
# The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling

Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout, *UC San Diego*

https://www.usenix.org/conference/nsdi25/presentation/guo

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

# The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling

Linsong Guo, Danial Zuberi, Tal Garfinkel, Amy Ousterhout
*UC San Diego*

## Abstract

Preemptive scheduling promises to mitigate head-of-line blocking and enable flexible scheduling while retaining a simple programming model. Despite this, preemption is underutilized in server-side software today. Instead, high-performance datacenter systems and language runtimes often rely on cooperative concurrency, or else use preemption only at very coarse timescales, limiting its effectiveness. A key reason that preemption is underutilized today is that existing preemption mechanisms have high and unpredictable overheads.

Intel recently introduced support for user interrupts, a new feature that offers an opportunity to change this. By enabling interrupts to be sent and received entirely in user space, user interrupts can significantly lower the overhead of preemption. In this paper, we shed light on how user interrupts impact the landscape of preemption mechanisms. We build two user-level schedulers that leverage user interrupts for low-overhead preemption. We find that user interrupts are not a panacea. For example, they provide limited benefits when other software layers constrain the kinds of scheduling policies that can be used. Still, user interrupts can match or exceed the performance of existing mechanisms for all but the highest preemption rates, while achieving much more consistent overheads and retaining a user-friendly programming model.

## 1 Introduction

Datacenter applications today suffer from high tail latency. For example, a recent study of remote procedure calls (RPCs) at Google showed that 90% of RPC methods were capable of completing in hundreds of microseconds or less, and yet 90% of them had median latencies in the milliseconds, with even higher tail latencies [52]. A myriad of factors contribute to this high tail latency, one of which is queuing of requests at servers. When requests have variable service times, as is the case for Google's RPCs, short requests can queue behind long requests and suffer from *head-of-line blocking*.

Head-of-line blocking is common because many applications have tasks with heterogeneous service times. For example, in hybrid transactional and analytical processing (HTAP) databases [47], short transactional tasks coexist with long-running analytical tasks and may be delayed by them unless the database is provisioned with extra resources [28,35,36,41]. A well-known way to reduce head-of-line blocking without over-provisioning is preemption [39, 56]. With preemption, a scheduler periodically interrupts running tasks, giving shorter, more urgent tasks an opportunity to run and complete quickly. However, preemption is underutilized today.

Though kernel schedulers do implement preemption, they typically do so only at millisecond timescales, which is not sufficient to ensure sub-millisecond tail latencies. Furthermore, kernel-based scheduling adds microseconds of overhead to scheduling operations [23]. As a result, many users turn to userspace schedulers and runtimes instead, employing an *M:N* threading model that schedules *M* user-level threads across *N* kernel threads [6, 10, 33, 37, 45, 49, 57]. We focus on such user-level schedulers.

Today's user-level schedulers rarely implement preemption. Most runtime systems and programming languages eschew preemption in favor of cooperative concurrency, which runs tasks to completion and relies on the developer to manually yield cores as needed (e.g., async tasks in Rust and coroutines in C++). Runtimes like Go [6] that do implement preemption do so only at coarse granularities (every 10 ms) so applications cannot rely on them to provide sub-millisecond tail latency. Similarly, high-performance kernel-bypass systems in datacenters also commonly rely on cooperative concurrency [25, 33, 45, 48, 49, 57].

One reason that preemption is underutilized is the high and unpredictable overheads of the typical mechanisms for user-level preemption today: signals and compiler instrumentation. With *signals*, a dedicated timer thread determines when a user-level thread should be preempted and sends a signal to interrupt it [6, 53]. This involves transitioning to the kernel and back on both the timer core and the core of the preempted thread, adding significant overhead on each preemption (§2.1), resulting in practical limits on preemption frequency.

In contrast, with *compiler instrumentation*, the compiler inserts code throughout a program that checks if it should voluntarily yield the CPU [6,24,37,43], for example, by polling a variable in shared memory. Here, the overhead of each check is much lower than a signal, so it can sometimes enable more efficient fine-grained preemption. However, the checking overhead is hard to predict, as it is heavily dependent on program control flow. Thus this approach can also result in overheads that are high, variable, and workload dependent (§2.2).

Fortunately, a new hardware feature called *user interrupts* recently introduced in Intel's Sapphire Rapids CPU offers the potential for a better approach. User interrupts offer a new way to send and receive inter-processor interrupts entirely in userspace, eliminating the multiple kernel transitions required to send and receive signals (§2.3).

In this work, we seek to understand the potential benefits of user interrupts for preemptive scheduling. We ask: *can user interrupts improve our ability to achieve fine-grained preemptive scheduling with low and predictable overheads, compared to existing approaches?* We answer this question with a measurement study and analysis of two user-level schedulers. While we are not the first to utilize user interrupts for preemptive scheduling [32, 38, 42], we are the first to shed light on the tradeoffs between user interrupts and compiler-based approaches, and to analyze fine-grained preemptive scheduling in the context of a widely used runtime (the Go runtime).

We began by studying the performance impact of preemption with each mechanism on a collection of different workloads (§2). We found, unsurprisingly, that user interrupts consistently outperform signals, reducing the per-preemption overhead from 2.4 $\mu$s to 0.4 $\mu$s. Consequently, user interrupts enable more frequent preemption at the same cost. In contrast, with compiler instrumentation, overheads can be quite low, but are unpredictable and workload dependent. We investigated several mitigations from other work [37] such as loop unrolling, but we found that rather than solving this problem, they merely shift the costs and complexity to other parts of the system. We also found that the overhead of compiler instrumentation is impacted little by the preemption frequency. Thus, compiler instrumentation shines for ultra-high frequency preemption, e.g., every 5 $\mu$s or less, but with a larger preemption quantum, the relative performance depends on the application and user interrupts often provide lower overhead.

Next, we built two user-level schedulers that perform preemptive scheduling with user interrupts (§3). First, **Aspen-KB** extends Caladan [33]—a highly optimized **k**ernel-**b**ypass runtime—with support for user interrupts, allowing us to explore the limits of user-level preemption with user interrupts. Aspen-KB also has support for signals and compiler instrumentation, which makes apples-to-apples comparisons among all three mechanisms possible, without the confounding effects of different scheduling algorithms, programming languages, or runtimes. Second, **Aspen-Go** extends the popular Go runtime, enabling comparisons between preemption mechanisms in the context of a runtime that is used by millions of developers. Both Aspen-KB and Aspen-Go leverage known techniques but carefully apply them to optimize for fine-grained preemptive scheduling. They are available at `https://github.com/LinsongGuo/aspen.git`.

We analyzed the performance of Aspen-KB and Aspen-Go by running applications such as key-value stores (RocksDB [15] and BadgerDB [1]) and a data analysis application (DataFrame [2, 50]) (§4). We have two key findings.

First, in the context of the kernel-bypass runtime Aspen-KB, we found that user interrupts enable Aspen to achieve much better performance than signals. Compared to compiler instrumentation, user interrupts generally deliver better or similar performance when the preemption quantum is greater than 10 $\mu$s. With an extremely small quantum (10 $\mu$s or lower), compiler instrumentation may offer a small performance benefit over user interrupts. However, compiler instrumentation requires manual intervention to determine whether loop backedges, function calls, or both should be instrumented, and to determine the optimal depth for loop unrolling. Thus, *user interrupts are the best choice for most applications*, due to their improved usability and similar or better performance.

Second, in contrast, we found that user interrupts provided only modest benefits for preemptive scheduling in Aspen-Go. The reason for this is that significantly shrinking the preemption quantum (e.g., to less than 50 $\mu$s) proved counterproductive. The design of Go constrains the kinds of scheduling policies that are possible, making preemption less effective at avoiding head-of-line blocking, and every preemption comes at the cost of software overheads in the Go runtime. With preemption at the coarse granularity of 50 $\mu$s, replacing signals with lower-overhead user interrupts provides only a small benefit. Thus, *when a system is not fully designed for fine-grained preemptive scheduling, user interrupts provide limited benefits*.

## 2 Preemption Mechanisms

Three preemption mechanisms are available to userspace schedulers today: signals (§2.1), compiler instrumentation (§2.2), and user interrupts (§2.3). Here, we explore how these mechanisms work and why the first two can have high and unpredictable overheads. We then examine user interrupts (§2.3) and how they can offer lower and more consistent overheads.

### 2.1 Signals

Signal-based preemption [6, 53, 54] involves two key components: a timer thread and a signal handler. The timer thread typically runs on its own core and is responsible for determining when user-level threads should be preempted.

To preempt a user-level thread running on kernel thread $k$, the timer thread sends a signal to $k$. The signal triggers the invocation of $k$'s signal handler function, on the core that $k$ is currently running on. The signal handler then calls into the user-level scheduler, which selects the thread to run in the next quantum. The primary limitation of signal-based preemption is that the signal handler is expensive, incurring multiple costly OS context switches, which ultimately limits preemption frequency.

