Optimizing Storage Performance with Calibrated Interrupts

AMY TAI, VMware Research, USA
IGOR SMOLYAR, Technion – Israel Institute of Technology, Israel
MICHAEL WEI, VMware Research, USA
DAN TSAFRIR, Technion – Israel Institute of Technology, Israel

3

After request completion, an I/O device must decide whether to minimize latency by immediately firing an interrupt or to optimize for throughput by delaying the interrupt, anticipating that more requests will complete soon and help amortize the interrupt cost. Devices employ adaptive interrupt coalescing heuristics that try to balance between these opposing goals. Unfortunately, because devices lack the semantic information about which I/O requests are latency-sensitive, these heuristics can sometimes lead to disastrous results.

Instead, we propose addressing the root cause of the heuristics problem by allowing software to explicitly specify to the device if submitted requests are latency-sensitive. The device then "calibrates" its interrupts to completions of latency-sensitive requests. We focus on NVMe storage devices and show that it is natural to express these semantics in the kernel and the application and only requires a modest two-bit change to the device interface. Calibrated interrupts increase throughput by up to 35%, reduce CPU consumption by as much as 30%, and achieve up to 37% lower latency when interrupts are coalesced.

CCS Concepts: • **Information systems** → **Storage architectures;**

Additional Key Words and Phrases: NVMe, hardware interrupts, calibrating interrupts

ACM Reference format:
Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2022. Optimizing Storage Performance with Calibrated Interrupts. *ACM Trans. Storage* 18, 1, Article 3 (February 2022), 32 pages.
https://doi.org/10.1145/3505139

1 INTRODUCTION

Interrupts are a basic communication pattern between the operating system and devices. While interrupts enable concurrency and efficient completion delivery, the costs of interrupts and the context switches they produce are well documented in the literature [7, 30, 67, 74]. In storage, these costs have gained attention as new interconnects such as **NVM Express™ (NVMe)** enable applications to not only submit millions of requests per second, but up to 65,535 concurrent requests [18, 21, 22, 25, 76]. With so many concurrent requests, sending interrupts for every

Amy Tai and Igor Smolyar contributed equally to this research.

Authors' addresses: A. Tai and M. Wei, VMware Research, 3401 Hillview Ave, Palo Alto, CA 94304; email: amy.tai.2009@gmail.com; I. Smolyar and D. Tsafrir, Technion – Israel Institute of Technology Faculty of Computer Science Technion – Israel Institute of Technology Haifa 3200003 Israel; email: xrevolver@gmail.com.

1553-3077/2022/02-ART3 $15.00
https://doi.org/10.1145/3505139

completion could result in an interrupt storm, grinding the system to a halt [40, 55]. Since CPU is already the bottleneck to driving high IOPS [37, 38, 41–43, 46, 70, 77, 82, 83], excessive interrupts can be fatal to the ability of software to fully utilize existing and future storage devices.

Typically, interrupt coalescing addresses interrupt storms by batching requests into a single interrupt. Batching, however, creates a tradeoff between request latency and the interrupt rate. For the workloads we inspected, CPU utilization increases by as much as 55% without coalescing (Figure 15(d)), while under even the minimum amount of coalescing, request latency increases by as much as 10× for small requests, due to large timeouts. Interrupt coalescing is disabled by default in Linux, and real deployments use alternatives (Section 2).

This article addresses the challenge of dealing with exponentially increasing interrupt rates without sacrificing latency. We initially implemented adaptive coalescing for NVMe, a dynamic, device-side-only approach that tries to adjust batching based on the workload, but find that it still adds unnecessary latency to requests (Section 3.2). This led to our core insight that device-side heuristics, such as our adaptive coalescing scheme, cannot achieve optimal latency because the device lacks the semantic context to infer the requester's intent: Is the request latency-sensitive or part of a series of asynchronous requests that the requester completes in parallel? Sending this vital information to the device bridges the semantic gap and enables the device to interrupt the requester when appropriate.

We call this technique *calibrating*[1] interrupts (or simply, cinterrupts), achieved by adding two bits to requests sent to the device. With calibrated interrupts, hardware and software collaborate on interrupt generation and avoid interrupt storms while still delivering completions in a timely manner (Section 3).

Because cinterrupts modifies how storage devices generate interrupts, supporting them requires modifications to the device. However, these are minimal changes that would only require a firmware change in most devices. We build an emulator for cinterrupts in Linux 5.0.8, where requests run on real NVMe hardware, but hardware interrupts are emulated by interprocessor interrupts (Section 4).

Cinterrupts is only as good as the semantics that are sent to the device. We show that the Linux kernel can naturally annotate all I/O requests with default calibrations, simply by inspecting the system call that originated the I/O request (Section 4.1). We also modify the kernel to expose a system call interface that allows applications to override these defaults.

In microbenchmarks, cinterrupts matches the latency of state-of-the-art interrupt-driven approaches while spending 30% fewer cycles per request and improves throughput by as much as 35%. Without application-level modifications, cinterrupts uses default kernel calibrations to improve the throughput of RocksDB and KVell [48] on YCSB benchmarks by as much as 14% over the state-of-the-art and to reduce latency by up to 28% over our adaptive approach. A mere 42-line patch to use the modified syscall interface improves the throughput of RocksDB by up to 37% and reduces tail latency by up to 86% over traditional interrupts (Section 5.5.1), showing how application-level semantics can unlock even greater performance benefits. Alternative techniques favor specific workloads at the expense of others (Section 5).

Cinterrupts can, in principle, also be applied to network controllers (NICs), provided the underlying network protocol is modified to indicate which packets are latency sensitive. We demonstrate that despite being more mature than NVMe drives, NICs suffer from similar problems with respect to interrupts (Section 2). We then explain in detail why it is more challenging to deploy cinterrupts for NICs (Section 6).

[1]To calibrate: to adjust precisely for a particular function [53].

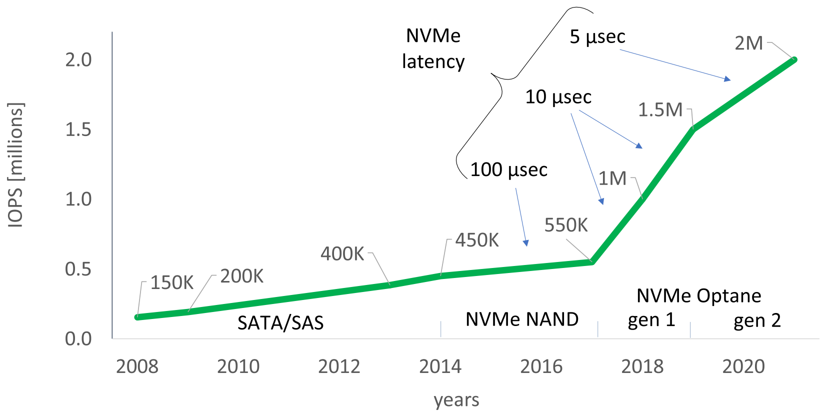

Fig. 1. Storage hardware trends include changes in device interconnect and device media. The NVMe interconnect exposes internal device parallelism, which increases maximum throughput of devices, and new technologies such as Intel Optane decreases device latency when compared to traditional flash.

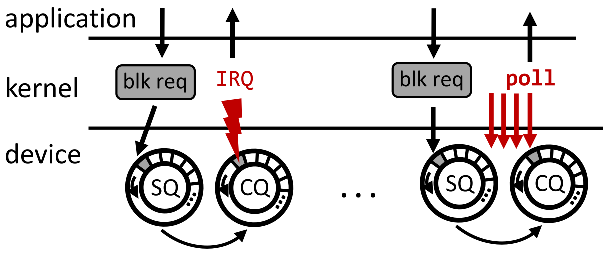

Fig. 2. NVMe requests are submitted through submission queues (SQ) and placed in completion queues (CQ) when done. Applications are notified of completions via either interrupts or polling.

2 BACKGROUND AND RELATED WORK

Disks historically had seek times in the milliseconds and produced at most hundreds of interrupts per second, which meant interrupts worked well to enable software-level concurrency while avoiding costly overheads. However, new storage devices are built with solid-state memory which can sustain not only millions of requests per second [25, 76], but also multiple concurrent requests. Figure 1 shows how storage devices have become both exponentially higher throughput and lower latency in the past decade. Higher throughput is supported through the NVMe specification [57], which exposes underlying device parallelism to software by providing multiple queues, up to 64K per device, where requests, up to 64K per queue, can be submitted and completed; Linux developers rewrote its block subsystem to match this multi-queue paradigm [9]. Figure 2 shows a high-level overview of NVMe request submission and completion.

On the other hand, lower latency is supported through a switch in device media from NAND chips to technologies such as Intel Optane, whose Gen 1 devices can achieve latencies of 10 μs [23] and Gen 2 devices can achieve latencies of 5 μs [24]. Numerous kernel, application, and firmware-level improvements have been proposed in the literature to unlock the higher request rate of these devices [13, 39, 46, 48, 60, 66, 83, 84], but they focus on increasing I/O *submit* rate without directly addressing the problem of higher *completion* rate. These devices are capable of higher interrupt rates that both occupy valuable CPU time and add noticeable latency to device latencies that are approaching the order of microseconds.

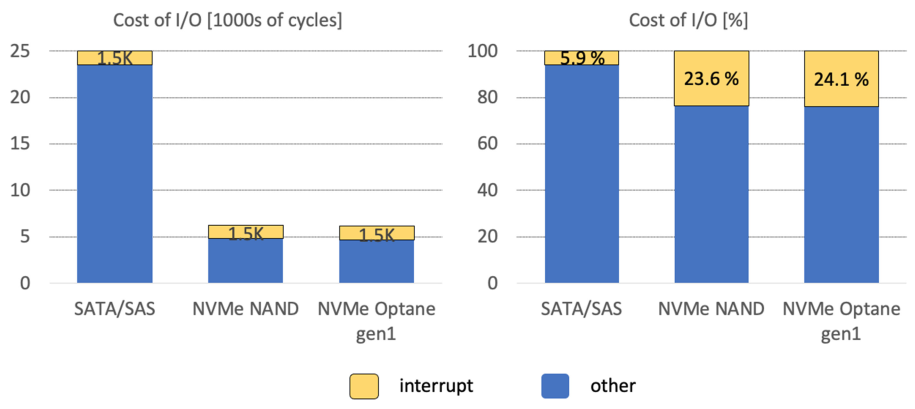

Fig. 3. As devices support higher throughput, the cost of an interrupt at the CPU, which is constant, becomes a higher percentage of total I/O cost. In this experiment, a process submits reads at I/O depth 256 to varying devices. The graphs compare the absolute and relative cost of an interrupt to the total I/O cost.

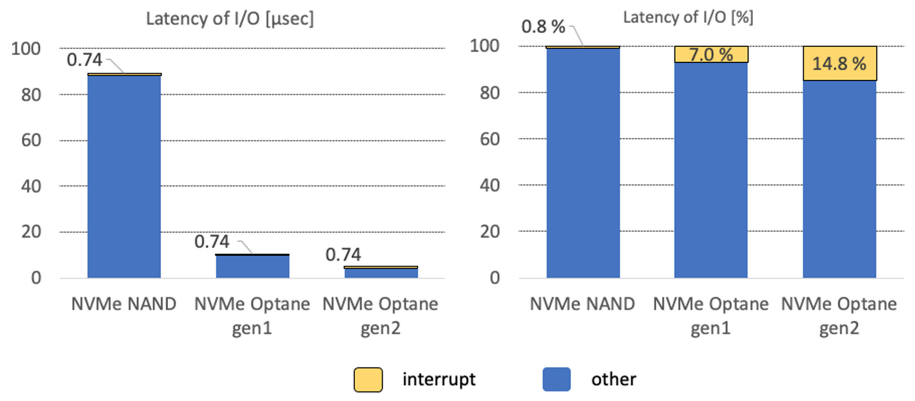

Fig. 4. As devices support lower latency (from 85 μs MLC NAND to 5 μ Intel Optane Gen 2), the latency of handling an interrupt at the CPU becomes a higher percentage of total I/O latency. In this experiment, a process submits a single read request at a time to varying devices. The graphs compare the absolute and relative latency of an interrupt to the total latency of the I/O.

High Cost of Interrupts. Figures 3 and 4 show how interrupt completion overheads become a significant portion of total IO time in new storage devices. Figure 3 shows results from running a CPU-bound (throughput-sensitive) workload: a single process submits 4K block reads to the device via libaio at an I/O depth of 256. Our experimental setup is described in Section 5.1. The left graph of Figure 3 shows the total cost of an I/O request in CPU cycles, while the right graph shows the relative cost of handling an interrupt with respect to total I/O cost.

Note that from the SATA device [65] to the NVMe devices [21, 23], the CPU can send many requests in parallel, which makes the total cost of an I/O lower. Across all storage devices, the cost of handling an interrupt remains constant, about 1,500 cycles, which accounts for the context switch into interrupt handling code, running the interrupt handler, and context-switching back to kernel code. Therefore, as shown in the right graph of Figure 3, as devices can handle higher throughput, the *relative* cost of an interrupt increases, and is up to 24% in NVMe Optane devices.

Figure 4 shows results from running a device-bound (latency-sensitive) workload: a single process submits a single 4K block read at a time. The left graph shows the total latency of the read, and the right graph shows the percentage of this latency that is attributed to interrupt handling. As with the previous experiment, the cost of an interrupt, around 740 ns, stays the same for varying storage devices. However, the latency of the request at the device falls considerably in the switch from NAND media [21] to Optane [23, 25]. Note that we do not change the interconnect (NVMe) in this experiment to show that the latency difference comes solely from the underlying media of the device. When the total request is 5 μs, as in Intel Optane Gen 2, the overhead of interrupt handling is almost 15% of the total IO latency.

These two experiments show that, with current interconnect and device media trends, the cost of a simple interrupt is now nontrivial.

Lessons from Networking. Networking devices have had much higher completion rates for a long time. For example, 100 Gbps networking cards can process over 100 million packets per second in each direction, over 200× that of a typical NVMe device. The networking community has devised two main strategies to deal with these completion rates: interrupt coalescing and polling.

