Extended User Interrupts (xUI): Fast and Flexible Notification without Polling

Berk Aydogmus
Purdue University
West Lafayette, USA

Linsong Guo
UC San Diego
La Jolla, USA

Danial Zuberi
UC San Diego
La Jolla, USA

Tal Garfinkel
UC San Diego
La Jolla, USA

Dean Tullsen
UC San Diego
La Jolla, USA

Amy Ousterhout
UC San Diego
La Jolla, USA

Kazem Taram
Purdue University
West Lafayette, USA

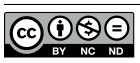
Abstract

Extended user interrupts (xUI) is a set of processor extensions that builds on Intel's UIPI model of user interrupts, for enhanced performance and flexibility. This paper deconstructs Intel's current UIPI design through analysis and measurement, and uses this to develop an accurate model of its timing. It then introduces four novel enhancements to user interrupts: tracked interrupts, hardware safepoints, a kernel bypass timer, and interrupt forwarding. xUI is modeled in gem5 simulation and evaluated on three use cases – preemption in a high-performance user-level runtime, IO notification in a layer3 router using DPDK, and IO notification in a synthetic workload with a streaming accelerator modeled after Intel's Data Streaming Accelerator. This work shows that xUI offers the performance of shared memory polling with the efficiency of asynchronous notification.

CCS Concepts: • **Computer systems organization** → **Pipeline computing**; **Processors and memory architectures**; • **Hardware** → Networking hardware.

Keywords: Interrupt, User Space, Preemption, Processor Architecture

ACM Reference Format:
Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. 2025. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3716028



ASPLOS '25, Rotterdam, Netherlands.

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716028

1 Introduction

With the rise of language-level concurrency and kernel bypass interfaces, user-level runtimes are increasingly responsible for tasks that once belonged to the OS—such as preemption, synchronization, and scheduling (CPU and IO).

User-level threads, goroutines (Go), and coroutines (C++) have allowed data center applications to scale to millions of concurrent requests [17, 60] and support microsecond timescales [26, 37, 48, 52] by avoiding the high cost of kernel threads. This has increasingly shifted responsibility for CPU scheduling to user-level runtimes. Similarly, kernel-bypass interfaces such as DPDK [1] and SPDK [9] achieve high performance I/O at microsecond timescales by bypassing the kernel, shifting IO scheduling to user level.

However, user-level runtimes are missing a key support that operating systems rely on to coordinate these activities—the interrupt system. The interrupt system delivers low-latency asynchronous notifications from devices, cores (IPIs), and timers. Asynchronous notification *minimizes latency*, as it immediately changes the control flow on the receiving core, and *maximizes efficiency*, because it doesn't require the receiver to waste cycles polling for notifications.

Without this support, user-level runtimes are forced to rely on expensive OS interfaces, or fall back to shared memory polling that trades poor efficiency for lower latencies and higher throughput [29]. This translates directly into inefficiencies and functional limitations at user level.

For example, preemption is essential for enabling precise control over scheduling. In server workloads, it can bound tail latency and increase useful throughput by mitigating head-of-line-blocking [37, 61], and it is mandatory for running untrusted code in multi-tenant settings such as serverless edge platforms [21, 59]. However, many programming languages and high-performance runtimes don't support

preemption [48, 52], limit preemption frequency (e.g. Go preempts every 10ms), or will not preempt loops [6, 7, 10] due to the high and unpredictable overheads that current user-level timing and notification mechanisms impose.

Next, while kernel bypass interfaces are increasingly popular for accessing high-performance network and storage (e.g., ultra-low latency (ULL) flash and NVMe), the only way to receive the microsecond-scale notifications these devices deliver is to poll. This can waste significant cycles and energy as cores busy-spin [29]. It also scales poorly, as each queue being polled burns an additional cache line, and processors offer no way to idle (e.g. `mwait`) on more than a single queue. The growing use of specialized accelerators [63] (e.g., DSA [42], QAT [33]) will only make this more challenging.

Other use cases where low-cost notifications would enable more efficient implementations include IPC notifications, syncing changes to shared data structures, and processing real-time tracing events.

Intel's recently released user-level interprocessor interrupts (UIPI) offer a promising starting point for bringing interrupt support to user space. For example, it offers significant benefits for implementing preemption with low, predictable overheads [25, 35, 44]. However, our in-depth analysis of its design and implementation (§3) and exploration of existing user-level notification mechanisms (§2) reveal significant limitations in performance, precision, and support for diverse notification types (i.e. timers and devices).

To overcome these limitations, we present xUI, a set of processor extensions that build on UIPI to meet the needs of user-level runtimes. xUI offers four key capabilities: *tracked-interrupts* (§4.2)—reduce interrupt delivery overhead by 3x-9x vs. UIPI—closing the performance gap between interrupts and memory-based notification (polling); *hardware safepoints* (§4.4)—offer precise control over where interrupts are delivered at near zero cost, reconciling the tension between interrupts and precise garbage collection; the *kernel bypass timer* (§4.3)—enables low-cost user-space timers, eliminating the need for expensive OS-based timers, or dedicating cores to polling high precision counters; *interrupt-forwarding* (§4.5)—extends interrupt routing, allowing devices to send interrupts to user-level threads.

We simulate xUI (and UIPI) in gem5 (§5); and evaluate its performance for preemption and I/O notification in several high-performance workloads (§6), including: serving database requests with RocksDB on Aspen [30]—a high-performance user-level runtime; routing with l3fwd [19]—a DPDK based layer 3 router; and offloading with a simulated streaming accelerator based on Intel's DSA [42].

We find using xUI instead of UIPI for preemption in Aspen improves RocksDB throughput by 10% (§6.2.1), and eliminates the need to waste core(s) to provide high precision timer(s) (§6.1). We find using xUI for IO notification in l3fwd achieves the same responsiveness (latency) as busy spinning, while significantly improving efficiency. For example, at 40% load, xUI reduces CPU consumption by 45% compared to busy spinning (§6.2.2). We observe similar efficiency gains using xUI for IO notification with simulated DSA; for example, at 50K IPOS (20μs average request latency), xUI maintains the same responsiveness as busy spinning while incurring negligible CPU overhead (§6.2.3).

2 Limitations of User-Level Notification

Current options for asynchronous notification and timing at user level have significant limitations, especially for high-performance applications.

Signals: high overheads, imprecise. Signals are the only option for asynchronous notification on most systems at user level today. Like interrupts, signals are free when no events occur, but are orders of magnitude more expensive than polling when events do arrive, making them unsuitable for high-performance applications.

For example, we measured the impact of signal delivery on some common benchmarks (e.g., linpack2, base64 encoding) and found each signal imposed roughly 2.4 μs of overhead at 2 GHz (without KPTI). About 1.4 μs of this comes from OS context switch overheads. The remainder is mostly due to added branch mispredictions and cache misses caused by contention with the kernel signal-handling code.

This is expensive enough to make signals a impractical for preemption and IO notification in high-performance applications, where requests arrive on a microsecond timescale [29, 30, 34], as signal delivery imposes a significant tax on throughput. Even in common server workloads where requests are often in the 100s of microseconds or less [53, 56], they are still a noticeable source of overhead.

Signals are also imprecise, that is, they change control flow at arbitrary points in the receiver code. This is a problem for runtimes that rely on precise garbage collection (e.g., most Java and .NET implementations), as precise GCs rely on stackmaps. Stackmaps are metadata generated at compile time that tell the GC which stack elements are pointers—each map is valid only for a particular *safepoint* [45] in program code. Thus, if a thread is being preempted somewhere other than a safepoint, a GC cannot garbage collect it, as it has no way of identifying which objects that thread can reference.

In recent years, Go developed a unique solution to this problem by moving from precise GC to a hybrid approach [10] that uses conservative GC for the most recent stack frame, and precise GC for the rest. Unfortunately, this is not a solution for most runtimes, as moving garbage collectors—which are quite popular—modify pointers on the stack [36]. Thus, they cannot tolerate the imprecision of conservative GC.

Polling: unpredictable, inefficient, unscalable. User-level notification can also be implemented by polling a variable in shared memory. When polling, each negative check is quite cheap (load with an L1 cache hit, and a correctly predicted branch), and each notification is still relatively

cheap (load with an L2 cache miss, and a likely mispredicted branch), thus polling can support both low latency and high bandwidth notification. Polling is also precise, as a programmer or compiler has total control over where polling occurs. However, while signal costs scale in the number of notifications arriving, polling costs scale in the number of checks being performed, and it is hard to avoid wasting significant cycles when no notifications are pending.

For example, polling is a common way to implement preemption in a programming language. To implement this, the compiler adds instrumentation that polls a shared variable at every function entry point and loop back-edge, ensuring a thread will yield, regardless of control flow. Unfortunately, polling overheads can be quite high and workload-dependent [30], as even a simple check can be quite expensive in tight loops.

To avoid this, Go did not preempt loops for its first ten years. Instead, it required developers to insert explicit calls to `runtime.Gosched()` (the scheduler) in loops when necessary. However, this was fragile, leading to hard-to-debug slowdowns and program freezes [10] when overlooked. When the Go team explored mitigating this by adding checks to loops, they found it slowed down programs by a geomean of roughly 7%, and in the worst case, by up to 96% [7].

Similarly, we measured the cost of preemption in Wasmtime, a WebAssembly runtime used to run serverless applications by Fastly, Shopify, Microsoft, etc. that relies on polling for preemption. On some common benchmarks containing tight loops or short functions (e.g., linpack2 [18]), we found it imposed slowdowns of up to 50%. Unsurprisingly, when Java decided to add support for virtual threads in recent years [6], they decided not to support loop preemption.