To explore the basic overhead of each of our three mechanisms, including signals, we ran the following experiment. A dedicated *timer core* sends a notification (signal, user interrupt, etc.) to a single application core at a regular interval (the *preemption quantum*). This notification triggers the application core to run an empty signal-handler function that immediately returns to the application thread. As there is no context switch to a different thread and the scheduler is not invoked, this benchmark illustrates the minimum possi-

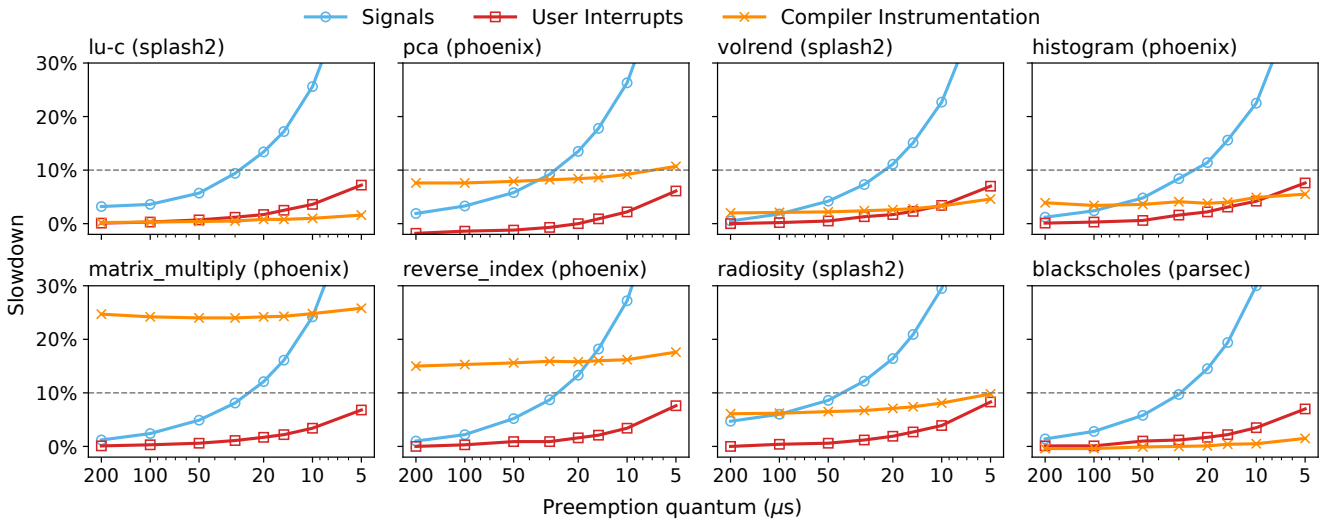



Figure 1: Slowdown of eight programs with different preemption mechanisms. The preemption quantum (x-axis) represents the time between consecutive preemptions and the slowdown (y-axis) indicates how much slower the program runs (as a percentage) compared to running without preemption. For a slowdown tolerance of 10% or less, user interrupts support a 5 $\mu$s preemption quantum, while signals support about 30 $\mu$s. Compiler Instrumentation is implemented using Concord [37].

ble overhead of each mechanism. We ran this experiment with 24 programs from three benchmark suites: Splash-2 [16], Phoenix [13], and Parsec [26]. In addition, we report the end-to-end cost of preemption (including context switching to a different thread) for latency-sensitive applications in §4.2.

The full results for all benchmark programs are provided in Appendix C. For a representative set of eight programs, Figure 1 shows how much slower each program runs (y-axis) due to receiving periodic signals (blue lines), as we vary the preemption quantum (x-axis). With a large quantum, the slowdown is negligible. For example, with a quantum of 100 $\mu$s, the slowdown for most programs is around 3%. However, as the frequency increases so does the cost, so that with a preemption quantum of 10 $\mu$s, the eight programs each experience a slowdown of roughly 25% or more, due to the overhead of signals.

Overall, handling a single signal on an application core slows down the program by about 2.4 $\mu$s. The dominant cost is the transitions between user and kernel space to receive and deliver the signal; these add up to about 1.4 $\mu$s. The remaining cost is due to factors such as additional branch mispredictions and cache misses that occur within the program, due to running kernel code and accessing kernel data structures while handling the signal. Leveraging virtualization features to use normal hardware interrupts rather than signals can reduce the per-signal overhead to 0.6 $\mu$s [39], but in practice it can be challenging to deploy systems that require these features.

## 2.2 Compiler Instrumentation

With compiler instrumentation, preemption is implemented by inserting code at specific points throughout an application that checks if the currently running user-level thread should yield the CPU. Typically, checks are added at function entry points and loop back edges, ensuring that the thread will yield regardless of control flow.

Similar to signals, compiler instrumentation typically relies on a timer thread. However, in this approach, the timer thread indicates that a user-level thread should yield by updating a shared variable in memory. This variable is periodically checked (polled) by the user-level thread to determine if it should yield. While each check is relatively inexpensive, the polling frequency depends on program control flow. Consequently, polling overhead varies across workloads and can be quite high and difficult to predict. This raises several issues. In code with tight loops or small recursive functions, these costs can become prohibitive [14]. Also, at higher preemption frequencies, polling may not take place often enough to ensure that a thread always yields in a timely manner.

Different approaches have been explored to mitigate the problem of overhead. For example, in its first decade Go only instrumented function entry points, to avoid the impact of polling in tight loops. Unfortunately, this required that programmers understand how Go's preemption mechanism worked and manually added explicit calls to the Go scheduler in loops when needed. Aside from the added cognitive burden this imposed, when overlooked, this could lead to high latencies and even program freezes that were difficult to debug [14]. When Go developers attempted to add checks to loop back edges they found that while this only increased the average runtime on a suite of benchmarks by around 7%, at worst it could increase runtime for an individual benchmark by as much as 96% [14]. Thus, they abandoned this approach in favor of signals.

Recent systems have attempted to address these overhead challenges through a collection of different approaches, such as loop unrolling [37, 43]. We use Concord [37] as a repre-

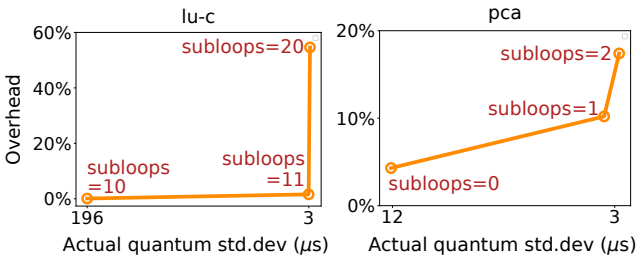



Figure 2: Small changes to the subloops parameter in Concord's compiler instrumentation can significantly impact preemption overhead and timeliness (standard deviation of the measured interval between preemptions). The x-axis uses a log scale.

sentative example that employs compiler instrumentation and reduces polling overhead by placing probes more strategically. We re-run the microbenchmark described above (§2.1), using Concord's LLVM passes to instrument each program. Figure 1 (orange lines) shows the results. While compiler instrumentation can achieve low overhead for some programs (such as *lu-c* and *blackscholes*) when they are well configured, other programs (*pca*, *matrix_multiply*, *reverse_index*) still suffer from significant overhead due to tight loops.

Furthermore, it is non-trivial to find a good configuration for each program, as there are multiple parameters that need to be tuned. For example, Concord features a parameter called `subloops` that limits the number of sub-loops in which probes can be inserted within each outermost loop. Slight variations in this parameter can result in large changes in preemption timeliness and overhead, as shown in Figure 2. For *lu-c* (left), setting `subloops` to 10 results in a high standard deviation of preemption intervals, indicating poor timeliness, while increasing the value to just 11 brings the standard deviation to an acceptable level. Further increasing `subloops` to 20 or more increases preemption overhead to greater than 50%. Similarly, in the benchmark *pca*, increasing `subloops` from 0 to 2 raises the overhead by more than 13%.

Another parameter than can be tuned with compiler instrumentation is the degree of loop unrolling; Concord uses loop unrolling to reduce preemption overhead in programs with tight loops. While loop unrolling can indeed reduce overhead for some programs, such as *blackscholes*, excessive unrolling can increase pressure on the instruction cache and microop cache [11], potentially causing some programs to run more slowly [3, 4]. For example, configuring *matrix_multiply* to unroll loops four times results in a preemption slowdown of 31%, compared to 25% without loop unrolling. Figure 1 illustrates the performance of compiler instrumentation under the best configuration of these parameters that we could find, but tuning these parameters required significant manual effort.

## 2.3 User Interrupts

Recently Intel's Sapphire Rapids CPUs [20] introduced support for sending inter-processor interrupts directly in user space, via *user interrupts*. Similar to signals, user interrupts enable asynchronous preemption that avoids the polling overheads imposed by compiler instrumentation. In contrast with signals, user interrupts avoid the costly overheads of transitions between kernel and user space.

Normally, inter-processor interrupts (IPIs) are sent from one core to another by programming the APIC from the kernel. This directs control flow to go through the interrupt vector table on the receiving core when an interrupt is delivered. Unfortunately, when communicating user-space events from one core to another, i.e., with signals, this requires multiple expensive protection boundary crossings.

User interrupts offer an alternative by providing a hardware fast path for sending and receiving interrupts. User space code cannot simply be allowed to send interrupts to other processes without access control, so, similar to other kernel-bypass mechanisms, user interrupts delegate access control to the operating system. For a *sender thread* to send user interrupts to a *receiver thread*, both must register with the kernel first. Once registration is complete, the kernel does not participate in the sending and receiving of user interrupts. The processor can directly consult a kernel-managed table about where to route interrupts, allowing a sender thread to ensure that user interrupts are delivered to a specific receiver thread.

To measure the overhead of receiving user interrupts, we again configure a timer core to periodically interrupt an application running on a separate core, this time by sending user interrupts. Figure 1 (red lines) shows that, as with signals, the program slowdown increases as the preemption frequency increases. However, the overhead of handling a single user interrupt is much lower than that of handling a signal—only 0.4 $\mu$s compared to 2.4 $\mu$s—so the slowdown remains tolerable down to much smaller quantum sizes. User interrupts have so much lower overhead because they avoid transitions between user and kernel space as well as extra branch mispredictions and cache misses within the program.[1] Thus, for a given acceptable amount of application slowdown, user interrupts can enable much more frequent preemption. For example, for a slowdown of at most 10%, user interrupts can support a preemption quantum of 5 $\mu$s, whereas for signals this is only about 30 $\mu$s.