To avoid bombarding the processor with interrupts, network devices apply interrupt coalescing [75], which waits until a threshold of packets is available or a timeout is triggered. Network stacks may also employ polling [16], where software queries for packets to process rather than being notified. IX [7] and DPDK [33] (as well as SPDK [68]) expose the device directly to the application, bypassing the kernel and the need for interrupts by implementing polling in userspace. Technologies such as Intel's DDIO [19] or ARM's ACP [56] enable networking devices to write incoming data directly into processor caches, making polling even faster by turning MMIO queries into cache hits. The networking community has also proposed various in-network switching and scheduling techniques to balance low-latency and high-throughput [3, 7, 35, 49].

Storage Is Adopting Networking Techniques. The NVMe specification standardizes the idea of interrupt coalescing for storage devices [57], where an interrupt will fire only if there is a sufficient threshold of items in the completion queue or after a timeout. There are two key problems with NVMe interrupt coalescing. First, NVMe only allows the aggregation time to be set in 100μs increments [57], while devices are approaching sub-10μs latencies. For example, in our setup, small requests that normally take 10 μs are delayed by 100μs, resulting in a 10× latency increase. Intel Optane Gen 2 devices have latencies around 5 μs [24], which would result in a 20× latency increase. The risk of such high latency amplification renders the timeout unusable in general-purpose deployments where the workload is unknown.

Second, even if the NVMe aggregation granularity were more reasonable, both the threshold and timeout are *statically* configured (the current NVMe standard and its implementations have no adaptive coalescing). This means interrupt coalescing easily breaks after small changes in workload—for example, if the workload temporarily cannot meet the threshold value. The NVMe standard even specifies that interrupt coalescing be turned off by default [58], and off-the-shelf NVMe devices ship with coalescing disabled.

Indeed, despite the existence of hardware-level coalescing, there are still continuous software and driver patches to deal with interrupt storms through mechanisms such as fine-tuning of threaded interrupt completions and polling completions [11, 47, 50]. Mailing list requests and product documentation show that Azure observes large latency increases when using aggressive interrupt coalescing to deal with interrupt storms, which they try to address with driver mitigations [26, 47]. Because of the proprietary nature of Azure's solution, it is unclear whether their interrupt coalescing is standard NVMe coalescing or some custom coalescing that they develop with hardware vendors.

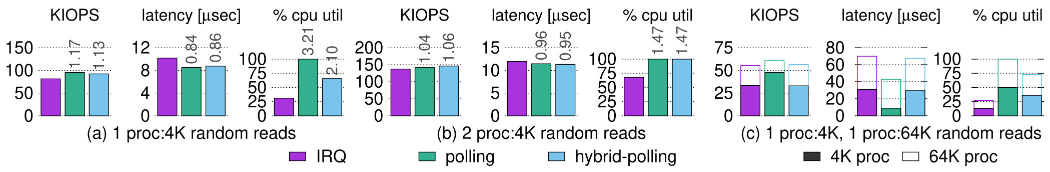

Fig. 5. Hybrid polling is not enough to mitigate polling overheads. All experiments run on a single core. (a) With a single thread of same-sized requests, hybrid polling effectively reduces the CPU utilization of polling while matching the performance of polling. (b) With more threads, hybrid polling reduces to polling in terms of CPU utilization without providing significant performance improvement over interrupts. (c) With variable I/O sizes, hybrid polling has the same throughput and latency as interrupts for both I/O sizes while using 2.7x more CPU. Labels show performance relative to IRQ.

Polling Is Expensive. μdepot [43] and Arrakis [61] deal with higher completion rates by resorting to polling. Directly polling from userspace via SPDK requires considerable changes to the application, and polling from either kernel-space or userspace is known to waste CPU cycles [42, 78, 83]. FlashShare only polls for what it categorizes as low-latency applications [83], but acknowledges that this is still expensive. Cinterrupts exposes application semantics for interrupt generation so that systems do not have to resort to polling.

Even hybrid polling [80], which is a heuristic-based technique for reducing the CPU overhead of polling by sleeping for a period of time before starting to poll, is insufficient, breaking down when requests having varying size [5, 41, 45].

Figure 5 compares the performance and CPU utilization of hybrid polling, polling, and interrupts for three benchmarks on an Intel Optane DC P4800X [23].[2] We note that in all cases, polling provides the lowest latency because the polling thread discovers completions immediately, at the expense of 100% CPU utilization. When there is a single thread submitting requests through the read syscall (Figure 5(a)), hybrid polling does well because request completions are uniform. However, when more threads or I/O sizes are added, as in Figures 5(b)–(c), hybrid polling still has 1.5×–2.7× higher CPU utilization than interrupts without providing noticeable performance improvement. These results match other findings in the literature [5, 6, 41, 45].

Even though polling wastes cycles, it can provide lower latency than interrupts, which is desirable in some cases. As such, cinterrupts coexists with kernel-side polling, such as in Linux NAPI for networking [15, 16], which switches between polling and interrupts based on demand.

Heuristics-Based Completion. vIC [2] tries to moderate virtual interrupts by estimating I/O completion time with heuristics. It primarily relies on inspecting the number of "commands-in-flight" to determine whether to coalesce interrupts, also employing smoothing mechanisms to ensure that the coalescing rate does not change too dramatically. To prevent latency increase in low-loaded scenarios, vIC also eliminates interrupt coalescing when the submission rate is below a certain threshold. vIC is a heuristic-based coalescing algorithm, similar to our adaptive algorithm (Section 3.2). Consequently, vIC also lacks semantic information necessary to align interrupt delivery.

NICs and their software stack are higher-performing and more mature than NVMe drives and their corresponding stack. As with NVMe devices, NICs must balance two contradictory goals: (1) reducing interrupt overhead via coalescing to help throughput-oriented workloads, while (2) providing low latency for latency-sensitive workloads by triggering interrupts upon incoming packets as soon as possible. Notably, NICs employ more sophisticated, adaptive interrupt coalescing schemes (implemented inside the device and helped by its driver). Yet, in general-purpose

[2]See Section 5.1 for a detailed description of our experimental setup.

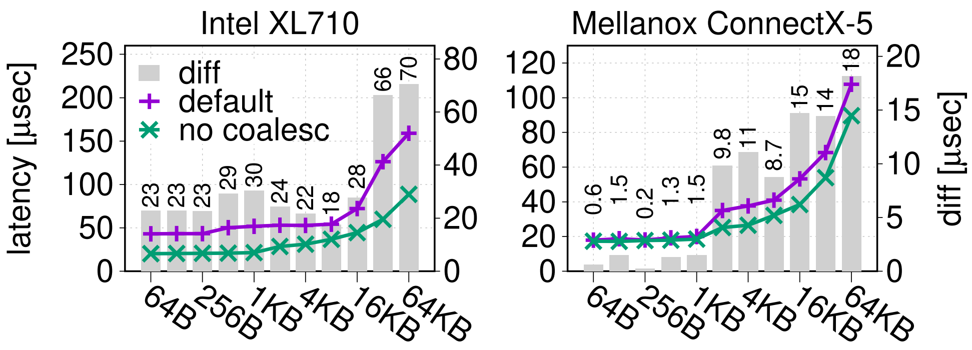

Fig. 6. NICs employ adaptive heuristics that try to minimize interrupt overheads without unnecessarily hurting latency. The inherent imperfection of these heuristics is demonstrated using Intel and Mellanox NICs servicing the netperf request-response benchmark, which ping-pongs a message of a specified size. The labels show the latency difference between the default NIC scheme and a no-coalescing policy, which minimizes latency for this particular workload, but which harms performance for more throughput-oriented workloads.

settings that must accommodate arbitrary workloads, NICs are unable to optimally fire and coalesce interrupts, despite their maturity.

Figure 6 demonstrates that heuristics in even mature devices cannot optimally resolve the interrupt delivery problem for all workloads. Two NICs, Intel XL710 40 GbE [20] and Mellanox ConnectX-5 100 GbE [72], run the standard latency-sensitive netperf TCP request-response (RR) benchmark [34], which repeatedly sends a message of a specified size to its peer and waits for an identical response. In this workload, the challenge for the NIC is identifying the end of the incoming message and issuing the corresponding interrupt. The Intel NIC heuristic results in increased latency regardless of message size, and the Mellanox NIC heuristic adds latency if the message size is greater than 1,500 bytes, the maximum transmission unit (MTU) for Ethernet, because the message becomes split across multiple packets.

Knowledgeable users with admin privileges may manually configure the NIC to avoid coalescing, which helps identify message boundaries and thus yields better results for this *specific* workload. But such a configuration is ill-suited for general-purpose setups that must reasonably support co-located throughput-oriented workloads as well.

Exposing Application-Level Semantics. Similar to [39, 79], and [83], cinterrupts augments the syscall interface with a few bits so applications can transmit information to the kernel. Cinterrupts further shares this information with the device, which is only also done in [83], which shares with the device SLO information used to improve internal device caching.

3 CINTERRUPTS

3.1 Design Overview

The initial design of cinterrupts focused on making NVMe coalescing adapt to workload changes. Our first contribution captures this intuition with an adaptive coalescing strategy to replace the static NVMe algorithm (Section 3.2).

While our adaptive coalescing strategy improves over static coalescing, there are still cases that an adaptive strategy cannot handle, such as workloads with a mix of latency-sensitive and throughput-sensitive requests. The adaptive strategy also imposes an inevitable overhead from detecting when a workload has changed.

This observation led to the core insight of cinterrupts: without prior knowledge about application behavior, device-level heuristics for coalescing will always fall short due to a semantic gap

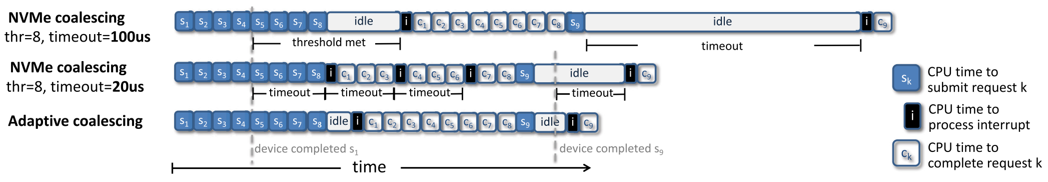

Fig. 7. NVMe coalescing with its current 100-μs granularity (top row) causes unusable timeouts when the threshold is not met (c_9). Even if the timeout granularity were smaller (middle row), NVMe coalescing cannot adapt to the workload. For bursts of requests, the smaller timeout will limit the throughput with interrupts ($c_1 - c_8$), while bursts that do not meet the threshold must still wait for the timeout (c_9). Note that idle periods occur when the CPU is done submitting requests, but is waiting for the device to complete I/O.

between the requester and the device, which sees a stream of requests and cannot determine which requests require interrupts in order to unblock the application. To bridge the semantic gap, the application issuing the I/O request can inform the device when it wishes to be interrupted. Cinterrupts takes advantage of the fact that this semantic information is easily accessible in the storage stack and available at submission time.

Note on Methodology. The results throughout this section are obtained on a setup fully described in Section 5.1. We use an Intel Optane DC P4800X, 375 GB [23], installed in a Dell PowerEdge R730 machine equipped with two 14-core 2.0-GHz Intel Xeon E5-2660 v4 CPUs and 128 GB of memory running Ubuntu 16.04. The server runs cinterrupts' modified version of Linux 5.0.8 and has C-states, Turbo Boost (dynamic clock rate control), and SMT disabled. We use the maximum performance governor.

All results are obtained with our cinterrupts emulation, as described in Section 4.2.1. Our emulation pairs one dedicated core to one target core. Each target core is assigned its own NVMe submission and completion queue. All results in this section are run on a single target core.

3.2 Adaptive Coalescing

Ideally, an interrupt coalescing scheme adapts dynamically to the workload. Figure 7 shows that even if the timeout granularity in the NVMe specification were smaller, it is still *fixed*, which means that interrupts will be generated when the workload does not need interrupts ($c_1 - c_8$), while completions must wait for the timeout to expire (c_9) when the workload does need interrupts.

Instead, as shown in the bottom row of Figure 7, the adaptive coalescing strategy in cinterrupts observes that a device should generate a single interrupt for a burst, or a sequence of requests whose interarrival time is within some bound.

Algorithm 1 shows the adaptive strategy. The burst detection happens on Line 6, where the timeout is pushed out by Δ every time a new completion arrives. In contrast, NVMe coalescing cannot detect bursts because it does not dynamically update the timeout, which means it can only detect bursts of a fixed size. To bound request latency, the adaptive strategy uses a thr that is the maximum number of requests it will coalesce into a single interrupt (Lines 14–15). This is necessary for long-lived bursts to prevent infinite delay of request completion. With Algorithm 1, a device will emit interrupts when either it has observed a completion quiescent interval of Δ or thr requests have completed. In Section 5, we explain how device manufacturers and system administrators can determine reasonable Δ and thr configurations.

Comparison to NVMe Coalescing. The adaptive strategy outperforms various NVMe coalescing configurations, even those with smaller timeouts, across different workloads. We compare the adaptive strategy, configured with $thr = 32$, $\Delta = 6$, to no coalescing (default), nvme100, which uses a timeout of 100 μs, the smallest possible in standard NVMe, nvme20, which uses a theoretical

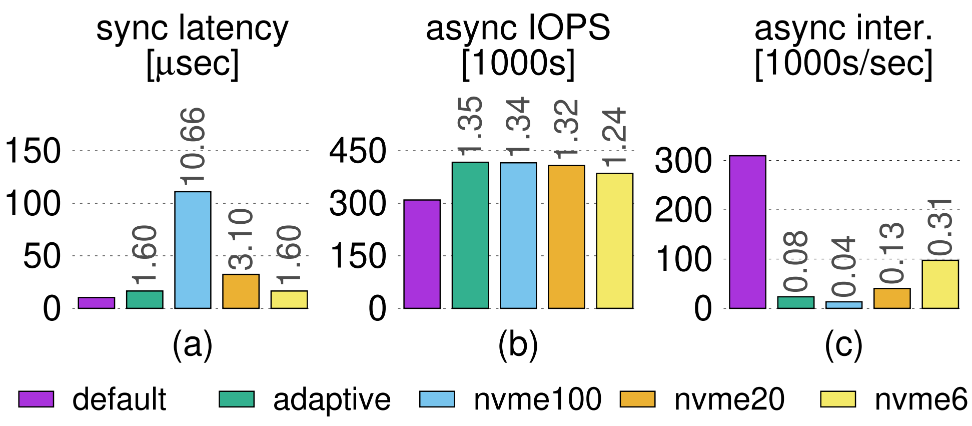

Fig. 8. Adaptive strategy has better performance for both types of workloads regardless of how NVMe coalescing is configured. (a) Latency of a synchronous read request. (b) Throughput of an asynchronous read workload with high iodepth. (c) Interrupt rate for async workload. Labels show performance relative to default.

