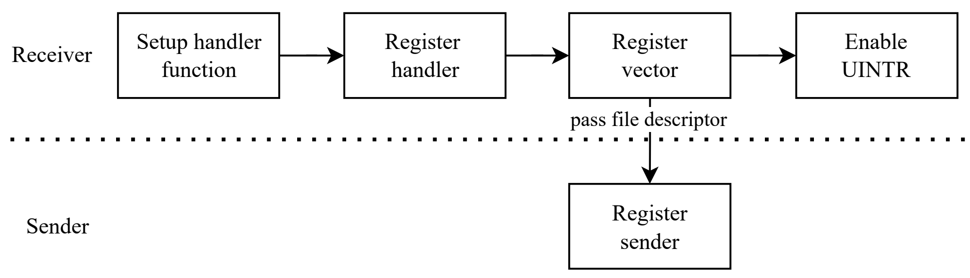

Figure 2.2: Setup for sending and receiving UINTR. Passing the file descriptor to the sender grants permission to send UINTR.

forwarding as well. Lastly, there will be an overview of research done on this new hardware feature and some benchmark results that highlight why this new notification method appears like a viable alternative to existing mechanisms.

2.2.1 Setup and Components

The process of setting up a receiver and sender is illustrated in Figure 2.2. The receiver thread first registers a handler function to execute upon receiving a UINTR. It then registers a vector for that particular UINTR and gets a file descriptor in return. The thread can then enable delivery by executing the `stui` instruction, which sets the User Interrupt Flag (UIF) in the `UINTR_MISC` Model-Specific Register (MSR).

The file descriptor needs to be passed to the sender for its setup. A system call using this descriptor links the sender to the receiver and returns an index. The sender uses this index with the `senduipi` instruction to deliver UINTRs. The essential data structures required for the delivery of UINTRs are explained below:

UPID. The User Posted Interrupt Descriptor (UPID) is a per-thread data structure for receiving User-Interrupts. It is created when a thread first registers a UINTR handler. The structure uses 64 bits to keep track of any pending interrupt requests, and each bit corresponds to a vector. Setting a bit (e.g., via `senduipi`) marks the vector as pending. Subsequently, the CPU copies these bits into the `UINTR_RR` MSR that is used to keep track of UINTRs that have not been processed yet.

UITT. The User Interrupt Target Table (UITT) is the counterpart of the UPID and the per-process data structure associated with the UINTR sender. Each table entry holds a pointer to a receiver's UPID and represents permission to send a User-Interrupt to that thread. When a sender registers with a receiver's file descriptor, the kernel creates a UITT entry. The entry index is then used as the operand to `senduipi`.

UI_FRAME. When a UINTR handler executes, it is passed a pointer to a `ui_frame` struct by the CPU. This `struct` contains the saved state of the interrupted thread (`RIP`, `RSP`, and `RFLAGS`). This state is not meant to be accessed by the handler itself, but rather allows the `uiret` instruction to restore execution at the exact point where the interrupt was taken.

In comparison, conventional Inter-Process Communication (IPC) interrupts require the sender to trigger a notification that is handled by the kernel [BC05]. This involves a context switch into the kernel, execution of the interrupt handler, and a return to user-space, which introduces non-negligible latency. UINTR, by contrast, allows the sender to signal the receiver directly through the UPID without kernel involvement after setup. Each UINTR is delivered straight to the registered user-space handler, which avoids context switches and makes delivery significantly faster. This capability is particularly beneficial for frequent events such as page fault forwarding, where the overhead of ordinary kernel-handled notification would otherwise dominate the end-to-end latency.

To demonstrate the performance of the stock UINTR feature and why it serves as the delivery mechanism for the proposed design, some microbenchmarks are shown next.

2.2.2 Microbenchmark

The microbenchmarks highlight the potential of this relatively new feature by comparing the latency of User-Interrupts to established IPC mechanisms such as POSIX signals and shared memory. They focus solely on delivery latency and do not perform any additional computation. These experiments are conducted on a server equipped with an Intel Xeon Gold 6438Y+ CPU with 64 cores and 2 TiB of memory. For the operating system, a Linux v6.0.0 kernel with Intel's UINTR patches is installed. First, a summary of the individual microbenchmarks is provided:

Setup

All three microbenchmarks share the following characteristics:

- *CPU affinity*: Sender and receiver threads are pinned to separate CPU cores using `pthread_setaffinity_np` to reduce the interference from scheduling and ensure deterministic measurements.
- *Time measurement*: Round-trip latency is measured using `CLOCK_MONOTONIC` before sending a notification and upon receipt. The values are stored in an array for `NUM_ITER` iterations. Minimum, maximum, total and average latencies are calculated after the benchmark completes.

- *Thread synchronisation*: Volatile or atomic variables coordinate the start and completion of each iteration between sender and receiver.

This shared setup ensures that differences in measured latency reflect only the notification mechanism under test.

UINTR. This microbenchmark measures the latency of UINTR delivery between two threads. It sets up a sender thread and a receiver thread as described in Section 2.2:

- *Receiver*: The registered handler records the timestamp when a UINTR is received and calculates the latency relative to when the sender issued it.
- *Sender*: The sender repeatedly sends UINTRs using the `senduipi` instruction after ensuring the receiver is ready. For each iteration, it records the timestamp right before sending.

This microbenchmark isolates the delivery overhead of UINTRs and provides a baseline for evaluating the efficiency of User-Interrupts.

Signal. This microbenchmark measures the latency of POSIX signals (`SIGUSR1`) exchanged between two threads.

- *Receiver*: A signal handler for `SIGUSR1` is installed. The handler captures the timestamp upon signal delivery and calculates the latency.
- *Sender*: After ensuring the receiver is ready, the sender thread repeatedly sends `SIGUSR1` signals to the receiver using `pthread_kill`. As with UINTR, a timestamp is recorded just before sending the signal.

The analysis of the overhead of signal delivery serves as a reference point for a notification mechanism that is handled by the kernel.

Shared Memory. The final microbenchmark evaluates the latency of communication between two threads using a shared memory page and coordinated access.

- *Receiver*: For each iteration, it waits for the sender's notification, reads data from the shared memory page, and notifies the sender that it has completed its read.
- *Sender*: The sender repeatedly writes data to the shared memory page, signals the receiver, waits for the receiver to finish and then reads back the updated content. The timestamp immediately before writing is taken to compute the latency for each iteration.

The shared memory benchmark provides an additional baseline for comparing notification mechanisms. It represents a purely user-space communication path with minimal kernel involvement.

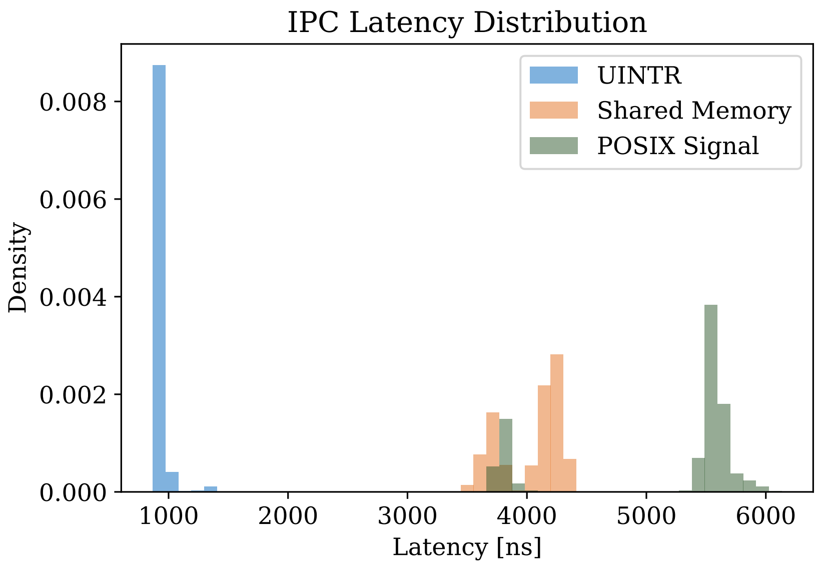

Figure 2.3: Histogram of IPC latencies from the microbenchmarks, showing all measurements below the 99th percentile. UINTR has the lowest latency and the most concentrated distribution.

Analysis of Microbenchmarks

Figure 2.3 shows the latency distributions of the three mechanisms plotted as normalised histograms. To improve readability, extreme outliers beyond the 99th percentile were removed from the data sets. The x-axis represents latency in nanoseconds, while the y-axis indicates the relative probability density. In addition, the key metrics from the microbenchmarks are summarised in Table 2.1, without accounting for outliers. The distribution of UINTR has its peak near the lowest latencies. Almost all samples lie within a narrow band around 900 ns, which aligns with the values in Table 2.1. The closeness of mean and median, combined with the small standard deviation, indicates both low latency and high predictability.

