TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10

Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal, *Cornell University*

https://www.usenix.org/conference/nsdi20/presentation/hwang

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10

Jaehyun Hwang Qizhe Cai Ao Tang Rachit Agarwal
Cornell University

Abstract

This paper presents design, implementation and evaluation of i10, a new remote storage stack implemented entirely in the kernel. i10 runs on commodity hardware, allows unmodified applications to operate directly on kernel's TCP/IP network stack, and yet, saturates a 100Gbps link for remote accesses using CPU utilization similar to state-of-the-art user-space and RDMA-based solutions.

1 Introduction

The landscape of cloud infrastructure has changed rapidly over the last few years. Two trends stand out:

- First, network and storage hardware has improved significantly, *e.g.*, network access link bandwidth has transitioned from 1Gbps to 40Gbps or even 100Gbps [4, 37]; and, fast non-volatile memory express (NVMe) storage devices that deliver more than a million input/output operations per second (IOPS) are being widely deployed [15, 23].
- Second, the need for fine-grained resource elasticity and high resource utilization has led to large-scale deployments of disaggregated storage [40, 43]; also see [9, 11, 12, 23, 24]. As a result, increasingly more applications now access storage devices over the network.

These changes have shifted performance bottlenecks back to the software stack — while network and storage hardware is able to sustain high throughput, traditional remote storage access stacks (that make remote storage devices available as local block devices, *e.g.*, iSCSI [35] and light-weight servers based on Linux) have unsustainable CPU overheads. For instance, traditional iSCSI protocol is known to achieve merely 70K IOPS per CPU core due to its high protocol processing and synchronization overheads [12, 15, 23, 24]. While this was not a bottleneck for slower storage devices and/or networks, saturating a single modern NVMe storage device now requires 14 cores, and saturating a 100Gbps link now requires 40 cores!

Responding to this challenge, both academic and industrial communities have been taking a fresh look at the problem of designing CPU-efficient remote storage stacks. Recently standardized NVMe-over-Fabrics (NVMe-oF), specifically, NVMe-over-RDMA [3, 30] keeps the kernel storage stack, but moves the network stack to the hardware. A complementary approach argues for moving the entire storage and network stack to the user space [24]. These proposals can achieve high performance in terms of IOPS per core, but require changes in applications and/or network infrastructure; such changes would be acceptable, if utmost necessary.

This paper explores a basic question: *"are infrastructure changes really necessary for efficient remote storage access"?* Exploring this question may help clarify our response to the challenges introduced by the two cloud infrastructure trends discussed above. An affirmative answer would make a strong argument for user-space stacks and/or specialized network hardware. However, if performance similar to above solutions can be achieved by re-architecting the kernel, then it changes the lens through which we view the design and adoption of software stacks in future: for example, rather than every organization asking *whether* to perform a ground-up redesign of their infrastructure, organizations that are already adopting user-space stacks and/or modern network hardware could ask *how* to port kernel remote storage stacks to integrate with their infrastructure. Thus, while the rest of the paper occasionally descends into kernel minutiae, the question we are asking has important practical implications.

Our exploration of the above question led to design and implementation of i10, a new *remote* storage stack within the kernel. i10 demonstrates that, at least for applications that are throughput bound, performance similar to state-of-the-art user-space and RDMA-based solutions can be achieved with minimal modifications in the kernel. i10 offers a number of benefits. First, i10 requires no modifications outside the kernel; thus, existing applications can operate directly on top of i10 without any modifications, whatsoever. Second, i10 operates directly on top of existing TCP/IP kernel protocol stack; thus, it requires no modifications in the network stack and/or hardware. Third, i10 complies with recently standardized NVMe-oF specification [3]; thus i10 can work with all emerging NVMe devices. Finally, over benchmark workloads, i10 saturates a 100Gbps link using a commodity server with CPU utilization similar to state-of-the-art user-space stacks and NVMe-over-RDMA products [24, 30]. The last benefit is perhaps the most interesting one as it fills a gaping hole in our understanding of kernel bottlenecks: as we discuss in Figure 1 and §4, existing bottlenecks for remote storage access are neither in the storage stack nor in the network stack; rather, the inefficiency lies at the boundary of the two stacks!