Polling for IO poses its own set of challenges. As Golestani et al., observed "you can't always spin to win" [29]. Busy spinning tends to waste significant cycles because I/O is unpredictable, and when I/O events are not arriving, the core is wasted. Polling also scales poorly as overheads increase linearly with the number of queues being polled, and each one burns a cache line. Wasting cycles translate to energy inefficiency, and the primary tool that processors offer to mitigate this cost (`mwait` and its equivalents), only works with a single queue. Less frequent polling can reduce these costs, but this forces a trade-off between overhead and responsiveness. In user-level runtimes, periodic polling is also messy because it becomes intertwined with scheduling, i.e., one can't poll when a user thread is running.

Timers: expensive and complex. Timers enable many scheduling tasks including preemption, periodic device polling, and synchronization timeouts. Unfortunately, user-level runtimes have limited options for microsecond-scale timing. Hardware timers are hidden behind expensive OS interfaces that rely on signals (`setitimer()`, `timer_create()`) and system calls (`nanosleep()`,`select()`) for notification.

In the absence of direct access to efficient hardware timers, user-level runtimes [7, 30, 44, 54] implementing threads often designate one kernel thread (core) to act as a timer, and notify other kernel threads when they need to be preempted, either through signals, shared memory, and more recently UIPI. Unfortunately, OS overheads can add up quite quickly for the timer thread (see Figure 6). In higher performance systems [30, 34, 44] it's not uncommon to implement timer thread(s) by busy spinning on a high-precision timer (`rdtsc`), burning the entire core.

UIPI: more expensive than polling, imprecise, narrow. Receiving a notification with UIPI is roughly 3x-5x cheaper than signals— roughly 2800 cycles vs. 600-900 cycles on our Saphire Rapids processor. Unfortunately, UIPI is still roughly 6x-9x slower than memory-based notifications, which are roughly 100 cycles.

The gap between UIPI and polling overheads will increase in future processors due to the growing size of speculation windows. To wit–the pipeline flush induced by UIPI is a significant source of overhead (§ 3.5). As the number of in-flight instructions in future processors continues to increase, this will become more expensive. This trend will also hurt future polling performance, as the flush to handle branch mispredictions becomes more expensive.

UIPI is also imprecise, similar to signals, thus it does not play well with precise GC. Finally, UIPI only supports a narrow subset of the features APICs offer, i.e., no device interrupts or timer interrupts, which limits its usefulness. Taken together, this makes UIPI a useful tool for some situations [25, 30, 35, 44], but still quite far from a substitute for all the interrupt system offers, and certainly not a viable replacement for polling in high-performance systems.

3 Characterizing Intel UIPI

Here, we provide the first in-depth analysis of Intel UIPI by reverse engineering its microarchitectural details. We discuss how xUI builds on this in section 4.

3.1 UIPI Overview

User interprocessor interrupt (UIPI) support was recently introduced by Intel in their Sapphire Rapids processors.[1] UIPI allows IPIs to be sent directly from one kernel thread (e.g. pthread) to another at user level, avoiding the context-switch overheads of signals.

There are several challenges to enabling user-level IPIs on existing hardware:

Adding access control. Access control is necessary to prevent threads from sending IPIs to other non-consenting threads. UIPI delegates this task to the kernel via a new hardware interface—programmed through MSRs and in-memory

[1]User interrupts have been shipping in both server and consumer class chips since Fall 2024 (e.g., Grand Rapids, Sierra Forest, Grand Ridge, Arrow Lake, Lunar Lake) [32].

Table 1. User Posted Interrupt Descriptor (UPID) Fields.

Name	Description	Bit Range
Outstanding Notification (ON)	if notification is outstanding for one or more UIs	0:0
Suppressed Notification (SN)	if the senders should avoid sending a notification	1:1
Notification Vector (NV)	the interrupt vector on which the UI should be sent	23:16
Notification Destination (NDST)	APICID of the core this thread running on	63:32
Posted Interrupt Requests (PIR)	interrupts that have been posted to this thread	127:64

tables—called the *User Interrupt Target Table (UITT)*. A UITT is a per-process table that specifies which other threads a process (and all of its threads) can send UIPIs to. At a high level, each entry is a tuple $\langle UPID, vector \rangle$—where UPID (User Posted Interrupt Descriptor) is a thread-specific, in-memory descriptor. The presence of a UPID pointer in the UITT of a process implicitly grants it permission to send interrupts to the thread associated with that UPID. The kernel lets authorized processes set up these mappings with system calls. As we discuss later, the UITT and its UPID entries are also essential for UIPI routing.

Routing interrupts at user level. The interrupt system is responsible for routing messages from sources (devices, cores, timers) to destinations. Destinations are cores (addressed by APICID), and messages are 8-bit values (vectors) that uniquely identify the cause of the interrupt. By design, routing is quite static—APICIDs are typically assigned to cores at startup time, and rarely change.

To extend x84-64 with user interrupts, there were several limitations of the existing interrupt routing scheme to overcome: (1) there was no notion of addressing threads—which are dynamically created, destroyed, and migrated between cores. (2) the x86 per-core (APICID) vector space is quite small (256 possible values), and parts of it are already dedicated to other purposes—so sharing this space with user IPIs would be quite limiting. (3) cores are always available, but threads are not; when an interrupt arrives, the destination thread may be context-switched out.

To address these limitations, UIPI creates its own orthogonal virtual namespace (thread IDs) and vector space (a 6-bit user vector, or UV). To deliver interrupts in this new namespace, it introduces two new types of data structures that are shared in memory between cores—the per-process UITTs and per-thread UPIDs previously mentioned. Next, we look at these data structures in more detail and how UIPI uses them to send and receive interrupts.

3.2 Sending and Receiving UIPIs

At a high level, when a UIPI is sent, the sending core looks up the UPID of the destination thread in the UITT, modifies the UPID for the destination thread, and then sends a conventional IPI to the receiving core to notify it of a pending UIPI. The receiving core then uses the state in the UPID to deliver the request.

Expanding on this, we begin by noting that each thread that can receive UIPI's has an associated UPID that is allocated by the kernel. Each UPID is a `struct` shared in memory among all cores. We will refer to its fields, shown in Table 1, in our description below.

To start, before a UIPI can be sent, a user space program asks the kernel to set up a route [5]. On the receiving thread, the `register_handler(...)` system call allocates a UPID, and stores a pointer to a user-level interrupt handler that will be invoked when a UIPI is received. On the sending process, the `register_sender(...)` system call allocates a new entry in the UITT, and stores a pointer there to the appropriate UPID and a user-supplied vector.

User interrupts are sent using the new `senduipi` instruction, which takes one operand, an index into the UITT. To send a UIPI, the sending core looks up the UPID and vector in the UITT, sets the bit in the PIR field of the UPID to indicate which vector was posted, and sets the ON-bit indicating an interrupt is pending. It then sends a conventional IPI to the core where the thread is running (the address in NDST), with the vector (NV) to indicate that there is a pending UIPI.

When this IPI arrives, the receiver core compares the vector it received with a field of an MSR (UINV) that stores the vector for UIPI notifications. If they match, the receiver then consults the PIR field of the UPID of the current thread to determine if the UIPI should be delivered to the current thread or if it needs to be handled by the OS.

If the UIPI is destined for the currently running thread (*fast path*), it delivers it by invoking the handler registered by the receiver, which returns using the new `uiret` instruction when it completes. Otherwise, it is delivered later by the kernel on the *slow path*. On the slow path, when the kernel resumes the thread an interrupt was destined for, it will repost the captured UIPI as a self-UIPI through the local APIC.

However, it might slow down other programs running on the receiver core if we frequently receive and process interrupts that are destined for out-of-context processes (slow path). To mitigate this, UIPI includes a "suppressed notification" (SN) bit field in the UPID, indicating when the receiver is not ready, thus preventing the sender from continuing to send interrupts to that thread. When a thread gets context switched out, the kernel sets the SN bit, halting further UIPIs from sender threads.

Threads can also manually toggle interrupt delivery using the new `clui` and `stui` (clear and set user interrupt flag, respectively), which work similarly to their kernel-mode counterparts, and use `testui` to query if interrupts are blocked.

Finally, a receiver thread might migrate from one physical core to another. Thus, a sending core must correctly determine which destination core it should send the IPI to. To solve this, the sending core reads the NDST field of the UPID that stores the APICID of the core the thread is currently

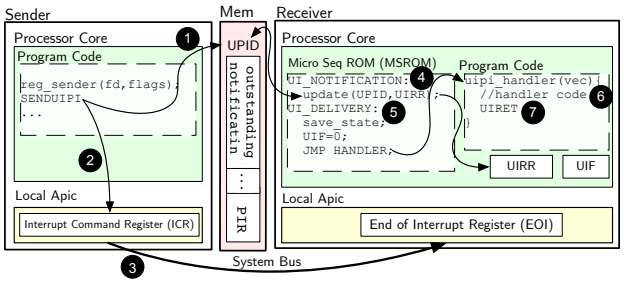


Figure 1. Architecture View of UIPI: *UIPI is largely implemented in microcode. The sender communicates the UIPI vector through shared memory (UPID). Steps described in §3.3.*

running on. Thus, to migrate a thread to a different core, the OS simply updates this field.

3.3 Architecture View

To support our analysis in 3.5 and design discussion in section 4, let's zoom in on the UIPI sending/receiving path at the architecture level. Figure 1 illustrates the end-to-end process of sending and receiving a posted UIPI. Let's examine each step: (❶) The sending core looks up the receiver's UPID in the UITT, updates the UPID (PIR) to reflect the vector being sent, and sets its outstanding interrupt (ON) bit, indicating an interrupt is pending. (❷), The sending core reads the APICID and vector of the receiving core from the UPID, and writes it to interrupt command register (ICR). This causes the local APIC to send an interrupt to the receiving core. (❸) The IOAPIC sends the interrupt message over the system bus to the local APIC on the receiver. The local APIC notifies the core by raising an interrupt signal line [4]. (❹) The receiving core issues a full pipeline flush and starts executing a microcode procedure called *notification processing* that reads the user interrupt vector from the UPID of the current thread, saves it in a dedicated register (UIRR), and clears the outstanding notification bit in the current thread's UPID. (❺) The core executes the *user interrupt delivery* microcode procedure. It pushes the stack pointer, PC, and user vector onto the stack. It clears the user interrupt flag (UIF) disabling interrupt delivery for the handler. It updates the UIRR to show the interrupt has been processed. And finally, it jumps to the user-level interrupt handler specified in the `UINT_Handler` register. (❻) The handler executes. (❼) The handler executes a `uiret`, popping the stack pointer and PC from the stack, and sets the user interrupt flag (UIF), reenabling interrupt delivery, and resuming normal control flow on the receiver.