While user interrupts uniformly outperform signals, the comparison with compiler instrumentation is more nuanced. For small quanta (10 $\mu$s or lower), Concord can achieve lower overhead than user interrupts for some programs such as *lu-c*. In contrast, for larger quanta (greater than 10 $\mu$s), the slowdown from user interrupts is lower or at least comparable to that of compiler instrumentation, especially in programs with tight loops, where it is 15–25% lower, as seen in *matrix_multiply* and *reverse_index*. This is because, regardless of the quanta, programs using compiler instrumentation must

[1] For example, using perf [12] to measure the *linpack_bench* program [8], we observed that user interrupts result in 29% fewer L1 cache misses and 12% fewer branch mispredictions compared to signals at a 5 $\mu$s quantum.

check the instrumented probes at the same locations as they would for smaller quanta, whereas user interrupts incur overhead only when preemption actually occurs.

## 2.4 Summary

We find that user interrupts consistently provide lower overhead than signals, across all programs we evaluated. While the overhead of compiler instrumentation can be lower than that of user interrupts for very fine-grained preemption with some programs, this overhead varies significantly and can be quite high. Furthermore, striking a good balance between preemption overhead and timeliness can require careful tuning of multiple parameters. While it may be possible to mitigate some of these issues with more sophisticated compiler techniques, with existing approaches to compiler instrumentation, fine-grained preemption remains fragile, entailing unpredictable and potentially high costs. User interrupts offer a compelling alternative due to their consistent overheads, which remain low for all but the highest preemption frequencies.

# 3 Preemptive User-Level Schedulers

To understand how user interrupts impact preemptive scheduling in practice, we implemented preemptive user-level scheduling in two different runtime systems. The two systems allow us to explore different regions of the design space. First, Aspen-KB extends Caladan [33], a user-level runtime and scheduler implemented in C that uses kernel-bypass networking and is heavily optimized to achieve microsecond-scale latencies. Second, Aspen-Go extends the popular Go runtime, which relies on the underlying operating system's network stack and is designed for programmer productivity rather than bleeding-edge performance. At a high level, these two runtime systems have similar overall architectures. Each runtime core has its own runqueue(s) of user-level threads (or goroutines) and runtime cores balance work using work stealing.

To enable preemptive scheduling, both runtime systems dedicate one core in each runtime to act as the timer core. The timer core periodically sends user interrupts to runtime cores, preempting the currently running user threads. Note that the timer core is not required to process incoming or outgoing packets (though in Go it may), enabling the timer core to scale to support larger numbers of cores than in systems where the timer core is also responsible for processing all incoming network traffic and dispatching packets to cores [37, 39, 43].[2]

In the remainder of this section, we describe four general factors that impacted the design of both of these preemptive schedulers (§3.1), as well as the specific design and implementation of our kernel-bypass scheduler Aspen-KB (§3.2) and Go-based scheduler Aspen-Go (§3.3).

[2]This decision involves a tradeoff. Distributed load balancing via work stealing enables better scalability than systems in which a centralized core dispatches incoming tasks, such as Shinjuku [39], Concord [37], or TQ [43]. Yet, this comes at the cost of potentially less effective load balancing [44].

## 3.1 Design Factors

### 3.1.1 Unnecessary Preemptions

Existing systems implement a preemption policy that issues preemptions to all runtime cores at a given interval to preempt all currently running tasks [37,38,42]. However, preemption is not always necessary—for example when there are no queued tasks for a core to run or when a thread has just yielded voluntarily—and preempting unnecessarily slows down running tasks. Eliminating these unnecessary preemptions can significantly reduce overhead.

To reduce the number of preemptions as much as possible, the timer core requires visibility into all sources of queued work (e.g., threads, incoming network packets) so that it can determine if there is another task that the core could handle. In addition, the timer core needs visibility into when context switches between user-level threads occur, so that it can avoid preempting a thread that just began running.

### 3.1.2 Non-Preemptible Code

User-level threads are not always safe to preempt, often because of nonreentrant code. For example, code that may not be safe to preempt includes:

**Scheduler code that holds locks.** Since the thread scheduler runs in userspace, a user interrupt may arrive while a thread is executing scheduler code. If that thread holds a lock that protects scheduler state, then it may be unsafe to preempt it, because it will not be possible to acquire the scheduler's lock to reenter the scheduler and context switch to a different user-level thread. This is an example of non-reentrant code.

**Other code that holds locks.** In addition, preempting other code that holds locks may lead to poor performance, e.g., if it delays critical functionality such as polling the network stack, or if it prevents other threads from being able to run.

**Application code that uses kernel-level-thread state.** Existing applications often rely on state that is associated with a kernel-level thread. For example, thread-local storage (TLS) is associated with kernel-level threads, and many library functions such as `malloc` rely on TLS [27, 54]. When existing applications are ported to use user-level threads, regions of code that use TLS or library functions that use TLS may not be safe to preempt. This is because multiple user-level threads can reside on the same kernel-level thread, thus sharing the same kernel-thread-local state. As a result, preemption may cause them to have interleaved access to the same state, resulting in incorrect behavior.

In some cases, non-preemptible code is isolated in distinct regions of code, e.g., in calls into the scheduler or to `malloc`. Existing runtimes typically handle these kinds of non-preemptible code by detecting when a signal arrives during non-preemptible code, and deferring or skipping the preemption. For example, in Go, the runtime consults metadata generated by the compiler to determine if the program is at

| Task | Rate of Non-Preemptible Code Regions | Use of Extended Registers |
|---|---|---|
| SPEC-mcf | only a few | none |
| linpack_bench | only a few | continuously |
| base64 | only a few | only at initialization |
| RocksDB GET | 4 calls/$\mu$s | 30 calls/$\mu$s |
| RocksDB SCAN | 0.039 calls/$\mu$s | 38 calls/$\mu$s |

Table 1: The frequency of non-preemptible code regions and vector register usage across four programs. Numbers in the table with the unit "calls/$\mu$s" represent the number of calls into functions involving non-preemptible code or extended registers per microsecond.

an unsafe point, and simply returns to the interrupted goroutine if so [6]. In other runtimes, the linker or developer wraps non-preemptible regions of code so that the runtime can track whether it is currently executing code that is not safe to preempt. If a signal arrives during one of these regions, the signal handler defers the preemption until the user-level thread exits the region [27, 33, 37, 45].

User interrupts provide a promising new approach to handling non-preemptible code, which is to defer interrupt delivery in hardware. With this approach, preemption is deferred by using the new `clui` and `stui` instructions to disable and reenable delivery of user interrupts. The benefit of this approach is that if multiple user interrupts arrive within a region of non-preemptible code, only one will be delivered when user interrupts are reenabled; this interrupt coalescing avoids the 0.4 $\mu$s overhead of receiving each unnecessary user interrupt.

However, we found that hardware-based deferral can add significant overhead, because of the cost of the current hardware instructions and because non-preemptible code regions can be short and frequent. We found that a single pair of `stui` and `clui` instructions costs about 18 ns. This is much lower than the 600 ns required to mask and unmask signals, but is still non-trivial. Table 1 shows that some programs execute non-preemptible regions of code infrequently, but that requests for RocksDB [15], a key-value store, frequently call `malloc` and `free`. Deferring preemption in hardware would slow down RocksDB GET requests by 7%, whereas deferring preemption in software adds only 1-2 ns of overhead for each region of code (less than 1% slowdown overall).

#### 3.1.3 Extended Registers

When context switching between user-level threads, it is essential to properly save and restore registers, including extended registers such as SIMD or matrix registers. Table 1 presents the frequency and usage patterns of extended registers in some application code. With compiler instrumentation, the compiler can save caller-saved registers, including extended registers if necessary. In contrast, with user interrupts, the hardware only saves the flags, instruction pointer, and stack pointer, requiring the user-level scheduler to save extended registers. Our approach conservatively saves all registers that might be used.

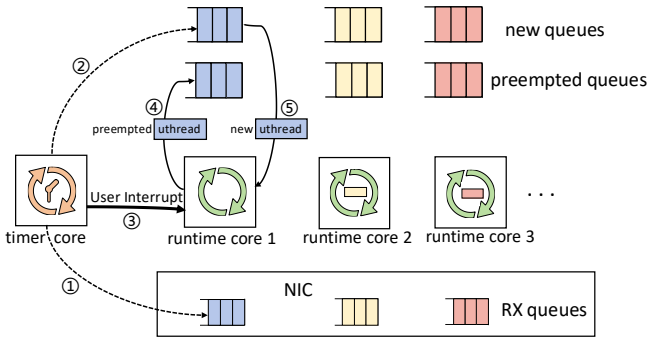



Figure 3: Architecture of Aspen-KB. To preempt a user thread on a runtime core, the timer core first checks if there are pending tasks in its RX queue ① or *new queue* ②. If so, the timer core sends a user interrupt to the runtime core ③. Then, the currently running user thread is suspended and added to the *preempted queue* ④ and the core dequeues and runs the next thread from the *new queue* ⑤.

While this increases the saved state size per thread (e.g., by 2 KB), our measurements show that the overhead is negligible in most cases. A detailed discussion of this and alternative approaches is provided in Appendix A.

#### 3.1.4 Head-of-Line Blocking

Even with periodic preemption of running threads, head-of-line blocking can still occur. First, if the network stack is not polled frequently, newly arriving tasks can experience queueing delays there while lower priority tasks occupy the CPU. Second, if newly runnable tasks are enqueued to a runqueue behind preempted tasks, the new tasks must wait for each preempted task to run for a preemption quantum before they are able to run. Prior work has addressed the latter problem by identifying the task type during processing in the network stack and explicitly prioritizing latency-sensitive tasks [30, 39], but we avoid this approach because task types or service times are not always available a priori. Instead, to prevent head-of-line blocking throughout the runtime, we focus on frequently polling for incoming network packets and prioritizing newly incoming tasks over preempted tasks.