i10 achieves the above benefits using a surprisingly simple design that integrates two ideas (§2):

End-to-end dedicated resources and batching. Using dedicated resources and batching to optimize the software stack is a well-known technique, also used in CPU-efficient network

stacks [6, 13, 19, 29]; however, as we will show, performance bottlenecks being at the boundary of storage and network stacks means that prior solutions of dedicating resources at per-core granularity and batching packets at socket level are no longer sufficient. i10 dedicates resources at the granularity of an i10-ℓane, creating a highly optimized pipe between each (core, target) pair, where target refers to the storage stack at the remote server (and not necessarily individual storage devices at the remote server). i10-ℓanes have the property that all control and data packets in any queue are destined to the same destination, and are to be transmitted on the same TCP session; thus, they can be efficiently batched at the entrance of the pipe to reduce network processing overheads (§2).

Delayed Doorbells. When accessing a local NVMe device, existing storage stacks "ring the doorbell" (signal the storage device about a new request) immediately upon receiving the request. i10 observes that while this is efficient for relatively low-overhead communication over PCI express, immediately ringing the doorbell leads to high context switching overheads for remote accesses where requests traverse over a high-overhead network stack and network fabric. i10, thus, introduces the idea of "delayed doorbells", where the worker thread in the storage stack delays ringing the doorbell until multiple requests are processed (or a timeout event happens). i10 also shows how the granularity of dedicated resources and batching in i10 interplays well with delayed ringing of doorbells to achieve the final performance (§2, §4).

i10 design simplicity also enables i10 implementation in Linux with modest modifications, while operating directly on an unmodified TCP/IP stack over commodity hardware[1]. i10 evaluation, over both benchmark workloads and real applications, demonstrates that i10 achieves throughput-per-core comparable to NVMe-over-RDMA [30]. Compared to state-of-the-art in-kernel remote storage stacks like NVMe-over-TCP [26], which was incorporated within the Linux kernel in March 2019, i10 both enables new operating points (*e.g.*, being able to saturate 100Gbps links) and also reduces the CPU utilization by 2.5× for already feasible operating points. Moreover, i10 maintains these benefits for all evaluated workloads including varying read/write ratios, request sizes and device types. Finally, i10 scales well with number of cores and with number of remote storage devices.

Going back to our starting point, i10 answers the original question about the necessity of application and/or network infrastructure changes for throughput-bound applications. Of course, this is already a large class of applications; however, batching used in i10 leads to the same latency-throughput tradeoff as in CPU-efficient network stacks [6, 13, 19, 29]: at low loads, latencies may be high (albeit, still at hundred-microsecond granularity and within 1.7× of NVMe-over-RDMA latency over storage devices). While not completely satisfying, our exploration has led us to believe that user-space stacks and RDMA-enabled solutions may be more useful for applications demanding absolutely minimal latency for each individual request. The question, however, remains open for such applications.

[1] i10 implementation, along with documentation that enables reproducing all results, is available at https://github.com/i10-kernel/.

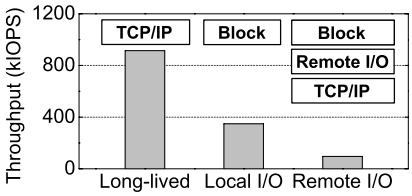


Figure 1: **Existing bottlenecks for remote storage access are at the boundary of the storage and network stacks.** The figure shows throughput-per-core for kernel network, local storage and remote storage stacks. We use 4KB random read requests between two servers with NVMe solid state drives connected via a 100Gbps link (detailed setup in §4). For long-lived flow, the network stack can sustain as much as ∼30Gbps (roughly 915 kIOPS) per core using well-known optimizations (*e.g.*, TCP segmentation offload (TSO) and generic receive offload (GRO)); similarly, the storage stack for local I/O can sustain ∼350 kIOPS per core. However, when integrated together with existing remote storage stacks, the achievable throughput reduces to 96 kIOPS (§4).