3.4 UIPI Performance

Table 2 summarizes the key performance metrics of UIPI. The setup for measuring these is described in §5. Next, we describe the experiments we use to reverse engineer UIPI.

Table 2. Key performance metrics of UIPIs.

End-to-End Latency	Receiver Cost	SENUIPI	CLUI	STUI
1360 cycles	720 cycles	383 cycles	2 cycles	32 cycles

3.5 Deconstructing the UIPI Microarchitecture

Interrupt handling strategy. The first step processors take when handling interrupts is to redirect control flow to the interrupt handler. However, with reorder buffers in modern processors capable of holding 500 or more in-flight instructions, the strategy they choose in dealing with those in-flight instructions has significant performance implications. *Flush*ing all in-flight program instructions is the conventional solution, as it minimizes latency to handle the interrupt. But *drain*ing all in-flight instructions is also an option. Here, all (correctly fetched) in-flight instructions would be allowed to retire. Once the pipeline is empty of pre-interrupt instructions, the program counter points to the last instruction in the program before the interrupt. The PC for the next instruction can be saved, designating where the processor should resume execution after the interrupt handler. This approach introduces greater latency (waiting for the pipeline to drain), but minimizes the wasted work.

To discover what our Sapphire Rapids processor does, we built two test programs that expose the effects of this feature. We exploit the fact that with flushing, the end-to-end latency of a UIPI is independent of the in-flight instructions on the receiving core. With a drain strategy, latency depends on what type of instructions are in flight.

Our first test program does pointer chasing, allowing us to vary the time it would take to resolve/drain the chain. We gradually increase the size of the workload, increasing the percentage of pointers that miss in the cache, and thus the length of time it would take to drain the chain of dependent fetched instructions from the pipeline. We observe the change in the end-to-end UIPI latency, expecting that as cache misses increase, this would affect the end-to-end UIPI latency if a drain strategy is employed. However, such latency variations were not observed, suggesting that our processor employs a flush strategy. Our next test program used Intel's hardware performance counters to measure the number of instructions that are dispatched but not committed (i.e., flushed) when we receive interrupts. We find that flushed micro-ops increase exactly linearly with the number of interrupts received, again, implying a flush strategy.

Where are the receiver overheads coming from? To understand the cost of receiving a user interrupt, we design a measurement program that records the costs of each individual step of receiving a UIPI as described in §3.3. As each step has an observable effect on a shared data structure (e.g., UPID), our test program running in a parallel thread can monitor these locations for changes made by the receiver core and measure the duration between the occurrence of

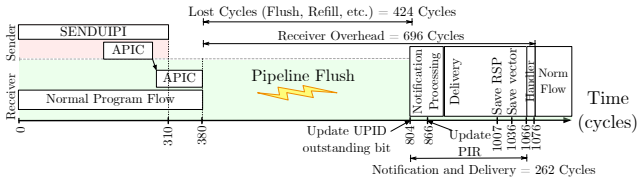


Figure 2. UIPI Latency Timeline: *The average latency for each step of delivering a UIPI as inferred by our measurements.*

these events. As these changes take place in kernel mode, our measurement program is implemented as a kernel module.

If our measurement program and the observed receiver thread run on different physical cores, memory writes to the UPID must traverse the last-level cache— introducing unpredictable overheads into our measurements. To avoid this, we pin our measurement kernel module to the sibling SMT context of the receiver thread.

To determine when an interrupt begins, we need to record the timestamp of the last instruction in the normal program flow to be executed before the interrupt is processed. To facilitate this, the receiver process executes the `rdtsc` instruction within a loop to record the current time. In the interrupt handler, this recorded value is saved as the final point in the pre-interrupt execution.

We repeat these measurements 400K times and report the average of each time interval. Figure 2 summarizes the results of our measurements. On the sender, we start measuring the time right before we execute the `senduipi` instruction that marks time 0 in Figure 2. The senduipi instruction causes an interrupt to be sent from the sender's APIC to the receiver's APIC which then interrupts the receiver program flow at cycle 380. After that, the next observable event on the receiver side is the update to the outstanding notification bit in the UPID during the notification processing. Our observations indicate that a significant portion of the time on the receiver side–424 cycles–is spent between the last program instruction and the first notification processing event—we believe this latency is a combination of: (1) the latency of the pipeline flush—as the processor can only flush a limited number of micro-ops per cycle, flushing all the in-flight instructions might take time, (2) the latency of filling the pipeline with the interrupt notification processing micro-ops, and (3) the latency of executing any interrupt notification processing micro-op that precedes the observable event that we monitor.

The notification and delivery take at least 262 cycles, while executing the `uiret` is relatively quick, taking only 10 cycles.

Where do the costs of `senduipi` come from? We ran a test program that repeatedly executes successful `senduipi` instructions (we found that unsuccessful UIPIs show different performance characteristics) and measured available hardware performance counters. Averaging across 300 million runs, we observe that each successful `senduipi` takes a total of 383 cycles. Out of those cycles, the processor stalls for 279 cycles per `senduipi` due to serializing operations. Some of the micro-ops of `senduipi` write to MSRs (e.g., the one that actually writes to the ICR register), and we know that writes to MSRs are serializing operations [31], which explains the stall cycles. Our measurements also show that the `senduipi` is implemented using microcode, as we see 57 micro-ops delivered through the Micro Sequencing ROM (MSROM) for each `senduipi`.

4 xUI: Extended User Interrupts

xUI attempts to meet the following design goals.

- **Fast:** faster is better; notification is a basic building block for IO, preemption, synchronization, etc. xUI aims to offer performance on par with shared memory notification to eliminate the need for polling.
- **Non-intrusive:** xUI should be adoptable with minimal changes in existing high-performance processors. Thus, it should have a small bill of materials, require minimal changes to the existing ISA and microarchitecture, and not impose a performance tax on other features.
- **Simple and compatible:** xUI should work with existing software and enable simpler solutions than current user-level notification.
- **General Purpose:** xUI should support all forms of notification offered by the interrupt system including timers, device interrupts, and IPIs.

4.1 Overview

Tracked interrupts: closing the memory gap. Tracked interrupts offer an alternative approach to interrupt delivery that avoids the high costs of flushing the pipeline. This enables notification with latency and performance costs comparable to memory-based notification (polling), with the efficiency and responsiveness of asynchronous notification. Tracked interrupts improve on the overhead of UIPI by roughly 3x-9x, with a notification taking 231 or 105 cycles, depending on interrupt type (IPI, device, timer interrupt). This offers significant benefits for performance and efficiency in high-performance applications (§6).

HW safepoints: precise interrupts for precise GC. Hardware safe points provide an explicit *safepoint instruction*. This allows compilers to communicate precisely where it is safe to deliver an interrupt. Safepoints address the needs of languages with precise GC at near zero cost, with minimal architectural complexity. While UIPI could implement this with `clui` and `stui`, this pair of instructions costs 34 cycles together (Table 2), expensive enough that when used in a tight loop or hot code path they are prohibitively expensive. For example, using them to protect a critical section in `malloc()` in RocksDB incurred a 7% throughput penalty.

KB_Timer: fast efficient user-level timer interrupts. The KB_Timer (Kernel Bypass timer) eliminates the cost and complexity of OS-based timers and dedicated timer threads.

With KB_Timer, each timer interrupt is 105 cycles. As each core gets its own timer, it eliminates the added notification overheads that scale with the number of cores (Figure 6). KB_Timers are also directly programable from user space, allowing efficient access to change the timer frequency, disable the timer, or use a different mode of operating (one-shot vs. periodic).

Interrupt forwarding: device interrupts for threads. Kernel bypass interfaces for NICs and accelerators force user-level code to poll for IO completion events. xUI aims to mitigate this inefficiency. Device interrupts are normally routed to cores (by APICID) rather than threads, and UIPI offers no solution. Interrupt forwarding (§4.5) overcomes this limitation by routing interrupts destined for a particular APICID/vector to a user-level thread/vector, allowing user-level code to enjoy the benefits of xUI with devices.

4.2 Tracked Interrupts

Current strategies for interrupt handling in CPU pipelines require difficult trade-offs, forcing a choice between either a loss of instruction throughput, or high latency. Tracked interrupts offer a new strategy for coping with the challenges of speculative execution (§3.5) that minimizes interrupt handling latency without sacrificing throughput.

To recap, *Flushing*—the common approach on modern out-of-order processors—flushes the pipeline immediately when an interrupt is received. This throws away all in-flight work, which hurts throughput but minimizes latency. In modern architectures with over 500 in-flight instructions, the lost throughput (for the interrupted thread) on every interrupt can represent hundreds or thousands of cycles of lost progress. There can also be a latency cost as many architectures will have a limit on the number of micro-ops that can be squashed in a cycle. Another possible approach would be *Draining*—retiring all in-flight instructions before delivering an interrupt. This does not waste work, but it adds significant latency. The latency is necessary because we cannot begin fetching until all sources of speculation are resolved; otherwise, we risk losing the interrupt. The cost of both methods is only increasing, as successive generations of processors increase the size of instruction windows—enabling more in-flight work.