ALGORITHM 1: Adaptive coalescing strategy in cinterrupts

```
Parameters: $\Delta$, $thr$
coalesced = 0, timeout = now + $\Delta$;
while true do
    while now < timeout do
        while new completion arrival do
            /* burst detection,update timeout */
            timeout = now + $\Delta$;
            if ++coalesced $\geq$ $thr$ then
                fire IRQ and reset;
    /* end of quiescent period */
    if coalesced > 0 then
        fire IRQ and reset;
    timeout = now + $\Delta$;
```

timeout of 20 μs, and nvme6, which uses a theoretical timeout of 6 μs. All NVMe coalescing configurations have threshold set to 32.

We run two single-threaded synthetic workloads with fio [4]. In the first workload, the thread submits 4-KB read requests via read, which blocks until the system call is done. In the second workload, the thread submits 4-KB read requests via libaio in batches of 16, with iodepth=512.[3]

Figure 8(a) reports the latency of the read requests for the synchronous workload. As expected, the default strategy has the lowest latency of 10μs, because it generates an interrupt for every request. All coalescing strategies add exactly their timeout to the latency of each read request (c_9 in Figure 7). Because they have the same timeout, nvme6 and adaptive have the same latency, but nvme6 pays the price for this low timeout in the next workload.

Figure 8(b) reports the read IOPS for the second workload and shows that if there are enough requests to hit the threshold, the timeout adds unnecessary interrupts ($c_1 - c_8$ in Figure 7). The

[3]iodepth represents the number of in-flight requests.

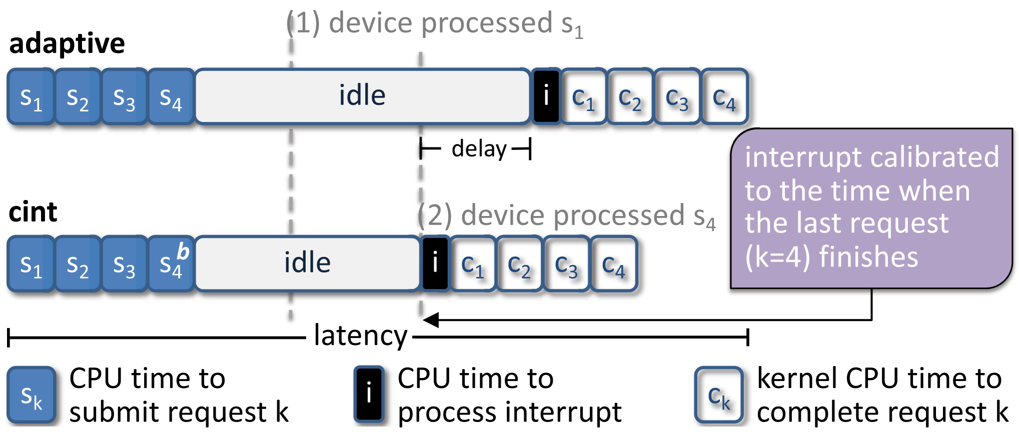
adaptive
(1) device processed s1
s1 s2 s3 s4 idle i c1 c2 c3 c4
delay
cint
(2) device processed s4
s1 s2 s3 s4 b idle i c1 c2 c3 c4
interrupt calibrated to the time when the last request (k=4) finishes
latency
sk CPU time to submit request k
i CPU time to process interrupt
ck kernel CPU time to complete request k

Fig. 9. Completion timeline for multiple submissions. The adaptive strategy can detect them as part of a burst, but only after the delay expires. Cinterrupts explicitly marks the last request in the batch. Idle periods occur when the CPU waits for I/O to complete.

default strategy's throughput is limited because it generates too many interrupts, as shown in Figure 8(c).

The workload has enough requests in flight that waiting for the threshold of 32 completions does not harm throughput. However, nvme20 and nvme6 *must* fire an interrupt every 20 μs or 6μs, respectively: Figure 8(c) shows that nvme20 generates 1.7× more interrupts than adaptive, and nvme6 generates 4.2× more interrupts than adaptive, explaining their lower throughput.

The adaptive strategy can accurately detect bursts, although it adds Δ delay to confirm the end of a burst; without additional information, this delay is unavoidable. Figure 9 shows how cinterrupts addresses this problem by enhancing adaptive with two annotations, Urgent and Barrier, which software passes to the device. We now describe both annotations.

3.3 Urgent

Urgent is used to request an interrupt for a single request: the device will generate an immediate interrupt for any request annotated with Urgent. The primary use for Urgent is to enable the device to calibrate interrupts for latency-sensitive requests. Urgent eliminates the delay in the adaptive strategy.

To demonstrate the effectiveness of Urgent, we run a synthetic mixed workload with fio with two threads: one submitting 4-KB read requests via libaio with iodepth =16 and one submitting 4-KB read requests via read, which blocks until the system call is done. In cinterrupts, the latency-sensitive read requests are annotated with Urgent, which is embedded in the NVMe request that is sent to the device (see Section 4.1.1). Results are shown in Figure 10.

Without cinterrupts, the requests from either thread are *indistinguishable* to the device. The default (no coalescing) strategy addresses this problem by generating an interrupt for every request, resulting in 2.7× more interrupts than cinterrupts (Figure 10(d)).

On the other hand, with Urgent, cinterrupts calibrates interrupts to the latency-sensitive read requests, enabling low-latency without generating needless interrupts that hamper the throughput of the asynchronous thread. This results in both higher asynchronous throughput and lower latency for the synchronous requests. The adaptive strategy is unable to identify the latency-sensitive requests and in fact tries to minimize interrupts for all requests, resulting in higher asynchronous throughput but a corresponding increase in read request latency (Figure 10(c)).

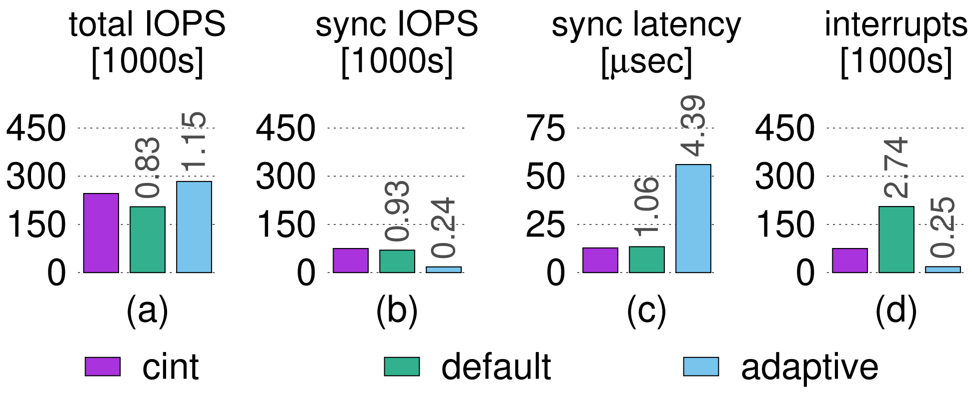

Fig. 10. Effect of Urgent. Synthetic workload with two threads running a mixed workload: one thread submitting synchronous requests via read, one thread submitting asynchronous requests via libaio. Cinterrupts achieves optimal synchronous latency and better throughput over the default (no coalescing). The adaptive strategy achieves better overall throughput, at the expense of synchronous latency. Labels show performance relative to cinterrupts. Note that cinterrupts achieves lower total throughput than the adaptive strategy, which we explain in Figure 21.

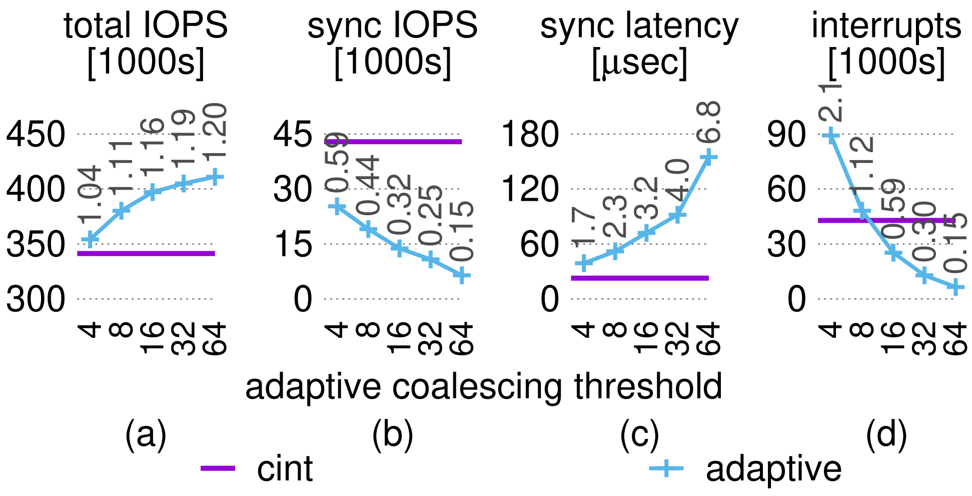

Fig. 11. In a mixed workload, increasing the coalescing threshold increases the latency of synchronous requests proportionally to the coalescing rate. Labels show performance relative to cinterrupts.

In fact, the more aggressive the coalescing, the more unusable synchronous latencies become. Figure 11 shows the same experiment with higher iodepth (iodepth = 256), which drives enough load to the device for coalescing to be beneficial. As the target coalescing rate increases, there is a corresponding linear increase in the synchronous latency. On the other hand, the purple line in Figure 11(c) shows that Urgent in cinterrupts makes synchronous latency acceptable. This latency comes at the expense of less asynchronous throughput, as shown in Figure 11(a), but we believe this is an acceptable tradeoff.

3.4 Barrier

To calibrate interrupts for batches of requests, cinterrupts uses Barrier, which marks the end of a batch and instructs the device to generate an interrupt as soon as all preceding requests have

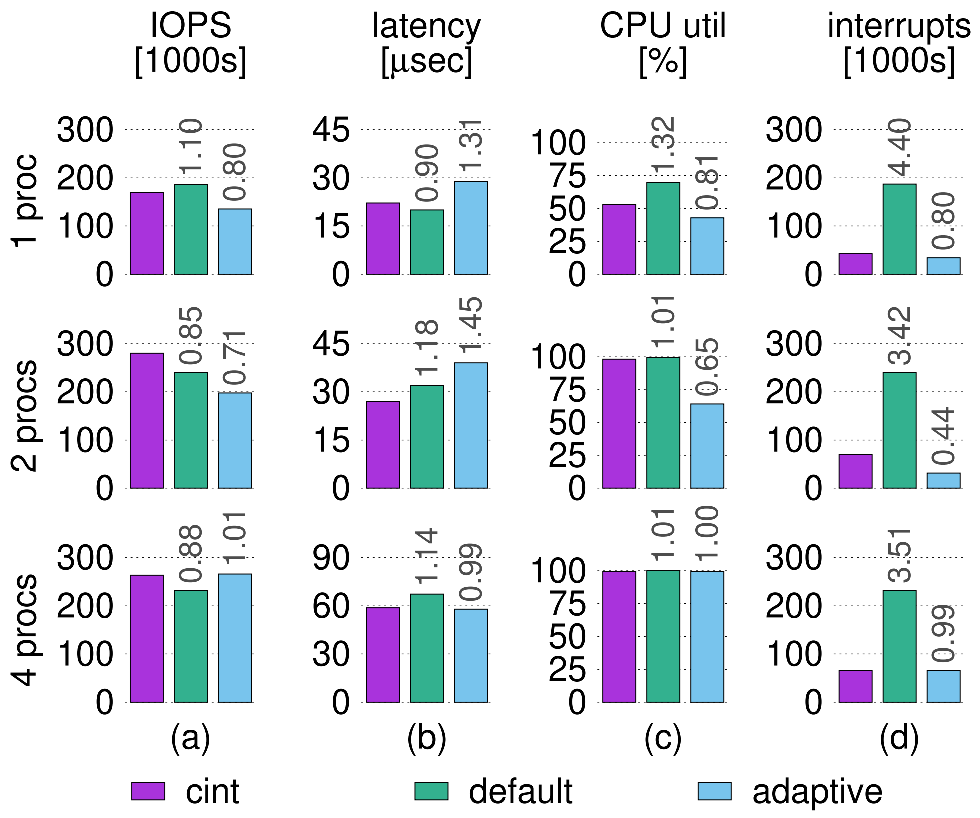

Fig. 12. Effect of Barrier. Each process submits a batch of 4 requests at a time, submitting a new batch after the previous batch has finished. Cinterrupts always detects the end of a batch with Barrier. Note that when there is CPU idleness, adaptive always adds Δ delay to the latency. Labels show performance relative to cinterrupts.

finished. The semantic difference between Urgent and Barrier is that an Urgent interrupt is generated as soon as the Urgent request finishes, whereas the Barrier interrupt may have to wait if requests are completed out of order.

Barrier minimizes the interrupt rate, which is always beneficial for CPU utilization, while enabling the device to generate enough interrupts so that the application is not blocked. For example, in the submission stream $s_1 - s_4$ in Figure 9, the last request in the batch, s_4, is marked with Barrier.

To demonstrate the effectiveness of Barrier, we run an experiment with a variable number of threads on the same core, where each thread is doing 4-KB random reads through libaio, submitting in fixed batch sizes of 4. The trick is determining the end of the batch without additional overhead, which is only possible in cinterrupts: we modify fio to mark the last request in each batch with a Barrier. Figure 12 shows the throughput, latency, CPU utilization, and interrupt rate.

