Optimizing storage I/O with calibrated interrupts

Anonymized, Under submission – please do not distribute
Paper #279

Abstract

NVMe storage devices are beginning to approach the performance of networking devices, which means interrupt storms, which first appeared in fast networks, are starting to affect storage stacks. We find that interrupt coalescing, borrowed from networking and adopted by NVMe, is not only suboptimal but unusable for practical workloads. Instead, we propose calibrated interrupts, which includes an adaptive coalescing strategy to replace NVMe coalescing, and leverages a slight modification to the device interface to enable software to annotate when requests should be interrupted, exploiting the fact that in storage, unlike networking, I/O is requested and completed by the same entity. Calibrated interrupts spend 30% fewer cycles per request and improve throughput by as much as 35% over state-of-the-art approaches.

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 [29, 37, 63, 65]. In storage, these costs have gained attention as new interconnects such as NVM Express^{TM} (NVMe) enable applications to not only submit millions of requests per second, but up to 65,535 concurrent requests [5, 10, 13, 15, 19]. With so many concurrent requests, sending interrupts for every completion could result in an interrupt storm, grinding the system to a halt [46, 58]. Since CPU is already the bottleneck to driving high IOPS [43, 44, 47, 48, 49, 51, 64, 67, 71, 72], excessive interrupts can be fatal to the ability of software to fully utilize existing and future storage devices.

Networking devices, which handle many more I/O requests than storage devices, also experience interrupt storms. As a result, storage stacks have adopted techniques from networking such as interrupt coalescing, available in NVMe devices. We find that NVMe interrupt coalescing is unusable for most storage workloads because it is practically impossible to select batch sizes or timeouts that do not cause excessive latency or stall the storage stack. For the workloads we inspected, CPU utilization increases by as much as 55% without coalescing (Figure 10(d)), while request latency increases by as much as 9× with coalescing, due to large timeouts. In fact, interrupt coalescing is disabled by default in Linux and real deployments work around it (§2).

In this paper, we address the challenge of dealing with exponentially increasing interrupt rates without sacrificing latency by leveraging two insights. First, NVMe coalescing must be adaptive rather than static, as it is today, in order to be practical. We implement adaptive coalescing for NVMe, a dynamic approach that tries to adjust batching based on the workload, but find that it still adds unnecessary latency to requests (§3.2).

Our second insight is that storage stacks, where requests originate from the host, are fundamentally different from networking stacks, where requests arrive unsolicited at line rate. The I/O requester knows when it should be notified of completions, but there is no way to express this information to the device, leaving the device to use heuristics to guess when to notify software. Storage stacks have an opportunity to exploit this difference.

We bridge this semantic gap by allowing the requestor to inform the device of when it wants to be interrupted. 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 (§3).

For application developers that do not want to reason about individual device requests, we augment the kernel to annotate all IO requests with *default* calibrations, based on the system call that originated the IO request (§4.2), which improves the throughput of unmodified applications by up to 14%. We also modify the kernel to expose a system call interface that allows applications to override these defaults. A mere 42-line patch to use this interface can improve the throughput of RocksDB by up to 37% and reduce the tail latency by up to 86% over traditional interrupts (§5.5.1), showing how application-level semantics can unlock performance benefits greater than those achieved by the default calibrations.

Because cinterrupts modifies how storage devices generate

[1]To calibrate: to adjust precisely for a particular function [57].

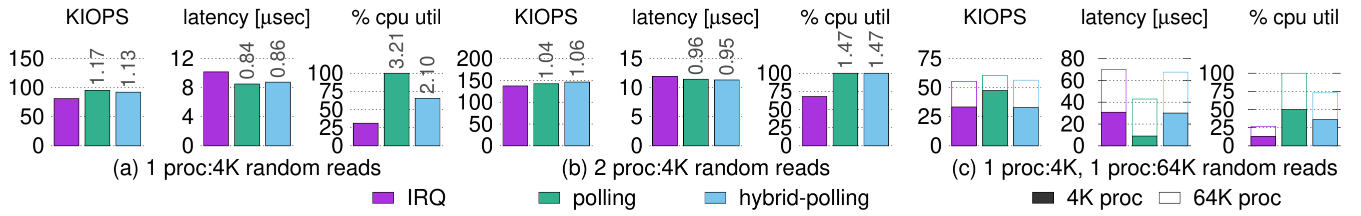


Figure 1: Hybrid polling is not enough to mitigate polling overheads. (a) With a single thread of same-sized requests, hybrid polling effectively reduces the CPU utilization of polling while matching the performance of polling. (b) With more threads, hybrid polling reduces to polling in terms of CPU utilization without providing significant performance improvement over interrupts. (c) With variable IO sizes, hybrid polling has the same throughput and latency as interrupts for both IO sizes while using 2.7x more CPU. Labels show performance relative to IRQ.

interrupts, supporting it requires modifications to the device. However, these are minimal changes that would only require a firmware change in most devices, while enabling significant performance gains. We build an emulator for cinterrupts in Linux 5.0.8 which runs on real NVMe hardware (§4). In microbenchmarks, cinterrupts can match 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 [53] on YCSB benchmarks by as much as 14% over the state-of-the-art and improve latency by up to 28% over our adaptive approach. Alternative techniques favor specific workloads at the expense of others (§5).

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 [10, 19], but also multiple concurrent requests per channel. The NVMe specification [6] exposes this parallelism to software by providing multiple queues where requests can be submitted and completed; Linux rewrote its block subsystem to match this multi-queue paradigm [30].