Our approach, *Tracking*, avoids both of these compromises by exploiting the observation that interrupts are *inherently asynchronous*, and thus the processor has some flexibility as to where in the instruction stream the interrupt handling code issues. To wit—both flushing and draining treat the precise point in the instruction stream where the interrupt signal was detected as meaningful, i.e., as a synchronous event, and issue the interrupt handling micro-ops immediately after that instruction (either the last instruction retired or the last instruction fetched, respectively).

Tracking exploits the fact that the precise point in the instruction stream when an interrupt is detected isn't semantically meaningful—interrupts are inherently asynchronous, thus the processor can choose where to issue the interrupt handler to achieve the best possible latency, without sacrificing throughput.

Even though Tracking appends instructions to the end of the inflight instruction stream, like Draining, we do not experience the same latency. Because we are (immediately) injecting instructions without dependences on prior instructions, we expect them to enter execution quickly with minimal latency cost, even in the presence of a non-empty instruction window.

Microarchitecture design. In contrast to flushing or draining, tracking is primarily implemented at the front-end of the processor (the fetch unit), rather than the back-end (commit unit).

At a high level, tracking delivers an interrupt in two steps. To start, as soon as an interrupt is received and accepted by the local APIC, tracking forces the instruction fetch unit to redirect control flow to MSROM, where the micro-ops for interrupt notification processing are stored—effectively injecting these instructions into the micro-op stream. Next, it *tracks* whether these micro-ops are committed or not, ensuring that: (1) they will be reissued if the processor flushes the younger in-flight instructions on mispredictions, and (2) we can accurately identify and save the last program instruction—the place execution will resume after the interrupt handler runs, i.e. after an (`iret`) instruction. This method guarantees that the interrupt will eventually be delivered, even if an in-flight branch is misspeculated, without throwing away useful work. Figure 3 provides an overview of the architectural changes introduced by our approach.

Now let's examine tracking in more detail. To start, when an interrupt arrives from the local APIC, the "next PC logic" in the front-end checks if it is at an instruction boundary—ensuring it is not in the midst of decoding micro-ops for a single instruction. Once this condition is satisfied, it then immediately sets the micro-PC to the beginning of the interrupt notification processing micro-routine in the MSROM and continues execution.

Next, because we neither flush nor drain, we have several new challenges to cope with:

First, we have to deal with the possibility of misspecualation. For example, on a branch misprediction, the processor normally recovers by flushing all the instructions younger than the branch, then resuming execution. However, this will also flush our interrupt processing code, causing us to lose an interrupt.

To prevent this, we implement a state machine in the front-end to track if we are in the middle of handling an interrupt. If there is a flush due to a misspeculation while an interrupt is being handled, we immediately redirect the fetch unit

to our interrupt processing microcode—re-injecting these micro-ops before resuming execution.

The interrupt processing microcode will remain the default misspeculation recovery path until the first interrupt micro-op commits. It is possible that the processor misspeculates and then restarts multiple times, however, the interrupt processing microcode—and the interrupt handler, will eventually get committed. Subsequent restarts may end up being faster, as the interrupt handler's code/data may already be in the cache.

Second, we need to be able to identify the next_pc value following the last committed program instruction before the interrupt, to return to the correct address on an `iret` (return from the interrupt handler). This correct return address may change as we handle misspeculations. However, our solution to misspeculation also handles this seamlessly, without significantly changing interrupt processing microcode. To start, we read the next_pc from the front-end when we redirect execution. If we observe no misspeculations, then this value is correct. If there is a recovery from a misspeculation, the front-end will (as always) calculate a new next_pc value which our restarted interrupt process microcode will again use.

Cheaper than shared memory notification? Our results suggest that tracked interrupts could be cheaper than shared memory notifications on the receiver side for reasons similar to why they are cheaper than normal interrupts—shared memory notification is inherently synchronous, thus it can introduce unexpected control flow.

To wit—polling typically involves a read of a shared variable, with the common path being a cache hit and no pending notifications. When a notification does arrive, it will typically involve both a long memory operation (cache miss due to an invalidation event by the remote write, followed by a slow read of the remote line) and a branch mispredict—resulting in a flush of instructions younger than the mispredicted branch. Further, these steps will be serialized, because only when the long-latency load retrieves its value will the branch mispredict resolve and recovery begin. Again, similar to interrupts, this cost will increase as speculation windows grow in future processors.

At present, whether tracked interrupts are as cheap (or cheaper) than polling depends on the cost of interrupt routing. Recall the steps for delivering a UIPI described in §3.3. Without the costs of flushing, the next major overhead of UIPI is reading the UPID, which is equivalent to polling, as this will similarly miss the cache if a remote core has updated the UPID to send an IPI. However, as we discuss next, neither timers (§4.3), nor device (§4.5) interrupts with xUI need to modify this structure. Thus, the microcode for interrupt delivery can start at step 5 of our description in §3.3, that takes only 105 cycles to execute.

Hardware cost and complexity. Our approach to implementing tracked interrupts aims to minimize additional cost and complexity in existing high-performance CPU pipelines. For example, current processors don't support partial flushes, thus xUI does not rely on them. Thus, if a misspeculation on an instruction that was fetched before an interrupt is detected, the normal misspeculation recovery logic flushes all the instructions after the misspeculation, including those from the interrupt path, even though we will need to reissue these later. While a partial flush could potentially offer some performance benefits, the improvements appear negligible, and not worth the added complexity [15, 20].

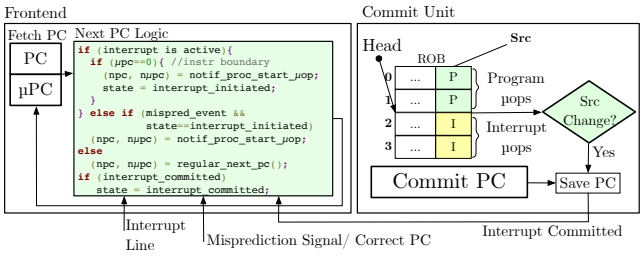


Figure 3. Tracking Architecture: *High-level overview of the Tracking architecture, which avoids flushing or draining when delivering an interrupt. The architectural changes are limited to the next PC logic and minor modifications to the commit logic.*

Our bill of materials includes: one bit per ROB entry to indicate if a micro-op's source is from normal program execution or an interrupt path (i.e. the interrupt processing microcode or interrupt handler), and a bit per micro-op from the front-end to the back-end to carry this source information to the ROB; a handful of logic gates to implement next-pc logic to steer instruction fetch to the interrupt notification microcode on an interrupt (Figure 3); a handful of logic gates and a wire, to extend the commit logic to detect if an interrupt has been committed and to signal the front-end, so it knows the interrupt does not need to be issued again.

4.3 The Kernel Bypass (KB) Timer

The Kernel bypass timer (KB_Timer) offers a per-thread user-level timer, similar to the local APIC timer. As with the APIC timer, it is not intended for direct use by application developers. Instead, it offers a low-level primitive that user-level runtimes can use to implement software timers for tasks like preemption, periodic polling, timeouts, etc.

The timer is used through two new instructions: `set_timer(cycles, mode)` and `clear_timer()`. `cycles` is a 64-bit unsigned integer and the mode is a one-bit flag, indicating if the timer is a periodic or a one-shot timer. If the `mode` is periodic, cycles is interpreted as a period in cycles. If the `mode` is one-shot, cycles is interpreted as a deadline, again, in keeping with the traditional APIC design that makes it simple to specify the next deadline when implementing multiple software timers.

Microarchitecture design. There is one KB_Timer per physical core, which is multiplexed among multiple threads by the operating system (as discussed below). The KB_Timer has its own physical timer, rather than simply re-using the local APIC timer, as that is already in use by the kernel. Similar to modern local APIC timers, the timer uses the system clock to offer high resolution.

KB_Timer interrupts are delivered by invoking the user-level interrupt handler, and passing it the vector that was assigned by the kernel. KB_Timer interrupt delivery is very inexpensive (105 cycles), as it can skip the microcode steps related to the UPID and routing (`notification_process`) (§3.3) and invoke the `interrupt_delivery` microcode directly.

Unlike IPIs, there is no special slow path support, i.e., it is not intended to fire when either the kernel or any thread other than the owner is executing. If the timer reaches its target in kernel mode, it will trap. If it is in user mode, it will deliver an interrupt to whatever thread is currently executing. Consequently, it is up to the kernel to manage the timer state.

Multiplexing the KB_Timer. The kernel enables and disables the KB_Timer, and sets the timer interrupt vector by writing into the `kb_config_MSR`. Similar to UIPI, this allows the kernel to control access to this mechanism through system calls (`enable_kb_timer()`, `disable_kb_timer()`).

To keep a consistent model for each kernel thread, on a context switch the OS saves the current timer by reading the kb_timer_state_MSR which returns the timer deadline (time + the current timestamp), and saves it along with the assigned vector, period, and mode so the timer can be restored when the thread is resumed.

On the slow path (i.e., delivering an interrupt from the kernel), we opt to simply check if the deadline has been exceeded on context restore and deliver the interrupt—similar to the normal UIPI slow path. However, the kernel could also continue tracking the timer using a kernel timer (which ultimately uses the local APIC timers) while the thread is not running, and immediately reschedule the thread when the timer expires.

4.4 Hardware Safepoints

As discussed in §2, the lack of precise control over where interrupts are delivered with UIPIs and signals conflicts with the needs of precisely garbage-collected languages, that expect to be preempted at safepoints. To recap, safepoints are precise locations in code—determined at compile time through static analysis—where precomputed stack maps required for precise GC are valid [10]

Hardware safepoints address this by providing a precise and near-zero-cost mechanism for the compiler to indicate where the processor can safely deliver an interrupt. Conceptually, safepoints can be thought of as a special form of NOP—a `safepoint` instruction. Their actual implementation is architecture-dependent. On x86, safepoints can be encoded using an x86 instruction prefix [4], effectively transforming any instruction into a hardware safepoint and eliminating the need for a separate instruction. Since safepoints are a common primitive in compilers for languages with precise GC (e.g., they are supported in LLVM IR [45]), compilers can be modified to emit safepoints with minimal changes. xUI's safepoint mode—where interrupts are only delivered at safepoints—is toggled via a system call.