Single Thread. When there is a single thread, the default (no coalescing) strategy can deliver lower latency than cinterrupts. This is because there is CPU idleness and no other thread running. However, the default strategy generates 4.4× the number of interrupts as cinterrupts, which results in 1.32× CPU utilization. The default strategy can also process some completions in parallel with device processing, whereas cinterrupts waits for all completions in the batch to arrive before processing. On the other hand, the Δ delay in the adaptive algorithm is clear: the latency of requests is 29 μs, compared to 22 μs with cinterrupts.

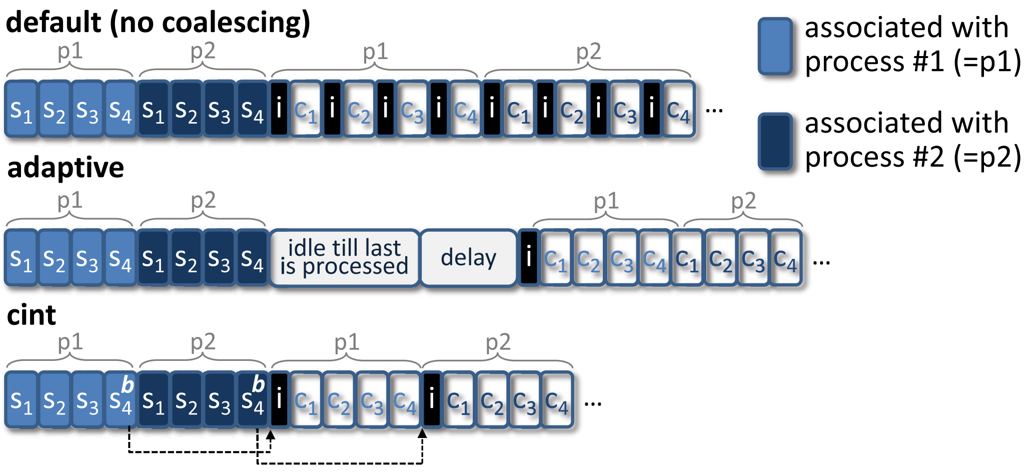
default (no coalescing)
adaptive
cint
p1
p2
idle till last is processed
delay
b
associated with process #1 (=p1)
associated with process #2 (=p2)

Fig. 13. Completion timeline for two threads submitting request batches. The adaptive strategy experiences CPU idleness both because of the delay and because it waits to process any completions until they all arrive. On the other hand, due to Barrier, cinterrupts can process each batch as soon as it completes.

Two Threads. When there are two threads in the experiment, the advantage of the default strategy goes away: the 3.4× interrupts taxes a saturated CPU. On the other hand, cinterrupts has the best throughput and latency because calibrating interrupts enable better CPU usage.

Figure 13 shows that the adaptive strategy exhibits highest synchronous latency due to CPU idleness, which comes from waiting for completions and the delay used to detect the end of the batch. This idleness is eliminated in the next experiment, where there are enough threads keep the CPU busy.

Four Threads. With four threads, the comparison between cinterrupts and the default NVMe strategy remains the same. However, at four threads, the adaptive strategy matches the performance of cinterrupts because without CPU idleness, the delay is less of a factor. Although the adaptive strategy does well in this last case, we showed in Section 3.3 that this aggregation comes at the expense of synchronous requests.

Note that Figure 13 is a simplification of a real execution, because it conflates time spent in userspace and the kernel, and does not show completion reordering. The full cinterrupts algorithm addresses reordering by employing the adaptive strategy to ensure no requests get stuck.

3.5 Out-of-Order Urgent

The full cinterrupts interrupt generation strategy is shown in Algorithm 2. Requests are either unmarked or marked by Urgent or Barrier. Unmarked requests are handled by the underlying adaptive algorithm and can of course piggyback on interrupts generated by Urgent or Barrier.

We noticed that Urgent requests sometimes get completed with other requests, which increases their latency because the interrupt handler does not return until it reaps all requests in the interrupt context. To address this, cinterrupts implements out-of-order (OOO) processing, a driver-level optimization for Urgent requests. With OOO processing, the IRQ handler will only reap Urgent requests in the interrupt context, which enables faster return-to-userspace of the Urgent requests.

Unmarked requests will not be reaped until a completion batch consists only of those requests, as shown in Figure 14. The driver also does not ring the CQ doorbell until it completes a contiguous range of entries. *thr* ensures non-Urgent requests are eventually reaped, even if there is a continuous stream of Urgent requests. For example, suppose in Figure 14 that *thr* = 9. Then an interrupt

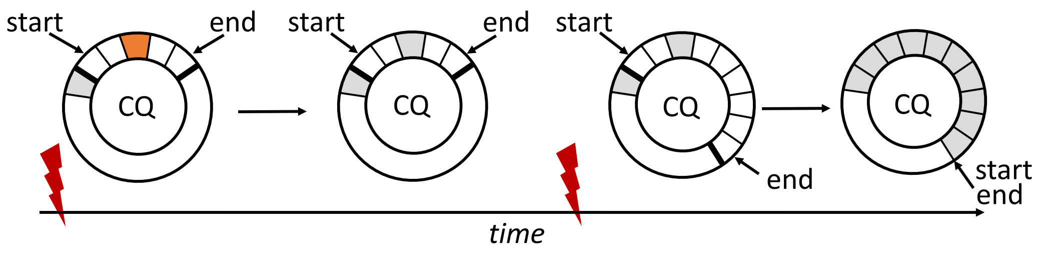

Fig. 14. OOO Urgent processing. Grayed entries are reaped entries. Urgent requests in an interrupt context (first interrupt) are processed immediately, and the interrupt handler returns. The other requests are not reaped until the next interrupt, which consists only of non-Urgent requests. After the second IRQ, the driver rings the completion queue doorbell to signal that the device can reclaim the contiguous range.

ALGORITHM 2: cinterrupts coalescing strategy

```
1  Parameters: $\Delta$, $thr$
2  coalesced = 0, timeout = now + $\Delta$;
3  while true do
4      while now < timeout do
5          while new completion arrival do
6              timeout = now + $\Delta$;
7              if completion type == Urgent then
8                  if ooo processing is enabled then
                       /* only urgent requests */
9                      fire urgent IRQ;
10                 else
                       /* process all requests */
11                     fire IRQ and reset coalesced;
12             if completion type == Barrier then
13                 fire IRQ and reset coalesced;
14             else
15                 if ++coalesced ≥ $thr$ then
16                     fire IRQ and reset coalesced;
       /* end of quiescent period */
17     if coalesced > 0 then
18         fire IRQ and reset coalesced;
19     timeout = now + $\Delta$;
```

for the non-OOO requests will fire as soon as 9 entries (already reaped or otherwise) accumulate in the completion queue.

The tradeoff with OOO processing is an increase in the number of interrupts generated and an increase in latency for non-OOO requests. Figure 15 reports performance metrics from running the same mixed workload as in Figure 11. OOO processing generates 2.4× the number of interrupts in order to reduce the latency of synchronous requests by almost half. The impact of the additional interrupts is noticeable in the reduced number of asynchronous IOPS.

Incidentally, these additional interrupts, as well as the interrupts in the default strategy, act as an inadvertent tax on the asynchronous thread. If we instead limit the number of asynchronous requests, the need for these additional interrupts goes away. In the second row of Figure 15, we

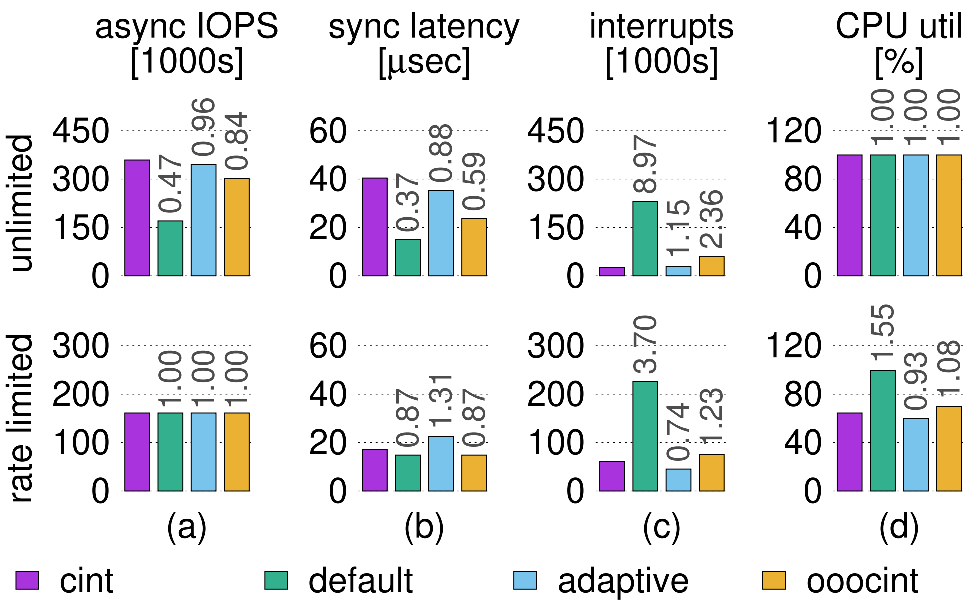

Fig. 15. Mixed workload. Out-of-order (OOO) driver processing of Urgent requests enables lower latency at the expense of more interrupts. Limiting the number of async requests (bottom row) reduces this overhead. Labels above bars indicate performance ratio compared to cinterrupts.

throttle the asynchronous thread with the blkio cgroup [10] to its throughput in the default scenario (green bar in the first row). In this case, OOO cinterrupts only generates 23% more interrupts, and its synchronous latency matches that of the default strategy while using 30% less CPU.

OOO processing is turned on by default in the cinterrupts NVMe driver but can be disabled with a module parameter.

4 IMPLEMENTATION

4.1 Software Modifications

4.1.1 Kernel Modifications. It is software's responsibility to pass request annotations to the device. To minimize programmer burden, our implementation includes a modified kernel that sets default annotations. Table 1 summarizes how the kernel sets these defaults, which naturally derive from the programming paradigm of each system call: any system call that blocks the application, such as read and write, is marked Urgent, and any system call that supports asynchronous submission, such as io_submit, is marked Barrier. System calls in the sync family are blocking, so they are marked Urgent. By following the programming paradigm of each system call, cinterrupts can be supported in the kernel with limited intrusion; the changes to the Linux software stack described in this section total around 100 LOC.

The cinterrupts kernel propagates annotations from the system call layer through the filesystem and block layers to the device. In the system call handling layer, cinterrupts embeds any bits in the iocb struct. The block layer can split or merge requests. In the case of request split—for example, a 1M write will get split into several smaller write blocks—each child request will retain the bit of the parent. In the case of merged requests, the merged request will retain a superset of the bits in its children. If this superset contains both Urgent and Barrier, we mark the merged request as Urgent for simplicity. This is not a correctness issue because the underlying adaptive algorithm will ensure that no request gets stuck.

Table 1. Summary of Storage I/O System Calls and the Corresponding Default Bits Used by the Kernel

System call	Kernel default annotations
(p)read(v), (p)write(v)	Urgent if fd is blocking or if write is `O_DIRECT`
preadv2, pwritev2	If `RWF_NOWAIT` is not set, use Urgent
io_submit	Barrier on the last request
(f)(data)sync, syncfs	Urgent
msync	With `MS_SYNC`, Barrier on the last request

For cases in which these defaults do not match application-level semantics, we expose a system call interface for applications to override these defaults. We leverage the preadv2/pwritev2 system call interface [63], which already exposes a parameter that accepts flags:

```
ssize_t preadv2(int fd, const struct iovec *iov,
                int iovcnt, off_t offset, int flags)
```

We create two new flag types, RWF_URGENT and RWF_BARRIER, which the application can use to pass bits as it sees fit. The application can explicitly ask for a request to be unmarked by passing both flags. We explain how applications can use this interface in the next section.

4.1.2 Application Case Studies. Ultimately the application has the best knowledge of when it requires interrupts, so cinterrupts enables the application to override kernel defaults for even better performance, using the syscall interface described previously. We modified RocksDB to use these flags.

RocksDB Background Tasks. Flushing and compaction are the two main sources of background I/O in RocksDB. We modify RocksDB to explicitly mark these I/O requests as non-Urgent. Since RocksDB already isolates the environment for interacting with files, our changes were minimal and involved replacing the calls to pread/pwrite with preadv2/pwritev2, creating a new file option to express urgency of I/O for that file, and modifying the Flush and Compaction jobs to explicitly mark I/O as non-Urgent, which totaled around 40 lines of code. We show in Section 5.4.1 that these manual annotations especially help RocksDB during write-intensive workloads.

RocksDB Dump. RocksDB includes a dump tool that dumps all the keys in a RocksDB instance to a file [1]. Typically, this tool is used to debug or migrate a RocksDB instance. As it is a maintenance-level tool, dump requests do not need to be Urgent, so we manually modify the dump code to mark dump read and write I/O as non-Urgent. In this way, dump I/O requests are completed when interrupts are generated by the underlying adaptive coalescing strategy. On top of the RocksDB changes described in the previous paragraph, marking dump requests as non-Urgent only required two lines of code. We show in Section 5.5.1 that modifying the dump tool can increase the throughput of foreground requests by 37%.

4.2 Hardware Modifications

Cinterrupts modifies the hardware-software boundary to support Urgent and Barrier. The key hardware component in cinterrupts is an NVMe device that recognizes these bits and implements Algorithm 2 as its interrupt generation strategy. Device firmware, which is responsible for interrupt generation, is the only device component that must be modified in order to support cinterrupts. Device firmware is typically a blackbox, so we chose to emulate the interrupt generation portion of cinterrupts while leveraging real NVMe hardware for I/O execution.