Numerous kernel, application, and firmware-level improvements have been proposed in the literature to unlock the higher request rate of these devices [32, 45, 51, 53, 60, 62, 72, 73], but they focus on increasing IO *submit* rate without directly addressing the problem of higher *completion* rate.

Lessons from networking. Networking devices have had much higher completion rates for a long time. For example, 100Gbps 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.

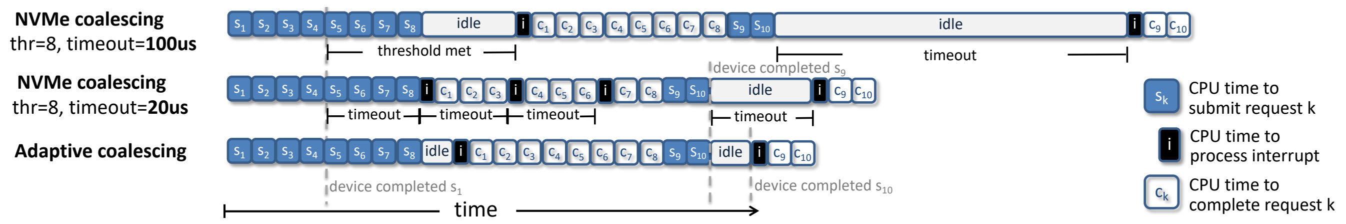
To avoid bombarding the processor with interrupts, network devices apply interrupt coalescing [66], which waits until a threshold of packets are available or a timeout is triggered. Network stacks may also employ polling [35], where software queries for packets to process rather than being notified. IX [29] and DPDK [40] 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 [11] or ARM's ACP [59] 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 [24, 29, 42, 54].

Storage is adopting networking techniques. The NVMe specification standardizes the idea of interrupt coalescing for storage devices [6], but there are two key problems with NVMe interrupt coalescing. First, NVMe only allows the aggregation time to be set in 100μs increments [6], while devices are approaching sub 10μs latencies. This means that requests that do not meet the threshold would incur unacceptably long latencies[2], rendering the timeout unusable in general-purpose deployments.

Second, even if the NVMe aggregation granularity were more reasonable, for example 10μs, NVMe interrupt coalescing is fundamentally unusable because both the threshold and timeout are *statically* configured. 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 [7], and off-the-shelf NVMe devices ship with coalescing disabled. Real NVMe deployments resort to driver workarounds and employ non-standard proprietary coalescing schemes rather than using NVMe coalescing [20, 31, 52, 55].

Polling is expensive. μdepot [49] 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 kernel-polling is known to waste CPU cycles [48, 68, 72]. FlashShare only uses polling for what it categorizes as low-latency applications [72], but

[2]On our setup, requests that normally take 11μs are delayed to 100μs, resulting in a 9× latency increase.



NVMe coalescing thr=8, timeout=100us
NVMe coalescing thr=8, timeout=20us
Adaptive coalescing
idle
threshold met
timeout
device completed s9
device completed s1
device completed s10
time
CPU time to submit request k
CPU time to process interrupt
CPU time to complete request k

Figure 2: NVMe coalescing with its current 100μs granularity (top row) causes unusable timeouts when the threshold is not met ($c_9 - c_{10}$). 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 ($c_9 - c_{10}$).

acknowledges that this is still is expensive. Cinterrupts addresses this by exposing application-semantics to interupt generation so that systems do not have to resort to polling.

Even hybrid polling [70], which is a heuristic-based technique for reducing the CPU overhead of polling, is insufficient, breaking down when requests having varying size [26, 47, 50].

Figure 1 compares the performance and CPU utilization of hybrid polling, polling, and interrupts for three benchmarks on an Intel Optane DC P4800X [16] [3]. We note that in all cases, polling provides the lowest latency, at the expense of 100% CPU utilization. When there is a single thread submitting requests (Figure 1(a)), hybrid polling does well because request completions are uniform. However, when more threads or IO sizes are added, as in Figures 1(b)-(c), hybrid polling still has 1.5x-2.7x higher CPU utilization than interrupts without providing noticeable performance improvement. These results match other findings in the literature [26, 28, 47, 50].

Even though polling is 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 [34, 35], which switches between polling and interrupts based on demand.

Exposing application-level semantics. Similar to [38, 39, 45, 69, 72], cinterrupts uses a few bits to transmit information about the application to the device. vIC [23] tries to moderate virtual interrupts by estimating IO completion time. Hybrid polling also tries to leverage this by polling when the completion is expected from the device.

3 Cinterrupts

3.1 Design overview

To address completion overheads in emerging storage devices, cinterrupts employs two main insights.

Insight 1: NVMe coalescing must be adaptive. As described in Section 2, NVMe interrupt coalescing is unusable because its coalescing rate cannot adapt to workload changes. Interestingly, the large 100μs timeout exacerbates but is not the fundamental cause. Figure 2 shows that even if the timeout granularity were smaller, it is still *fixed*, which means that when the workload doesn't need interrupts ($c_1 - c_8$), interrupts will be generated, while when the workload does need interrupts, completion must wait for the timeout to expire ($c_9 - c_{10}$).

Ideally, as shown in the bottom row of Figure 2, coalescing rate adapts dynamically to the workload. To capture this intuition, our first contribution is an adaptive coalescing strategy to replace the static NVMe algorithm (§3.2).