Both shared memory and POSIX signal differ in their shape: their histograms exhibit two distinct peaks instead of one. For shared memory, the two peaks are relatively close together, which results in a mean latency of around 4000 ns. For the POSIX signal, the peaks are further apart, which leads to a greater spread. This observation is also reflected in the larger maximum value and higher standard deviation in Table 2.1.

The microbenchmark analysis demonstrates that UINTR consistently achieves lower and more predictable latencies than shared memory and POSIX signals. This shows that UINTR-based notification can substantially reduce the cost of fault handling compared to traditional mechanisms. While the evaluation in this section focuses on microbenchmarks rather than real workloads, the observed improvements in both mean and tail latency suggest that realistic applications could benefit as well.

Table 2.1: Key metrics of the IPC methods in [ns].

	UINTR	Shared Memory	POSIX Signal
Mean	924	4082	5203
Min	867	3504	3719
Max	4889	17995	20898
Median	891	4159	5549
STD	267	640	948

These results highlight the potential of low-latency communication mechanisms based on UINTR. Building on this observation, the next subsection examines prior research that leverages UINTR for user-space scheduling and preemption, and illustrate how such mechanisms can be applied in practical systems.

2.2.3 User-Interrupts for Low-Latency User-Space Scheduling

Recent studies have shown that UINTR enables user-space preemption, which makes it possible to extend these techniques to other areas, such as event handling.

Skyloft [Jia+24] presents a user-space scheduling framework that leverages UINTR for scheduling that supports both preemptive and non-preemptive policies. It integrates with DPDK and other high-performance I/O frameworks to achieve microsecond-scale thread preemption and, thus, improves responsiveness in latency-critical workloads.

PreemptDB [Hua+25] applies UINTR to database transaction scheduling. It enables fine-grained preemption of transactions and reduces latency for high-priority tasks compared to traditional cooperative or First In - First Out (FIFO) scheduling approaches.

LibPreemptible [Li+24] is a user-space library that provides adaptive preemption via UINTR. It supports multiple scheduling policies, runs entirely in user-space, and requires minimal changes to existing applications. This shows that UINTR can deliver low-overhead task management without kernel involvement.

Collectively, these works showcase practical UINTR-based user-space scheduling and preemption. However, their focus lies on delivering low-latency control entirely in user-space, which is restricted to the existing model of user-to-user notifications. How UINTR can be extended to kernel-originated events, such as page faults, has not been explored to the same extent. xUI [Ayd+25] proposes a hypothetical architecture

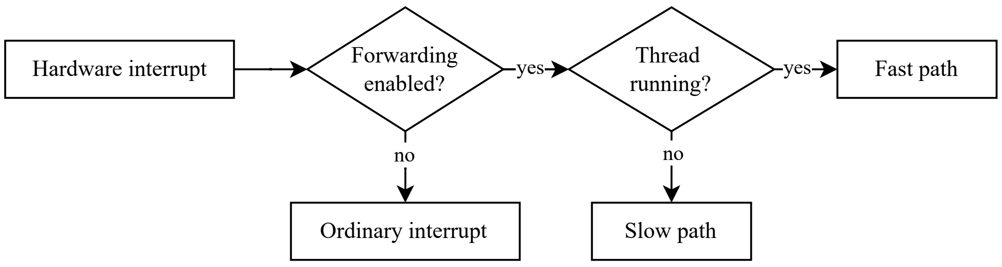

Figure 2.4: xUI Interrupt Forwarding Workflow.

that generalises UINTR to support direct delivery of hardware interrupts to user space. Their work provides the foundation for the mechanism studied in this thesis. The next section introduces this architecture in detail.

2.3 xUI

Motivated by UINTR's restrictions, xUI [Ayd+25] proposes a hypothetical architecture that enables hardware interrupts to be delivered directly to user-space, among other modifications to stock UINTR. This extension addresses the limitation that UINTR notifications can only be sent between user-space threads. Interrupt forwarding allows, for instance, an Ethernet controller to notify a process of each incoming packet without kernel intervention or the need for active polling.

2.3.1 xUI Design

The design proposed by Aydogmus et al. [Ayd+25] enables hardware interrupts to be forwarded to user-space with relatively modest modifications to the existing hardware. A simplified overview of the workflow is illustrated in Figure 2.4. The mechanism relies on extending the Local Advanced Programmable Interrupt Controller (LAPIC)s with two additional registers, `forwarding_enabled` and `forwarding_active`. Each register contains 256 bits, which encode the interrupt vectors that should be forwarded to user-space threads.

When an interrupt arrives, the LAPIC first consults the `forwarding_enabled` register to see if interrupts with that particular vector should be forwarded to user-space. If that bit has not been set, the interrupt will take the normal route, and the event will be handled by the kernel. If the bit corresponding to the interrupt vector has been set, the LAPIC will start with the UINTR delivery process by setting the same bit in the `UINTR_RR` MSR. The `forwarding_active` register indicates whether the target thread is currently running and whether the interrupt can be delivered immediately (called the fast path). If the thread is not running, delivery is deferred until the thread is rescheduled (slow path). Similar to UINTR, the kernel sets up the mappings between user threads and interrupt vectors by writing the relevant bits in the two new registers of the LAPIC. In a multithreaded scenario, the kernel ensures that the `forwarding_active` register holds the bits of the running thread.

2.3.2 Interrupt Forwarding Implementation

To evaluate this hypothetical architecture, Aydogmus et al. [Ayd+25] performed a series of benchmarks in gem5, a platform for simulating and testing computer architectures. The authors had to extend gem5 because the simulator did not support UINTR before xUI. They added User-Interrupt support to the out-of-order (O3) CPU and integrated it into the existing CPU pipeline and interrupt logic. For this, they determined the cycle costs for each stage of the UINTR delivery on a real system so that the implementation could model realistic latency. Their system uses the O3 CPU as it is the most realistic model for modern, high-performance CPUs. It simulates a full out-of-order pipeline with stages such as fetch, execute and commit, and it is able to simulate micro-ops execution.

Since gem5 does not model a LAPIC as a stand-alone unit, UINTR delivery from the interrupt forwarding mechanism is implemented directly in the interrupt logic of the X86ISA. Consequently, `forwarding_enabled` and `forwarding_active` are not added as separate registers in the simulated system. Instead, their behaviour is integrated into the interrupt handling logic. This approach is used in the xUI benchmark for testing interrupt forwarding from the Ethernet controller: each network packet received on the controller schedules a UINTR event with a fixed, specific interrupt vector. The event is propagated through the generic device interface and the board's I/O subsystem until it reaches the I/O APIC. The I/O APIC then broadcasts the interrupt to all cores associated with that vector, and each core schedules the corresponding User-Interrupt for future handling. Unlike stock User-Interrupt, the UINTR issued in interrupt forwarding are not sent via the `senduipi` instruction and therefore, do not trigger a write to the UPID of the receiver. This results in two positive side effects: Firstly, not having to access the UPID data structure for UINTR delivery saves CPU cycles. Secondly, the sender side does not need to go through the registration process, and the receiver does not need to pass a file descriptor to the sender.

During the commit stage of the O3 CPU model, the processor checks for any scheduled interrupts. If a User-Interrupt is pending, the CPU traps and executes a microcode routine corresponding to the interrupt vector. This is done by creating a fault object that represents the event. For instance, a standard `PageFault` object for page faults, and a `UserPci` object for network packet interrupts, respectively. Generally, the fault object sets up the necessary CPU registers and adjusts the program counter to the start of the microcode routine. The routine then prepares the CPU state before transferring control to the appropriate interrupt or fault handler. For UINTRs and network packet interrupts, the microcode routine involves disabling further UINTRs for the duration of the handler, saving the thread state (`RIP`, `RSP` and `RFLAGS`) and queueing up the registered UINTR handler to run afterwards.

xUI's modified gem5 system serves as the basis for further modifications in the context of this thesis. The exact changes to enable page fault forwarding in gem5 are covered in Chapter 5.

3 Overview

This chapter provides a high-level view of the proposed page fault forwarding mechanism. It explains how selected page faults are routed from the Memory Management Unit (MMU) to user-space threads using a combination of User-Interrupt and Userfaultfd, outlines the workflow, and describes the main components involved. The overview establishes the foundation for the subsequent design and implementation chapters.

3.1 Hardware Overview

The system selectively forwards page faults to threads in user-space, while kernel page faults and user-space faults that are not UFFD-backed are handled by the OS. A high-level overview of the page fault forwarding is shown in Figure 3.1. The design leverages the interrupt forwarding architecture of xUI: an interrupt from the MMU is treated like any other hardware interrupt and forwarded directly to the recipient thread in user-space when the relevant bits in the `forwarding_enabled` and `forwarding_active` registers are set.