### 3.2 Kernel-Bypass Preemptive Scheduler

Our first preemptive user-level scheduler, Aspen-KB, is built by extending Caladan [33], a user-level runtime and scheduler implemented in C. Figure 3 shows the architecture of Aspen-KB. Aspen-KB bypasses the kernel for both threading and networking. Each core in each runtime has its own runqueue of user-level threads and its own RX queue which it polls to receive incoming packets. Runtime cores balance work—including both threads and incoming packets—among themselves using work stealing. Aspen-KB dedicates one core per runtime to serve as the timer core.

**Unnecessary preemptions.** In Aspen-KB, the timer core checks if preemption is necessary at a fixed preemption quan-

tum. However, the timer core may choose not to preempt every runtime core. The timer core has visibility into incoming packet queues (RX queues) and scheduler state (a timestamp indicating when the currently running thread started running) via shared memory, and only preempts the currently running thread if it has been running for at least the preemption quantum and there is other work (incoming packets, runnable threads, etc.) available on that core. We found that this approach significantly reduces the rate of preemptions (§4.4).

**Head-of-line blocking.** Aspen-KB avoids head-of-line blocking with two techniques. First, it polls the RX queue for newly arriving packets after every preemption; the overhead from this is small due to the kernel-bypass network stack. Second, each core employs a simple two-queue scheduling approach. This is a simplified variant of multi-level feedback queues [29] in which newly runnable threads are initially enqueued to the *new queue*, but once they are preempted, they are enqueued to the *preempted queue* instead. Each core prioritizes the new queue over the preempted queue and uses a longer quantum (e.g., 100 $\mu$s) for the preempted queue. With this approach, the system requires no a priori knowledge of how long each thread will run for, and latency-critical threads that run for less than the preemption quantum are still handled quickly.[3]

**Implementation.** We built Aspen-KB by extending Caladan [33] with a timer core, scheduler features to support user interrupts, and the two-queue scheduling approach. Aspen-KB handles extended registers by saving all general-purpose and AVX-512 registers on every context switch. Additionally, it extends Caladan's shim layer to fully support kernel-level-thread state, including TLS. Specifically, each user-level thread maintains a private copy of its kernel-level-thread state, preventing interleaved access to the same state. Aspen-KB does not modify Caladan's software-based approach to handling non-preemptible code, which defines the functions `preempt_disable()` and `preempt_enable()` to disable and re-enable preemption, respectively, and wraps them around non-preemptible scheduler code. All modifications are implemented in 1,849 LOC. Running an application on Aspen-KB requires a `makefile` to link it with the runtime.

To enable an apples-to-apples evaluation of different preemption mechanisms, we also implement signal-based preemption and compiler instrumentation in Aspen-KB. To implement *signal-based preemption*, we modify Aspen-KB's timer core to send signals in lieu of user interrupts. To implement *compiler instrumentation*, we integrate Concord's approach into Aspen-KB [37]. Specifically, we borrow Concord's LLVM instrumentation passes and apply them to applications. The instrumented applications then run on Aspen-KB, and Aspen-KB's timer core preempts user-level threads by writing to shared memory.

[3] Under overload conditions, preempted threads can languish in the preempted queue. However, we assume that a load balancer will prevent any server from operating in a continuously overloaded state.

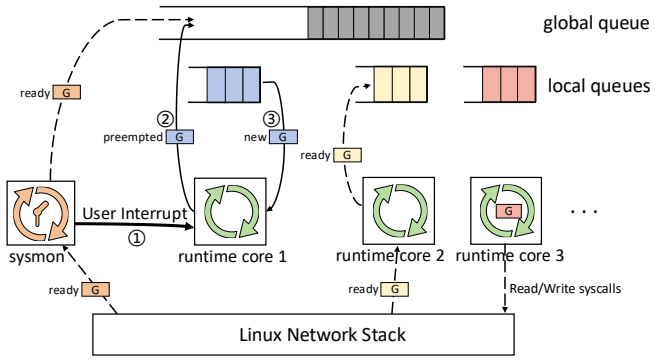



Figure 4: Architecture of Aspen-Go. To preempt a goroutine on a runtime core, sysmon sends a user interrupt to the core ①, triggering it to suspend the current goroutine and place it in the global queue ②, and dequeue a new goroutine from the local queue ③. When an application invokes Go's network APIs, the Go runtime issues a nonblocking system call on its behalf (core 3). To handle I/O completions, runtime cores poll the network stack and add readied goroutines to their local runqueue (core 2). Sysmon can poll as well, but places goroutines in the global queue.

While Junction [32]'s implementation bears similarity to Aspen-KB's—both extend Caladan [33] with support for user interrupts—Aspen-KB differs in its two-queue scheduling approach, its support for compiler instrumentation and signal-based preemption, and how it handles extended registers.

## 3.3 Go-Based Preemptive Scheduler

Our second preemptive user-level scheduler, Aspen-Go, is built by extending the Go runtime [6] (Figure 4). The Go runtime is designed to support high concurrency and implements user-level scheduling via lightweight threads called goroutines. The scheduler maintains one queue of runnable goroutines for each active core in the runtime, as well as a global queue for preempted goroutines or those that become runnable when a system call returns; goroutines in per-core queues are prioritized over those in the global queue. For networking, Go relies on the underlying operating system's network stack. To perform network I/O, the runtime typically issues non-blocking system calls (e.g., `read` and `write`), and a runtime component called *netpoller* handles polling for completion of those operations (e.g., via `epoll` or `kqueue`).

As described above (§2.2), Go implements two forms of preemption. First, it leverages compiler instrumentation by instrumenting function entry points. To handle programs that do not contain frequent function calls (e.g., those with tight loops), Go also uses signals for periodic preemption [14]. A dedicated thread called *sysmon* preempts running goroutines periodically (every 10 ms by default). The sysmon thread also polls the network stack for incoming packets occasionally, if no other runtime core has polled the network stack for at least 10 ms. The sysmon thread sleeps when it is inactive.

Our goal with Aspen-Go is not to overhaul the Go runtime, but rather to make the minimal set of changes necessary in

order to support fine-grained preemption with user interrupts. We configured the sysmon thread to busy spin instead of sleeping, to enable precise preemption at fine granularity, and additionally made the modifications described below.

**Unnecessary preemptions.** The sysmon thread has visibility into when a context switch last occurred on each core, and so it can skip preemptions when the current goroutine has been running for less than the preemption quantum. However, because the Go runtime relies on the OS's network stack, the sysmon thread cannot easily determine whether there are packets queued or not. Thus if the current goroutine has exceeded its quantum, the sysmon thread will preempt it, even if there is no other work (e.g., goroutine or packet) available.

**Head-of-line blocking.** In Go, solely decreasing the preemption quantum provides little performance benefit for networked applications, because incoming packets are not necessarily processed in a timely manner. Incoming packets can be received in one of three ways: (1) an application issues a non-blocking read to check for packets for a specific connection, (2) the scheduler invokes the netpoller to check for incoming packets on any connection; this only occurs when there are no runnable goroutines in the local or global runqueue, or (3) the sysmon thread invokes the netpoller every 10 ms. Thus the Go scheduler generally prioritizes runnable goroutines over incoming network packets. Even with fine-grained preemption of goroutines, packets can remain queued in the network stack until all goroutines in the local and global runqueues have completed or the sysmon thread invokes the netpoller.

To minimize head-of-line blocking in the network stack, the scheduler could invoke the netpoller after every preemption, to check for newly arriving packets. However, in Go, this requires a system call, adding at least 2 $\mu$s of overhead to every preemption. We found that this degrades performance prohibitively, especially for applications that rarely perform network I/O. Instead, Aspen-Go modifies the sysmon thread to invoke the netpoller more frequently, every 100 $\mu$s. While this allows queued packets to be processed much more quickly, it does not entirely eliminate head-of-line blocking. This is because once sysmon processes incoming packets and marks their corresponding goroutines as runnable, it enqueues them to the global runqueue (to avoid contention for per-core runqueues), where they must queue behind preempted goroutines.

An alternative approach for those willing to tolerate more significant changes to Go would be to leverage a kernel-bypass network stack. For example, this could be done using Junction [32]. In this case, polling the network would be much cheaper and could be performed more frequently. However, given our goal of making minimal modifications to Go and to how it can be deployed, we do not adopt these approaches.

**Implementation.** We modified the Go runtime (version 1.21) to support preemption via user interrupts and modified the sysmon thread as described above. Additionally, Aspen-Go disables preemption at `Lock()` and re-enables it at `Unlock()` to prevent preemption in critical sections. This is particularly useful since locks are commonly used in networking packages such as `fasthttp`. These modifications amount to 733 LOC.

## 4 Evaluation

We answer four questions in this evaluation:

1. How do different preemption mechanisms impact the tail latency and throughput of applications in Aspen-KB and Aspen-Go? (§4.1)
2. What is the cost of an individual preemption in Aspen-KB and Aspen-Go? (§4.2)
3. How does the preemption mechanism impact the ideal choice of preemption quantum? (§4.3)
4. What is the impact of Aspen-KB's design decisions? (§4.4)

**Experimental setup.** We conduct experiments using two dual-socket servers, each with 28-core Intel Xeon Gold 5420+ CPUs operating at 2.0 GHz and 256 GB of RAM. Each server is equipped with a 100 Gbits/s Mellanox ConnectX-6 Dx NIC and the two NICs are connected via a 100GbE Mellanox SN2700 switch. We disable hyperthreads because we found that our workloads achieved higher performance in this configuration. We disable TurboBoost, frequency scaling, and c-states. Both of our servers run Ubuntu 22.04.4. Our load-generating machine uses kernel version 6.8.0 while our server machine uses a custom kernel provided by Intel that is based on kernel version 6.0.0 and supports user interrupts [9, 20].