Insight 2: Storage is different from networking. Ultimately, device-level heuristics for coalescing will always fall short due to missing semantics: the device sees a stream of requests and cannot determine which requests require interrupts in order to unblock the application. Even sophisticated coalescing strategies such as those in high-speed NICS are suboptimal. Figure 3 shows the interrupt rate and latency of the the TCP_RR, i.e. ping-pong, benchmark of *netperf* [41] with varying message sizes on two NICS, an Intel XL710 40 GbE [12] and a Mellanox ConnectX-5 100 GbE [17].

On both devices, interrupt coalescing helps reduce the interrupt rate but causes an increase in latency. In the Intel NIC, coalescing results in increased latency regardless of message

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

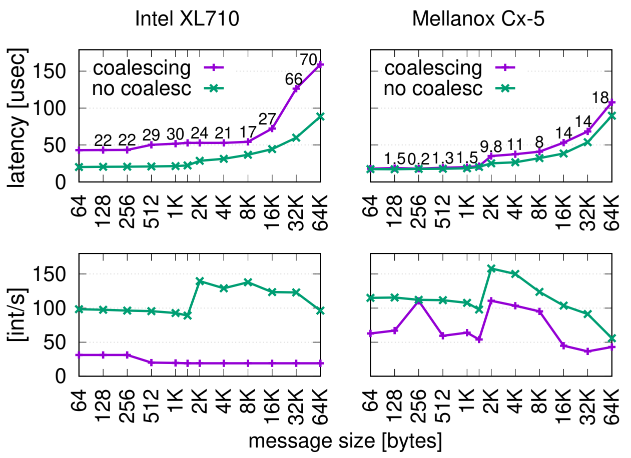
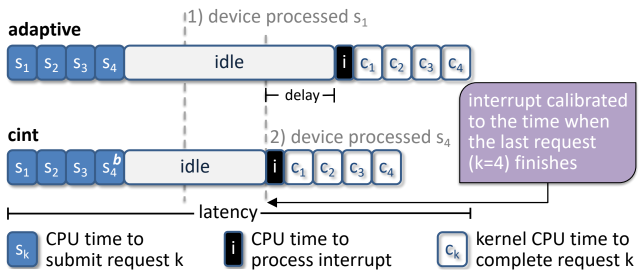


Figure 3: Missing semantics causes the additional latency imposed by coalescing. Latency and interrupt rate of *netperf* TCP_RR with no coalescing and with adaptive interrupt coalescing enabled. Labels show the differences between latencies.

Algorithm 1: Adaptive coalescing strategy in cinterrupts

```
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
               /* burst detection,update timeout */
6              timeout = now + $\Delta$;
7              if ++coalesced $\geq thr$ then
8                  fire IRQ and reset;
       /* end of quiescent period */
9      if coalesced > 0 then
10         fire IRQ and reset;
11     timeout = now + $\Delta$;
```



adaptive
1) device processed s_1
s_1 s_2 s_3 s_4
idle
i
c_1 c_2 c_3 c_4
delay
cint
2) device processed s_4
s_1 s_2 s_3 s_4^b
interrupt calibrated to the time when the last request (k=4) finishes
latency
CPU time to submit request k
CPU time to process interrupt
kernel CPU time to complete request k

Figure 4: Completion timeline for multiple submissions. The adaptive strategy can detect them as part of a burst, but only after the delay expires, which confirms the end of the batch. Cinterrupts explicitly marks the last request in the batch.

size. In the Mellanox NIC, coalescing adds latency if the message size is greater than 1500 bytes, the maximum transmission unit for Ethernet, because the message becomes split across multiple packets. To address this, NIC vendors have tried to add even more advanced heuristics such as Intel's low-latency interrupts (LLI) [18, 21]. In the end, however, users have had difficulty using these heuristics [36], and they have been dropped from recent devices [12, 14].

Due to the lessons from networking, cinterrupts avoids pushing more policy into the hardware by letting software dictate completion semantics. Fortunately, unlike network devices, where unsolicited incoming packets can arrive at line rate, every completion (and resulting interrupt) from a storage device is the result of a software submission. Cinterrupts takes full advantage of this difference by propagating the semantic information available at submission time to the device.

3.2 Adaptive Coalescing

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. To detect these bursts, the adaptive strategy uses a parameter, Δ, which is a bound on request interarrival time.

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. 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 §5, we explain how device manufacturers and system administrators can determine reasonable Δ and *thr* configurations.

Figure 4 shows a completion timeline with the adaptive strategy. The adaptive strategy can accurately detect bursts, although it adds Δ delay to confirm the end of a burst. Without additional information, this Δ delay is unavoidable, because the device cannot tell which request is the last one. Cinterrupts addresses this problem by enhancing adaptive coalescing 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 [25] with two threads: one submitting 4 KB read requests via libaio with iodepth=16[4] 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 §4.2.1). Results are shown in Figure 5.

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 2.7x more interrupts than cinterrupts (Figure 5(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 read 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 5(c)).

Even the adaptive coalescing schemes in NICs cannot differentiate request semantics. These adaptive schemes favor

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

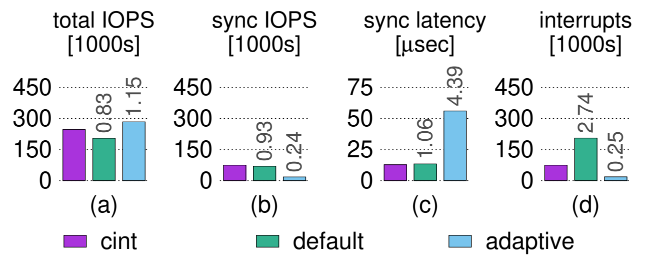


Figure 5: Effect of Urgent. Cinterrupts achieves optimal synchronous latency and better throughput over the default (no coalescing). The adaptive strategy achieves better overall throughput, at the expense of synchronous latency. Synthetic workload mixing libaio and read requests. Labels show performance relative to cinterrupts.