4.2.1 Firmware Emulation. To emulate interrupt generation in cinterrupts, we explored using several existing aspects of the NVMe specification, all of which were insufficient. We considered

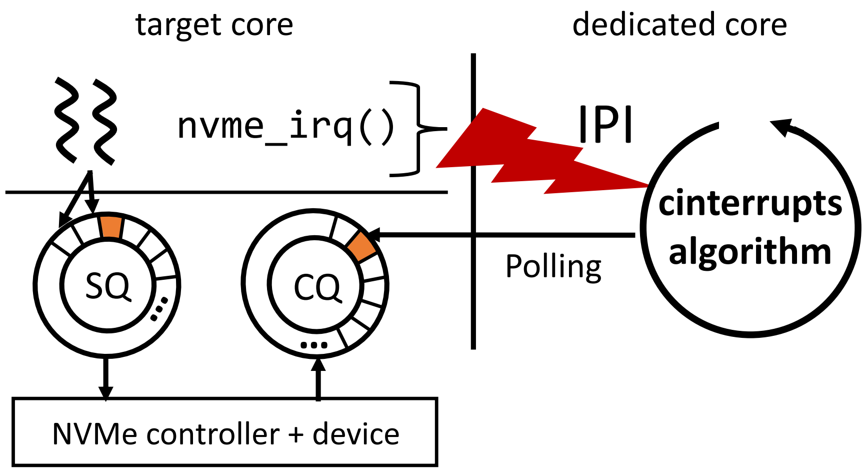

Fig. 16. Cinterrupts emulation: The dedicated core polls on the NVMe completion queue of the target core and sends IPIs for any completion. IPIs emulate hardware interrupts of a real device that supports cinterrupts. The target core submits requests normally to hardware by writing to the NVMe submission queue.

using the urgent priority queues to implement Urgent. While this would have worked for Urgent, there is still no way to implement Barrier or Algorithm 2. Furthermore, in NVMe devices, while it is possible to have a dedicated urgent priority queue, hardware queues are still limited in NVMe devices; Azure NVMe devices have eight queues that must be shared across many cores [26], while Intel devices have 32-128 queues [17, 71]. By labeling individual requests, cinterrupts is explicitly designed to work in a general context where queues cannot be differentiated due to resource limitations.

We also considered using special bogus commands to force the NVMe device to generate an interrupt. The specification recommends that "commands that complete in error are not coalesced" [57]. Unfortunately, neither device we inspected [22, 76] respected this aspect of the specification.

Instead, we prototype cinterrupts by emulating interrupt generation with a dedicated sidecore that uses interprocessor interrupts (IPIs) to emulate hardware interrupts. We implement this emulation on Linux 5.0.8.

Dedicated Core. Our emulation assigns a dedicated core to a target core. The target core functions normally by running applications that submit requests to the core's NVMe submission queue, which are passed normally to the NVMe device. The dedicated core runs a pinned kernel thread, created in the NVMe device driver, that polls the completion queue of the target core and generates IPIs based on Algorithm 2. Cinterrupts annotations are embedded in a request's command ID, which the polling dedicated core inspects to determine which bits are set. When the target core handles the completion entry, it clears any cinterrupts bits in the command ID, returning it to the original blk_mq tag so that the block subsystem can complete the correct request. Finally, to support OOO Urgent, the driver allocates the third most significant bit as a "completion" flag, which is used to prevent already-reaped Urgent requests from getting completed in future interrupt contexts.

In a hardware-level implementation of cinterrupts, Urgent and Barrier can be communicated in any of the reserved bits in the submission queue entry of the NVMe specification [57].

To faithfully emulate the proposed hardware, we disable hardware interrupts for the NVMe queue assigned to that core; in this way, the target core only receives interrupts iff ideal cinterrupts hardware would fire an interrupt. Figure 16 shows how our dedicated core emulates the proposed interrupt generation scheme. Importantly, we still leverage real hardware to execute the I/O requests, and the driver still communicates with the NVMe device through the normal SQ/CQ pairs, but we replace the device's native interrupt generation mechanism with the dedicated core. Section 5.1 shows that this emulation has a modest 3–6% overhead.

4.3 Discussion

Other I/O Requests. We initially did not annotate requests generated by the kernel itself, for example from page cache writeback and filesystem journaling. But because filesystem journaling is on the critical path of write requests, non-Urgent journal transactions caused a slight increase in latency of application-level requests. Hence, by default, we mark journal commits as Urgent. Because journaling does not generate a large interrupt rate for our applications, marking these requests tightened application latency without adding overhead.

On the other hand, our applications did not see significant benefit in marking writeback requests. As such, we rely on the application to inform us when these requests are latency-sensitive, for example page cache flushes will be Urgent when they are explicitly requested through fsync.

Other Implementations. The Barrier implementation can be strict or relaxed, where a strict version only releases the interrupt if all requests in front of it in the submission queue have been completed. A relaxed Barrier is equivalent to Urgent and works well assuming that requests do not complete too far out of order; it does not require the interrupt generation algorithm to record any additional state. The cinterrupts prototype evaluated in this article uses a relaxed Barrier, which already enjoys significant performance benefits. We have retained a separate flag because Barrier is semantically different to the application and to enable future implementations to choose to implement strict Barrier.

The strict Barrier requires more accounting overhead to keep track of which requests have completed: we explored a preliminary implementation of the strict Barrier in our emulator but its overheads were larger than its benefit. We suspect firmware implementations of a strict Barrier will be more efficient. Alternatively, this strict ordering could be enforced in the kernel: the driver can withhold completions from userspace until all other requests have completed. Such an implementation might be efficient by piggybacking on the accounting mechanisms in the block layer of the Linux kernel.

Urgent Storm. If all requests in the system are marked as Urgent, this can inadvertently cause an interrupt storm. To address this, cinterrupts has a module parameter that can be configured to keep the interrupt rate below a fixed value, similar to NICs, enforced with a lightweight heuristic based on **Exponential Weighted Moving Average (EWMA)** of the interrupt rate.

Figure 17 shows an experiment where four processes are submitting 4-KB reads in a loop via pread. Because all the processes are using a blocking system call, all these requests will be annotated with Urgent by the kernel.

Without storm protection, the core receives 150K interrupts per second, which is noticeable in the saturated CPU, as shown in Figure 17(d). Cinterrupts storm protection is parameterized with two values: the maximum acceptable interrupt rate, and the weight of the EWMA algorithm (λ). The maximum acceptable interrupt rate is exposed to the user as a configurable module parameter. λ is not currently exposed as a module parameter, but this can easily be done.

The storm protection algorithm uses the EWMA of the interrupt rate to modify the threshold (thr) and Δ values of the cinterrupts adaptive algorithm to target the configured maximum interrupt rate. Generally, any interrupt rate less than 100K IRQ/s will not tax the CPU. Setting the maximum interrupt rate to a value greater than 100K IRQ/s will cause the storm protection algorithm to throttle interrupt generation if it is over this maximum, but the resulting interrupt rate could still perturb CPU processing. For example, in Figure 17, if the maximum interrupt rate is set

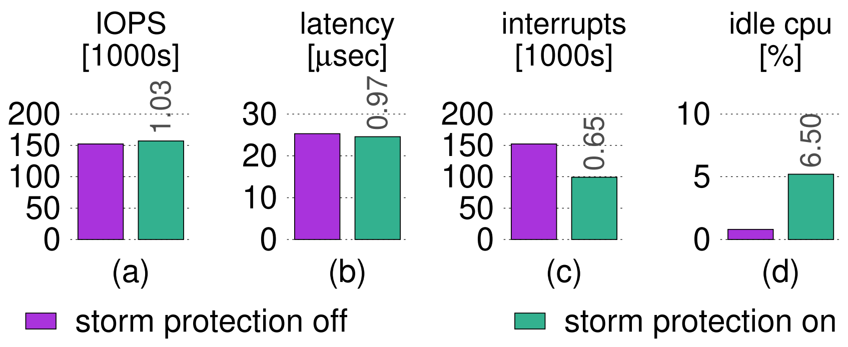

Fig. 17. Users can maliciously or accidentally abuse Urgent, for example if an application submits many synchronous I/O requests in parallel. To address this, we enable storm protection, which throttles interrupts to a user-configurable limit. In this experiment, four processes are all submitting synchronous reads in a loop. Without storm protection, the Urgent storm is noticeable in (c), which shows an interrupt rate of 150 K/sec. With storm protection, which we configure to throttle interrupts to 100 K/sec, cinterrupts adaptively sets the threshold and Δ parameters of the adaptive algorithm to keep interrupts below the limit. Doing so frees up CPU time (d) and marginally improves the throughput and latency of the workload. Importantly, if the Urgent intensity subsides, cinterrupts adapts automatically and turns off the throttling mechanism.

to 145K IRQ/s, then the storm protection algorithm will do little to change the CPU overhead of interrupts because the workload only generates 150K IRQ/s.

However, if the maximum interrupt rate is too small, this could result in negative consequences for the application. For example, a maximum interrupt rate of 10K IRQ/s means requests have completion latency of at least 100 μs. Throttling interrupt rate to this low value might be beneficial for batching workloads, but is detrimental to latency-sensitive workloads.

The weight (λ) of the EWMA algorithm represents how much to weigh recent statistics versus historical data. We currently set $\lambda = \frac{1}{128}$, which means each new interrupt rate measured contributes λ to the EWMA, while the previous value contributes $1 - \lambda$. We chose this value of lambda so that the IRQ generation algorithm would not be sensitive to sudden changes in workload. The larger the λ, for example, $\lambda = \frac{1}{4}$, the more sensitive cinterrupts is to bursty workloads. Because I/O completions are streams of requests and the calculation of interrupt rate lags slightly behind the workload, we chose smaller λ to enable smoother transitions in interrupt rate.

Figure 17(c) shows that with this storm protection algorithm, the interrupt rate does not exceed 100 K/sec. Crucially, limiting IRQ rate in this manner does not affect the performance of the workload. In fact, there is a slight increase in throughput (Figure 17(a)) and a slight decrease in request latency (Figure 17(b)) due to less interrupt perturbation. CPU utilization also decreases from 100% to 94% with storm protection.

Lines of Code. Linux modifications to support cinterrupts total around 100 LOC. The cinterrupts emulator in the NVMe driver is around 500 LOC, with an additional 200 LOC for implementations of strict Barrier and Urgent storm.

5 EVALUATION

These questions drive our evaluation: What is the overhead of our cinterrupts emulation (Section 5.1)? How do device vendors and admins select Δ and thr (Section 5.2)? How does cinterrupts compare to the default and the adaptive strategies in terms of latency and throughput (Section 5.3)? How much does cinterrupts improve latency and throughput in a variety of applications (Section 5.4)?

Table 2. Emulation Overhead Is Comparable with Overhead of Mitigations

	Sync latency of 4 KB, μs			
mitigations	off	default	off	off
system	baremetal interrupts	baremetal interrupts	emulation interrupts	baremetal polling
P3700	$80_{\pm 29.0}$	$81_{\pm 29.1}$	$82_{\pm 28.2}$	$78_{\pm 28.1}$
Optane	$10_{\pm 1.3}$	$11_{\pm 1.3}$	$10_{\pm 1.2}$	$8_{\pm 1.2}$

Cinterrupts runs with mitigations disabled to compensate for the emulation overhead.

5.1 Methodology

Experimental Setup. We use two NVMe SSD devices: Intel DC P3700, 400 GB [21] and Intel Optane DC P4800X, 375 GB [23]. We refer to them as P3700 and Optane.

Both SSDs are installed in a Dell PowerEdge R730 machine equipped with two 14-core 2.0 GHz Intel Xeon E5-2660 v4 CPUs and 128 GB of memory running Ubuntu 16.04. The server runs cinterrupts' modified version of Linux 5.0.8 and has C-states, Turbo Boost (dynamic clock rate control), and SMT disabled. We use the maximum performance governor.

Our emulation pairs one dedicated core to one target core. Each core is assigned its own NVMe submission and completion queue. The microbenchmarks are run on a single core, but we run macrobenchmarks on multiple cores. For our microbenchmarks, we use fio [4] version 3.12 to generate workloads. All of our workloads are non-buffered random access. We run the benchmarks for 60 seconds and report averages of 10 runs.

For a consistent evaluation of cinterrupts, we implemented an emulated version of the default strategy. Similar to the emulation of cinterrupts we described in Section 4.2.1, device interrupts are also emulated with IPIs.

Emulation Overhead. The cinterrupts emulation is lightweight and its overheads come from sending the IPI and cache contention from the dedicated core continuously polling on the CQ memory of the target core. Table 2 summarizes the overhead of emulation. We also show the overhead of mitigations for CPU vulnerabilities [31] to show that the overhead of our emulation is comparable to the overhead of the default mitigations for CPU. Therefore, our performance numbers with mitigations disabled and emulation on mirrors results from a server with mitigations enabled and emulation off (cinterrupts implemented in real hardware).

Emulation imposes a modest 3–6% latency overhead for both devices. There is a difference in emulation overhead between the devices, which we suspect is due to each device's time lag between updating the CQ and actually sending an interrupt. As the difference between the last column and the first column shows, this lag varies between devices, and the longer the lag, the smaller the overhead of emulation.

Baselines. We compare cinterrupts to our adaptive strategy and to the default interrupt strategy, which does not coalesce. The adaptive strategy is a proxy for comparison to NVMe coalescing, which it outperforms (Section 3.2).

5.2 Selection of Δ and thr

Δ should approximate the interarrival time of requests, which depends on workload. Figure 18 shows the interarrival time for two types of workloads. The first workload is a single-threaded workload that submits read requests of size 4 KB with libaio and iodepth = 256. The second workload is the same workload, except with batched requests. We run the same workloads on P3700 and Optane, to show that vendors or sysadmin will pick different Δ for different devices.

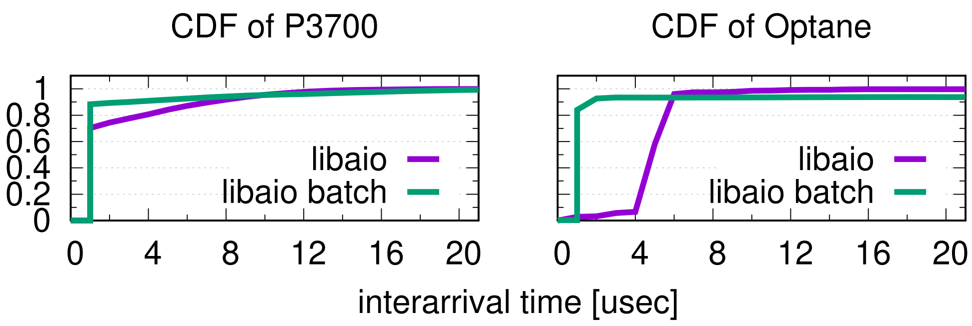

Fig. 18. Using interarrival CDF to determine Δ.