2 i10 Design

In this section, we describes the i10 design. We start with an overview (§2.1), followed by a detailed description of how i10 creates highly optimized i10-ℓanes using dedicated resources (§2.2), batching (§2.3) and delayed doorbells (§2.4). The next section provides some of the interesting implementation details for i10.

2.1 i10 Design Overview

i10 is designed and implemented as a shim layer between the kernel storage stack (specifically, the block device layer) and the kernel network stack. Figure 2 shows the high-level design of i10, including the i10 layer and end-to-end path between the host application and the target NVMe storage device. i10 does not control how applications are scheduled on the cores; each application may run on one or more cores, and multiple applications may share the same core. Applications submit remote read/write requests to the kernel through the standard read/write APIs; i10 requires no modifications to these APIs.

i10 achieves its goals using the core abstraction of an i10-ℓane— a dedicated pipe that is used to exchange both control and data plane messages along a set of dedicated resources. i10 creates i10-ℓanes and dedicates resources to each i10-ℓane at the granularity of (core, target)-pair, where target refers to the block device at the remote server (and not necessarily individual storage devices). For example, consider that (potentially more than one application running at) a core c submits read/write requests to two target servers `t1` and `t2`, each having multiple storage devices `d11, d12, ...` and `d21, d22, ...`. Then, i10 creates two i10-ℓanes, one `c` $\rightsquigarrow$ `t1` for all