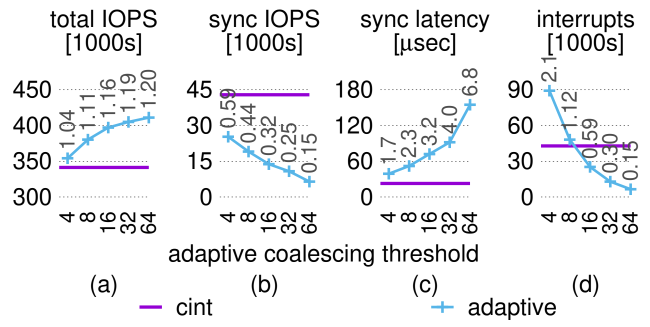


Figure 6: Sophisticated adaptive coalescing, such as that found in NICs, tries to throttle interrupt rate with aggressive coalescing. In a mixed workload, increasing the coalescing threshold will only increase the latency of synchronous requests proportionally to the coalescing rate. Labels show performance relative to cinterrupts.

the asynchronous thread with aggressive coalescing and interrupt throttling [22, 56], which simply overwhelms the synchronous requests, leading to unusable synchronous latencies.

Figure 6 shows how these sophisticated adaptive schemes behave at higher throughputs. We run the same experiment as in Figure 5 with higher iodepth. As the target coalescing rate increases for the adaptive strategy, there is a corresponding linear increase in the synchronous latency. On the other hand, the purple line in Figure 6(c) shows that Urgent in cinterrupts makes synchronous latency acceptable. This latency comes at the expense of less asynchronous throughput, as shown in Figure 6(a), but we believe this is an acceptable trade-off. §3.5 will show that Urgent can be combined with throttling to achieve more rigorous performance guarantees.

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 finished. For example, in the submission stream $s_1 - s_4$ in Figure 4, the last request, s_4, is marked with Barrier. Barrier minimizes the interrupt rate, which is always beneficial for

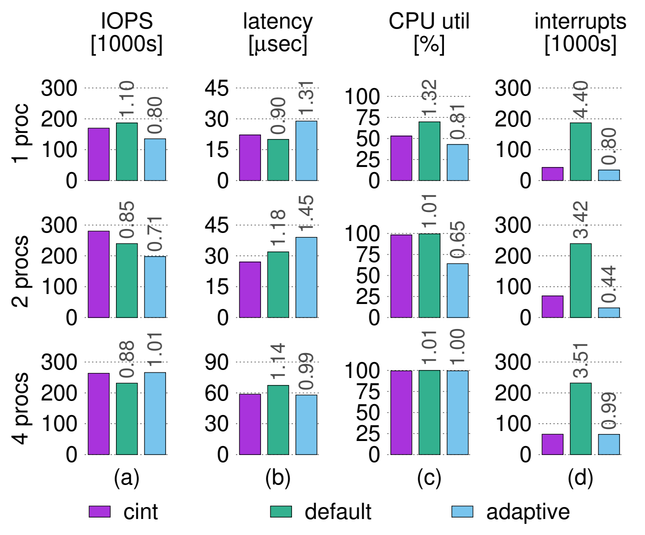


Figure 7: Effect of Barrier. Cinterrupts always detects the end of a batch with Barrier. Each process submits four requests per batch. Labels show performance relative to cinterrupts.

CPU utilization, while enabling the device to generate enough interrupts so that the application is not blocked.

To demonstrate the effectiveness of Barrier, we run an experiment in fio that generates a completion timeline similar to that found in Figure 4. In the experiment, we run 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 7 shows the throughput, latency, CPU utilization, and interrupt rate for the experiment.

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 4.4x the number of interrupts as cinterrupts, which results in 1.32x CPU utilization. Figure 4 also shows that the default strategy can 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.

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

Figure 8 shows that the adaptive strategy exhibits highest synchronous latency due to CPU idleness, which comes from waiting for completions and the delay built into the algorithm to detect the end of the batch. This idleness is eliminated in

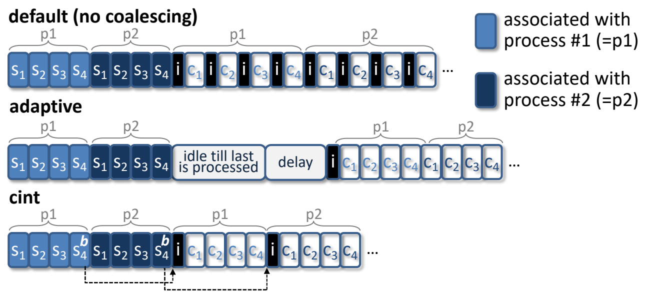

default (no coalescing)
adaptive
cint
p1
p2
idle till last is processed
delay
associated with process #1 (=p1)
associated with process #2 (=p2)

Figure 8: 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.

Algorithm 2: cinterrupts coalescing strategy

```
Parameters: $\Delta$, $thr$
coalesced = 0, timeout = now + $\Delta$;
while true do
    while now < timeout do
        while new completion arrival do
            timeout = now + $\Delta$;
            if completion type == Urgent then
                if ooo processing is enabled then
                    /* only urgent requests */
                    fire urgent IRQ;
                else
                    /* process all requests */
                    fire IRQ and reset coalesced;
            if completion type == Barrier then
                fire IRQ and reset coalesced;
            else
                if ++coalesced $\geq$ thr then
                    fire IRQ and reset coalesced;
    /* end of quiescent period */
    if coalesced > 0 then
        fire IRQ and reset coalesced;
    timeout = now + $\Delta$;
```

the next experiment, because there are enough threads to 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 §3.3 that this aggregation comes at the expense of synchronous requests.