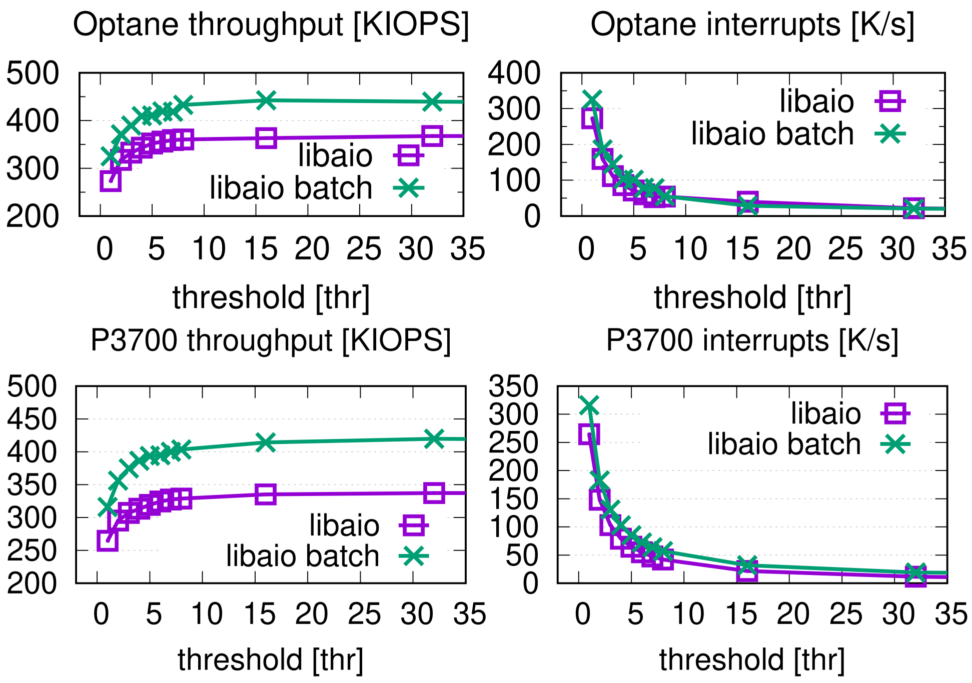

Fig. 19. Determining thr under a fixed Δ (Δ= 6 μs for Optane and Δ = 15 μs for P3700). thr is the smallest value where throughput plateaus, which is between 16–32, so we set *thr* = 32 for both devices. Note that the P3700 and Optane behave similarly, except the P3700 has about 15K lower throughput than the Optane.

When libaio submits batches, the CPU can send many more requests to the device, resulting in lower interarrival times—a 90th percentile of 1 μs in the batched case versus 6 μs in the non-batched case for Optane. For P3700, both workloads have a 99th percentile of 15 μs. We pick Δ to minimize the interrupt rate without adding unnecessary delay, so for P3700 we set Δ = 15 μs and for Optane we set Δ = 6 μs.

After fixing Δ, we sweep thr in the $[0, 256)$ range and select the lowest thr after which throughput plateaus; the results are shown in Figure 19. thr = 32 achieves high throughput and low interrupt rate for both devices.

In practice, hardware vendors should use this methodology to set default values to Δ and thr for their devices, and system administrators could tune these values for their workloads.

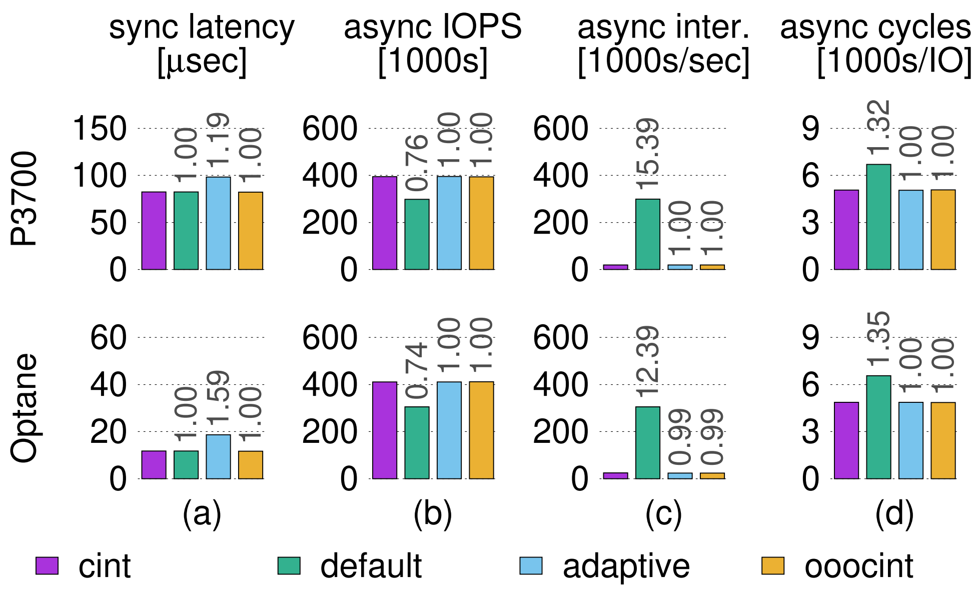

Fig. 20. Workloads with only one type of request. Column (a) shows latency of synchronous requests (lower is better); (b), (c) and (d) show metrics for the asynchronous workload.

5.3 Microbenchmarks

We use fio to generate two workloads to show how cinterrupts behaves at the extremes. The synchronous-only workload submits blocking 4 KB reads via read. The asynchronous-only workload submits 4 KB reads via libaio with iodepth 256 and batches of size 16. For each device, cinterrupts and the adaptive strategy are configured with the same Δ and thr. The results are shown in Figure 20.

As in Section 3.3, the synchronous workload shows the drawback of the adaptive strategy, which adds exactly Δ= 15 μs to request latency for P3700 and Δ= 6 μs for Optane (first column of Figure 20). Cinterrupts remedies this with Urgent. The default strategy performs as well as cinterrupts in the synchronous workload, because it generates an interrupt for every request. This strategy is penalized in the asynchronous workload, where the default strategy generates 12–15× the number of interrupts as cinterrupts.

Cinterrupts matches the synchronous latency of default, while achieving up to 35% more asynchronous throughput, and matches the asynchronous throughput of adaptive while achieving up to 37% lower latency. Finally, OOO does not add overhead to cinterrupts performance when it is not triggered.

CPU Fairness. Another tradeoff introduced by cinterrupts and OOO is CPU fairness. In particular, in the mixed workload originally described in Section 3.3, the nature of the interrupt coalescing scheme affects how much of the CPU each process can use.

Figure 21 shows the CPU utilization for each system while running the mixed workload. From the process scheduler's point of view, a fair schedule would enable each thread to get nearly 50% of the CPU, as is the case in the baseline, where all requests receive interrupts. However, with the adaptive strategy, the device inadvertently prevents this fair scheduling because it withholds interrupts from the synchronous thread: the synchronous thread only gets 23% of the CPU because it is blocked.

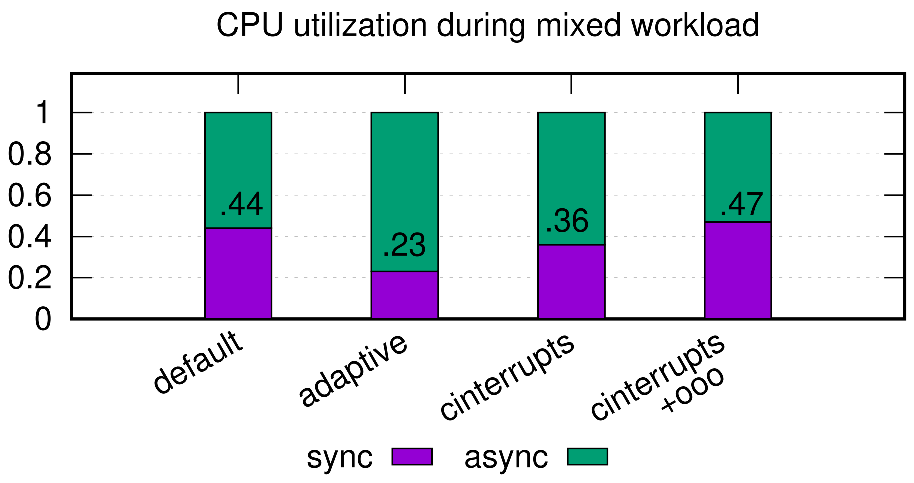

Fig. 21. CPU utilization for each process for the mixed workload. Both the baseline and cInterrupts + OOO allow the processes to use the CPU fairly.

Cinterrupts uses the Urgent hint to better approximate fair scheduling, and OOO processing achieves the same CPU allocation as the baseline, due to immediate delivery of interrupts for synchronous requests. This imbalanced CPU utilization also explains why cinterrupts has lower total throughput than adaptive in Figure 10. Per CPU unit, the synchronous process achieves lower throughput than the asynchronous process. Therefore, since cinterrupts gives the synchronous process more CPU time, the *total* throughput of both processes decreases with cinterrupts.

5.4 Macrobenchmarks

To evaluate the effect of cinterrupts on real applications, we run three application setups on Optane: RocksDB [64], KVell [48], and RocksDB and KVell colocated on the same cores. RocksDB is a widely used key-value store that uses pread/pwrite system calls in its storage engine, and KVell is a new key-value store employing Linux AIO in its storage engine. Both applications use direct I/O. KVell uses default kernel annotations (Barrier) while we will note when RocksDB uses default annotations or the modified annotations described in Section 4.1.2.

We run each application on two cores. In KVell, an additional four cores are allocated for clients. Cinterrupts is the only strategy that performs the best across all three setups.

5.4.1 RocksDB. Load. Using db_bench [8], we load a database with 10 M key-value pairs, with 16 byte keys and 1 KiB values. During the load phase, we compare the results under cinterrupts where RocksDB is unmodified and modified. In unmodified RocksDB, every I/O is labeled Urgent by default. In modified RocksDB, background activity is non-Urgent as described in Section 4.1.2. Table 3 shows the performance results.

We see that marking background activity as non-Urgent has a modest but significant 4% increase in throughput without affecting latency (app-cint vs cint). This is because delaying the interrupts of background I/O does not affect foreground latency. In fact, doing so actually decreases the tail latency of foreground writes by 15%. Hence reducing the CPU pressure caused by interrupts enables better p99 latency.

Steady State. After loading the database to 20 GB, we run two experiments from db_bench: readrandom, where each thread reads randomly from the key space, and readwhilewriting, where one

Table 3. Modifying RocksDB with Annotations That Make the Flush Non-Urgent (App-cint)

interrupt scheme	thruput [KIOPS]	norm	avg lat [ms]	norm	p99 lat [ms]	norm
cint	$388_{\pm 5.6}$	1.00	$1.3_{\pm 0.3}$	1.00	$15.0_{\pm 0.8}$	1.00
default	$391_{\pm 1.8}$	1.01	$1.3_{\pm 0.1}$	1.00	$14.4_{\pm 0.4}$	0.96
adaptive	$391_{\pm 4.6}$	1.01	$1.3_{\pm 0.4}$	1.00	$14.2_{\pm 0.9}$	0.95
app-cint	$405_{\pm 5.6}$	1.04	$1.3_{\pm 0.1}$	1.00	$12.7_{\pm 0.3}$	0.85

Results are for database load (fillbatch experiment in db_bench). Note that cint and default have the same performance, within error bounds, which is expected for RocksDB.

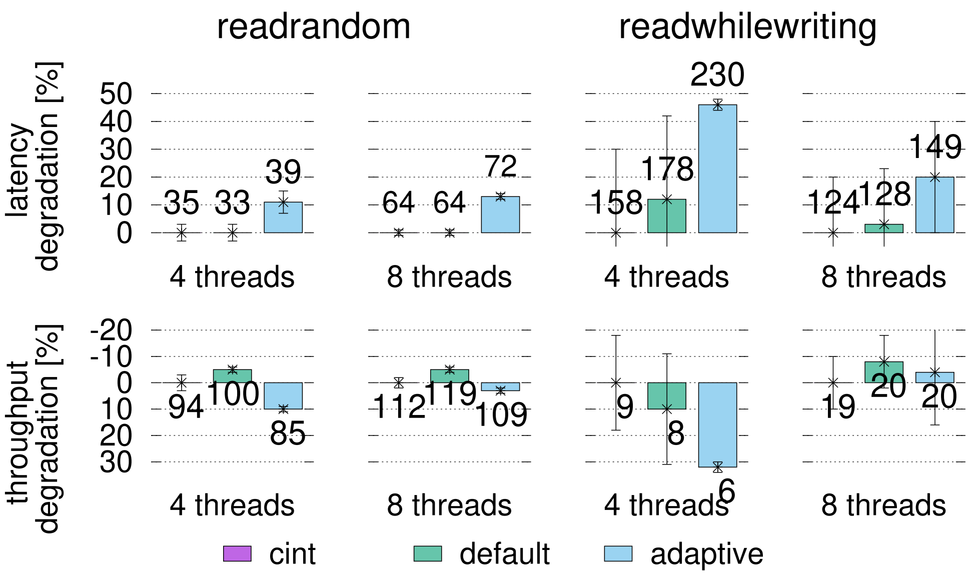

Fig. 22. Latency of get operation and throughput in RocksDB for varying workloads. We show performance degradation with respect to cinterrupts. Labels show absolute values in μs and KIOPS, respectively. As expected, cinterrupts and the default strategy have nearly the same performance within error bounds, but the adaptive strategy has up to 45% worse latency and 5-32% worse throughput due to the Δ delay.

thread inserts keys and the rest of the threads read randomly. The readwhilewriting experiment runs for 30 seconds. For each experiment, we also vary the number of threads. The latency of the get operation and throughput for both experiments is shown in Figure 22.