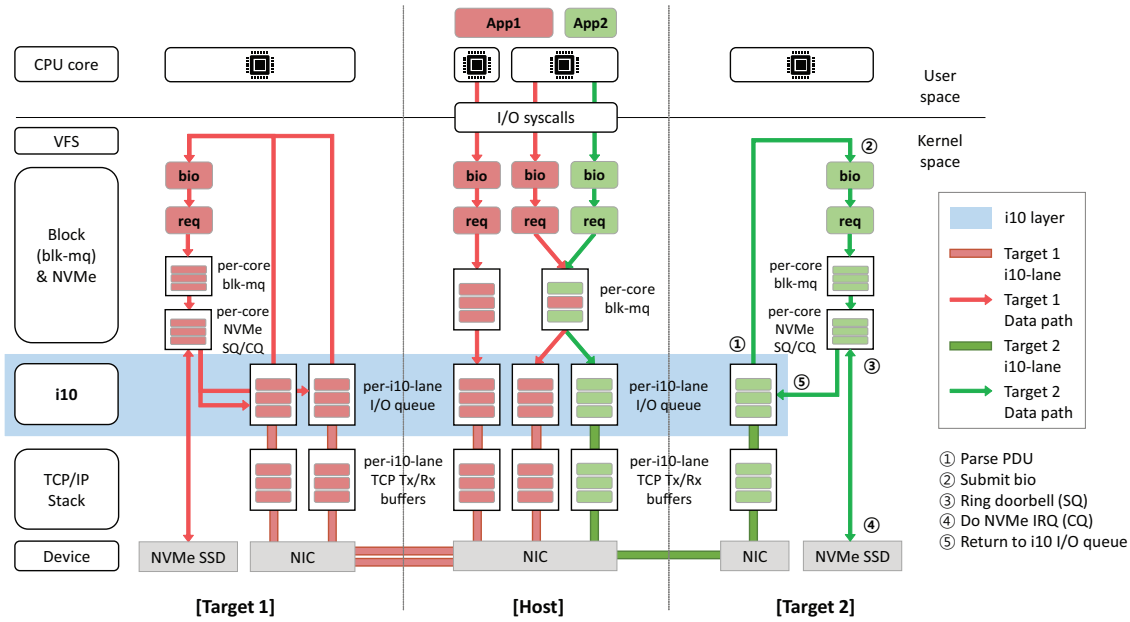


Figure 2: **End-to-end data path in i10 between one host and two target servers.** In this example, `app1` is sending read/write requests to `target1` using both core1 and core2, and `app2` is sending read/write requests to `target2` using core2. Thus, i10 creates three i10-ℓanes — one for each of (core1, target1), (core2, target1) and (core2, target2) pairs. Moreover, (green) requests submitted from `app2` are copied to the block layer request queue of core2 and (red) requests submitted from `app1` are copied to either the block layer request queue of core1 or core2 depending on which core the request comes from. While block layer request queues may contain requests going to different target servers (*e.g.*, the right block layer request queue at `host1`), each request is copied to the i10 I/O queue for i10-ℓane corresponding to the (core, target)-pair. Finally, if there were an `app3` running on `core2` sending requests to `target2`, it would completely share the i10-ℓane with `app2`.

requests going to the former set of storage devices and the other `c` $\rightsquigarrow$ `t2` for all requests going to the latter set of storage devices. Note that this is independent of the number of applications running on the core `c`. Moreover, if a single application running on two cores `c1 and c2` submits read/write requests to two target servers `t1` and `t2`, then i10 will create four i10-ℓanes, one for each (core, target) pair.

i10 uses three set of dedicated resources for each i10-ℓane (both at the host and at the target side). We first describe these dedicated resources and then discuss how they integrate into an end-to-end path between the host core and the target. The first dedicated resource is an I/O queue in the i10 layer (shown in the blue horizontal bar in Figure 2). The second is a dedicated TCP connection, along with its buffers, between the host and target i10 queues. Finally, a dedicated i10 worker thread for each core that interacts with i10 at the host side and for each core that is needed at the target side. Note that, for reasons that we will discuss in §2.2, i10 queues and TCP connections are at the per-lane granularity, and the i10 worker threads are at a per-core granularity.

We are now ready to describe the end-to-end path between the host core and the target. Upon receiving a request from a core, the block layer does its usual operations — generates a `bio` instance (that represents an in-flight block I/O operation in the kernel [28]), initializes the corresponding `request` instance using Linux kernel's support for multiple per-core block queues (`blk-mq`) [7, 14] and then, copies the request to the block layer's request queue for that core (these request queues are different from i10 queues). Finally, the block layer's `request` instance is converted to an i10 request; to be compliant with NVMe-oF standards, i10 requests are similar to a command Protocol Data Unit (PDU) [3]. Finally, using the context information within the block layer's request data structure, i10 requests are copied to the i10 queue for the corresponding i10-ℓane.

Having a dedicated queue for each i10-ℓane implies that all requests and data packets in a queue are destined to the same target server, and will be transmitted over the same TCP connection. Thus, i10 is able to batch multiple requests and data packets into i10 "caravans"[2], all to be processed and transmitted over the same TCP connection. This allows i10 to significantly reduce the network processing overheads by aggregating enough data to benefit from well-known optimizations like TSO and GRO. We discuss the precise details in §2.3.

i10 observes that the original NVMe specification was designed for accessing storage devices over PCI express (PCIe). Since PCIe provides a low-latency low-overhead communication between the storage stack and the local storage devices,

[2]i10 "caravans" are nothing but batches of requests; we use the term caravans to avoid confusion between i10 batches of requests and traditional batches — while traditional batches correspond to packets going to the same application port, i10 batches may be going to different storage devices, albeit within the same target server.

it was useful for the case of local access to "ring the doorbell" (provide a signal to the storage device that a new I/O request is ready to be served) immediately upon creating a request. However, in the case of remote accesses where requests traverse through a relatively high-latency high-overhead network, immediately ringing the doorbell leads to high context switching overheads for the worker threads. To alleviate these overheads, i10 introduces the idea of delayed doorbells, where the block layer worker thread processes multiple requests (or times out) before ringing the doorbell to wake up the i10 worker thread. This not only reduces the context switching overheads significantly, but also provides i10-ℓane queues with enough requests/data to generate right-sized caravans. We describe the precise mechanism in §2.4.