Note that Figure 8 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.

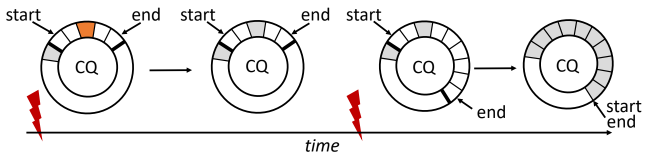


Figure 9: OOO Urgent processing. Grayed entries are reaped entries. Urgent requests in an interrupt context (first interrupt) are processed immediately, and the interrupt handler returns. The other requests are not reaped until the next interrupt, which consists only of non-Urgent requests. After the second IRQ, the driver rings the completion queue doorbell to signal that the device can reclaim the contiguous range.

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.

When running the full cinterrupts algorithm, we noticed that Urgent requests sometimes get completed with other requests, which increases the latency of the Urgent requests because the userspace thread is forced to block until the driver reaps all requests in the batch.

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 current interrupt context, returning immediately after these requests are reaped. This enables faster unblocking of the userspace thread that was waiting for the Urgent requests. The interrupt handler leaves the remaining requests for the next context, as shown in Figure 9. After reaping them, the IRQ handler marks OOO Urgent requests with a special flag so they are ignored by future interrupt contexts.

Unmarked requests will not be reaped until a completion batch consists only of those requests. 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. For example, suppose in Figure 9 that $thr = 9$. Then an interrupt will fire as soon as 9 entries (already reaped or otherwise) accumulate in the completion queue.

The trade-off with OOO processing is an increase in the number of interrupts generated. In Figure 10 we report performance metrics from running the same mixed workload as in Figure 6. OOO processing generates 2.4x 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 10, we throttle the asynchronous thread with the blkio cgroup [3] to its through-

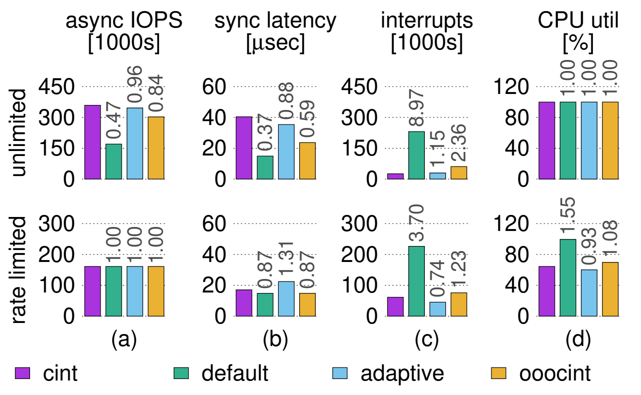


Figure 10: Mixed workload. Out-of-order (OOO) driver processing of Urgent requests enables lower latency at the expense of more interrupts. Limiting the number of async requests (bottom row) reduces this overhead.

put 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

In this section, we describe the hardware and software changes necessary to support cinterrupts.

4.1 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 IO execution.

4.1.1 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 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.

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" [6]. Unfortunately, neither device we inspected [10, 15] respected this aspect of the specification.

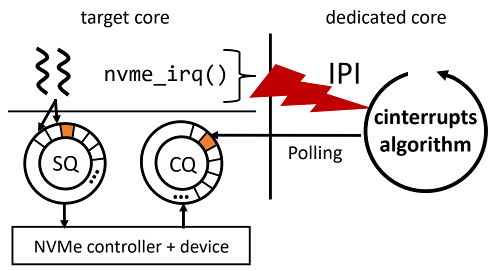


Figure 11: Cinterrupts emulation: dedicated core sends IPIs, which emulate hardware interrupts of a real device that supports cinterrupts.

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. 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 [6].

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 11 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.

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.

4.2 Software modifications

4.2.1 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, based

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
fdatasync, fsync, sync, syncfs	Urgent
msync	With `MS_SYNC`, Barrier on the last request

Table 1: Summary of storage I/O system calls and the corresponding default bits used by the kernel.

on the programming paradigm of each system call. Generally, 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. 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 cover such cases in Section 4.2.2.

4.2.2 Application case studies

Ultimately the application has the best knowledge of when it requires interrupts, so we also enable the application to override kernel defaults for even better performance. We leverage the preadv2/preadw2 system call interface [8], 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 now describe how we modified RocksDB to use these flags.

RocksDB background tasks. Flushing and compaction are the two main sources of background IO in RocksDB. We modify RocksDB to explicitly mark these IO 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 IO for that file, and modifying the Flush and Compaction jobs to explicitly mark IO 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 IO as non-Urgent. In this way, dump IO requests are completed when interrupts are generated by the underlying adaptive coalescing strategy. On top of the RocksDB changes described in the previous section, 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%.

4.3 Discussion

Other IO requests. We initially did not annotate requests generated by the kernel itself, for example from page cache writeback and filesystem journalling. But because filesystem journalling 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 journalling 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. There are multiple ways to implement cinterrupts in hardware. For example, a hardware implementation could enforce a stricter Barrier ordering, only releasing a Barrier interrupt if all requests in front of it in the submission queue have been completed. This strict ordering can even be enforced in the kernel: even if the driver receives completion notifications through device interrupts, it can withhold the completions from userspace until all other requests have completed. The cinterrupts implementation in this paper shows that even with a relaxed implementation of Barrier, which uses the timeout from the adaptive strategy to cleanup, cinterrupts enjoys significant performance benefits.