As expected, for both metrics, cinterrupts and the default strategy perform nearly the same because both generate interrupts for every request; in the next two applications, the default strategy will suffer due to this behavior. On the other hand, adaptive does consistently worse because of its Δ delay; this is particularly noticeable in the latency measurements. With eight threads, this delay penalty is amortized across more threads, which reduces the performance degradation.

Interestingly, modified RocksDB had similar performance to unmodified RocksDB during these benchmarks. This is because there is very little if any background I/O in the readrandom benchmark, and the write rate is not high enough for the background I/O interrupts to affect foreground performance in the readwhilewriting benchmark.

Table 4. Summary of YCSB Workloads

Workload	Description
A	update heavy: 50% reads, 50% writes
B	read mostly: 95% reads, 5% writes
C	read only: 100% reads
F	read latest: 95% reads, 5% updates
D	read-modify-write: 50% reads, 50% r-m-w
E	scan mostly: 95% scans, 5% updates

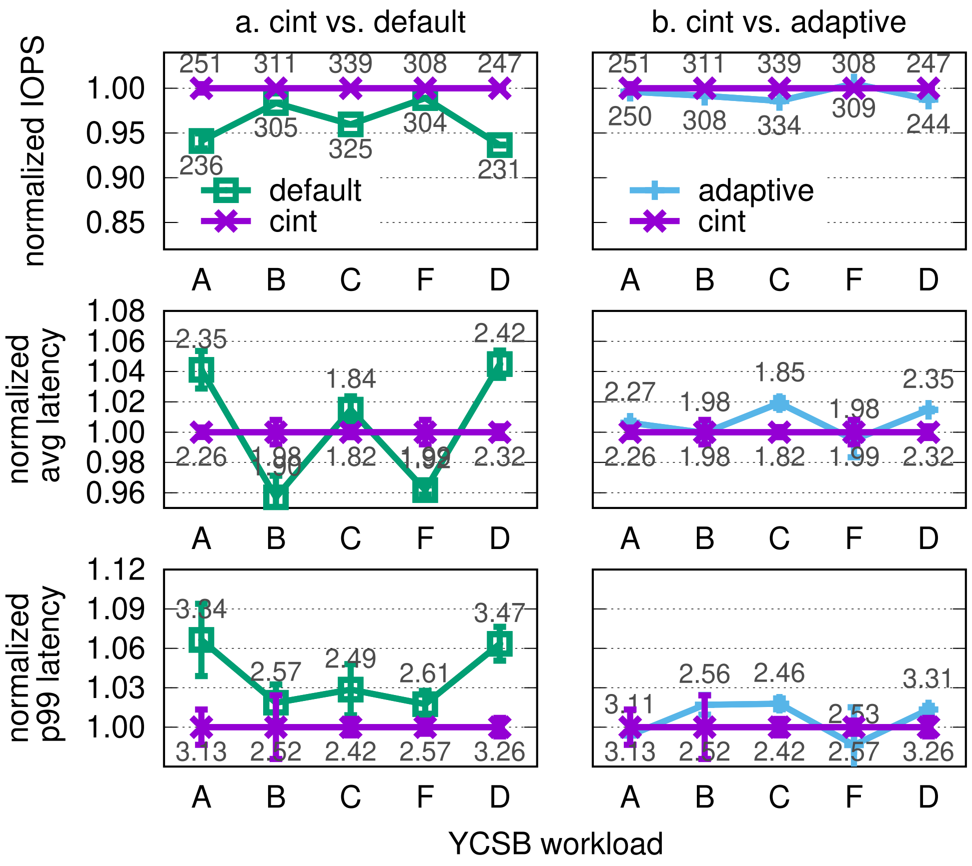

Fig. 23. Throughput and latency results for YCSB on KVell. Labels show absolute throughput in KIOPS and latency in ms.

5.4.2 KVell. We use workloads derived from the YCSB benchmark [14], summarized in Table 4. We load 80-M key-value pairs, with 24-byte keys and 1-KB item sizes for a dataset of size 80 GB. Each workload does 20M operations. Figure 23 shows KVell throughput, average latency, and 99th percentile latency for each YCSB workload.

Throughput. Cinterrupts does better than default for throughput, because default generates an interrupt for *every* request. In contrast, cinterrupts uses Barrier to generate an interrupt for a single batch, which consists of 10s of requests. The difference between cinterrupts and default is more pronounced for write-heavy workloads (A, D), but less pronounced for read-heavy workloads

Table 5. YCSB-E Throughput Results for KVell

	length=16		length=256	
interrupt scheme	scans [KIOPS]	normalized	scans [KIOPS]	normalized
cint	$26.2_{\pm 0.5}$	1.00	$1.6_{\pm 0.04}$	1.00
default	$23.1_{\pm 0.3}$	0.88	$1.5_{\pm 0.06}$	0.95
adaptive	$24.9_{\pm 0.6}$	0.95	$1.6_{\pm 0.02}$	0.97

Excessive interrupt generation limits default throughput to 86%-89% of cinterrupts'.

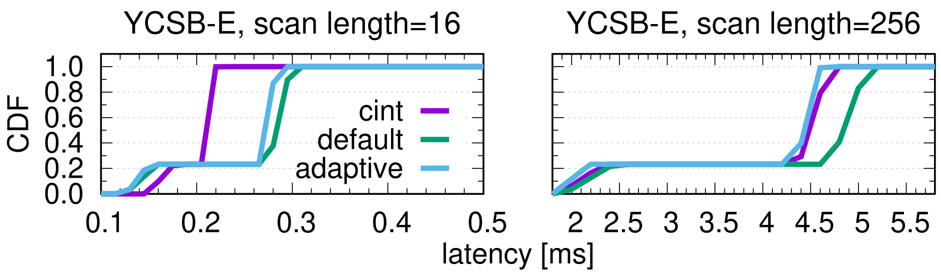

Fig. 24. Latency CDF of scans of length 16 and 256 in KVell.

(B, C, F). This is because reads are efficient in KVell, so there is some CPU idleness in these workloads (3% idleness under default and 14% idleness under cinterrupts).

The adaptive strategy performs similarly to cinterrupts because it is designed to detect bursts. Its delay is more pronounced in latency measurements.

Latency. The adaptive strategy has 5–8% higher average and 99th percentile latency than cinterrupts in all workloads. Again, this is the effect of the Δ delay, which cinterrupts remedies with Barrier. Cinterrupts latency also does better than the default, where interrupt handling and context switching both add to the latency of requests and slow down the request submission rate. The high number of interrupts in the default strategy also adds to latency variability, which is noticeable in the larger 99th percentile latencies.

YCSB-E. Scans are interesting because their latency is determined by the completion of requests that can span multiple submission boundaries. Table 5 shows throughput results for YCSB-E with different scan lengths, and Figure 24 shows latency CDFs for scans of length 16 and 256.

Similar to the other YCSB workloads, the adaptive strategy again can almost match the throughput of cinterrupts, because it is designed for batching. At higher scan lengths, factors such as application-level queueing begin affecting scan throughput, reducing the benefit of cinterrupts.

Figure 24 shows that there is a notable difference in scan latency between cinterrupts and the default for both scan lengths; the difference in 50th percentile latencies between default and cinterrupts is around 600 μs for both scan lengths. This difference is maintained at the 99th percentile latencies.

Notably, there is a 400-μs difference between cinterrupts and adaptive 50th percentile latencies when the scan length is 16, which goes away when the scan length is 256. The adaptive strategy does well in KVell's asynchronous programming model and longer scans are able to amortize the additional delay over many requests.

5.5 Colocated Applications

We run two types of colocated applications to see the effects of cinterrupts in consolidated datacenter environments.

Table 6. Performance of RocksDB Readrandom while the RocksDB Dump Tool Is Running in the Background

interrupt scheme	thruput [KIOPS]	norm	get lat [ms]	norm	p99 lat [ms]	norm
cint	$24.8_{\pm 3.9}$	1.00	$30.4_{\pm 0.4}$	1.00	$421_{\pm 41}$	1.00
default	$24.9_{\pm 1.6}$	1.00	$30.9_{\pm 0.3}$	1.02	$420_{\pm 25}$	1.00
adaptive	$20.2_{\pm 0.7}$	1.02	$43.9_{\pm 0.5}$	1.44	$85.1_{\pm 7.0}$	0.15
app-cint	$33.1_{\pm 1.2}$	1.37	$20.7_{\pm 0.4}$	0.68	75.7±0.1	0.14

app-cint modifies the dump tool to mark its I/O requests as non-urgent, boosting throughput and latency of the foreground get operations. Interestingly, the adaptive strategy can tame the tail latency (p99 lat) of get requests, but does so at the expense of limited IOPS.

5.5.1 RocksDB + Dump Tool. First, we run two colocated instances that both use pread/pwrite, which means by default the kernel marks all I/O as Urgent. The first is a regular RocksDB instance, and the second is a RocksDB instance running the RocksDB dump tool. As described in Section 4.1.2, we modify the RocksDB dump tool to explicitly disable the Urgent bit on its I/O requests.

We load two databases with 10 M key-value pairs, as in the previous section. Then, one database runs readrandom, as in the previous section, while we run the dump tool on the second database. We compare the performance of get requests under modified and unmodified RocksDB in Table 6. app-cint shows the results when the dump tool is modified.

By disabling Urgent in the dump tool, we increase the throughput of get requests by 37%, decrease the average latency by 32%, and decrease the 99th percentile latency by 86% compared to the kernel annotations cinterrupts. This is not only from the reduced interrupt rate generated by the dump tool, but also from the reduced I/O bandwidth generated by the dump tool. On the other hand, the throughput of the dump tool decreases by 11% under app-cint, but this is an acceptable trade-off for the foreground improvements.

5.5.2 RocksDB + KVell. Finally, we run RocksDB and KVell on the same cores. RocksDB runs the readrandom benchmark from before, and KVell runs YCSB-C. We run two experiments, varying the number of threads of the RocksDB instance. The latency of RocksDB requests and the throughput of KVell is shown in Tables 7 and 8.

When there are four RocksDB threads, the default strategy matches the RocksDB latency of cinterrupts, but has 12% less KVell throughput due to the excessive interrupt rate. Conversely, the adaptive strategy can match the KVell throughput of cinterrupts, but has 11% worse RocksDB latency.

As before, when there are more RocksDB threads, the effect of cinterrupts is less pronounced, because the CPU spends less of its time handling interrupts and more of its time context-switching and in userspace. Even so, cinterrupts still achieves a modest 5–6% higher throughput and up to 6% better latency than the other two strategies.

6 CINTERRUPTS FOR NETWORKING

Figure 6 (in Section 2) shows that NICs suffer from similar problems as NVMe drives with respect to interrupts. A natural future direction is applying cinterrupts to the networking stack. Accomplishing this goal, however, is more challenging, as explained next in the context of Ethernet.

Cooperation. For network cinterrupts to work, modifying a single host (as in storage) is insufficient. Multiple communicating parties should be changed to agree on how cinterrupts semantics

Table 7. Results from Colocated Experiment: 4 RocksDB Threads and KVell

interrupt scheme	RocksDB get lat [μs]	normalized	KVell [KIOPS]	normalized
cint	$116_{\pm 0.8}$	1.00	$171_{\pm 2.8}$	1.00
default	$115_{\pm 0.8}$	0.99	$153_{\pm 2.0}$	0.89
adaptive	$129_{\pm 0.0}$	1.11	$171_{\pm 2.0}$	1.00

As expected, cinterrupts both has lower latency than the adaptive strategy and higher throughput than the baseline.

Table 8. Results from Colocated Experiment: 8 RocksDB Threads and KVell

interrupt scheme	RocksDB get lat [μs]	normalized	KVell [KIOPS]	normalized
cint	$164_{\pm 0}$	1.00	$131_{\pm 1}$	1.00
default	$163_{\pm 0}$	0.99	$123_{\pm 0}$	0.94
adaptive	$174_{\pm 1}$	1.06	$124_{\pm 0}$	0.95

The performance gains of cinterrupts is reduced with respect to Table 7, because the CPU is both context-switching more and spending more time in userspace.

are communicated. In particular, transmitters should be modified to send network packets that indicate whether to fire an interrupt immediately upon reaching their destination, and receivers should be modified to react accordingly. Any interrupt-driven software routers along the way should also preferably support cinterrupts.

Propagation. NVMe controllers deal with plaintext read or write requests associated with pointers to buffers; the controller either copies bytes from the buffers to the drive or vice versa. This is true even when NVMe is encapsulated in other, higher-level protocols, as is the case with, e.g., NVMe over TCP over TLS encryption (denoted NVMeTLS [62]).

In contrast, network stacks are layered. NICs may operate at the Ethernet level, but the content of Ethernet frames frequently encapsulates higher-level protocols, like IP, TCP, UDP, and VXLAN. Crucially, the transmitted payload, which includes headers of higher-level protocols, is oftentimes encrypted. For example, the payload in tunnel-mode IPsec packets [36] encapsulates encrypted IP and TCP headers. Bits in these headers are thus unsuitable for communicating cinterrupts information, as the receiving NIC might not be able to observe them. Consequently, to support cinterrupts, each layer at the sender should be modified to explicitly propagate cinterrupts information to its encapsulating layer, until a low-enough protocol level is reached.

Within a data center, it seems reasonable to choose Ethernet as the aforementioned low-enough protocol. In practice, however, there are no free Ethernet reserved bits or flags that can be used for this purpose [69]. Cinterrupts bits can instead reside one level higher, at the least-encapsulated IP layer, as its headers are not encrypted, and its "options" field [44] can be used to add the extra bits. The downside is that other, non-IP networks—such as RDMA over Converged Ethernet (RoCE) [73] and Fiber Channel over Ethernet (FCoE) [32]—should be handled separately, in some other way.

Segmentation. TCP performance is accelerated by NIC offloading functionality, which significantly reduces CPU processing overhead. Notably, upon transmit, software may use TSO (TCP segmentation offload) to hand a sizable (≤64KB) TCP segment to the NIC, relying on the NIC to split the outgoing segment into a sequence of (≤MTU) Ethernet frames [51]. Likewise, with LRO (large receive offload), the NIC may reassemble multiple incoming frames into a single sizable segment before handing it to software [51, 52].