To avoid unnecessary overhead, the MMU checks whether a faulting address belongs to a UFFD-backed region immediately after detecting a page fault. Due to the high number of regularly occurring page faults during system execution, it would be

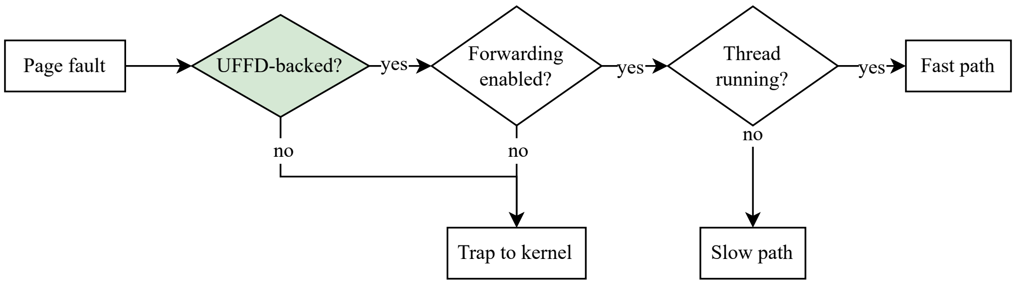

Figure 3.1: Overview of page fault forwarding: the design builds on top of xUI and utilises its forwarding mechanism.

detrimental to the performance of an application if every page fault were forwarded and the thread had to check whether the address belonged to a UFFD-region. At the same time, re-routing a page fault from the UINTR handler back to the kernel will further increase the latency of all page faults that are not UFFD-backed. Therefore, an intermediate step is introduced between the fault event and forwarding the interrupt. As described in Section 2.1, this check is currently done by the kernel as part of the default page fault routine. For this to take place before the trap and in hardware, a new mechanism to keep track of UFFD regions and to verify faulting addresses has been implemented.

A requirement for managing UFFD-backed regions is that the lookup of address ranges during page fault events should be fast and with minimal overhead. Even with a high number of tracked UFFD regions, determining whether a faulting address lies within a tracked region is performed in minimal cycles, avoiding linear or worse runtime complexity.

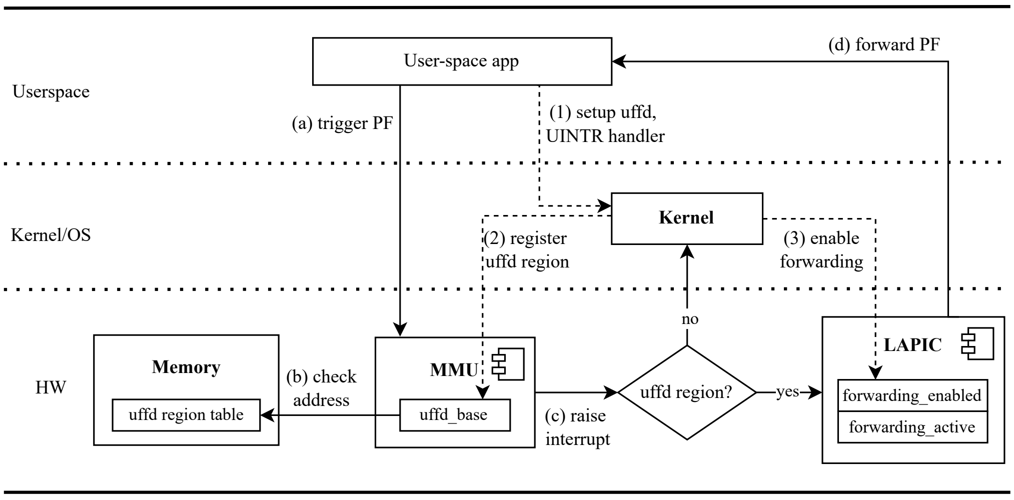

Figure 3.2: Detailed page fault forwarding workflow with component interaction.

3.2 Linux Kernel Modifications

UINTR has been extended to deliver the faulting address and error code as additional payload. Originally, UINTR passes only the vector, interrupt flags, stack and instruction pointers during an interrupt. As with other modifications, the changes to UINTR are done in a non-intrusive and minimal way. The delivery mechanism, stack access, and return routine remain unchanged to ensure compatibility with xUI's interrupt forwarding and sending UINTRs via the `senduipi` instruction. The two additional values necessary for page fault handling are passed through the stack, without affecting how `RSP`, `RIP`, and `RFLAGS` are shared with user-space.

3.3 Workflow

The page fault forwarding utilises a combination of xUI's architecture, Userfaultfd, and User-Interrupts to enable user-space handling of selected page faults. Applications register memory regions to be monitored and a corresponding UINTR handler. When a page fault occurs, the system determines whether the faulting address belongs to one of these monitored regions. Faults outside such regions follow the standard kernel handling path. On the other hand, monitored faults are forwarded to the user-space for resolution if the core has forwarding enabled. The workflow above with all major system components is illustrated in Figure 3.2.

4 Design

This chapter details the architecture and key design decisions of the page fault forwarding mechanism. It covers how UFFD-backed regions are tracked in hardware, how the CPU and MMU handle fault detection and forwarding, and how the faulting address and error code are delivered to the user-space handler. The design discussion highlights trade-offs and motivates the chosen solutions.

The proposed mechanism builds on xUI's interrupt forwarding and combines elements from Userfaultfd and User-Interrupt. During setup, the user-space application registers a reserved memory range with UFFD, and configures a UINTR handler to receive forwarded page faults and resolve them with any of the `UFFDIO` options from Section 2.1. Lastly, interrupt forwarding needs to be enabled through the kernel as per xUI. This will set bit 14 in the `forwarding_enabled` and `forwarding_active` registers of the LAPIC as it corresponds to the interrupt vector for page faults. At runtime, when the MMU detects a page fault, it checks whether the faulting address falls within a UFFD-registered range. If not, the fault follows the standard kernel-handled page fault path. If it does, the CPU consults the `forwarding_enabled` and `forwarding_active` registers and decides which path the page fault will take.

4.1 MMU

The MMU plays a central role in the page fault forwarding mechanism: In addition to managing memory accesses, it maintains a record of UFFD-backed regions and verifies whether a faulting address belongs to any of the tracked regions. When setting up Userfaultfd, the `ioctl` call for registering a memory region provides the start and end address of the region, which the MMU stores in a reserved 4 KiB in-memory table. Two new MSRs, `uffd_base` and `uffd_size`, hold the pointer to the beginning of this table and its size, respectively. The entries in the table are sorted by the start address when inserted. During lookup, binary search is used on the sorted start addresses to quickly determine whether a faulting address lies within a tracked region. With a table of 4 KiB, up to 256 start/end pairs can be stored, which should be enough for the expected number of UFFD regions per core while keeping the lookup overhead minimal. For n memory regions, binary search requires $O(log\ n)$ comparisons. The following describes the check that the MMU performs for a page fault event:

Algorithm 1 MMU lookup process for a faulting address.

```
$Input = fault\_addr;$
$Registers:$
  $base \leftarrow uffd\_base;$
  $size \leftarrow uffd\_size;$
$low \leftarrow 0,\ high \leftarrow size - 1;$
while $low \leq high$ do
    $mid \leftarrow (low + high)/2;$
    $start \leftarrow MEM[base + mid].start;$          ▷ read start address (included)
    $end \leftarrow MEM[base + mid].end;$              ▷ read end address (excluded)
    if $fault\_addr < start$ then
        $high \leftarrow mid - 1;$
    else if $fault\_addr \geq end$ then
        $low \leftarrow mid + 1;$
    else $return\ true;$
    end if
end while
$return\ false;$
```

For this process, it is reasonable to estimate a cost of 3 to 5 cycles per comparison. Combined with the number of lookups required, the overhead is approximately

$$5\ cycles * log_2(n)$$

on average, and at most

$$log_2(256) * 5 = 40\ cycles$$

. An even faster check could be achieved by introducing new registers for directly storing the addresses of UFFD regions, similar to Memory Type Range Registers (MTRR). The address check could then happen in parallel across all region registers, resulting in a constant O(1) runtime. The drawback is that the number of concurrent UFFD-regions would be very limited compared to an in-memory table, as only a handful of region registers could exist per core. Nevertheless, a small number of region registers per core may be sufficient, since a high number of concurrent UFFD regions is uncommon.