We explored a preliminary software implementation of the strict Barrier, but its overheads were larger than its benefit. We suspect firmware implementations of a strict Barrier will be more efficient.

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 target a fixed interrupt rate, similar to NICs, enforced with a lightweight heuristic based on Exponential Weighted Moving Average (EWMA) of the interrupt rate.

5 Evaluation

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

	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	$10_{\pm 1.3}$	$11_{\pm 1.3}$	$10_{\pm 1.2}$	$8_{\pm 1.2}$

Table 2: Emulation overhead is comparable with overhead of mitigations. 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 [13] and Intel Optane DC P4800X, 375 GB [16]. 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 [25] 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 §4.1.1, 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 [4]. The overhead of emulation is comparable with the overhead of the default mitigations for CPU. We disabled mitigations in our server in order to have performance numbers close to a real server with default mitigations enabled and to inspect the performance of cinterrupts in future architectures that do not require software-level mitigation [27].

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.

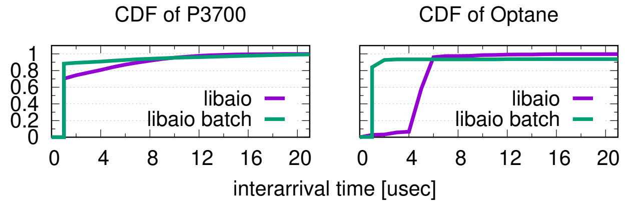


Figure 12: Using interarrival CDF to determine Δ.

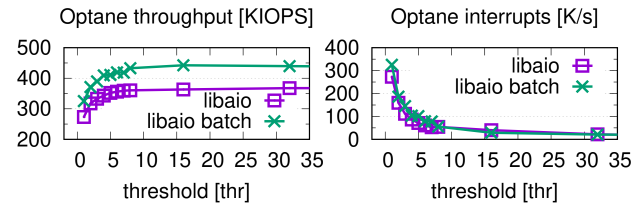


Figure 13: Determining thr under a fixed Δ (Δ=6 μs for Optane and $\Delta = 15$ μs for P3700). thr is the smallest value where throughput plateaus, which is between 16-32, so we set thr= 32 for both devices. We omitted P3700 thr results as it shows virtually the same behavior.

The adaptive strategy is a proxy for comparison to NVMe coalescing, because it has better semantics than NVMe coalescing, as described in Section 3.1.

5.2 Selection of Δ and thr

Δ should approximate the interarrival time of requests, which depends on workload. Figure 12 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 two NVMe devices, P3700 and Optane, to show that sysadmin will pick different Δ for different devices.

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, thr is straightforward to select. For our devices, we sweep thr in the $[0, 256)$ range and select the lowest thr after which throughput plateaus; the results are shown in Figure 13. 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.

5.3 Microbenchmarks

We use fio to generate two workloads to show how cinterrupts behaves at the extremes. The synchronous-only workload

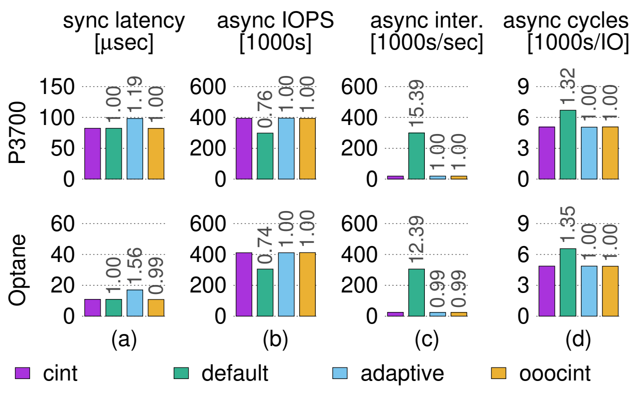


Figure 14: Workloads with only one type of request. Column (a) shows latency of synchronous requests (lower is better); (b), (c) and (d) show metrics for the asynchronous workload.

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 14.

As in §3.3, the synchronous workload shows the drawback of the adaptive strategy, which adds exactly Δ=15 *µ*s to request latency for P3700 and Δ=6 *µ*s for Optane (first column of Figure 14). 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 12-15x the number of interrupts as cinterrupts.

In summary, cinterrupts matches the synchronous latency of default, while achieving up to 35% more asynchronous throughput, and matches the asynchronous throughput of the adaptive strategy while achieving up to 36% lower latency. Finally, we note that OOO does not add overhead to cinterrupts performance when it is not triggered.

5.4 Macrobenchmarks

To evaluate the effect of cinterrupts on real applications, we run three application setups on Optane: RocksDB [9], KVell [53], 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 IO. KVell uses default kernel annotations (Barrier) while we will note when RocksDB uses default annotations or the modified annotations described in Section 4.2.2.

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

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

Table 3: Modifying RocksDB with annotations that make the flush non-urgent (app-cint) fillbatch performance with application-modified annotations.