Storage cinterrupts affect only the timing and number of device interrupts. Network cinterrupts can also increase the number of I/O requests and thus the CPU usage, assuming TSO and LRO are not applied beyond cinterrupts. To illustrate, assume $\{M_i\}_{i=0}^{15}$ is a consecutive series of 1-KB messages, each individually associated with a cinterrupt. To optimize latency, differently than what frequently happens on existing systems, $\{M_i\}_{i=0}^{15}$ should seemingly *not* be aggregated into a single 16-KB TCP segment at the sender before it is handed to the NIC (leveraging TSO), nor should it be aggregated to a single segment by the receiver NIC (leveraging LRO); otherwise only the cinterrupt of M_{15} will survive. But such a policy might inadvertently degrade both latency and throughput if the CPUs of the sender or receiver are saturated, necessitating a more sophisticated policy that considers CPU usage.

URG and PSH. The TCP flags URG and PSH seem related to cinterrupts. But even if ignoring the aforementioned propagation problem, in practice, the semantics of these flags are sufficiently different that they cannot be repurposed for cinterrupts. Specifically, URG is used to implement socket out-of-band communication [27, 29, 54], and its usage model involves the POSIX SIGURG signal. (URG also has security implications [28, 29, 81], and middleboxes and firewalls tend to clear it by default [12, 59].) When examining how PSH is used in the Linux network stack (see calls to the tcp_push and tcp_mark_push functions in the source code), we find that it is used in many more circumstances than is appropriate for cinterrupts. For example, sending a 16-KB message using a single write system call frequently results in four Ethernet frames encapsulating PSH segments, instead of one.

7 CONCLUSION

In this article, we show that the existing NVMe interrupt coalescing API poses a serious limitation on practical coalescing. In addition to devising an adaptive coalescing strategy for NVMe, our main insight is that software directives are the best way for a device to generate interrupts. Cinterrupts, with a combination of Urgent, Barrier, and the adaptive burst-detection strategy, enabling workloads to experience better performance even in a dynamic environment. In doing so, cinterrupts enables the software stack to take full advantage of existing and future low-latency storage devices.

ACKNOWLEDGMENTS

We are deeply indebted to Orran Krieger for helping shape the final version of this article. We are also grateful to the anonymous reviewers of all draft versions for their comments that have greatly improved this article. We also thank Marcos Aguilera, Nadav Amit, and Jon Howell for discussions and comments on earlier iterations of this work.

REFERENCES

[1] *Administration and Data Access Tool.* https://github.com/facebook/rocksdb/wiki/Administration-and-Data-Access-Tool. ([n.d.]). Accessed: May 2021.
[2] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. 2011. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (USENIX ATC).*
[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[4] Jens Axboe. *Flexible I/O Tester.* https://github.com/axboe/fio. ([n.d.]). Accessed: May 2021.
[5] Jens Axboe. 2016. Linux Kernel Mailing List, BLK-MQ: Make the Polling Code Adaptive. https://lkml.org/lkml/2016/11/3/548. (2016). Accessed: May 2021.
[6] Pavel Begunkov. 2019. Linux Kernel Mailing List, Blk-mq: Adjust Hybrid Poll Sleep Time. https://lkml.org/lkml/2019/4/30/120. (2019). Accessed: May 2021.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[8] *Benchmarking Tools*. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. ([n.d.]). Accessed: May 2021.

[9] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *International Systems and Storage Conference (SYSTOR).*

[10] Block IO Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt. ([n.d.]). Accessed: May 2021.

[11] Keith Bush. 2019. *Linux NVMe Mailing List: Nvme-PEI Interrupt Handling Improvements.* https://lore.kernel.org/linux-nvme/20191209175622.1964-1-kbusch@kernel.org/. (2019). Accessed: May 2021.

[12] Cisco Systems, Inc. 2021. Cisco ASA Series Command Reference: Urgent-flag. https://www.cisco.com/c/en/us/td/docs/security/asa/asa-cli-reference/T-Z/asa-command-ref-T-Z/u-commands.html#wp2606000884. (2021). Accessed: May 2021.

[13] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *USENIX Annual Technical Conference (USENIX ATC).*

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC).*

[15] Jonathan Corbet. Batch Processing of Network Packets. https://lwn.net/Articles/763056/. ([n.d.]). Accessed: May 2021.

[16] Jonathan Corbet. Driver Porting: Network Drivers. https://lwn.net/Articles/30107/. ([n.d.]). Accessed: May 2021.

[17] Intel Corporation. Intel Optane SSD DC D4800X Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-d4800x-product-brief.pdf. ([n.d.]). Accessed: May 2021.

[18] Intel Corporation. Intel Optane Technology for Data Centers. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html. ([n.d.]). Accessed: May 2021.

[19] Intel Corporation. 2012. Intel Data Direct I/O Technology (Intel DDIO): A Primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. (2012). Accessed: May 2021.

[20] Intel Corporation. 2014. Intel Ethernet Converged Network Adapter XL710. https://ark.intel.com/content/www/us/en/ark/products/83967/intel-ethernet-converged-network-adapter-xl710-qda2.html. (2014). Accessed: May 2021.

[21] Intel Corporation. 2014. Intel SSD DC P3700 Series. https://ark.intel.com/content/www/us/en/ark/products/79624/intel-ssd-dc-p3700-series-400gb-1-2-height-pcie-3-0-20nm-mlc.html. (2014). Accessed: May 2021.

[22] Intel Corporation. 2017. Intel Optane SSD 900P Series. https://ark.intel.com/content/www/us/en/ark/products/123623/intel-optane-ssd-900p-series-280gb-2-5in-pcie-x4-20nm-3d-xpoint.html. (2017). Accessed: May 2021.

[23] Intel Corporation. 2017. Intel Optane SSD DC P4800X Series. https://ark.intel.com/content/www/us/en/ark/products/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint.html. (2017). Accessed: May 2021.

[24] Intel Corporation. 2018. Intel Optane SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201861/intel-optane-ssd-dc-p5800x-series-400gb-2-5in-pcie-x4-3d-xpoint.html. (2018). Accessed: May 2021.

[25] Intel Corporation. 2019. Intel SSD DC P4618 Series. https://ark.intel.com/content/www/us/en/ark/products/192574/intel-ssd-dc-p4618-series-6-4tb-1-2-height-pcie-3-1-x8-3d2-tlc.html. (2019). Accessed: May 2021.

[26] Microsoft Corporation. 2019. Microsoft Documentation: Optimize Performance on the Lsv2-series Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/storage-performance. (2019). Accessed: May 2021.

[27] Kevin R. Fall and W. Richard Stevens. 2011. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley.

[28] Fernando Gont. 2012. *Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations.* Technical Report. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-tcp-security-03 Work in Progress.

[29] Fernando Gont and Andrew Yourtchenko. 2011. *On the Implementation of the TCP Urgent Mechanism.* RFC 768. Internet Engineering Task Force.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is dead: Long live KASLR. In *International Symposium on Engineering Secure Software and Systems (ESSoS).*

[31] *Hardware Vulnerabilities, The Linux Kernel User's and Administrator's Guide.* https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/index.html. ([n.d.]). Accessed: May 2021.

[32] John Hufferd. 2013. *Fibre Channel over Ethernet (FCoE).* https://www.snia.org/educational-library/fibre-channel-over-ethernet-fcoe-2013-2013. (2013). Accessed: May 2021.

[33] Intel. 2014. DPDK: Data Plane Development Kit. https://www.dpdk.org. (2014). Accessed: May 2021.

[34] Rick A. Jones. 1995. *Netperf: A Network Performance Benchmark*. https://github.com/HewlettPackard/netperf. (1995). Accessed: May 2021.
[35] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *3rd ACM Symposium on Cloud Computing (SoCC)*.
[36] S. A. Kent and R. Atkinson. 1998. *Security Architecture for the Internet Protocol*. RFC 2401. Internet Engineering Task Force. 66 pages. http://www.rfc-editor.org/rfc/rfc2401.txt.
[37] Byungseok Kim, Jaeho Kim, and Sam H. Noh. 2017. Managing array of SSDs when the storage device is no longer the performance bottleneck. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[38] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[39] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O path: A holistic approach for application performance. In *USENIX Conference on File and Storage Technologies (FAST)*.
[40] Avi Kivity. 2018. *Wasted Processing Time due to NVMe Interrupts*. https://github.com/scylladb/seastar/issues/507. (2018). Accessed: May 2021.
[41] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 2019. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency SSDs. In *IEEE International Symposium on Workload Characterization (IISWC)*.
[42] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. 2018. Exploring system challenges of ultra-low latency solid state drives. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[43] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *USENIX Conference on File and Storage Technologies (FAST)*.
[44] Charles M. Kozierok. *The TCP/IP Guide*. http://www.tcpipguide.com/free/t_IPDatagramOptionsandOptionFormat.htm. ([n.d.]). Accessed: May, 2021.
[45] Damien Le Moal. 2017. I/O latency optimization with polling. In *Linux Storage and Filesystems Conference (VAULT)* (2017).
[46] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *USENIX Annual Technical Conference (USENIX ATC)*.
[47] Ming Lei. 2019. Linux-NVMe Mailing List: NVMe-PCI: Check CQ after batch submission for Microsoft device. https://lore.kernel.org/linux-nvme/20191114025917.24634-3-ming.lei@redhat.com/. (2019). Accessed: May 2021.
[48] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*.
[49] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems (EuroSys)*.
[50] Long Li. 2019. *Linux kernel mailing list: Fix interrupt swamp in NVMe*. https://lkml.org/lkml/2019/8/20/45. (2019). Accessed: May 2021.
[51] Linux Kernel Documentation. 2021. *Segmentation Offloads*. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html. (2021). Accessed: May 2021.
[52] Mellanox Technologies. 2020. *How To Enable Large Receive Offload (LRO)*.https://community.mellanox.com/s/article/how-to-enable-large-receive-offload--lro-x. (2020). Accessed: May 2021.
[53] Merriam-Webster. 2020. "Calibrate". https://www.merriam-webster.com/dictionary/calibrate. (2020). Accessed: May 2021.
[54] Microsoft Corporation. 2018. *OOB Data in TCP*. https://docs.microsoft.com/en-us/windows/win32/winsock/protocol-independent-out-of-band-data-2#oob-data-in-tcp. (2018). Accessed: May 2021.
[55] Jeffrey C. Mogul and Kadangode K. Ramakrishnan. 1996. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX Annual Technical Conference (ATC)*.
[56] Rikin J. Nayak and Jaiminkumar B. Chavda. 2017. Comparison of accelerator coherency port (ACP) and high performance port (HP) for data transfer in DDR memory using Xilinx ZYNQ SoC. In *International Conference on Information and Communication Technology for Intelligent Systems (ICTIS)*.
[57] *NVM Express, Revision 1.3*. https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf. ([n.d.]). Accessed: May2021.
[58] *NVM Express, Revision 1.4*, Figure 284. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf. ([n.d.]). Accessed: May 2021.
[59] Palo Alto Networks. 2018. *How to Preserve the TCP URG Flag and Pointer*. https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000ClWACA0. (2018). Accessed: May 2021.
[60] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *USENIX Annual Technical Conference (USENIX ATC)*.

[61] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[62] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC offloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[63] *preadv2(2) – Linux Manual Page.* https://man7.org/linux/man-pages/man2/preadv2.2.html. ([n.d.]). Accessed: May, 2021.

[64] *RocksDB.* . https://github.com/facebook/rocksdb. ([n.d.]). Accessed: May 2021.

[65] Samsung. 2017. *Samsung SSD 850 PRO.* https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850pro. (2017).

[66] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. OS I/O path optimizations for flash solid-state drives. In *USENIX Annual Technical Conference (USENIX ATC).*

[67] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[68] *SPDK: Storage Performance Development Kit.* https://spdk.io/. ([n.d.]). Accessed: May 2021.

[69] Charles E. Spurgeon and Joann Zimmerman. *Ethernet: The Definitive Guide,* 2nd Edition. https://www.oreilly.com/library/view/ethernet-the-definitive/9781449362980/ch04.html. ([n.d.]). Accessed: May, 2021.

[70] Steven Swanson and Adrian M. Caulfield. 2013. Refactor, reduce, recycle: Restructuring the IO stack for the future of storage. *Computer* (2013).

[71] Billy Tallis. 2017. *Intel Optane SSD DC P4800X 750GB Hands-On Review.* https://www.anandtech.com/show/11930/intel-optane-ssd-dc-p4800x-750gb-handson-review/3. (2017). Accessed: May 2021.

[72] Mellanox Technologies. 2018. *Mellanox ConnectX-5 VPI Adapter*. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf. (2018). Accessed: May 2021.

[73] The RoCE Initiative. *RoCE Is RDMA over Converged Ethernet.* https://www.roceinitiative.org. ([n.d.]). Accessed: May 2021.

[74] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS).*

[75] John E. Uffenbeck. 1997. *The 80x86 Family: Design, Programming, and Interfacing*. Prentice Hall.

[76] Western Digital Corporation. *Ultrastar DC SN200.* https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-nvme-series/data-sheet-ultrastar-dc-sn200.pdf. ([n.d.]). Accessed: May 2021.

[77] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world dtabases. In *ACM International Systems and Storage Conference (SYSTOR).*

[78] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When poll is better than interrupt. In *USENIX Conference on File and Storage Technologies (FAST).*

[79] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: First class support for interactivity in commodity operating systems.. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[80] Tom Yates. *Improvements to the Block Layer.* https://lwn.net/Articles/735275/. ([n.d.]). Accessed: May 2021.

[81] Young Yoon, Jae Yong Oh, and Young Min Yoon. 2001. NIDS evasion method named "SeolMa". *Phrack Magazine, Volume 0x0b, Issue 0x39* (2001).

[82] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. 2014. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)* (2014).

[83] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 2018. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[84] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. 2020. Scalable parallel flash firmware for many-core architectures. In *USENIX Conference on File and Storage Technologies (FAST).*

Received October 2021; accepted December 2021