4.2 Faulting Address and Error Code

A key design question is how the faulting address and the associated page fault error code should be transferred to the user-space UINTR handler. The following options were considered:

Stack

One straightforward option is to extend the interrupt delivery mechanism so that the CPU pushes the faulting address and error code onto the stack of the UINTR handler. This method requires minimal changes to the delivery path as it reuses the same mechanism already employed to save the state before entering a UINTR handler. As described in Section 2.2, the CPU currently pushes `RSP`, `RIP`, and `RFLAGS` onto the handler's stack as part of the UINTR delivery routine, which can then be accessed via the `ui_frame` struct. Extending this process to also include the faulting address and error code would be a simple addition. This approach does not need any dedicated storage registers, and the values can be read from inside the handler without a privilege change or extra instructions.

However, there are practical concerns: Adding more pushes modifies the stack layout. Alignment must be preserved to avoid undefined behaviour. Furthermore, stack clean up through the `uiret` instruction must take care of the added stack entries while maintaining compatibility with stock UINTR and xUI's interrupt forwarding.

MSR

Another approach is to introduce new dedicated MSRs to hold the faulting address and error code, and leave the UINTR stack untouched. In this design, the CPU would store the values in these registers as part of the UINTR delivery preparation. Reading the values would be straightforward with the `rdmsr` instruction. The advantage of using MSRs is that they offer a clean interface for passing these parameters. The values would be stored in a fixed location and remain valid until explicitly updated.

However, this option has significant disadvantages in the UINTR use case. On x86 systems, MSRs are generally not directly accessible from user-space for security reasons. To obtain the faulting address or error code, the handler would need to make a system call to the kernel, which would read from the MSR and return the value. This additional context switch would increase latency. Aside from that, adding new MSRs requires changes to the CPU design, which is more complex than extending the stack frame.

Considering the trade-offs, the stack-based approach was chosen. It requires only minor modifications to the stack logic, and, most importantly, allows immediate access to the faulting address and error code without leaving user-space. While it slightly alters the handler stack and alignment must be ensured, these changes are minimal compared to the complexity and latency penalty of introducing new MSRs.

4.3 Discussion

Supporting UINTR-based page fault forwarding requires additions to the CPU and MMU beyond the xUI architecture. In the MMU, new memory is needed to store a table of UFFD-registered address ranges, along with a routine to populate this table during `UFFDIO_REGISTER`. Two registers store the pointer to the start of the table and its size, and a fast check is performed on each page fault to determine whether the faulting address lies within a registered region. In the CPU pipeline, a new routine pushes the faulting virtual address and error code onto the stack of the UINTR handler to allow it to correctly resolve the fault in user-space. Together, these extensions enable direct forwarding of page faults to user-space with minimal kernel involvement while maintaining backward compatibility.

5 Implementation

This chapter describes how the design was realised in practice. It explains the extensions to gem5 and the Linux kernel, the creation of a custom MMIO device for region tracking, the integration of UINTR for direct fault delivery, and modifications to the handler stack. The chapter demonstrates how the theoretical design was translated into a working prototype for evaluation.

5.1 Page Fault Forwarding

The main benchmarks take place in a gem5 simulation. As described in Section 2.3, xUI provides a mechanism for scheduling UINTR to simulate the behaviour of forwarding a hardware interrupt to user-space.

The page fault forwarding logic follows the same design principles as xUI's interrupt forwarding. Since the MMU is not modelled as a stand-alone unit in gem5, it cannot independently schedule a trap event when a page fault occurs. Instead, page faults are tracked during the execution stage of the O3 CPU, but handling is deferred until commit. At this point, the pipeline ensures that the faulting instruction has retired or is about to retire, which avoids hazards associated with pipeline reordering and guarantees that the fault can be forwarded safely. Therefore, the commit stage is the natural point in the CPU pipeline to implement page fault forwarding. Additionally, extending this logic at commit ensures consistent integration with the existing pipeline model.

Algorithm 2 illustrates the modified commit-stage workflow. First, the CPU checks whether a fault has been recorded during execution. If it has and it is of type page fault, the system queries the region manager to determine whether the faulting address belongs to a UFFD-backed region (see Section 5.2). Next, two additional conditions are verified: the `UIF` flag must be set to enable UINTR delivery, and the `IF` bit in `RFLAGS` must indicate that interrupts are not masked.

The `UserPageFault` object routine combines the functionality of a standard page fault object and the UINTR routine. It first invalidates any matching Translation Lookaside Buffer (TLB) entries, then stores the interrupt vector, error code, and the offset of the faulting instruction in the code segment into dedicated temporary registers. These registers are subsequently accessed during the microcode routine. The fault object

Algorithm 2 Page fault forwarding in the commit stage.

```
if $exec\_stage.hasFault()$ then
    $fault \leftarrow exec\_stage.getFault();$
    if $fault.type == PAGE\_FAULT$ then
        if $isUFFDRegion(fault.address)$ and $UIF\_set()$ and $RFLAGS.IF\_set()$ then
            $user\_fault \leftarrow UserPageFault(fault.vector, fault.error\_code);$
            $CPU.trap(user\_fault);$
        end if
    end if
end if
```

also sets UINTR flags to ensure that the `uiret` routine, responsible for cleaning up the stack and resetting CPU flags, is scheduled after the handler completes. Finally, the fault object sets the microcode routine according to the CPU mode (`long` or `protected`) and the context in which `trap` was called. Since page faults are always handled in a trap context and 32-bit mode is rarely used, the `UserPageFault` object effectively has a single routine option.

The microcode routine follows the structure defined by Aydogmus et al. [Ayd+25]. It begins by disabling further UINTR delivery while executing the routine. The routine then saves `RSP`, `RIP`, and `RFLAGS` on the stack and adjusts the instruction pointer to the UINTR handler. For page fault forwarding, two additional values are pushed onto the stack: the faulting address and the error code. Before pushing the address, the routine aligns it to the page boundary by masking out the lower bits. By default, a Linux page size of 4 KiB is assumed, which clears the lowest 12 bits. The routine could, however, dynamically adjust the alignment based on the page size in use by checking the page size bit in the page directory. For example, a set bit indicates a 2 MiB page in long mode. For standard Userfaultfd, the kernel guarantees that the address passed to user-space meets the alignment requirement.

To maintain compatibility with the existing `uiret` stack cleanup routine, the `RSP` is first saved in a temporary register. Any number of values can then be pushed to the stack, provided proper alignment is maintained. Finally, the saved `RSP` is pushed along with the other two members of the `ui_frame` structure. This ensures that the stack is returned to its original state when `uiret` executes. Access to the extra data pushed onto the stack is discussed further in Section 5.3.

5.2 Userfaultfd Region Tracking

Tracking registered Userfaultfd regions is implemented via a new MMIO device in gem5 that enables communication between the simulated system and the Linux kernel. To understand the device logic, it is helpful to review gem5's memory model briefly.

In gem5, memory is organised as address ranges mapped to different components, such as DRAM or devices. The system board connects these ranges via ports on various buses (e.g., memory, I/O, or PCIe buses). When a component, such as the CPU, issues a read or write, the request is encapsulated in a `Packet`, which includes the address, command type, size, and other metadata. The packet is then routed through the bus hierarchy to the target device or memory. For reads and certain writes, a response packet is returned.

For UFFD tracking, the custom MMIO device receives write packets containing the start and end addresses of the tracked region. The device triggers its tracking logic upon receiving a write and stores these addresses for later use during page fault forwarding. MMIO was chosen because it provides a simple mechanism for the simulated system to communicate with the kernel without requiring changes to the CPU or complex synchronisation mechanisms.

Algorithm 3 Merging of overlapping or adjacent regions during registration.

```
$Input:\ start,\ end$
**if** $start \geq end$ **then**
    **return**
**end if**
**for** each region $(r\_start, r\_end)$ overlapping or adjacent to $[start, end)$ **do**
    $start \leftarrow \min(start, r\_start)$
    $end \leftarrow \max(end, r\_end)$
    remove region $(r\_start, r\_end)$
**end for**
insert region $(start, end)$
```

The tracking logic uses a map that sorts inserted address regions by their starting addresses. This enables efficient lookups in $O(\log n)$ time for n tracked regions when checking whether a faulting address is backed by UFFD, as described in Chapter 4. When inserting a new region, the tracker first identifies any existing regions that overlap with or are directly adjacent to the new interval. It then computes the minimal start and maximal end addresses that cover both the new and existing regions, removes the old entries, and inserts a single merged region. This process ensures that the tracker maintains a minimal and non-overlapping set of registered ranges to reduce lookup complexity and avoid redundant entries. The pseudo-code for this merge process is shown in Algorithm 3. Conversely, when a previously tracked region is removed, the tracker may split a merged interval into multiple smaller regions to accurately reflect the remaining registered ranges.

To realistically include the cost of this check in the cycle count, the `isBacked(Addr address)` function returns both a boolean indicating whether the address is backed and the number of comparisons performed as a struct. After the check during the commit stage, these additional cycles are added to the base cost of a trap.