Microarchitecture design. To enable safepoints, we added a one-bit flag register—implemented as an MSR similar to UIPI's UIF flag—to indicate whether safepoint mode is enabled. When the safepoint flag is set, the processor delivers interrupts only at safepoint instructions.

Our tracked interrupts (§4.2), like normal processor interrupts, are delivered only at an instruction boundaries, ensuring they are not delivered in the middle of the micro-ops for a single instruction. Therefore, implementing safepoints only required a minor modification to the existing instruction boundary check.

To wit—we extended this check with an additional condition: if safepoint mode is enabled, the processor verifies if the instruction currently being decoded is a `safepoint` instruction, and only then delivers an interrupt. Speculation adds some complexity here, as we must handle cases where a safepoint is speculatively fetched and decoded on an incorrect path.

During recovery from a misspeculation, normally with tracking, if an interrupt is pending, the processor redirects to the interrupt processing path instead of the tracked program PC. However, if the processor is in safepoint mode, it cannot take the interrupt immediately after recovery since the safepoint was on the misspeculated path. In such cases, the processor goes into a state where normal execution continues until another `safepoint` is encountered, then the tracked interrupt logic redirects the processor to the interrupt processing path.

Another complication arises when the `safepoint` instruction is decoded through optimized frontend paths, such as the micro-op cache or the loop stream detector in Intel architectures. To ensure that safepoints function correctly alongside these optimizations, we add a bit to the encoding of each micro-op to indicate whether it is a safepoint. If the processor is in safepoint mode and an interrupt is pending, and it is executing micro-ops on one of these optimized paths, it uses this bit to detect if it is at a safepoint, and thus is allowed to deliver the interrupt.

4.5 Interrupt Forwarding: Routing Device Interrupts

We want the benefits of xUI for device I/O, however, UIPI has its own delivery scheme using UPIDs that the rest of the interrupt system knows nothing about. It only knows how to route interrupts to APICIDs, that are generally assigned to cores at boot time.

Interrupt forwarding solves this by extending the local APIC, allowing it to forward interrupts destined for the core directly to the current thread. If the destination thread is not running, the local APIC will tell the OS to handle the notification, similar to UIPI. To implement this, Interrupt Forwarding allows the kernel to set up mappings between per-core (APICID) vectors and user-level vectors. Thus, when an interrupt arrives on a mapped vector, the interrupt is forwarded, rather like port forwarding in a network.

To set up a mapping, a thread registers through the OS to receive interrupts from a device, and is assigned a notification vector. Interrupts are delivered as normal through the interrupt handler. However, similar to the KB_Timer, the UPID is never touched, thus fast path interrupt delivery (where the receiving thread is running) is faster and free of contention for shared memory.

Microarchitecture design. Interrupt forwarding extends the local APIC with two new 256-bit registers, `forwarding_enabled` and `forwarded_active`. Each bit in these registers corresponds to a vector. `forwarding_enabled` indicates which interrupts to forward on the current core. `forwarded_active` indicates which interrupts should be forwarded to the currently running thread.

For example, if an interrupt arrives with $vector == 8$. If bit 8 is set in `forwarding_enabled` the APIC will set bit 8 in the UIRR MSR. From here there are two choices, the fast path and the slow path.

If bit 8 is set in `forwarding_active`, the APIC will notify the core to deliver an interrupt—this is our fast path, the interrupt goes straight to the user-level thread. If bit 8 is not set, this means that the thread this is destined for is not currently running—this is our slow path. In this case the APIC issues a regular conventional interrupt, the kernel trap handler looks at the cause, then reads the vector from UIRR and stores it for later delivery in the DUPID (see below).

Multiplexing interrupt forwarding. Each kernel task (thread) has it's own 256-bit vector indicating which interrupts are forwarded to it. This vector is written to `forwarded_active` when a thread resumes execution.

During registration, the OS also allocates a data structure similar to the UPID for handling the slow path on each receiver—i.e when the destination thread is not running, that we call Device User Interrupt Posted Descriptor (DUPID). When a forwarded interrupt arrives that the current thread is not waiting for, the kernel updates the DUPID and delivers the interrupt via the normal slow path, similar to a UIPI.

Our interrupt forwarding scheme, while efficient and minimally intrusive, is constrained by the limited vector space of the underlying core as this must be shared by threads on the host. This limitation restricts the number of device/user pairs that can be supported simultaneously. One could imagine adding a new field to the message format of the interrupt system, or repurposing unused bits in the existing message format (e.g. the clusterID) to avoid this limitation.

Table 3. Architecture Detail for the Baseline x86 Core

Baseline Processor			
Frequency	2.0 GHz	I cache	32 KB, 8 way
Fetch Width	6 μops	D cache	32 KB, 8 way
Issue Width	10 μops	Retire Width	10 μops
Squash Width	10 μops	Decode Width	6 μops
SQ Size	72 entries	IQ	168 entries
LQ Size	128 entries	Functional	Int ALU(6), Mult(2),
ROB Size	384 entries	Units	FPALU/Mult(3)

5 Experimental Methodology

5.1 Hardware Platform

We ran all of our experiments on an Intel Xeon Gold 5420+ processor with 28 Sapphire Rapids cores, using Intel's fork of Linux v6.0.0 with their UIPI patches [5]. We ran our processor at 2 GHz with TurboBoost, frequency scaling, and c-states disabled to ensure controlled measurements. We used both Linux perf [23] and Agner tools [22] to read performance counters. To estimate the number of flushed micro-ops we calculated the difference between the number of committed micro-ops and decoded instructions—as there is no direct performance counter for flushed micro-ops.

5.2 gem5 Simulation

Our simulation is based on gem5 (version 23) [46] and extends the out-of-order CPU model. We configured the baseline architecture to model an Intel Sapphire Rapids processor such as the Intel Xeon Gold 5420+, using the parameters shown in Table 3. We ran gem5 in full-system mode, again with the Intel fork of the v6.0.0 Linux Kernel [5]. We implemented UIPI support in gem5, and used our characterization study (§3) to provide accurate costs for flushing, notification, and delivery, and built support for xUI on top of this. We also verified that the receiver overhead of our UIPI simulation in gem5 closely matches the overhead of UIPIs on our Sapphire Rapids processor (Table 2), with both around 720 ns. To support our I/O notification experiments (§5.4), we ported the gem5-dpdk [58] device model from ARM to x86, and added support for delivering I/O completions using xUI.

One interesting discovery was that gem5's model of interrupt performance is quite different from real hardware. For example, it drains the pipeline instead of flushing it, and a fixed 13 cycles was artificially added after each drain. To remedy this, we plan to upstream our improved interrupt model to gem5.

5.3 Preemptive Scheduling with UIPI and xUI

To evaluate UIPI and xUI with preemptive scheduling, we run Aspen [30] in gem5. Aspen is a user-level runtime based on Caladan [26]. It supports lightweight user-level threading and kernel-bypass I/O. Aspen implements preemptive scheduling via user interrupts and balances threads across cores using work stealing.

For the application, we run RocksDB [8] (v5.15.10), a popular key-value store, using a bimodal workload consisting of 99.5% GET (1.2 μs) and 0.5% SCAN (580 μs) requests. We

generate requests using Caladan's open-loop load generator, configured to use UDP with a Poisson arrival distribution. The load generator and the RocksDB server each run in an Aspen runtime on a single core.

To enable us to run this workload in gem5, we adapted Aspen in three ways. First, to simulate network traffic in gem5, we added an additional core that forwards packets between the load generator and the RocksDB server. Second, we disabled core reallocation to reduce noise and removed the kernel module that implements it (*ksched*). Instead, we pin each kernel thread to a specific CPU core. Finally, we extended Aspen to use our KB_Timer. With KB_Timer enabled, each core gets its own timer.

5.4 IO Notification: L3Fwd & Simulated Accelerators

DPDK-based L3 forwarding. We adopted a methodology similar to prior work to measure the inefficiencies associated with spin polling using DPDK [29] . In our experiments, we use DPDK's Layer 3 Forwarding (L3Fwd) application, with the Longest Prefix Match (LPM) algorithm, a routing table containing 16,000 entries, and 64-byte IPv4 UDP packets. We model a system with 1, 2, 4, or 8 different NICs, each with its own receive queue. Packets are sent back to the same NIC in the 1 NIC case, similar to [29].

We used a dedicated external device and an open-loop packet generator that are part of gem5-dpdk [58] to generate our experimental workloads. We modified the packet generator to use an exponential distribution for inter-packet arrival times to more accurately model the burstiness of real network traffic [14, 27].

On-chip accelerators. We built a simulated on-chip accelerator similar to Intel's DSA [42], with a configurable distribution of offload latencies. The accelerator is connected to our gem5 system via a simulated PCIe bus. User programs submit offloads to the simulated accelerator using an asynchronous interface similar to SPDK [9]. We evaluated our accelerator with a synthetic closed-loop workload that busy-spins while it waits for requests to complete.

Real offload latencies vary as a function of the requested operation, the request (buffer) size, and contention for the accelerator. To capture this variability, we model offload latencies by adding random noise with varying magnitude to the response time of the accelerator. We model two types of requests with response times of 2 μs and 20 μs. According to prior work [42] and our own measurements, examples of operations and data sizes that have these mean offload latencies include: (1) for 2 μs latency: copying one 16 KB buffer using DSA [42] or copying a batch of 8 buffers of up to 2048 bytes each; (2) for 20 μs latency: copying one 1 MB buffer using DSA [42].