Finally, the i10 caravan is transmitted through the in-kernel socket interface. As shown in Figure 2, when the caravan arrives in the target-side i10 queue, i10 parses the caravan to regenerate the `bio` instances, corresponding requests and submits them to the block layer. Upon receiving the requests, the block layer executes the same steps as it does for accessing PCIe-connected local storage devices: the request is inserted to the NVMe submission queue and upon completion, the result is returned to the NVMe completion queue. After the local access, the result goes back to the block device layer, and finally is abstracted as a response caravan by i10 and sent back to the host server over the TCP connection.

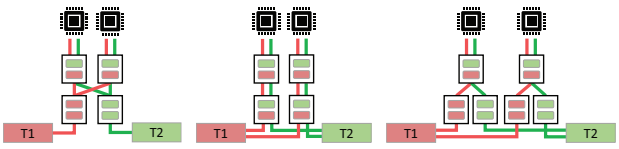
In the following three subsections, we present design details for the three building blocks of i10: i10-ℓane, i10 caravans, and delayed doorbells.

2.2 i10-ℓane

The two obvious options for creating i10-ℓane are (1) creating one i10-ℓane per target server, independent of the number of cores (Figure 3(a)); and (2) creating one i10-ℓane per core, independent of the number of target servers (Figure 3(b)). At high loads, the first option leads to high write contention among the block layer worker threads since they will need to write the `requests` to the same i10 queue. The second option is no better either — here, requests destined to different targets are forced to be in the same i10 queue resulting in preventing i10 caravans from batching enough requests, or high CPU overheads (for sorting requests to batch into the same caravans). Both these overheads become worse in the most interesting case of high-load regime. i10 avoids these overheads by creating an i10-ℓane for each (core, target)-pair (Figure 3(c)). That is, for applications that use P host cores to access data at T target servers, i10 creates $P \times T$ i10-ℓanes, independent of the number of storage devices at each target.

We now describe the resources dedicated to each i10-ℓane.

`blk-mq` level request queues. i10 exploits per-core request queue defined in the block layer (using `blk-mq` [7, 14]). Before the support for `blk-mq`, all block layer requests went to a single request queue per device. While queueing requests



(a) per-target (b) per-core (c) per-target/per-core

Figure 3: **Creating i10-ℓane for each (core, target) pair is the right design.** The figure shows host cores, `blk-mq`, i10 queues, TCP connections, and target devices T1 and T2. For discussion, see §2.2.

in a single queue could create head-of-line blocking, this design enabled scheduling so as to minimize the seek time on hard disks. For modern storage devices that support high-throughput random reads and writes using multiple cores, the equation is quite different due to two reasons: (1) multiple cores operating over a single queue becomes a performance bottleneck; and, (2) since seek time is not a problem, minimizing head-of-line blocking becomes more important. Thus, recent versions of Linux kernel enable the block layer to create per-core request queues and to maintain multi-queue context information for both `blk-mq` and underlying remote access layers. This enables i10 to efficiently demultiplex requests in `blk-mq` into individual i10-ℓane queues.

i10 I/O queue. i10 creates one dedicated queue for each individual i10-ℓane. These queues are equivalent to the I/O queues from the NVMe standard [2] with the only difference that they communicate with a remote target server, not with local SSD devices. Once the requests from `blk-mq` are converted to NVMe-compatible command PDUs, they are inserted to i10 queues. The NVMe standard allows creating as many as 64K NVMe queues to enable parallel I/O processing; since we expect i10 to have no more than 64K simultaneously active i10-ℓanes at any server for most deployments, i10 design should not be limited by the number of available queues.