A custom Linux kernel platform device and driver were implemented to support communication with the gem5 system via MMIO. The device exposes a fixed MMIO region at the physical address 0xc0100000, sized 16 bytes. Upon probing, the driver maps this MMIO region into the kernel's virtual address space and, thus, enables the kernel to perform memory-mapped I/O accesses. The MMIO region serves as the interface for passing registered Userfaultfd memory ranges from the kernel to gem5. Write operations to this region trigger updates in the simulator, allowing the kernel and simulator to synchronise the set of tracked memory areas without requiring additional complex communication protocols.

Address regions are communicated during Userfaultfd setup: as described in Section 2.1, a system call registers reserved memory from a `mmap` call to be monitored. Near the end of a successful registration, the kernel writes the start and end addresses into the MMIO region using `iowrite64`. This write triggers the region tracker described above. Importantly, how the kernel resolves page faults remains unchanged, as this design only introduces a new method for delivering fault notifications.

5.3 Usage

The page fault forwarding design keeps the standard Userfaultfd setup but modifies the notification path. Applications still create a file descriptor and register memory ranges via the usual `ioctls`, as explained in Section 2.1. However, a dedicated polling thread or event loop is no longer required. Instead, the thread that performs the UFFD setup must also install a UINTR handler, register that handler through the appropriate system call, and enable UINTR delivery via the `stui` instruction.

The remainder of the usage pattern is familiar. Inside the UINTR handler, any of the page-fault resolution methods described in Section 2.1 may be used. The handler can obtain the faulting address and error code from the stack by casting the supplied `ui_frame` to the extended frame layout (`uintr_frame_extended`), which includes `fault_address` and `error_code` fields. As an alternative, the handler may use pointer arithmetic: each struct member is a 64-bit value, so the offset to a field equals the frame base plus $8 \times (fieldindex)$.

The design is backward compatible: programs using stock userfaultfd or stock UINTR behaviour continue to work unchanged. Forwarding is opt-in: a thread only receives forwarded page-fault notifications if it performs the UFFD registration and also registers a UINTR handler and enables UINTR delivery.

One implementation caveat is simulator behaviour. Unlike the design described in Chapter 4, the kernel does not configure the `forwarding_enabled` and `forwarding_active` registers in gem5, since the LAPIC is not modelled. Instead, page fault forwarding is hard-coded: a thread automatically opts in by performing UFFD setup, registering a UINTR handler, and enabling UINTR delivery.

With the forwarding mechanism and usage pattern established, the next chapter evaluates performance and behaviour under a set of benchmarks.

6 Evaluation

This chapter evaluates the performance and applicability of the hardware-assisted page-fault forwarding mechanism. It measures latency improvements and runtime impact using both synthetic microbenchmarks and a real-world BFS workload. The evaluation demonstrates how the design choices affect fault handling efficiency and highlights the benefits of forwarding selected page faults directly to user-space.

With the help of experiments, the evaluation aims to answer the following research questions:

Q1 Latency improvement: Can UINTR-based page fault forwarding reduce the latency of user-space page fault handling compared to traditional mechanisms such as polling?

Q2 Real-world performance: How does page fault forwarding impact the performance of realistic workloads, such as graph traversal with BFS, compared to stock UFFD?

6.1 Page Fault

The benchmarks evaluate page fault forwarding in gem5 based on xUI's interrupt forwarding mechanism.

6.1.1 Setup and Methodology

As the benchmarks rely on a hypothetical architecture, they are run on a gem5 simulated system. The configuration for the gem5 system looks as follows:

- Simulation mode: Fullsystem.
- CPU: boot with KVM, switch to O3 (single core) for the benchmarks.
- Memory: 16 GB DDR3.

A modified Linux kernel, as described in Chapter 5, is used to support tracking of `UFFD`-registered regions by extending the Userfaultfd registration process and adding a platform device and driver.

Two variants of Userfaultfd are compared: the unmodified stock implementation and a UINTR-based implementation. In both cases, a memory region is reserved using `mmap`. The stock setup follows the procedure in Chapter 2, while the UINTR setup follows the configuration in Chapter 3. Each benchmark iterates over the reserved region and touches every page to trigger page faults. All page faults are then resolved using the `UFFDIO_ZEROPAGE ioctl`.

In the stock Userfaultfd version, the user-space handler is set up as a separate thread that polls for notifications. In contrast, the UINTR handler does not need to actively wait. Instead, notifications are delivered directly via user-interrupt. However, it relies on the extended struct introduced in Chapter 5 to access the faulting address and error code.

Latency is measured in CPU cycles using the `rdtsc` instruction since `CLOCK_MONOTONIC` is not reliable in gem5 due to the way simulated time progresses and how gem5 schedules events. Measurement points are placed at critical locations:

- right before accessing a page
- when entering a handler
- after returning from `ioctl`
- when the main thread resumes execution after the page fault has been resolved.

The first phase 'trigger' measures the time it takes from triggering a page fault to entering the fault handler. The next phase 'ioctl' contains the necessary preparations for the `ioctl` call, but the added cycle count from these steps is negligible compared to the cost of the `ioctl`. The last phase measures the duration of the return from the handler back to the main thread that was paused in the meantime.

To allow seamless measurements across context boundaries, and because `rdtsc` is not synchronised between CPU cores, the gem5 system is set up with a single core. This implies that the measured performance also depends on the simulator's scheduler: in particular, the time from triggering a page fault to entering the handler and the time from finishing the handler back to the main thread can vary depending on when the scheduler assigns execution to each context. In contrast, the time spent inside the handler itself is largely unaffected, since the handler is already running.

6.1.2 Analysis of Page Fault Results

Figure 6.1 compares the total CPU cycles required to handle page faults using both Userfaultfd implementations. The figure is divided into two subplots: absolute cycle

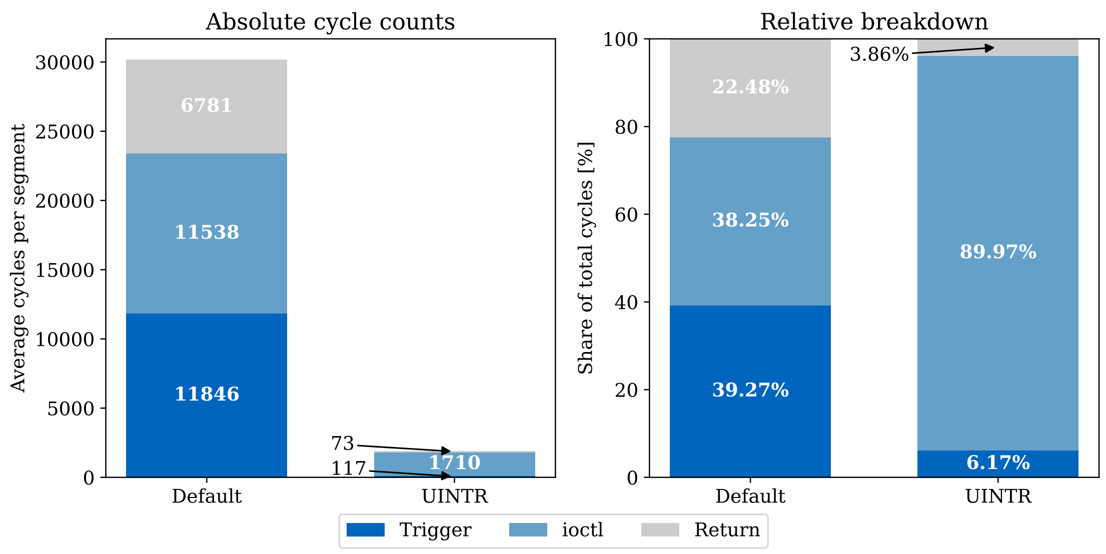

Figure 6.1: Page fault forwarding: absolute average cycles and percentage of each segment. Entry and return from the UINTR handler are a small fraction of the total cycle count.

counts and relative contribution of each phase to the total page fault latency. The phases correspond to the key measurement points introduced above.

The stock implementation exhibits a fairly balanced distribution of runtime across the three phases: the trigger phase consumes 11846, the `ioctl` call 11538 and the return phase 6781 cycles on average. This indicates that the overhead of actively polling for page fault events and context-switching to the handler thread is substantial, as more than a third of the total latency is incurred before the `ioctl` operation, and more than 22 % of the cycles are spent on returning to the main thread.

UINTR, on the other hand, dramatically reduces the trigger and return phases to 117 and 73 cycles on average, respectively. This demonstrates that forwarding page fault notifications directly via User-Interrupts bypasses the active polling overhead and eliminates most of the latency associated with context-switching between the faulting thread and the handler. The `ioctl` phase remains a major share of the total cost at 1710 cycles, but is still six to seven times faster than the same phase in the stock version.