**Kernel-bypass systems evaluated.** Ideally we would directly compare Aspen-KB against state-of-the-art systems that use other preemption mechanisms, such as Shinjuku [39] and Concord [37]. However, comparing against these systems is challenging. First, both were designed to run with an older kernel version (4.4.185), which is not supported by Aspen-KB. Second, all three systems use different policies for load balancing work across cores; Shinjuku dispatches work from a single centralized queue, Concord uses JBSQ [40], and Aspen-KB uses work stealing. These differences in load-balancing policy could significantly impact performance for multicore experiments [44], making it difficult to isolate the impact of the preemption mechanism itself.

Instead, we evaluate five variants of Aspen-KB (§3.2). This includes preemptive schedulers with *signal*-based preemption[4] and *user-interrupt*-based preemption (the default version of Aspen-KB). For compiler instrumentation, we evaluate *Concord*, which uses Concord's default configuration that instruments both function calls and loop backedges [37]. We also evaluate *Concord fine-tuned*, in which we carefully configure the subloops parameter, degree of loop unrolling, and whether function calls are instrumented, to achieve the

[4] Shinjuku uses virtualization hardware to further lower the overhead of preemption [39], so its performance would likely fall between that of our signal-based approach and our user-interrupt-based approach.

best performance for each application. We also evaluate *non-preemptive*, a run-to-completion baseline, by running Aspen-KB with preemption disabled. Unless stated otherwise, we configure the non-preemptive baseline to dedicate 25 cores to the evaluated application, and all other systems to dedicate 24 cores to the application and 1 core to the timer.[5]

**Go variants evaluated.** We evaluate Go under five configurations. *Unmodified* Go refers to the standard Go runtime, version 1.21, which uses both compiler instrumentation and signal-based preemption, every 10 ms. We also applied fine-grained preemption on it, triggering every 50 µs. The remaining configurations run Aspen-Go, as described above (§3.3). *Aspen-Go UINTR* and *Aspen-Go Signals* use user interrupts and signals, respectively. We also evaluate compiler instrumentation alone (*Aspen-Go Compiler*) by disabling preemption via signals or user interrupts. Unless stated otherwise, we configure 8 cores for the application and 1 for sysmon. Go dynamically adjusts its core usage, so at any given time it may use fewer cores.

**Workloads.** We evaluate three applications:

1. RocksDB [15] (version 5.15.10), a popular key-value database developed by Facebook, runs on Aspen-KB. We run bimodal workloads that include GET and SCAN operations, which take about 1 $\mu$s and 260 $\mu$s, respectively.
2. C++ DataFrame [2] (version 1.19.0), a data analysis application, runs on Aspen-KB. Our evaluation runs a workload with *decay*, *ad* (Accumulation/Distribution), *rmv* (Rolling Mid Value), *ppo* (Percentage Price Oscillator) and *kmeans*. Each task type makes up 20% of the total number of tasks.
3. BadgerDB [1] (version 1.6.2), a fast key-value database in Go, runs on Aspen-Go. We populate the database with 10 million keys. Our evaluation involves a bimodal workload, with GET operations taking about 5 $\mu$s and RangeSCAN operations around 800 $\mu$s.

We generate load using Caladan's load generator [33]. For RocksDB and DataFrame, the load generator is configured to send requests over UDP with a Poisson arrival distribution. For BadgerDB, it generates requests over HTTP connections, also following a Poisson arrival distribution.

## 4.1 Application Performance

In this section, we evaluate the impact of different preemption mechanisms on end-to-end application performance in Aspen-KB and in Aspen-Go. Because the preemption quantum that yields the best performance varies across preemption mechanisms and workloads, we show results with the best-performing quantum. We define the "best" quantum as the preemption quantum for which short operations achieve the highest throughput while tail latency remains below a threshold (e.g., in RocksDB, GET tail latency of at most 50 $\mu$s).

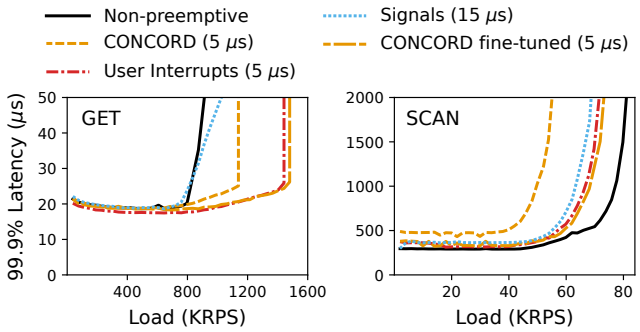



Figure 5: RocksDB performance in Aspen-KB under different preemption mechanisms. The workload is 95% GET and 5% SCAN. KRPS on the x-axis represents kilo-requests per second.

### 4.1.1 RocksDB (Aspen-KB)

RocksDB [15] exemplifies applications that benefit from an extremely low preemption quantum, as it has short-running GET tasks, which typically take less than 1 µs. Figure 5 shows the results of running RocksDB with 95% GET requests and 5% SCAN requests; experiments with different distributions of GET and SCAN requests (95.5%/0.5% and 50%/50%) yielded similar conclusions.

**User-interrupt-based preemption.** As shown in Figure 5, user interrupts (UINTR) are able to mitigate head-of-line blocking and significantly improve performance compared to a system without preemption. For example, for a tail latency limit of at most 50 $\mu$s, user interrupts improve GET throughput by 58.2% compared to the non-preemptive system.

**Signal-based preemption.** In contrast, signal-based preemption provides little benefit for GETs compared to no preemption and it increases the tail latency for SCANs compared to most other approaches. There are three reasons for this lackluster performance. First, as illustrated above, receiving a signal entails an overhead of about 2.4 $\mu$s (§2.3), which can increase execution time for SCANs by up to 16.7% with a preemption quantum of 15 $\mu$s. Second, with signals, the timer thread suffers from poor scalability (§4.4), due to the overhead of sending signals. For this workload, the timer core can only trigger 43% as many preemptions with signals as it can with user interrupts; it is not able to reliably sustain the target preemption quantum. Finally, receiving signals also exhibits poor scalability due to lock contention [54], so adding additional timer cores does not improve performance.[6]

**Compiler instrumentation.** Compiler instrumentation can yield similar improvements as user-interrupt-based preemption, as illustrated by the "CONCORD fine-tuned" lines in

[5] We also dedicate one core to Caladan's IOKernel. The IOKernel is responsible for reallocating cores across applications, but we disable this. Thus in our experiments the IOKernel is only used during initialization; this functionality could be combined with the timer core in the future.

[6] Note that Shinjuku's virtualization-based approach reduces the overhead of sending/receiving interrupts to 14%/48% of the overhead with signals and enables Shinjuku to scale to support at least 22 cores [39].

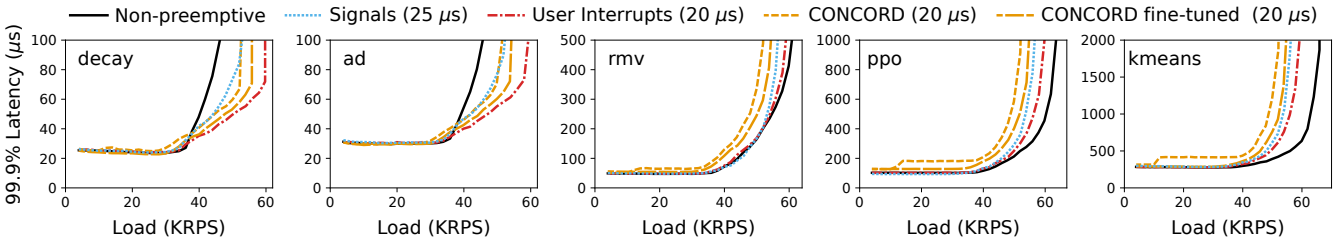



Figure 6: DataFrame performance in Aspen-KB under different preemption mechanisms. Each task type constitutes 20% of the total task count.

Figure 5. However, achieving this performance required deviating from Concord's default instrumentation. With the default instrumentation, a single SCAN operation triggers over 95,000 preemption checks, resulting in a 31.2% slowdown with a 5 $\mu$s quantum (as shown by "CONCORD"). GET operations suffer similarly. In the fine-tuned variant, we disable instrumentation of function calls, which are frequent in RocksDB. This reduces the number of preemption checks per SCAN to 5,000 and decreases the slowdown to only 2.3%.

#### 4.1.2 DataFrame (Aspen-KB)

We evaluate DataFrame [2] with a workload that includes 2 short tasks (*decay* (5 $\mu$s) and *ad* (7 $\mu$s)), one medium task (*rmv* (28 $\mu$s)), and two longer tasks (*ppo* (75 $\mu$s) and *kmeans* (250 $\mu$s)). Because its tasks are longer than the short GETs in RocksDB, DataFrame generally performs best with a longer preemption quantum. Figure 6 shows the results.

With a tail latency limit of 100 $\mu$s, user interrupts achieve the highest throughput for the two types of short-running tasks, about 30% higher than the non-preemptive system and 9% higher than the fine-tuned Concord. For longer tasks, user interrupts do sacrifice some throughput compared to the non-preemptive system. However, with the 20 $\mu$s preemption quantum, user interrupts typically cause less than 2% slowdown for programs (§2.3), thus improving the throughput of short requests with minimal sacrifice.

Compiler instrumentation improves performance for short tasks, but at the cost of significant throughput for longer tasks. Despite efforts to fine-tune Concord's parameters (the number of instrumented subloops and loop unrolling count), both *rmv* and *ppo* still experience significant preemption overhead (3.3 $\mu$s) due to tight loops (§4.2). In contrast, user interrupts incur only 0.37 $\mu$s and 0.32 $\mu$s of overhead, respectively.