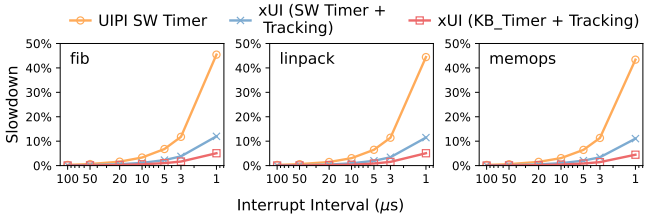


Figure 4. Reducing Receiver Overheads: *UIPI-based notifications impose the most overhead, due to their impact on the receiver pipeline and routing costs. Switching to notification using xUI tracked interrupts eliminates pipeline overheads. Switching to KB_Timers eliminates routing costs, for an overhead reduction of 6.9x vs. UIPI.*

6 Evaluation

We begin by evaluating xUI's features individually, then explore xUI in end-to-end benchmarks in §6.2.

6.1 xUI Features

Below we evaluate the receiver-side overheads with xUI, contrast receiver-side overheads for two precise mechanisms, i.e., hardware safepoints and polling, examine the reduction in sender-side overheads and core counts from KB_Timers, and explore the maximum latency of interrupts with xUI.

Receiver-side overheads: UIPI and xUI. To start, we look at how xUI reduces the overhead of receiving interrupts when compared to UIPI. Figure 4 shows how periodic interrupt delivery impacts three different benchmarks (*fib*, *linpack*, and *memops*).

Our experiment sends periodic interrupts to the running application and measures how much longer it takes to complete. We examined three different strategies to isolate the benefits of different xUI mechanisms: a dedicate timer core sending UIPIs (UIPI SW Timer), xUI (SW Timer + Tracking)—tracked interrupts also being sent with a dedicated timer core, and xUI (KB_Timer + Tracking)—tracked interrupts being sent by the KB_Timer.

To start, we can see that tracked interrupts reduce the per-event cost of delivering an interrupt from an average of 645 cycles with the UIPI baseline to 231 cycles with tracking. Switching to KB_Timer as our time source eliminates the added delivery costs of accessing the UPID, as discussed in §4.3, further reducing the per-event cost of an interrupt to an average of 105 cycles. Thus, the cost of delivering interrupts at a 5 μs interval decreases by roughly 6.9x from 6.86% with UIPIs to just 1.06% with KB_Timers and tracking.

Hardware safepoints vs. polling-based preemption. We compare the overhead of thread preemption with two precise mechanisms—xUI hardware safepoints and polling— as well as UIPIs. Our experiment preempts two programs: *matmul* and *base64*. To implement polling-based preemption, we use Concord [34], a state-of-the-art system for compiler-based

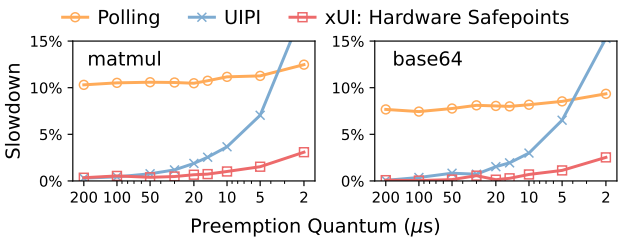


Figure 5. Preemption with Hardware Safepoints: *Preemption with xUI (tracking + KB_Timer) using hardware safepoints is precise and near zero-cost—visible overheads are primarily due to context switch costs. UIPI is imprecise and scales less gracefully due to its higher per-event cost. Polling is precise and scales better with increased preemption rates, but imposes up to 10x more overhead than UIPI or xUI.*

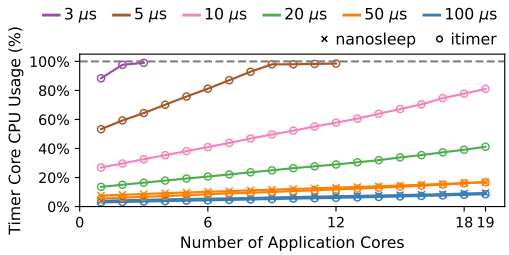


Figure 6. The Cost of a Timer: *OS overheads become increasingly costly for fine grain timing. Here we see CPU use on a single core to get time from the OS (using setitimer() or nanosleep()) and then preempt application cores with UIPI. Lines correspond to different preemption frequencies. The costs of OS timer notifications initially dominate, but the overhead of sending UIPIs increases with the number of receiving application cores. xUI eliminates these costs, as each core gets its own low-overhead high-resolution KB_Timer.*

instrumentation (polling) preemption as a representative system. To evaluate hardware safepoints, we modified Concord's compiler instrumentation passes to insert `safepoint` instructions rather than polling checks.

Figure 5 shows that hardware safepoints have much lower overhead than the other two approaches for both programs at any preemption quantum. With a quantum of 5 μs, our hardware safepoints show a slowdown of only 1.2%-1.5%—largely attributable to the overhead of context switching—while the compiler-based approach imposes 8.5%-11% overhead. This improvement is mainly due to (1) xUI's lower receiver costs, and (2) the near-zero costs of our `safepoint` instructions as they are just interpreted as NOPs when there is no event. Note that as we return to the same thread after handling a preemption, the costs of context switches are minimized vs. context switching to different threads, an I/O handler, etc.—which would pollute the microarchitectural state. While this has the benefit of isolating the improvements due to notification costs, it doesn't illustrate how these benefits are amortized in more complex settings. Our end-to-end evaluations on RocksDB in §6.2.1 capture these more realistic, higher context switching costs.

Benefits of eliminating timing cores with KB_Timers. As discussed in §2, to amortize the costs of OS timers and avoid contention [55], user-level runtimes [3, 30, 44, 54] often rely on notifications from a core that is partially or totally dedicated to timing. In Figure 6, we show the CPU utilization of a timer core using two different OS timer interfaces. In our experiment, we use `setitimer()`, which delivers a signal each interval, and `nanosleep()`, where a thread goes to sleep and is then woken up each interval. UIPIs are used to send notifications to other cores, and we show how overheads scale on the sender (timer core) as there are more cores to notify. As we can see OS timers exert a small, but still noticeable overhead at lower notification rates. And as the notification rate increases, they consume an increasingly large percentage of the core. xUI eliminates these costs entirely as the KB_Timer eliminates the need to use expensive OS interfaces (with multiple context switches per timer event), and each core can have its own timer, eliminating the need for IPIs.

Higher performance system [30, 34, 44], will sometimes dedicate an entire core to spinning on a high-precision timer (e.g., `rdtsc`) to avoid OS overhead. Using this approach, we found we could support up to 22 application cores at a 5 μs preemption interval [30]. As the number of receiver cores increases, notification costs (`senduipi`) become dominant. Using KB_Timer, we can eliminate the need for a dedicated timer core entirely, thus, as a baseline, it saves us 1 core for every 22 workers at 5μs.

Of course, the benefits of the KB_Timer depend on the number of available cores and the workload. For example, in the most optimal case, if we have two cores and can fully utilize an extra core to improve throughput, KB_Timer doubles throughput. In the least optimal case, i.e., 22 other worker cores fully utilized, it only improves throughput by 4.5%. KB_Timer also offers significant per-event performance benefits vs. UIPI (Figure 4).

Maximum interrupt latency. Tracking offers the benefit of maximizing throughput as it never discards work, however, latency depends on the nature of the in-flight instructions when an interrupt arrives.

To explore the limits of this, we sought to understand the potential worst-case latency. To measure this we constructed an example that fills the pipeline with a chain of long-latency load instructions that miss the CPU caches—and then our dependency chain ultimately produces the value for the stack pointer register. Note that that this is an extreme pathological case, where the interrupt code is in fact dependent on a long chain of inflight instructions. While it is not unlikely that the interrupted thread has a long chain, or that it modifies the stack pointer, it is highly unlikely the latter depends on the former.

With tracked interrupts, this case delays the execution of the interrupt delivery procedure (§3.3) as there is an instruction in the delivery that reads the value of the stack pointer register (to save it on the stack). The value remains unavailable until the dependency chain completes, potentially delaying interrupt delivery by hundreds of cycles.

When we run this experiment, we do see an absolute worst-case latency close to 7000 cycles with a chain of 50 or more long-latency loads. Utilizing Intel's flush mechanism has a worst-case latency an order of magnitude less (because all of those instructions just get squashed). However, experiments with our more typical benchmarks (e.g., linpack2, memops, fib) (in addition to our end-to-end workloads), confirm this is indeed an anomalous result. In these cases, tracking latencies were substantially better than flushing, owing to the fact that often there is no delay between when an interrupt arrives and when it issues with tracking, whereas flushing introduces an additional delay before an interrupt can issue. That said, we believe this is an area where we are up against the limits of what is knowable through reverse engineering and simulation. Thus, we refrain from drawing more definite conclusions.

6.2 End-to-End Evaluation

Here we evaluate xUI end-to-end in three use cases.

6.2.1 Preemption for RocksDB in Aspen. Preemption reduces head-of-line blocking and tail latency in workloads where request service times vary significantly [30, 37], leading to higher useful throughput. We evaluated these benefits for RocksDB in Aspen (§5.3) using three configurations: preemption with UIPI SW Timer (UIPI + dedicated timer core) and xUI (KB_Timer + Tracking), and a non-preemptive baseline. Again, our workload is a mix of 99.5% GET (1.2 μs) and 0.5% SCAN (580 μs) requests. We choose a preemption quantum of 5 μs to optimize GET throughput with a 1 ms tail latency target.

Figure 7 shows the tail latency for GET and SCAN requests as we vary the offered load. Without preemption, the tail latency for GET requests is hundreds of microseconds, even under very low load. In contrast, with preemption via UIPIs, Aspen is able to maintain low tail latency for GET requests (at most a few hundred microseconds) up to throughputs of over 100,000 requests per second. The addition of KB Timers and tracking further increases throughput for GETs by 10%, due to the lower overhead of receiving UIs. In both cases, preemption comes at the cost of slightly elevated tail latencies for SCAN requests at high load. This is because each SCAN request takes hundreds of microseconds to complete and is thus preempted many times.