The relative subplot highlights the shift in the UFFD path. In the stock Userfaultfd, `ioctl` makes up 38.2 % of the total latency, with trigger and return accounting for the remaining shares. With UINTR, however, the `ioctl` phase dominates with 90 % of the total duration, while trigger and return shrink down to 6.17 % and 3.86 %, respectively. This emphasises that UINTR shifts the dominant portion of latency from context management to the kernel's internal handling of the page fault. The user-level handler overhead becomes virtually negligible, and the total page fault handling time is determined primarily by the kernel's page allocation.

These results confirm that UINTR and directly forwarding interrupts to user-space remove the waiting and polling costs in the stock Userfaultfd implementation. It reduces the end-to-end latency of user-space page fault handling by an order of magnitude in the phases that are not directly contributing to fault resolution. Moreover, the relative contribution plot illustrates the remaining kernel-side cost, which offers a glimpse of where future optimisation effort could focus.

6.1.3 Detailed IOCTL Breakdown

Since page fault forwarding mainly aims to provide an alternative notification method, it is interesting to know where the large difference in the `ioctl` call comes from. To see what possible side effects page fault forwarding might have on the fault resolution, the `userfaultfd_zeropage` path is broken down into its main kernel actions:

- `copy_from_user`: copies input data (the `UFFDIO_ZEROPAGE` struct) from user-space into kernel space.
- `validate_range`: checks that the address range from the struct is valid and lies within the registered region.
- `mfill_zeropage`: allocates and maps a new physical page, initialising it to zeros.
- `put_user`: writes results back to user-space.
- `wake_userfault`: wake the faulting thread so it can resume execution after the page fault has been resolved.

This breakdown allows a detailed view of how latency is distributed to individual kernel operations, which helps identify where the UINTR-based forwarding differs most from the stock mechanism. To not affect the total duration of Userfaultfd, these cycles were measured separately. The results are shown in Figure 6.2.

Similar to the previous plot, this figure separates the main operations, with the first subplot showing the absolute cycle counts and the second subplot displaying the relative shares of total cycles for each operation. The lower total absolute cycle counts observed for both the stock and UINTR Userfaultfd implementations are partially due to the measurements not capturing any context switch between user and kernel space, nor any entry routine following the system call. Overall, the operations `copy_from_user`, `validate_range`, and `put_user` can be identified as quick and minor in both variants.

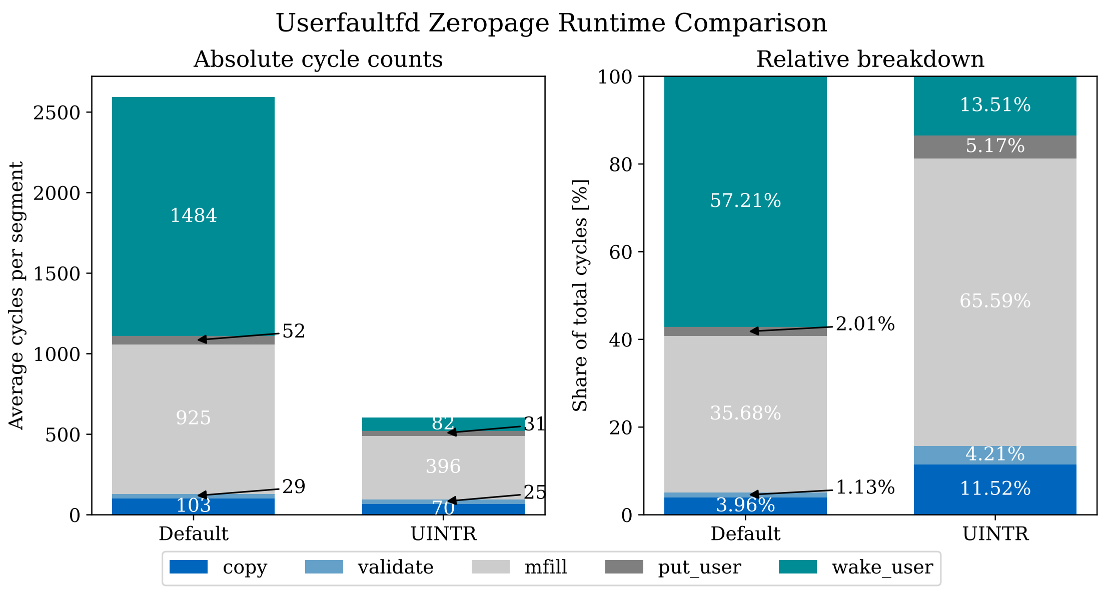

Figure 6.2: Detailed analysis of userfaultfd_zeropage. Allocating and zeroing a page and waking the paused main thread are the most expensive operations.

In the stock UFFD benchmark, these three operations have an average cycle cost of 184, which corresponds to 7.1 % of the total `userfaultfd_zeropage` runtime. For the UINTR version, the absolute number of cycles is similar; however, due to the lower total cycle count, these operations account for 20.9 % of the total cycles.

The largest discrepancy is observed in the `wake_userfault` method, which incurs an additional 1402 cycles for the unmodified Userfaultfd compared to the UINTR-based implementation. An analysis of this method clarifies the source of the higher cycle count: when a page fault occurs for a UFFD-backed address, the CPU traps and the kernel begins the fault-handling routine. As explained in Chapter 2, the kernel sets up a notification for user-space and blocks the faulting thread. To track blocked threads, the kernel places a pointer to the blocked thread in a `waitqueue`. Later, when the fault is resolved via the `ioctl`, the kernel checks whether any thread needs to be unblocked. In the stock Userfaultfd case, this check returns true, so the kernel must acquire a lock and iterate over the `waitqueue` to wake the appropriate thread.

In contrast, in the UINTR version, there is no kernel involvement after the page fault is first triggered, and the kernel does not block the faulting thread. As a result, the wake-up process in the `wake_userfault` method is skipped entirely.

As for `mfill_zeropage`, the benchmark shows it executes slightly faster under page fault forwarding. However, the reason for this difference cannot be attributed to any operation inside `mfill_zeropage` itself, since it does not block, interact with `waitqueues`, or wake threads. Similarly, the differences observed between measuring the `ioctl` duration from inside versus outside the kernel cannot be fully explained by the kernel operations alone. They likely include additional, unmeasured overheads from context switches, kernel entry/exit, and bookkeeping in the stock Userfaultfd path, which are largely bypassed in the UINTR path. Therefore, while the measurements clearly show lower cycles for UINTR, the precise distribution of overhead contributing to these differences cannot be completely isolated from the observed data.

6.2 BFS

6.2.1 Setup and Methodology

The BFS benchmarks aim to evaluate whether the proposed design for page fault forwarding offers the desired benefits, or if the speedup from using UINTRs for delivery is too insignificant and cancelled out or outweighed by the overhead from the forwarding mechanism. The benchmark is adapted from the UMap project [ref], but it does not use the UMap library. Breadth-First Search (BFS) is chosen because it has irregular memory access patterns and frequently touches pages that are not yet resident in memory. The graph used in this benchmark contains 4,194,303 vertices and 134,217,728 edges and is mapped into the process address space at runtime. Three page fault handling strategies are assessed:

- `Mmap`-only (baseline): This setup represents default system behaviour and provides a reference runtime for page fault handling without user-space intervention.
- Stock Userfaultfd: The memory is mapped with `MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORSERVE` to ensure that pages are initially not backed, private to the process, and do not reserve swap space. Pages are only created after the handler resolves the fault. When a fault occurs, the handler reads the missing page from the backing file and copies it into the faulting address using `UFFDIO_COPY`. This setup makes the handler solely responsible for populating pages.
- UINTR-based page fault forwarding: The same `mmap` flags and page-in operation as stock UFFD are used to resolve page faults.

For each setup, the total BFS runtime is captured using `rdtsc`. Furthermore, stock Userfaultfd and page fault forwarding record the time spent in the handler to obtain the percentage of time spent on fault handling.