TCP socket instance. Each i10-ℓane maintains its own TCP socket instance and establishes a long lived TCP connection with the target, along with corresponding TCP buffers. The state needed to be maintained in the kernel for each individual TCP connection is already quite small. The only additional state that i10 requires is the mapping between the TCP connections and the corresponding i10-ℓane, which again turns out to be small (§5).

i10 worker thread. i10 creates a dedicated per-core worker thread whose responsibility is to conceptually move i10 caravans bidirectionally on i10-ℓanes. This worker thread starts executing when a doorbell is rung for any of the i10-ℓanes on the same core, and aggregates command PDUs in the queue of the corresponding i10-ℓane into caravans. The worker then moves the caravans to TCP buffers for the corresponding i10-ℓane. Finally, the worker thread goes into the sleep mode until a new doorbell is rung.

When the caravan reaches the target's TCP receive buffers,

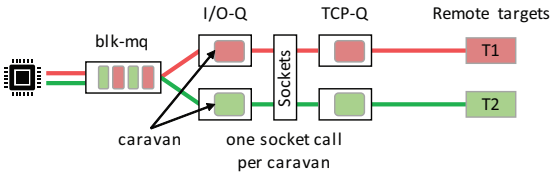


Figure 4: **Creating i10 caravans at the i10 queues is the right design for reducing per-request network processing overhead**. See discussion in §2.3.

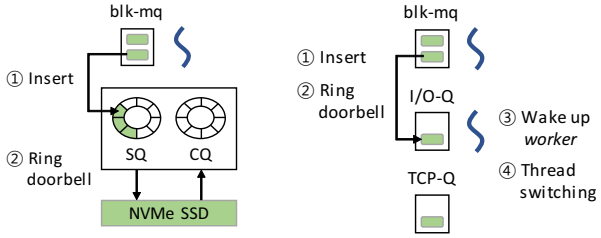
the corresponding worker thread starts processing the requests in the caravan. First, it regenerates `bio` for each request in the caravan, followed by processing the requests as needed at the block layer. Upon receiving the signal from completion processing, the results are inserted into target's i10 queue and a caravan is created. It is not necessary for response caravans to have the same set of requests as in host caravans, because caravan's size can be different between host and target (§2.3).

2.3 i10 Caravans

Given that all requests in an i10 queue are going to the same destination over the same TCP connection, i10 batches multiple requests into an i10 caravan. This allows i10 to benefit from standard optimizations like TSO and GRO, which significantly reduces the network processing overheads. Our key insight here is that i10 queues are precisely the place to create caravans because of two reasons. First, at the block layer, the `blk-mq` is per-core and at any given point of time, may have requests belonging to different targets (as in Figure 3(c)); thus, batching the requests at the block layer would require significant CPU processing to sort the requests going to the same target device. Second, batching at the TCP layer would require i10 to process each request individually to send to TCP buffers, thereby creating one event per request; prior work [19] has shown that such per-request events results in high CPU processing overheads. By batching at its own queues, i10 reduces both of these overheads (Figure 4). We set the maximum amount of data carried by a caravan to be 64KB to align it with the maximum packet size supported by TSO. However, to prevent a single caravan from batching too many small-sized requests, each caravan may batch no more than a pre-defined aggregation size number of requests (§3).

2.4 Delayed Doorbells

The original NVMe specification was designed for accessing storage devices over PCI express (PCIe). Even though the standard itself does not prescribe how to use doorbells, the current storage stack simply rings the doorbell (that is, updates the submission queue doorbell register) whenever a request is inserted into the NVMe submission queue (Figure 5(a)). Since PCIe provides a low-latency low-overhead communication between the storage stack and the local storage devices, ringing the doorbell on a per-request basis reaches the maximum throughput of the device with minimal latency. However, in the case of remote accesses where requests traverse



(a) Local I/O (b) Remote I/O

Figure 5: **Ringing the doorbell per request is effective for local PCIe-attached local storage, but not for remote storage access since the latter results in high context switching overhead.** See discussion in §2.4.

through a relatively high-latency high-overhead network, ringing the doorbell on a per-request basis results in high context switching overheads. In the specific context of i10, ringing the doorbell implies that as soon as the block layer thread inserts an i10 request to the i10 queue, it wakes up the i10 worker thread to handle the request immediately (Figure 5(b)). This incurs a context switch, which at high loads, could result in high CPU overheads (§4).