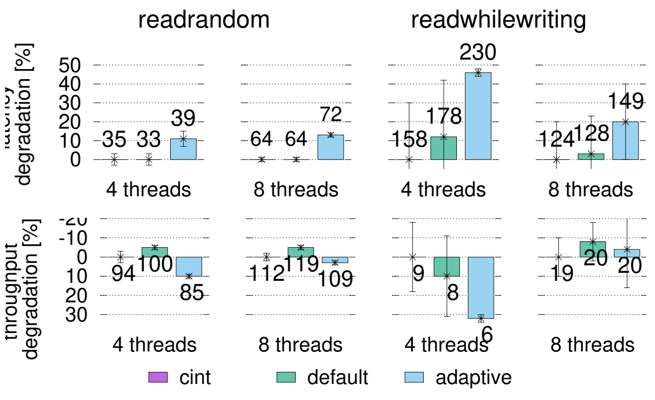


Figure 15: Latency of get operation and throughput in RocksDB for varying workloads. We show performance degradation with respect to cinterrupts. Labels show absolute values in *µ*s and KIOPS, respectively. As expected, cinterrupts and the default strategy have nearly the same performance (within error bounds), but the adaptive strategy has up to 45% worse latency and 5-32% worse throughput due to the Δ delay.

5.4.1 RocksDB

Load. Using db_bench [2], 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 IO is labelled Urgent by default. In modified RocksDB, background activity is non-Urgent as described in Section 4.2.2. 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 IO 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 20GB, we run two experiments from db_bench: readrandom, where each thread reads randomly from the key space, and readwhilewriting, where one 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 15.

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

Table 4: Summary of YCSB workloads.

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 8 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 IO in the readrandom benchmark, and the write rate is not high enough for the background IO interrupts to affect foreground performance in the readwhilewriting benchmark.

5.4.2 KVell

We use workloads derived from the YCSB benchmark [33], 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 16 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 (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 add to latency variability, which is noticeable in the larger 99th percentile latencies.

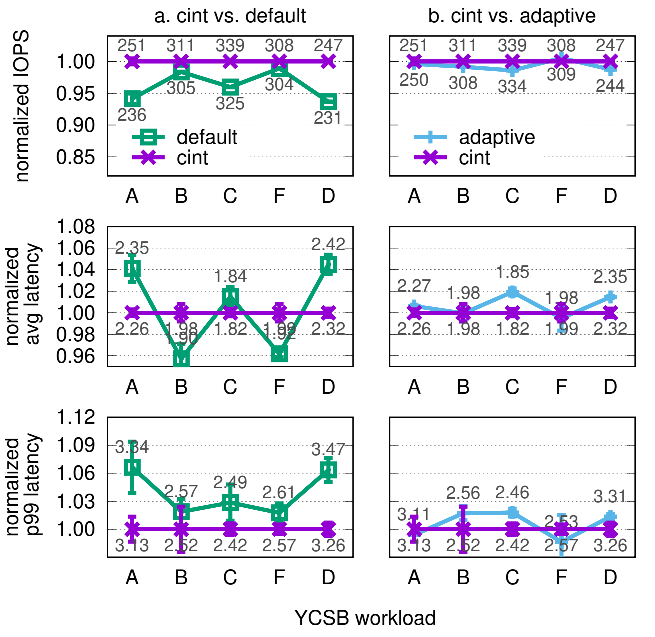


Figure 16: Throughput and latency results for YCSB on KVell. Labels show absolute throughput in KIOPS and latency in ms.

	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

Table 5: YCSB-E throughput results for KVell. Excessive interrupt generation limits default throughput to 86%-89% of cinterrupts'.

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 17 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, which is why the benefit of cinterrupts on throughput reduces for higher scan lengths.

Figure 17 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 400μs 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

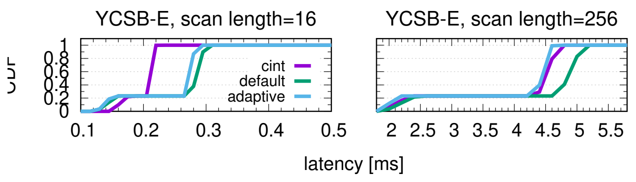


Figure 17: Latency CDF of scans of length 16 and 256 in KVell.

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

Table 6: Performance of RocksDB readrandom while the RocksDB dump tool is running in the background. app-cint modifies the dump tool to mark its IO 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.

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.

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 IO 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 4.2.2, we modify the RocksDB dump tool to explicitly disable the Urgent bit on its IO 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 can 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 IO 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 tradeoff for the foreground improvements.

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

Table 7: Results from colocated experiment: 4 RocksDB threads and KVell. As expected, cinterrupts both has lower latency than the adaptive strategy and higher throughput than the baseline.

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

Table 8: Results from colocated experiment: 8 RocksDB threads and KVell. 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.

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 Conclusion

In this paper 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, generates interrupts exactly when a workload needs them, 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.

References

[1] Administration and data access tool. https://github.com/facebook/rocksdb/wiki/Administration-and-Data-Access-Tool. Accessed: December, 2020.

[2] Benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed: December, 2020.

[3] Block IO Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt. Accessed: December, 2020.

[4] Hardware vulnerabilities, The Linux kernel user's and administrator's guide. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/index.html. Accessed: December, 2020.

[5] Intel Optane Technology for Data Centers. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html. Accessed: December, 2020.

[6] NVM Express, Revision 1.3. https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf. Accessed: December, 2020.

[7] NVM Express, Revision 1.4, Figure 284. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf. Accessed: December, 2020.

[8] preadv2(2) — linux manual page. https://man7.org/linux/man-pages/man2/preadv2.2.html. Accessed: December, 2020.

[9] RocksDB. https://github.com/facebook/rocksdb. Accessed: December, 2020.

[10] Ultrastar DC SN200. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-dc-ha200-series/data-sheet-ultrastar-dc-sn200.pdf. Accessed: December, 2020.