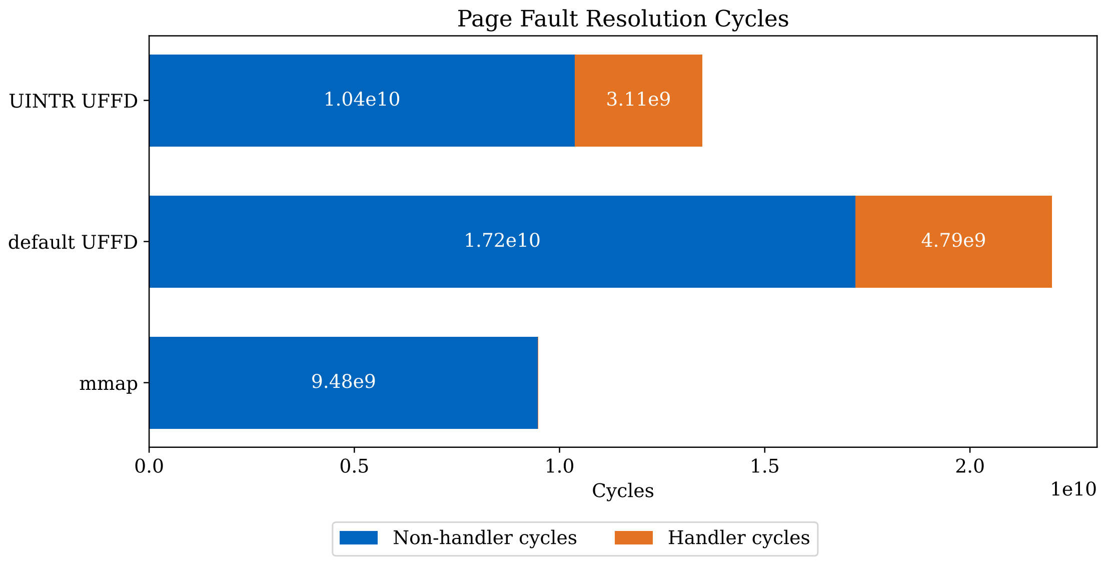

Figure 6.3: BFS runtime analysis: Runtime of the BFS benchmarks measured in CPU cycles, with `mmap` as a baseline. UINTR-based page fault forwarding significantly reduces

6.2.2 Analysis of BFS Results

The results of the three BFS runs are depicted in Figure 6.3. It shows the total CPU cycles consumed during BFS traversal, divided into cycles spent in the page fault handler and cycles spent in the rest of the program for default and UINTR-based page fault forwarding. The baseline `mmap`-only version completes the traversal in 9.48×10^9 cycles. Stock Userfaultfd records 4.79×10^9 cycles inside the handler and 1.72×10^{10} cycles outside, resulting in a total of 2.20×10^{10} cycles. UINTR-based page fault forwarding reduces handler cycles to 3.11×10^9 and non-handler cycles to 1.04×10^{10}, for a total of 1.35×10^{10} cycles. This demonstrates a clear reduction in both handler and total runtime. The relative cycle plot has been omitted because the fraction of time spent in and outside the handler is nearly the same due to the handler and non-handler cycles being reduced almost proportionally.

The BFS benchmark allows a quantification of the speedup introduced by UINTR-based page fault forwarding. Three metrics are considered: the speedup of time spent in the page fault handler, the speedup of the remaining program outside the handler, and the resulting overall speedup. UINTR reduces cycles spent in the handler by about 1.54 times. A higher speedup is achieved outside the handler: here, the improvement is a speedup by roughly 1.65 times. The resulting total decrease in cycle count is therefore approximately 1.63 times from stock Userfaultfd to UINTR-based.

The handler speedup reflects the side effects of the page fault forwarding mechanism on the operations required to resolve a page fault. With UINTR, the faulting thread does not block, which avoids wake-up operations and the associated context switch. Additionally, because no thread is placed on a `waitqueue`, the kernel skips the bookkeeping normally required for managing blocked threads. These factors together make the `ioctl` call faster, consistent with the findings in Subsection 6.1.2, and contribute to the measurable reduction in cycles spent inside the handler.

The speedup outside the handler can be attributed to the more efficient delivery method provided by page fault forwarding. Since notifications are delivered directly to user-space, the system avoids context switches between the main program and fault handler, removes the need for polling or blocking as in the stock Userfaultfd path, and eliminates kernel involvement during delivery.

Overall, the BFS benchmarks confirm that UINTR-based page fault forwarding reduces the latency of the notification delivery and accelerates the resolution of page faults using UFFD and `ioctl`. These improvements are consistent with the page fault experiments. Thus, programs with frequent page faults or requiring custom resolution/page-in are likely to benefit from the forwarding mechanism.

An additional benefit of page fault forwarding with UINTR is that it removes the need for a dedicated polling thread. Since the UINTR handler can execute on the same core as the main program, no extra core is occupied for fault handling. This frees computational resources and allows the available cores to be used more effectively for application work.

6.3 Discussion of Research questions

6.3.1 Latency Improvement

The page fault results demonstrate that UINTR-based page fault forwarding reduces the latency of user-space fault handling compared to the stock Userfaultfd path. As a bonus, a positive side effect can be observed in the handler: here, cycle counts were consistently lower with UINTR, primarily due to the elimination of thread wake-ups, context switching, and bookkeeping of blocked threads. Combining the side effects and the more efficient delivery based on xUI's interrupt forwarding, page fault forwarding achieves its intended goal of lowering the overhead of fault resolution.

6.3.2 Real-world Applicability

The BFS benchmark shows that these improvements carry over into an application-level workload. While BFS still spends a large fraction of its time inside the handler, the faster delivery and resolution of page faults result in an overall runtime improvement. This indicates that applications with frequent page faults or the need for custom page-in logic can benefit from the mechanism. The results, therefore, suggest that page fault forwarding is not only effective in synthetic benchmarks but can also provide measurable speedups in realistic use cases.

7 Related Work

Research relevant to this thesis can be divided into two main categories. The first category consists of memory management frameworks that demonstrate how applications can manage virtual memory more flexibly, using kernel interfaces for fault notification and resolution. The second category focuses on kernel-bypass and low-latency event delivery mechanisms, which aim to reduce or eliminate kernel involvement for performance-critical operations, such as I/O.

7.1 Memory Management Frameworks

This section covers frameworks that provide applications with more control over memory allocation and page fault handling, while still relying on the kernel for fault delivery and resolution.

uMMAP-IO [Riv+19] employs the `SIGSEGV` signal to detect page faults in user-space by marking memory regions as PROT_NONE. This allows applications to handle faults asynchronously and gives fine-grained control over memory management. While effective, the signal-based approach introduces higher latency and, thus, highlights the need for more efficient fault-forwarding mechanisms in user-space, which is the focus of this thesis.

UMap [Pen+22] uses Linux's Userfaultfd system call to register memory regions and receive notifications on page faults. Its design offloads fault handling to user-space more efficiently than signal-based methods. This makes it directly relevant as a precursor to page-fault forwarding techniques that minimise kernel involvement.

LightSwap [Zho+23] moves page fault handling completely into user-space and avoids the kernel's involvement in reading or writing memory pages. It handles faults using lightweight threads so that application threads do not block, which reduces page fault latency compared to kernel-based swapping. This shows that user-space mechanisms can improve fault handling efficiency and motivates hardware-assisted forwarding approaches.

These papers illustrate different approaches for user-space page-fault handling. They present strategies for detecting and managing faults entirely outside the kernel, and they highlight both the potential performance benefits and the limitations of existing approaches. The examination of signal-based methods, Userfaultfd notifications, and user-space swapping with lightweight threading establishes the foundation for forwarding page faults directly to user-space and motivates the design choices and goals of the present thesis.

7.2 Kernel-Bypass Frameworks

For kernel-bypass frameworks, the subsequent works show how moving I/O operations and network handling to user-space can reduce latency and increase throughput. While these frameworks target I/O, they illustrate the benefits of user-space control over hardware resources.

DPDK [CS18] provides a set of libraries and drivers that allow applications to process network packets directly in user-space, bypassing the kernel's network stack. This reduces context switching and kernel overhead, achieving lower latency and higher throughput for high-performance networking applications.

SPDK [Yan+17] implements a similar approach for storage devices, giving applications direct access to NVMe and other storage hardware. By avoiding the kernel I/O path, SPDK enables extremely low-latency storage operations, which makes it suitable for latency-sensitive workloads such as databases or real-time analytics.

Although these frameworks focus on I/O rather than memory management, they exemplify principles of low-latency user-space control. The present thesis applies a similar principle to memory management by forwarding page faults directly to user-space via UINTR, achieving low-latency fault handling without relying on polling.

8 Summary and Conclusion

This thesis examined page fault forwarding to user-space with support from Intel User-Interrupts [Cor24], Userfaultfd [The22], and the xUI interrupt forwarding architecture [Ayd+25]. The goal was to cut page fault latency and kernel involvement while keeping compatibility with existing Linux mechanisms. The work designed a forwarding path that sends selected page faults directly to a registered user-space thread if the faulting address lies in a UFFD region and interrupt forwarding is enabled on the core. The design keeps ordinary kernel handling for all other faults.