#### 4.1.3 BadgerDB (Aspen-Go)

We ran BadgerDB [1] with unmodified Go and Aspen-Go to evaluate how effectively different Go variants can mitigate head-of-line blocking. As shown in Figure 7, simply decreasing the preemption frequency (e.g., to 50 $\mu$s) in unmodified Go does not significantly reduce head-of-line blocking. This is because even with frequent preemption, the unmodified Go runtime does not attempt to poll incoming packets from network stack as long as there are available goroutines in

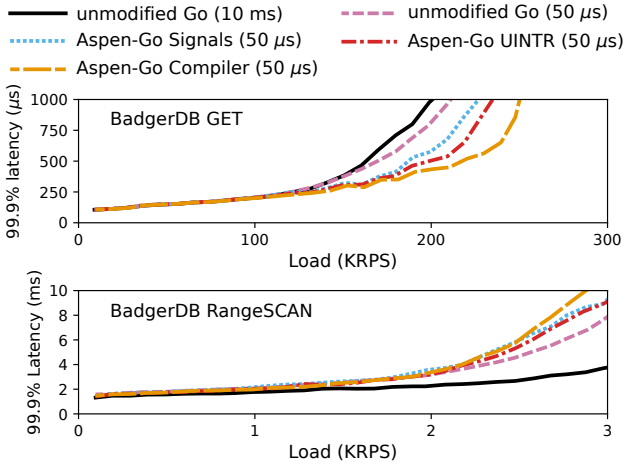



Figure 7: BadgerDB performance with unmodified Go and Aspen-Go. The workload consists of 99% GET and 1% RangeSCAN.

local or global queues (§3.3). As a result, the incoming packets remain blocked in the network stack and more frequent preemption only adds extra slowdown to running goroutines.

In contrast, Aspen-Go UINTR can achieve 17.5% higher throughput for GET requests than unmodified Go (while maintaining a tail latency of at most 1000 $\mu$s), primarily because Aspen-Go actively polls ready packets from the network stack, allowing them to run earlier than in the unmodified version. However, the performance gains from user interrupts in Aspen-Go are significantly smaller than those in Aspen-KB for the following reasons. First, goroutines resulting from newly arrived packets may be placed at the back of the global runqueue (e.g., if they are polled by sysmon) and still need to wait, whereas new user threads in Aspen-KB can be placed in local queues dedicated to new threads and processed more quickly. Second, sysmon cannot detect whether packets are queued or not due to its lack of visibility into the Linux network stack, and thus longer tasks are preempted even when there are no other tasks to handle. Third, the cost of context switching during preemption in Aspen-Go is higher than in Aspen-KB (§4.2). Overall, the design of the Go runtime makes it difficult to adopt some policies that could help reduce head-of-line blocking.

Finally, compiler instrumentation (Aspen-Go Compiler) achieves 6% higher GET throughput than user interrupts, for two reasons. First, polling at function calls serves multiple purposes beyond preemption and, therefore, cannot be disabled in

Aspen-Go UINTR/Signals. Thus, Aspen-Go-UINTR/Signals still incur the overhead of polling. Second, even though these systems are configured with the same preemption quantum (50 $\mu$s), Aspen-Go UINTR/Signals experiences 22% fewer preemptions than Aspen-Go Compiler, making it less effective at mitigating head-of-line blocking. This is because with compiler instrumentation, preemptions occur only at safe points, whereas user interrupts or signals can be delivered anywhere and are ignored if they arrive at unsafe points. In the BadgerDB workload, most unsafe points occur during the execution of `memmove`, which is heavily used in Go, e.g., in assignment statements.

#### 4.1.4 Summary and Takeaways

We found that the preemption mechanism does not significantly impact performance in systems like Go, which are not fully designed for fine-grained preemption. Below, we summarize key takeaways for Aspen-KB and similar low-overhead, kernel-bypass preemptive systems.

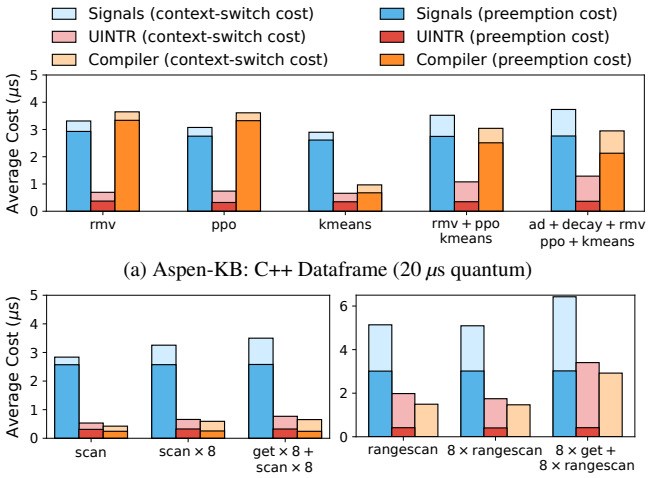
The best choice of preemption mechanism for a given application depends on the optimal preemption quantum. One heuristic for determining a reasonable preemption quantum is that it should be at least as long as the tail runtime of "short" tasks, i.e., tasks that should not be preempted. The guidance in § 4.3 can help more precisely identify the optimal quantum, which might be longer than the runtime of short tasks in order to avoid over-preemption. In either case, if the chosen preemption quantum is at least 10 $\mu$s, user interrupts generally deliver better or similar performance compared to compiler instrumentation. The DataFrame results and some benchmark programs in Figure 1 illustrate this.

In scenarios where extremely low quanta (less than 10 $\mu$s) are necessary, compiler instrumentation may offer better performance, as long as an application is not dominated by tight loops (§2.2). However, the performance gains are limited, as observed in the RocksDB results. This is because compiler instrumentation only reduces the overhead of the preemption mechanism and does not address the context switching overhead (§4.2). Overall, we conclude that user interrupts are likely a better choice for the majority of applications.

## 4.2 The Cost of Preemption

In Figure 1 we measured the cost of preempting a thread, executing an empty handler function, and then immediately returning to the same thread (*preemption cost*). In this section, we also report the *context-switch cost* of preemption, which includes the costs of traversing the scheduler, context switching to a different thread, and any resulting cache contention. Together, the preemption cost and context-switch cost represent the total overhead incurred per preemption.

Figure 8 illustrates the preemption and context-switch costs for various preemption mechanisms, applications, and task combinations. Each application uses its optimal quantum as identified in §4.1. The Aspen-Go results (Figure 8c) compute costs relative to a different baseline than those in Aspen-KB. In Go, compiler instrumentation is used for multiple purposes (e.g., to check if more stack space is needed) and cannot easily be disabled. Thus all data in Figure 8c is collected with compiler instrumentation enabled and we cannot quantify the preemption cost associated with compiler instrumentation.




(a) Aspen-KB: C++ Dataframe (20 $\mu$s quantum)

(b) Aspen-KB: RocksDB (5 $\mu$s)

(c) Aspen-Go: BadgerDB (50 $\mu$s)

Figure 8: The costs of preemption (*preemption cost*) and context switching (*context-switch cost*), for 3 applications. The x-axis represents different task combinations, e.g., $get \times 8 + scan \times 8$ indicates that 8 GET tasks and 8 SCAN tasks run concurrently on one core.

We make four observations. First, with larger preemption quanta, the compiler instrumentation costs are amortized over fewer preemptions, yielding higher per-preemption costs, as illustrated by the higher costs for Dataframe than for RocksDB; this is consistent with Figure 1. Second, the context-switch cost in Aspen-KB is not negligible and can even exceed the preemption cost in some workloads (e.g., with 8 GET tasks and 8 SCAN tasks). This further indicates that, although compiler instrumentation may incur lower overhead than user interrupts in some workloads, the improvement in total cost is relatively small. Third, context-switch costs are slightly higher with user interrupts than with compiler instrumentation for some workloads, e.g., 36% higher for the *rmv*, *ppo*, *kmeans* combination. This may be due to saving extended registers (§3.1.3), which is unique to user interrupts. Finally, the context-switch cost in Aspen-Go is higher than in Aspen-KB (1.3-3.0 $\mu$s vs. 0.2-0.9 $\mu$s), mainly because of the Go runtime's more complex scheduler logic (e.g., checking safepoints). Despite this, user interrupts and compiler instrumentation still incur significantly lower total cost compared to signals.

## 4.3 Preemption Quantum

We evaluate how the preemption quantum impacts performance in Aspen-KB by running RocksDB with several different preemption quantum. In each case, we determine the

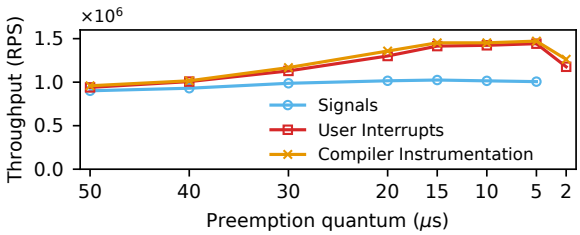



Figure 9: RocksDB GET throughput with different quanta.

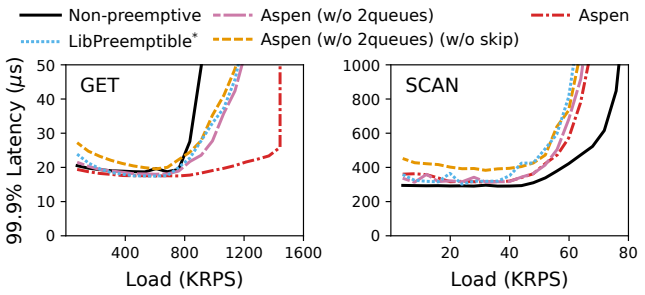



Figure 10: RocksDB performance achieved by LibPreemptible*, Aspen-KB, and other variants. "w/o 2queues" indicates that the 2-queue policy is disabled, while "w/o skip" means that the policy of skipping unnecessary preemption is disabled.