6.2.2 IO notification in l3fwd. Here we compare the performance and efficiency of polling vs. xUI (tracked interrupts + interrupt forwarding) for delivering device interrupts

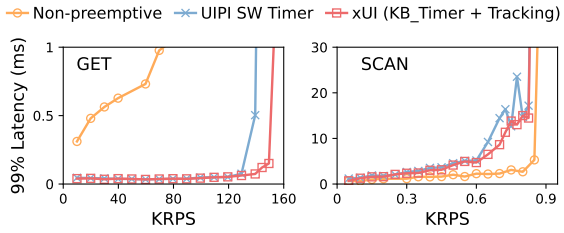


Figure 7. Improving RocksDB Throughput: *RocksDB throughput with a mix of 99.5% GET (1.2 µs) and 0.5% SCAN (580 µs) requests. We compare no-preemption and preempting every 5us with UIPI and xUI. Both UIPI and xUI significantly improve GET throughput by mitigating head-of-line blocking. However, xUI achieves 10% higher throughput than UIPI, and also free's up an extra core (not shown) that UIPI-based preemption requires as a time source.*

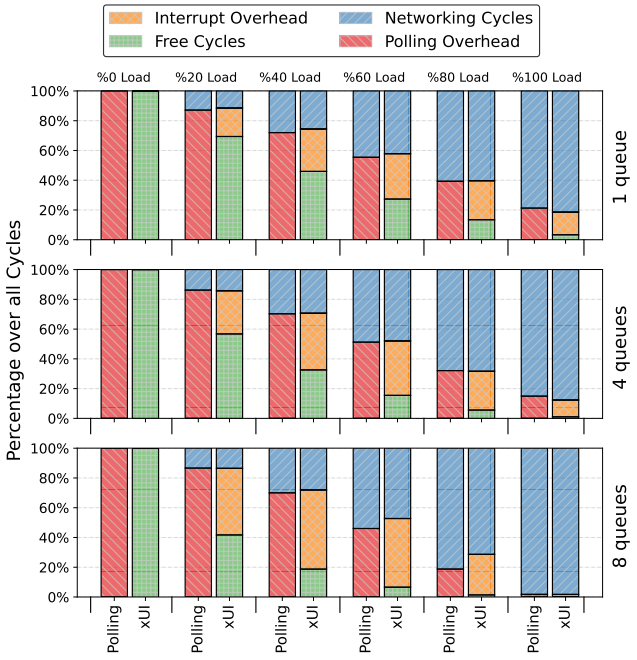


Figure 8. Improving l3fwd Efficiency: *Our layer 3 forwarding workload shows how polling always utilizes the entire core, while xUI frees up cycles that can be used for other useful work or power savings. At 0% load, xUI frees up 100% of cycles, at 40% load with one queue xUI leaves 45% free, etc.*

from simulated high-performance NICs to a layer 3 router (DPDK's l3fwd).

We observe that the throughput of both approaches is nearly identical, with xUI throughput falling short of polling by 0.08%. This is because with device UIs, the interrupt handler polls the network queue again before returning; at high loads the handler never returns, yielding the same throughput as with polling. xUI achieves a 95th percentile tail latency

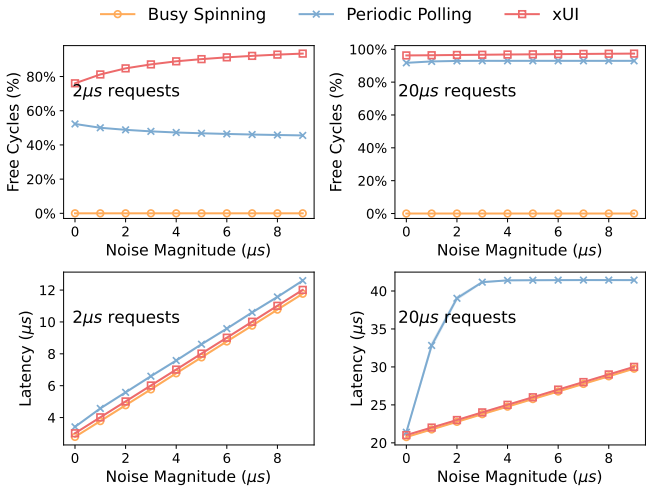


Figure 9. Optimizing Latency and Efficiency of DSA Response Delivery: *Here we see free cycles (top graphs) and latency (bottom graphs) while delivering responses from our DSA accelerator using busy spinning, periodic polling (OS interval timer), and xUI: Busy spinning burns the entire core waiting for DSA responses, i.e. no free cycles (top graph), but minimizes latency (bottom graph). Periodic polling sacrifices latency, especially as response times become less regular, but frees up many cycles compared to polling. xUI provides optimal responsiveness (latency), while minimizing CPU consumption, i.e., maximizing free cycles that can be used for other useful work or power savings.*

within +2%, -8%, and +65% of polling's latency for 1, 4, and 8 NICs, respectively.

We also see how xUI reduces CPU consumption, i.e., leaves more "free cycles" that could be spent on other useful work or power savings. We see in Figure 8, with polling, all CPU cycles are spent either handling packets ("Networking Cycles") or polling the network stack. In contrast, with device interrupts polling becomes unnecessary. Thus, there are free cycles. For example, with 1 queue and 40% load, xUI 45% of cycles are free cycles, compared to polling which always consumes all cycles.

6.2.3 IO notification in DSA. Here we explore the performance and efficiency benefits of xUI for delivering I/O completion events from our simulated accelerator modeled after DSA. We consider two criteria: latency—how long after a completion arrives is it delivered, and free cycles—how many cycles are left on the core after the cost of the notification mechanism. We evaluate three options for receiving I/O competitions: busy spinning, periodic polling, and xUI device interrupts.

To evaluate these different approaches, we offload tasks with two different response time distributions as described in §5.4. Figure 9 shows the free CPU cycles and average latency for the offloaded tasks.

Here we see our three strategies and their results: *Busy spinning*— busy spin continuously polls the completion queue, this offers excellent responsiveness (latency), but wastes all the CPU cycles not spent handling notifications.

Periodic polling—periodically check for completions—in our experiment, we implement these checks with the OS interval timer (`setitimer()`). Here we see that this frees up a significant portion of cycles, but may also hurt the latency of the offloaded tasks, as the arrival time of responses becomes less periodic/more noisy. For example, with 20 μs requests, the latency of periodic polling increases sharply as unpredictability rises. This is because the additional response latency prevents periodic polling from occurring precisely when requests finish, causing processing to wait until the next timer event. We don't see the same effect for shorter requests as the timer frequency is already very high (2 μs), almost at the limit of the OS interval timer, and missing one event does not significantly hurt latency.

With, *xUI*, in contrast, latency remains constant and within 0.2 μs of the latency achieved by polling for both types of requests. Overall xUI is able to achieve the best CPU efficiency with only a small latency penalty. For example, for 2 μs requests with no unpredictability (zero added noise), tracked interrupts free up 75% of CPU cycles.

7 Related Work

Tracked interrupts. xUI's tracked interrupts improve performance by reducing the impact of unpredicted control flow that undermines the benefits of speculative execution. In this sense, they are similar to techniques like predicated execution [50] or selective flushing mechanisms [20] that attempt to retain instructions when a mispredicted branch direction reconverges with the correct path.

Hardware support for user interrupts. Prior research made the case for the value of user-level handling of interrupts [51] and exceptions [57]. To the best of our knowledge, Intel's Sapphire Rapids processor, released in 2023, is the first commodity high-performance processor to offer hardware support for user-level interrupts.

Early versions of the RISC-V spec featured user interrupt support, but this extension was later dropped [24]. More recently, RISC-V added the CLIC extension [16], to support fast interrupts for embedded and real-time applications. While xUI builds on UIPI, most of it is applicable to other ISAs.

Accelerators and killer microseconds. Barroso et al. in their "Attack of the Killer Microseconds" [11] noted a growing number of high-performance IO devices (e.g., NICs, flash, non-volatile memory, accelerators) in the datacenter will be operating at microsecond timescales, and that current processor architectures and software were missing the ability to deliver event notifications, hiding latency, etc. at this

scale. They further predicted the growing use of accelerators to reduce the "datacenter tax", i.e., common microsecond scale tasks such as encryption, compression and serialization that consume 20-25% of the CPU cycles in Google's datacenters [40], would only exacerbate this problem.

Intel introduced UIPI to efficiently interface [38] with a whole range of new accelerators [63] —such as DSA [42]— that are intended to address the data center tax. xUI directly addresses some of the challenges raised in "attack of the killer microseconds", and builds on the vision behind UIPI [38].

Hardware virtualization at user level. Dune [12] leveraged hardware virtualization features (VT [4] and EPT [4]) to provide user-level access to low-level hardware features, including the interrupt system. Shinjuku [37] built on this approach, enabling direct use of IPIs for preemption (rather than signals) before UIPI became available. However, this approach introduces additional performance overhead and deployment challenges; as a result, user-level access to virtualization has never seen widespread adoption.

High-performance IO and polling. High-performance user-space IO stacks rely almost exclusively on polling to detect IO events [13, 26, 37, 39, 41, 48, 49, 52, 62]. However, polling is often quite wasteful especially when traffic is unpredictable and as the number of queues increases [29, 43]. Hyperplane [47] reduces polling overheads with a dedicated accelerator that offers a multiqueue `mwait`-type primitive. With xUI, we explore how polling can be eliminated entirely, demonstrating that interrupts can achieve low latency and high performance without sacrificing efficiency.

User-level preemption. Preemption is a well-known technique for achieving more precise scheduling control. With preemption, schedulers can improve throughput while bounding tail latency [34, 37]. Systems that have used UIPI to enable this [25, 30, 35, 44] could benefit from the improved performance and efficiency offered by xUI.