8.1 Overall work

High-level approach and design. The mechanism integrates xUI, UINTR, and UFFD. Applications register memory regions and a UINTR handler. On a fault, the system checks whether the address is inside a registered region. If that is the case, it forwards the fault to user-space. Otherwise, the kernel handles it. The forwarding path delivers the faulting address and page fault error code to the handler so it can resolve the fault via `ioctl` call.

Implementation. The prototype runs in gem5 with an out-of-order (O3) CPU. It extends xUI with UINTR-based delivery for page faults, adds a MMIO device and Linux kernel driver to synchronise UFFD regions with the simulator, and extends the UINTR frame so the handler can read the faulting address and error code directly from the stack.

Evaluation. The study compares stock UFFD with the UINTR-based forwarding path in two settings: a page fault benchmark and a BFS workload with irregular memory access. For measurement, `rdtsc` was used to obtain the CPU cycle count.

8.2 Key findings

Latency Reduction. In the page fault benchmark, stock UFFD shows on average 11,846 cycles in the trigger phase, 11,538 cycles in the `ioctl` phase, and 6,781 cycles in the return phase. The UINTR-based implementation reduces the average cycle counts to 117 cycles, 1,710 cycles, and 73 cycles, respectively. As a result, `ioctl` becomes roughly

90% of the remaining cost, and user-space management overhead becomes negligible in comparison.

Breakdown of ioctl. Minor work such as `copy_from_user`, `validate_range`, and `put_user` stays cheap in both paths. The large gap comes from `wake_userfault`: the stock path pays about 1,402 cycles per fault for `waitqueue` management and wake-ups. The UINTR path avoids this step because the kernel does not block the faulting thread.

Application level gains. In BFS, the baseline `mmap`-only run finishes in 9.48×10^9 cycles. Stock UFFD needs 2.20×10^{10} cycles in total (handler 4.79×10^9, non-handler 1.72×10^{10}). The UINTR path needs 1.35×10^{10} cycles from start to finish (handler 3.11×10^9, non-handler 1.04×10^{10}). This is an approximate total speedup of 1.63 times over stock UFFD (inside the handler by 1.54, outside the handler by 1.65). The reduction outside the handler reflects faster delivery with no polling or wake-ups. This indicates that workloads with frequent page faults or custom page-in logic can benefit in practice.

Resource efficiency. The page fault forwarding path removes the dedicated polling thread. The handler can execute on the same core as the application, which frees a core in setups that otherwise dedicate one to polling.

Impact. The results show that hardware-assisted forwarding can shift the critical path of user-space fault handling away from user-level orchestration and towards the kernel's page allocation work. This reduces end-to-end latency, improves application runtime, and opens a path for further gains through kernel-internal optimisations or additional hardware support. The prototype also establishes a reusable foundation in gem5 for research on user-level delivery of other events.

8.3 Source Code

Prototype implementations developed in this thesis include:

- gem5 modifications for UINTR-based page fault delivery and the MMIO region tracker
- Linux kernel modifications for the platform device and driver, and UFFD registration hook
- User-space benchmarks

The source code is provided at https://github.com/TUM-DSE/uintr.

8.4 Conclusion

This thesis shows the feasibility and effectiveness of forwarding selected page faults directly to user-space with UINTR. By removing polling and wake-ups, the mechanism lowers per-fault latency, reduces context switches, and improves overall runtime in applications with irregular memory access. At the same time, it remains compatible with the standard kernel path and doesn't change existing UFFD and UINTR usage. Building on this prototype and its evaluation, the next chapter outlines future directions.

9 Future Work

The evaluation demonstrates that UINTR-based page fault forwarding can significantly reduce latency in both microbenchmarks and real workloads such as BFS. While the current implementation already achieves substantial improvements, several directions remain to further enhance performance, scalability, and flexibility:

Forwarding to Other Cores

In the current setup, the UINTR handler executes on the same core as the main program and, thus, benefits from avoiding polling threads and context switches. For workloads with high fault rates or with dedicated page fault handling threads, forwarding faults to other cores could allow better load distribution and enable parallel fault handling. Investigating policies for optimal core assignment and balancing would extend the applicability of the mechanism to multi-core workloads.

Immediate Fault Resolution

The current design relies on an `ioctl` call for page fault resolution, which contributes significantly to the total latency in the handler. A mechanism to resolve page faults immediately, without having to call `ioctl`, could eliminate an additional context switch and reduce overhead further by removing kernel involvement altogether from the page fault handling routine. The method for page fault resolution could be provided as a configurable option during Userfaultfd registration so that applications with strict latency requirements can take advantage of it without being forced to change the kernel behaviour for other workloads.

Batch Processing of Page Faults

Many applications trigger bursts of page faults in rapid succession. Handling each fault individually incurs per-fault overhead for delivery and handler invocation. Introducing a buffer to collect multiple faults and process them in a single batch could reduce this overhead and improve the efficiency of both the handler and the overall program.

In the stock Userfaultfd implementation, batch processing is naturally supported due to the use of internal queues: multiple pending faults can be queued and processed sequentially by the handler thread, amortising the cost of context switches and `ioctl` calls. For the UINTR-based mechanism, a similar approach could be implemented, either by buffering multiple UINTR notifications or by collecting metadata for several faults before invoking the user-level handler. This would preserve the low-latency benefits of direct delivery while further reducing per-fault processing overhead in scenarios with very frequent page faults.

Collectively, these future directions aim to increase the applicability, performance, and flexibility of page fault forwarding. They build on the evaluation insights, such as the dominance of `ioctl` cycles and the benefits of eliminating polling, with the goal of making page fault handling more effective in latency-critical and memory-intensive applications. Furthermore, these extensions could enable the development of user-space page fault handling libraries or frameworks, similar in concept to UMap [Pen+22]. This would allow applications to implement custom page-in policies, batching strategies, or specialised memory management routines without modifying the kernel.

Abbreviations

UINTR	User-Interrupt
UFFD	Userfaultfd
IPC	Inter-Process Communication
UPID	User Posted Interrupt Descriptor
UITT	User Interrupt Target Table
LAPIC	Local Advanced Programmable Interrupt Controller
MSR	Model-Specific Register
MMU	Memory Management Unit
MTRR	Memory Type Range Registers
TLB	Translation Lookaside Buffer
MMIO	Memory-Mapped I/O
UIF	User Interrupt Flag
BFS	Breadth-First Search
FIFO	First In - First Out

List of Figures

List of Tables

Bibliography

[Ayd+25] B. Aydogmus, L. Guo, D. Zuberi, T. Garfinkel, D. Tullsen, A. Ousterhout, and K. Taram. "Extended User Interrupts (xUI): Fast and Flexible Notification without Polling." In: New York, NY, USA: Association for Computing Machinery, 2025, pp. 373–389. ISBN: 9798400710797.

[BC05] D. Bovet and M. Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly Media, 2005. ISBN: 9780596554910.

[Cor24] I. Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Accessed on 2 March 2025. 2024.

[CS18] R. Chen and G. Sun. "A Survey of Kernel-Bypass Techniques in Network Stack." In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. CSAI '18. Shenzhen, China: Association for Computing Machinery, 2018, pp. 474–477. ISBN: 9781450366069.

[Hei+18] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya. "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges." In: *ACM Comput. Surv.* 51.3 (June 2018). ISSN: 0360-0300.

[Hua+25] 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?" In: *Proc. ACM Manag. Data* 3.3 (June 2025).

[Jia+24] 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*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 265–279. ISBN: 9798400712517.

[Li+24] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou. "LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling." In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2024, pp. 922–936.

[MGR24] S. Mishra, B. N. Gohil, and S. Ray. "A survey on Persistent Memory indexes: Recent advances, challenges and opportunities." In: *Journal of Systems Architecture* 151 (2024), p. 103140. ISSN: 1383-7621. DOI: `https://doi.org/10.1016/j.sysarc.2024.103140`.

[Pen+22] I. B. Peng, M. B. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce. "Enabling Scalable and Extensible Memory-Mapped Datastores in Userspace." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 866–877.

[Riv+19] S. Rivas-Gomez, A. Fanfarillo, S. Valat, C. Laferriere, P. Couvee, S. Narasimhamurthy, and S. Markidis. "uMMAP-IO: User-Level Memory-Mapped I/O for HPC." In: *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2019, pp. 363–372.

[The22] The Linux Programmer's Manual. *userfaultfd(2) — Linux manual page*. Accessed: 2025-05-23. 2022.

[Yan+17] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. "SPDK: A Development Kit to Build High Performance Storage Applications." In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017, pp. 154–161.

[Zho+23] K. Zhong, W. Cui, X. Chen, Q. Li, Z. Yang, Y. Lu, X. Yan, S. Luo, Q. Yuan, and K. Huang. "Revisiting Swapping in User-Space With Lightweight Threading." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.11 (2023), pp. 4205–4218.