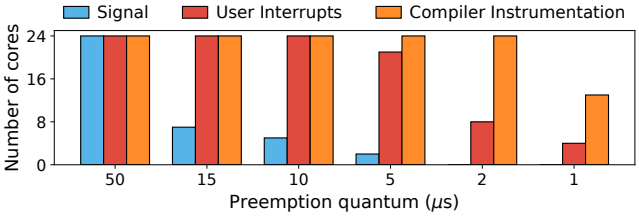



Figure 11: The number of application cores that a single timer core can support under the three preemption mechanisms.

max GET throughput for which GET operations maintain a tail latency of 50 $\mu$s or less; Figure 9 shows the results. For all mechanisms, decreasing the preemption quantum improves performance up to a point; beyond this point, the overhead of extra preemptions outweighs the benefits of mitigating head-of-line blocking and throughput degrades. For both user interrupts and compiler instrumentation, the best quantum for this workload is 5 $\mu$s; for signals it is 15 $\mu$s. The best quantum depends on the workload; across different applications and ratios of GET/SCAN requests in RocksDB, we have observed ideal quanta for user interrupts ranging from 2 $\mu$s to 20 $\mu$s.

In Aspen-Go, we found that with a slowdown tolerance of at most 5%, signal-based preemption limits the quantum to no lower than 100 $\mu$s. In contrast, user interrupts further reduce it to 30–50 $\mu$s, providing tighter tail latency guarantees (Appendix B).

## 4.4 Impact of Aspen-KB's Design Decisions

**Preemption Polices.** We evaluated the effectiveness of Aspen-KB's policies and compare them to the approach of LibPreemptible [42], a recent system that uses user interrupts for preemption. We run Aspen-KB with various policies disabled, and also approximate LibPreemptible's policies in Aspen-KB. Specifically, for a given load, we test different preemption quanta and select the best result as LibPreemptible's result, modeling its ability to find a suitable quantum through adaptive adjustment. This approximation achieves the best possible performance for a workload that does not have significant variation over time; we denote this as LibPreemptible*.While LibPreemptible's APIs allow users to use two queues to prioritize short requests over long, the user-defined queue for preempted tasks must be shared across all cores, as users can not control which cores tasks run on. Thus it would be difficult to support a two-queue policy in a scalable way with LibPreemptible, so we do not apply the two-queue policy to LibPreemptible*.

Figure 10 show the performance of RocksDB achieved by LibPreemptible, Aspen-KB, and other variants of Aspen-KB. Aspen-KB's two-queue policy yields significant performance improvements for both kinds of requests, by allowing incoming requests to be prioritized, and by reducing the amount of preemptions experienced by longer SCAN requests. The policy of skipping unnecessary preemptions benefits SCAN requests the most at lower loads, when there is less likely to be another task to run. Under such load, even though the preemption quantum is 5 $\mu$s, most preemptions can be skipped and the interval between consecutive preemptions is typically more than 30 $\mu$s.

Compared to LibPreemptible*, Aspen-KB handles head-of-line blocking more effectively, keeping the tail latency of short requests low. This is largely because of Aspen-KB's two-queue policy. Furthermore, Aspen-KB reduces the slowdown to preempted tasks. LibPreemptible adjusts the preemption quantum based on past latencies and offered load but updates slowly (every 10 seconds), making it unresponsive to real-time load changes. Instead of explicit quantum adjustments, Aspen-KB skips unnecessary preemptions, allowing rapid, real-time adaptation based on the current load.

**Timer Scalability.** We evaluated the scalability of Aspen-KB's timer core. To stress the timer core, we run a workload that consists of long-running tasks so that the timer core needs to issue a preemption to each core every preemption quantum. We vary the quantum, and for each quantum measure the maximum number of application cores to which the single timer core can reliably send at least 99% of the expected number of preemptions. Figure 11 shows the results.

With a preemption quantum as large as 50 $\mu$s, all three mechanisms are able to support all 24 cores that are available to them. With this large quantum, the timer core mostly calls `rdtsc` and busy-spins. With smaller quanta, the timer scalability depends on the cost of triggering preemptions, which varies significantly across the three mechanisms. Sending a signal takes 1.7 $\mu$s, a user interrupt 200 ns, and the compiler-based approach incurs negligible overhead, requiring just a

single shared memory write. Consequently, with a 5 $\mu$s quantum, the timer scales to only 2 cores with signals but reaches 22 with user interrupts and 24 with compiler instrumentation. This benchmark represents a lower bound on the performance of the timer core; in more realistic workloads one timer core can typically support more application cores due to Aspen-KB's preemption policy, which can skip 70% of preemptions.

## 5 Related Work

**User-level threading.** To avoid the overhead of kernel-based scheduling, many systems schedule threads in userspace, adopting an *M:N* threading model. Examples include the Go runtime [6], $\mu$Threads [18], Caladan [33], Arachne [49], and many others [10, 21, 37, 45, 53, 54, 57]. Aspen-KB and Aspen-Go are examples of such user-level schedulers.

**Preemption mechanisms.** Many existing user-level schedulers adopt cooperative concurrency; those that are preemptive typically employ signal-based preemption or compiler instrumentation. For example, Go [6] and Argobots [53] use signal-based preemption. Shinjuku improves upon signals by virtualizing the APIC for lower overhead IPIs [39].

Compiler instrumentation can be implemented with or without a timer core. With a timer core, the compiler inserts instrumentation that checks shared memory—which is written to by the timer core—to determine when the core should yield. Go [6], Wasmtime [19], and Concord [37] take this approach. Alternatively, the instrumentation itself can estimate how much time has elapsed, by tracking the number and/or duration of instructions that have elapsed, and yield the CPU once the metric exceeds a threshold [24, 34, 43].

Both user-level interrupts [46, 51] and user-level exceptions [55] were first proposed decades ago. The new "user interrupts" feature in Intel CPUs makes user-level interrupts available as a preemption mechanism for userspace schedulers today [20]. LibPreemptible [42], Junction [32], and Skyloft [38] use user interrupts for core scheduling or timeslicing. These works are complementary to ours; none studies how user interrupts compare to compiler instrumentation, addresses head-of-line blocking within the network stack, or studies fine-grained preemptive scheduling in the context of a widely used runtime such as Go. Finally, xUI proposes processor extensions that further reduce the overhead of receiving user interrupts [22]; these can be used to improve the performance of both Aspen variants.

## 6 Conclusion

Preemption can allow us to build more efficient and responsive systems by enabling precise scheduling and mitigating head-of-line blocking. We explored implementing preemption with user interrupts, a new hardware feature. We found that when a system is not constrained by its scheduler's design, user interrupts can offer performance comparable to or better than compiler-instrumentation, while offering a simpler developer experience and more predictable performance.

## Acknowledgments

We thank our shepherd Malte Schwarzkopf and the anonymous reviewers for their feedback. We also thank the Supermicro JumpStart program for providing access to 4th generation Intel Xeon processors for our initial benchmarks. This work was funded in part by a Google Research Scholar Award and a gift from Cisco.

## References

[1] BadgerDB: Fast key-value db in go. https://github.com/dgraph-io/badger.

[2] C++ DataFrame for statistical, financial, and ML analysis. https://github.com/hosseinmoein/DataFrame.

[3] Deep diving into llvm loop unroll. https://yashwantsingh.in/posts/loop-unroll.

[4] Excessive loop unrolling. https://github.com/llvm/llvm-project/issues/42332.

[5] Go performance dashboard. https://perf.golang.org/search?q=upload%3A20171003.1+%7C+upload-part%3A20171003.1%2F3+vs+upload-part%3A20171003.1%2F1.

[6] The Go programming language. https://go.dev/.

[7] Intel® 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[8] LINPACK_BENCH - the linpack benchmark. https://people.math.sc.edu/Burkardt/c_src/linpack_bench/linpack_bench.html.

[9] Linux kernel with support for user interrupts. https://github.com/intel/uintr-linux-kernel.

[10] Loom - fibers, continuations and tail-calls for the JVM. https://openjdk.org/projects/loom/.

[11] Optimizing subroutines in assembly language. https://www.agner.org/optimize/optimizing_assembly.pdf.

[12] perf: Linux profiling with performance counters. https://perfwiki.github.io.

[13] Phoenix benchmark suite. https://github.com/kozyraki/phoenix.

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

[15] RocksDB: A persistent key-value store for flash and RAM storage. https://github.com/facebook/rocksdb.

[16] Stanford parallel applications for shared-memory (SPLASH-2) programs. https://github.com/staceyson/splash2.

[17] User interrupt compiler guide. https://github.com/intel/uintr-compiler-guide/blob/uintr-gcc-11.1/UINTR-compiler-guide.pdf.

[18] uThreads: Concurrent user threads in C++(and C). https://github.com/samanbarghi/uThreads.

[19] Wasmtime. https://docs.wasmtime.dev/.

[20] x86 User Interrupts support. https://lwn.net/Articles/869140/.

[21] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[22] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. Extended user interrupts (xUI): Fast and flexible asynchronous notification without polling. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.

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

[24] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of ACM International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

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

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[27] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *USENIX Annual Technical Conference*, pages 465–477, 2020.

[28] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. Bytehtap: bytedance's htap system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment*, 15(12):3411–3424, 2022.

[29] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, pages 335–344, 1962.

[30] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 621–637, 2021.

[31] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–25, 2013.

[32] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 55–73, 2024.

[33] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–297, 2020.

[34] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[35] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[36] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. Low-latency transaction scheduling via userspace interrupts. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2025.

[37] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 466–481, 2023.

[38] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A general high-efficient scheduling framework in user space. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 265–279, 2024.

[39] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for $\mu$second-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 345–360, 2019.

[40] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX Annual Technical Conference*, pages 863–880, 2019.

[41] Guoliang Li and Chao Zhang. Htap databases: What is new and what is next. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 2483–2488, 2022.

[42] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 922–936, 2024.

[43] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–319, 2024.

[44] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for microsecond-scale tasks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–18, 2022.

[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 361–378, 2019.

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

[47] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner*, pages 4–20, 2014.