The tension between precise GC and efficient preemption has long been known in the Java runtime developer community [28], and was extensively explored by the Go developers [2, 10]. The lack of support for loop preemption in Java's virtual threads [6] is simply the latest example. xUI's hardware safepoints offer a simple and efficient solution to this decades-old problem.

Hacking around UIPI limitations. UIPI does not officially support timer interrupts. However, Skyloft [35] recently demonstrated a trick to get around this limitation, albeit in a very constrained way. Their approach starts by setting the UINV field of the IA32_UINTR_MISC MSR that a core uses to discriminate conventional interrupts and UIPIs, to match the vector of the local APIC timer. Thus, timer interrupts are forwarded to the current thread. However, the local APIC will not set the PIR (posted interrupt) register on the local core on a timer interrupt, which is required to deliver a UIPI. To get around this, Skyloft abuses `senduipi` in a novel way. At startup, its sets the SN bit on the UPIDs for all threads. Thus, when a thread then calls `senduipi` to send a UIPI to itself, the PIR will be set, but no UIPI will be delivered. Consequently, when a timer interrupt arrives, it will be delivered as expected. Finally, Skyloft repeats this self-`senduipi` trick in their interrupt handler after every interrupt to ensure the PIR is set before the next timer interrupt.

Unfortunately, this trick has significant limitations. First, because there is only one local APIC timer, the kernel no longer has use of the local APIC timer for scheduling, timekeeping, etc. Next, this also disables all other uses of user interrupts, first because the SN bit must be set for the self-`senduipi` trick, and more generally because overloading UINV for timer interrupts means the processor cannot disambiguate other UIPI's from timer interrupts. For these reasons, as well as others, e.g. direct user-space programmability, this trick is not a substitute for xUI's KB_Timer support.

8 Conclusion

xUI offers first class support for asynchronous notification at user level. It significantly improves on the performance and flexibility of UIPI, and offers a simpler and more efficient alternative to shared memory polling.

Acknowledgments

This work was supported by gifts from Cisco, Intel, and Google. Thanks to Chris Fallin, Shravan Narayan, Hosein Yavarzadeh, and Brendan Sweeney for insightful discussions and feedback on this work. We also thank the anonymous reviewers and our shepherd, Gabriel Parmer. And finally thanks to our families, without their support this work would not be possible.

A Artifact

A.1 Abstract

This artifact includes 7 different modules, 5 of which are based on our extended user interrupt model implemented in the gem5 simulator. We help reproduce l3 forwarding (Figure 8), accelerator (Figure 9), overhead (Figure 4), rocksdb preemption (Figure 7), and safepoint preemption (Figure 5). There are two other modules one of which measures core utilization of timers on a physical cpu (unlike the gem5 models, Figure 6), and the other one is a reverse engineering program to uncover latencies of user interrupts, again on a physical cpu (Figure 2).

A.2 Artifact check-list (meta-information)

- **Program: gem5, dpdk**
- **Compilation: gcc,clang**
- **Transformations: compiler insertion**
- **Hardware: Intel Sapphire Rapids Processor**
- **How much time is needed to prepare workflow (approximately)?: 4 hours**
- **How much time is needed to complete experiments (approximately)?: 3 days**

- **Publicly available?: Yes**

A.3 Description

A.3.1 How to access. https://github.com/ArchSecLab/xui

A.3.2 Hardware dependencies. These tests require a processor with Intel Sapphire Rapids Microarchitecture.

A.4 Installation

Please follow the step of the **README.md** in the repository. There are 7 folders in the xui repository, each of them has its own **README.md** that explains how to generate the experiment results. The folders are: rocksdb, accel, l3, overhead, safepoint, timer, and profile. The measurements on real hardware require a Sapphire Rapid processor, and at least a 40-core machine is suggested. There are a few Ubuntu disk images for use with the full system mode of gem5 simulator, you will find a link to download those disks on the **README.md** pages.

A.5 Evaluation and expected results

We expect the experiments to closely match those in the paper.

A.6 Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

References

[1] [n. d.]. DPDK. https://www.dpdk.org/.

[2] [n. d.]. Go Preemption Issue 10958. https://github.com/golang/go/issues/10958.

[3] [n. d.]. The Go Programming Language. https://go.dev/.

[4] [n. d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[5] [n. d.]. Linux Support For Sapphire Rapids' User Interrupts Still Awaiting Mainline. https://www.phoronix.com/news/Linux-SPR-Intel-User-Interrupts.

[6] [n. d.]. Loom - Fibers, Continuations and Tail-Calls for the JVM. https://openjdk.org/projects/loom/.

[7] [n. d.]. Proposal: Non-cooperative goroutine preemption. https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md.

[8] [n. d.]. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[9] [n. d.]. Storage Performance Development Kit. https://spdk.io/.

[10] Austin Clement. 2020. Pardon the Interruption:loop preemption in Go 1.14. https://www.youtube.com/watch?v=1I1WmeSjRSw

[11] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[12] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[13] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2016), 1–39.

[14] Balakrishnan Chandrasekaran. 2009. Survey of network traffic models. *Waschington University in St. Louis CSE* 567 (2009).

[15] Jamison D Collins, Dean M Tullsen, and Hong Wang. 2004. Control flow optimization via dynamic reconvergence prediction. In *International Symposium on Microarchitecture (MICRO)*. IEEE.

[16] RISC-V Community. 2025. RISC-V Fast Interrupts and Core-Local Interrupt Controller (CLIC). https://github.com/riscv/riscv-fast-interrupt/blob/master/src/clic.adoc

[17] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. 2022. The Go programming language and environment. *Commun. ACM* 65, 5 (2022), 70–78.

[18] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[19] DPDK Project. [n. d.]. *L3 Forwarding Sample Application User Guide*. Accessed: 2024-18-10.

[20] Stijn Eyerman, Wim Heirman, Sam Van Den Steen, and Ibrahim Hur. 2021. Enabling Branch-Mispredict Level Parallelism by Selectively Flushing Instructions. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 767–778. doi:10.1145/3466752.3480045

[21] Fastly. [n. d.]. Fastly Compute. https://www.fastly.com/resources/datasheets/edge-compute/fastly-compute/. Accessed: 2024-10-18.

[22] Agner Fog. 2024. Test programs for measuring clock cycles and performance monitoring. https://www.agner.org/optimize/

[23] Linux Kernel Foundation. [n. d.]. Performance Counters for Linux (perf). https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2024-18-10.

[24] RISC-V Foundation. 2017. Document 'N' extension. https://github.com/riscv/riscv-isa-manual/issues/16. Accessed: 2025-02-08.

[25] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making kernel bypass practical for the cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 55–73.

[26] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[27] V.S. Frost and B. Melamed. 1994. Traffic modeling for telecommunications networks. *IEEE Communications Magazine* 32, 3 (1994), 70–81. doi:10.1109/35.267444

[28] Go Team. 2015. runtime: tight loops should be preemptible. https://github.com/golang/go/issues/10958 GitHub issue #10958, accessed on 2025-02-08.

[29] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. 2019. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*. 337–350.

[30] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. 2025. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. To appear.

[31] Intel. 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[32] Intel. 2023. Intel® Architecture Instruction Set Extensions and Future Features. https://cdrdv2-public.intel.com/812218/architecture-instruction-set-extensions-programming-reference.pdf.

[33] Intel Corporation. 2024. What is Intel® QuickAssist Technology (Intel® QAT)? https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-qat.html. Accessed: 2024-18-10.

[34] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[35] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[36] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

[37] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[38] Utkarsh Y. Kakaiya, Sanjay Kumar, Rajesh Madukkarumukumana Sankaran, and Prashant Seth. 2022. Scalable I/O Between Accelerators and Host Processors. *Intel Developer Articles* (2022). https://www.intel.com/content/www/us/en/developer/articles/technical/scalable-io-between-accelerators-host-processors.html

[39] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[40] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[41] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash ≈ local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

[42] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. 2024. A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 37–54.

[43] Bojie Li, Zihao Xiang, Xiaoliang Wang, Han Ruan1 Jingbin Zhou, and Kun Tan. 2023. FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA. *Context* 5 (2023), 9.

[44] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–936.

[45] llvm.org. 2024. Garbage Collection Safepoints in LLVM. https://llvm.org/docs/Statepoints.html.

[46] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, et al. 2020. The gem5 Simulator: Version 20.0+. doi:10.48550/ARXIV.2007.03152

[47] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F Wenisch. 2020. HyperPlane: A scalable low-latency notification accelerator for software data planes. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 852–867.

[48] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[49] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[50] J.C. Park and M.S. Schlansker. 1991. *On predicated execution*. Technical Report. Hewlett Packard Tech. Rep. HPL-91-58.

[51] Mike Parker. 2002. A case for user-level interrupts. *ACM SIGARCH Computer Architecture News* 30, 3 (2002), 17–18.

[52] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 325–341.

[53] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[54] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. 2017. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 512–526.

[55] Shumpei Shiina, Shintaro Iwasaki, Kenjiro Taura, and Pavan Balaji. 2021. Lightweight preemptive user-level threads. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 374–388.

[56] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. 2023. μManycore: A Cloud-Native CPU for Tail at Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[57] Chandramohan A Thekkath and Henry M Levy. 1994. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. 110–119.

[58] Johnson Umeike, Siddharth Agarwal, Nikita Lazarev, and Mohammad Alian. 2024. Userspace Networking in gem5. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 179–191. doi:10.1109/ISPASS61541.2024.00026

[59] Kenton Varda. 2018. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[60] WhatsApp Blog. 2011. 1 million is so 2011. https://blog.whatsapp.com/196/1-million-is-so-2011

[61] Adam Wierman and Bert Zwart. 2012. Is tail-optimal scheduling possible? *Operations research* 60, 5 (2012), 1249–1257.

[62] Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt.. In *FAST*, Vol. 12. 3–3.

[63] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, et al. 2024. Intel accelerators ecosystem: An soc-oriented perspective: Industry product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 848–862.