[11] 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: December 2020.

[12] 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: December, 2020.

[13] 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: December, 2020.

[14] Intel Ethernet Converged Network Adapter X550. https://ark.intel.com/content/www/us/en/ark/products/88209/intel-ethernet-converged-network-adapter-x550-t2.html, 2016. Accessed: December, 2020.

[15] Intel Optane SSD 900P Series. https://ark.intel.com/content/www/us/en/ark/products/123626/intel-optane-ssd-900p-series-480gb-1-2-height-pcie-x4-20nm-3d-xpoint.html, 2017. Accessed: December, 2020.

[16] Intel Optane SSD DC P4800X Series. https://ark.intel.com/content/www/us/en/ark/products/97162/intel-optane-ssd-dc-p4800x-series-375gb-1-2-height-pcie-x4-3d-xpoint.html, 2017. Accessed: December, 2020.

[17] Mellanox ConnectX-5 VPI Adapter. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf, 2018. Accessed: December, 2020.

[18] Intel 82599 10 GbE Controller Datasheet. https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html, 2019. Accessed: December, 2020.

[19] 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: December, 2020.

[20] Microsoft Documentation: Optimize performance on the Lsv2-series virtual machines. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/storage-performance, 2019. Accessed: December, 2020.

[21] Intel Ethernet Controller X540 Datasheet. http://www.intel.com/content/www/us/en/network-adapters/10-gigabit-network-adapters/ethernet-x540-datasheet.html, 2020. Accessed: December, 2020.

[22] Tuning Throughput Performance for Intel Ethernet Adapters. https://www.intel.com/content/www/us/en/support/articles/000005811/network-and-i-o/ethernet-products.html, 2020. Accessed: December, 2020.

[23] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (USENIX ATC'11)*, 2011.

[24] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[25] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio. Accessed: December, 2020.

[26] Jens Axboe. blk-mq: make the polling code adaptive. https://lkml.org/lkml/2016/11/3/548, 2016. Accessed: December 2020.

[27] Andrew Baumann. Hardware is the New Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, 2017.

[28] Pavel Begunkov. blk-mq: Adjust hybrid poll sleep time. https://lkml.org/lkml/2019/4/30/120, 2019. Accessed: December 2020.

[29] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[30] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, 2013.

[31] Keith Bush. Linux-NVME mailing list: nvme pci interrupt handling improvements. https://lore.kernel.org/linux-nvme/20191209175622.1964-1-kbusch@kernel.org/, 2019. Accessed: December, 2020.

[32] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *USENIX Annual Technical Conference (USENIX ATC'20)*, 2020.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010.

[34] Jonathan Corbet. Batch processing of network packets. `https://lwn.net/Articles/763056/`. Accessed: December, 2020.

[35] Jonathan Corbet. Driver porting: Network drivers. `https://lwn.net/Articles/30107/`. Accessed: December, 2020.

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

[37] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems (ESSoS'17)*. Springer, 2017.

[38] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O Suminto, Cesar A Stuardo, Andrew A Chien, and Haryadi S Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[39] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[40] Intel. DPDK: Data Plane Development Kit. `https://www.dpdk.org`, 2014. Accessed: December, 2020.

[41] Rick A. Jones. Netperf: A Network Performance Benchmark. `https://github.com/HewlettPackard/netperf`, 1995. Accessed: December, 2020.

[42] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[43] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing array of SSDs when the storage device is no longer the performance bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[44] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[45] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O path: a holistic approach for application performance. In *15th USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[46] Avi Kivity. Wasted processing time due to nvme interrupts. `https://github.com/scylladb/seastar/issues/507`, 2018. Accessed: December, 2020.

[47] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency SSDs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019.

[48] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[49] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*, 2019.

[50] Damien Le Moal. I/O Latency Optimization with Polling. *Linux Storage and Filesystems Conference (VAULT'17)*, 2017.

[51] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency ssds. In *USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.

[52] Ming Lei. 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: December, 2020.

[53] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[54] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[55] Long Li. LKML: fix interrupt swamp in NVMe. `https://lkml.org/lkml/2019/8/20/45`, 2019. Accessed: December, 2020.

[56] Kan Liang, Andi Kleen, and Jesse Brandenburg. Improve Network Performance by Setting per-Queue Interrupt Moderation in Linux. `https://01.org/linux-interrupt-moderation`, 2017. Accessed: December, 2020.

[57] Merriam-Webster. "Calibrate". `https://www.merriam-webster.com/dictionary/calibrate`, 2020. Accessed: December, 2020.

[58] Jeffrey C. Mogul and Kadangode K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC'96)*, 1996.

[59] Rikin J Nayak and Jaiminkumar B Chavda. 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 2017)*, 2017.

[60] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, 2016.

[61] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[62] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y Yeom. OS I/O path optimizations for Flash Solid-State Drives. In *USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.

[63] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, volume 10, 2010.

[64] Steven Swanson and Adrian M Caulfield. Refactor, reduce, recycle: Restructuring the io stack for the future of storage. *Computer*, 46(8):52–59, 2013.

[65] Dan Tsafrir. The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops). In *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007.

[66] John Uffenbeck et al. *The 80x86 family: design, programming, and interfacing*. Prentice Hall PTR, 1997.

[67] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*. ACM, 2015.

[68] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[69] Ting Yang, Tongping Liu, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Redline: First class support for interactivity in commodity operating systems. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[70] Tom Yates. Improvements to the block layer. `https://lwn.net/Articles/735275/`. Accessed: December, 2020.

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

[72] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[73] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.