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

[49] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 145–160, 2018.

[50] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 315–332, 2020.

[51] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. *ACM SIGARCH Computer Architecture News*, 38(1):311–322, 2010.

[52] 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. A cloud-scale characterization of remote procedure calls. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 498–514, 2023.

[53] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level

threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.

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

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

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

[57] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 195–211, 2021.

# A Extended Registers

When context switching between user-level threads, care must be taken to ensure that the appropriate registers are saved and restored. This includes any extended registers that are being used by the preempted thread, such as SIMD registers or matrix registers.

When compiler instrumentation is used as the preemption mechanism, preempting a user-level thread simply involves calling a function. In this case, the registers that are in use are known at compile time, and the compiler can save just the caller-saved registers that are in use, including extended registers if necessary [31].

In contrast, with user interrupts, the user-level scheduler is responsible for saving almost all registers. When a user interrupt is delivered, the hardware saves the flags, instruction pointer, and stack pointer. The compiler also saves general-purpose callee-saved registers if they are used by the interrupt handler function [17]. However, the scheduler is responsible for saving any additional registers, including SIMD registers. The scheduler typically has no knowledge of which registers a program is using at runtime, and thus for correctness must conservatively save all extended registers.

We ran a microbenchmark program to measure the cost of saving extra extended registers. In this benchmark, multiple user-level threads perform pointer chasing, each within their own independent region of memory. All threads run on the same core; we preempt them periodically and measure the overhead per preemption. We found that, in the worst case, saving all AVX-512 registers (2 KB) could add over 700 ns of overhead to each preemption, compared to saving only the general-purpose registers. This worst-case overhead occurred when the memory accessed collectively by the threads just barely fit into a given cache layer (e.g., the L1 cache), so that the extra memory consumed by saving extra registers caused evictions to the next layer of cache. These evictions make the benchmark program run much slower, as the chasing pattern was deliberately designed to be highly random, minimizing the effectiveness of cache prefetching. However, for other memory region sizes, the overhead of saving all AVX-512 registers per context switch in this benchmark program remained below 40 ns.

We considered three alternative approaches for handling extended registers. First, one could defer preemption during code that uses them, but unfortunately this approach does not work well for many applications. As shown in Table 1, some workloads such as linpack use SIMD registers continuously throughout their execution, with no clear point to defer preemption to. With RocksDB operations, SIMD registers are confined to specific functions, but these functions are called so frequently that deferring preemption would slow down operations by several percent.

A second approach is to use the instructions `XSAVEC` and `XRSTOR` to save and restore registers, as in prior work [32].

These instructions detect which "state components" are in use and save only active components [7]. However, we found that using these instructions added 100-150 ns to each context switch, compared to only 10-30 ns when saving all AVX-512 registers one by one, due to the high fixed costs of these instructions. Furthermore, these instructions track state components only at coarse granularity (e.g., all AVX-512 registers are in the same state component), limiting their ability to optimize which registers are saved.

Finally, it may be possible to reduce overheads by having the compiler identify the set of registers that are used by each program and create custom functions that save and restore only those registers. For the applications we evaluated, saving and restoring all AVX-512 registers on every context switch yielded sufficiently low overhead, but if register state continues to grow in the future, or if applications use additional extended registers such as matrix registers, it may be beneficial to explore this approach.

## B Preemption Quantum in Aspen-Go

We ran several microbenchmarks on both Aspen-Go UINTR and Aspen-Go Signals to gain a broader understanding of the preemption overhead at different preemption quanta in Aspen-Go. For each benchmark program, we ran 10 goroutines of the same program concurrently on a single core, with preemption occurring at every quantum. These microbenchmarks are from the benchmark suite [5], which Go developers used to measure preemption slowdown [14]. Figure 12 presents the results. With a slowdown tolerance of at most 5%, using signals as the preemption mechanism limits the quantum to no lower than 100 $\mu$s. In contrast, user interrupts provide finer-grained preemption, allowing for preemption quanta as small as 30-50 $\mu$s.

## C Preemption Slowdown

Table 2 presents the preemption overhead of three mechanisms—signals, user interrupts, and compiler instrumentation—measured across 24 programs from the Splash-2, Phoenix, and Parsec benchmark suites. Section 2 highlights representative benchmarks and compares the three preemption mechanisms.

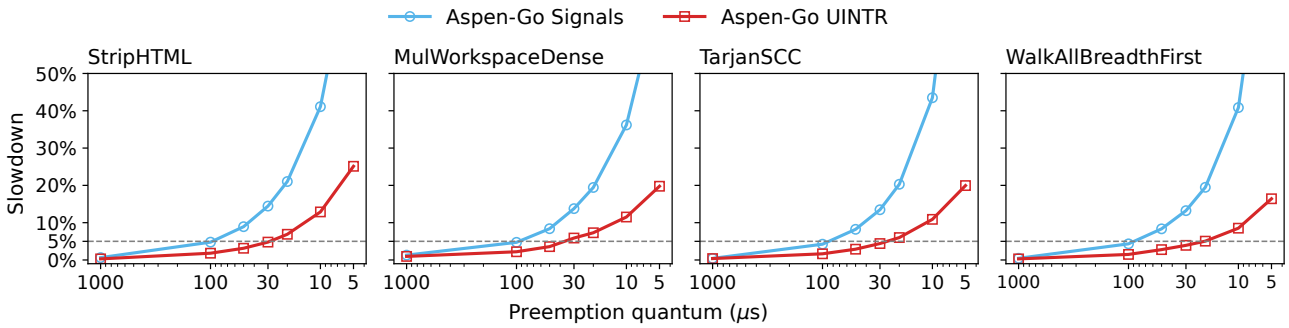



Figure 12: The preemption slowdown with different preemption quanta in Aspen-Go UINTR and Aspen-Go Signals.

| **Program** | **User Interrupts** | | | **Concord** | | | **Signals** | | |
|---|---|---|---|---|---|---|---|---|---|
| **quantum** | 50 $\mu$s | 20 $\mu$s | 5 $\mu$s | 50 $\mu$s | 20 $\mu$s | 5 $\mu$s | 50 $\mu$s | 20 $\mu$s | 5 $\mu$s |
| water-nsquared | 0.6% | 1.2% | 6.2% | 0.7% | 1.2% | 3.3% | 5.1% | 12.6% | 48.7% |
| water-spatial | 1.3% | 1.9% | 9.3% | 1.5% | 1.8% | 3.5% | 5.1% | 12.2% | 47.6% |
| ocean-cp | 0.3% | 1.9% | 6.1% | 1.7% | 2.4% | 3.9% | 4.8% | 11.1% | 47.7% |
| volrend | 0.8% | 1.9% | 7.8% | 2.2% | 2.6% | 4.6% | 4.2% | 11.1% | 45.3% |
| fmm | 0.6% | 1.6% | 7.3% | -1.7% | -1.7% | 0.7% | 5.2% | 13.4% | 47.6% |
| raytrace | 0.3% | 1.7% | 7.1% | 1.8% | 2.2% | 3.9% | 5.5% | 12.6 % | 52.1% |
| radix | 0.6% | 1.7% | 6.5% | -0.6% | -0.4% | -0.2% | 4.7% | 11.1% | 43.7% |
| fft | 0.8% | 1.9% | 7.3% | 2.5% | 2.7% | 4.4% | 5.2% | 12.2% | 43.0% |
| lu-c | 0.7% | 1.8% | 7.4% | 0.4% | 0.8% | 1.6% | 5.7% | 13.4% | 49.6% |
| lu-nc | 0.7% | 1.6% | 6.3% | -0.8% | -0.7% | -0.4% | 3.2% | 10.2% | 44.3% |
| cholesky | 0.9% | 2.0% | 7.7% | 6.9% | 7.3% | 9.6% | 4.8% | 12.0% | 46.9% |
| radiosity | 0.5% | 1.8% | 8.5% | 6.5% | 7.1% | 9.8% | 8.6% | 16.4% | 55.2% |
| histogram | 0.7% | 1.8% | 6.8% | 3.6% | 3.8% | 5.5% | 4.8% | 11.4% | 45.7% |
| pca | 0.9% | 2.2% | 7.9% | 7.9% | 8.4 % | 10.7% | 5.8% | 13.5% | 51.2% |
| string_match | 0.7% | 1.8% | 7.0% | 4.5% | 4.9% | 7.1% | 4.3% | 11.2% | 45.3% |
| linear_regression | 0.7% | 1.7% | 6.9% | 0.1% | 0.5% | 2.5% | 4.7% | 11.8% | 44.0% |
| word_count | 0.6% | 1.6% | 6.7% | 1.9% | 2.1% | 3.0% | 5.0% | 12.4% | 47.3% |
| matrix_multiply | 0.6% | 1.7% | 7.0% | 24.0% | 24.2% | 25.8% | 4.9% | 12.1% | 47.9% |
| reverse_index | 1.0% | 2.4% | 8.2% | 15.6% | 15.8% | 17.6% | 5.2% | 13.3% | 54.6% |
| blackscholes | 0.6% | 1.8% | 7.8% | -0.1% | 0.1% | 1.5% | 5.8% | 14.5% | 59.8% |
| fluidanimate | 0.5% | 1.5% | 7.7% | 0.0% | 0.1% | 0.9 % | 4.6% | 12.5% | 55.9% |
| swaptions | 0.3% | 1.4% | 7.3% | 5.9% | 5.7% | 7.2% | 7.4% | 17.4% | 65.5% |
| canneal | 0.5% | 2.4% | 7.2% | -0.2% | -1.1% | 0.9% | 5.1% | 12.5% | 58.5% |
| streamcluster | -0.5% | 1.8% | 6.6% | 4.9% | 6.2% | 7.0% | 6.1% | 16.3% | 60.8% |

Table 2: Preemption slowdown with different preemption mechanisms in benchmark programs.