i10 alleviates these overheads using the idea of delayed doorbells. When an i10 queue is empty, a doorbell timer is set upon arrival of the first request. Then, the doorbell is rung either when the i10 queue has as many requests as a pre-defined aggregation size or when the timer reaches a timeout value, whichever happens earlier. Whenever the doorbell is rung, i10 caravans are created with all the requests in the i10 queue and the doorbell timer is unset. We note that delayed doorbells can be used independent of whether or not requests are batched into caravans. Moreover, this design will cause extra latency if applications generate low load (resulting in requests observing "timeout" amount of latency).

3 i10 Implementation Details

We implement i10 host and target in the Linux kernel 4.20.0. i10 implementation runs on commodity hardware (we do use the TSO and GRO features supported by most commodity NICs) and allows unmodified applications to operate directly on kernel's TCP/IP network stack. In this section, we discuss some interesting aspects of i10 implementation.

`kernel_sendpage()` vs. `kernel_sendmsg()`. There are two options to transmit i10 caravans via kernel socket interfaces. The first interface, `kernel_sendpage()` allows avoiding transmission-side data copy when sending each page of the data, but limits the aggregation size to be no more than 16. The second, `kernel_sendmsg()` takes a kernel I/O vector as a function argument and internally copies every scattered data of the I/O vector into a single socket buffer. This allows i10 aggregation size to be larger than 16, which leads to lower network processing overhead in some cases. Our tests reveal that `kernel_sendmsg()` achieves slightly better overall

CPU usage (that is, including data copy as well as network processing overheads), resulting in better overall throughput. Therefore, we use `kernel_sendmsg()` for i10 caravans to achieve better throughput.

i10 no-delay path. For latency-critical applications, it may be more important to avoid the latency incurred by i10 batching and delayed doorbells. For example, it may be desirable to execute a read request on file system metadata such as inode tables immediately upon submission so as to avoid blocking further read/write requests that cannot be executed without the response to the original request. For such cases, i10 supports a *no-delay path* — when such a latency-sensitive request arrives in i10 queue, i10 flushes all outstanding requests in the queue and processes the latency-sensitive request immediately. This is implemented using a simple check during the delayed doorbell ringing process: upon receiving a latency-sensitive request, the doorbell can be rung immediately, forcing i10 to create a caravan using all the outstanding requests along with the latency-sensitive request.

i10 parameters. In general, we expect throughput-per-core in i10 to improve with increasing aggregation size due to reduced network processing overheads. However, as we show in Appendix A in the technical report [16], increasing aggregation size beyond a certain threshold would result in marginal throughput improvements while requiring larger doorbell timeout values to be able to aggregate larger number of requests (thus, inflating per-request latency at low loads). This threshold — that is, the value that reaches the point of marginal improvements — of course, depends on kernel stack implementation. For our kernel implementation, we find 16 to be the best aggregation size with 50μs doorbell timeout value. We will use these parameters by default in our evaluation.

TCP buffer configuration. i10 caravans may be as large as 64KB. To this end, the TCP transmit buffer should have enough space for receiving caravans. However, TCP buffer size is generally adjusted by TCP's auto tuning mechanism — the Linux TCP implementation automatically increases the buffer size based on the bandwidth-delay product estimate for the transmit buffer, unless users specify a static buffer size via `setsockopt()`. Therefore, if the remaining transmit buffer is currently less than 64KB, the caravan would be fragmented even if kernel can provide more memory for the buffer resulting in higher processing overheads due to more than one socket call per caravan. For this reason, we explicitly use a fixed size of TCP buffers via `kernel_setsockopt()` at the session establishment stage. We set the buffer size to 8MB in this paper, which is sufficiently large to avoid caravan fragmentation.

4 i10 Evaluation

In this section, we evaluate an end-to-end implementation of i10. We first describe our evaluation setup (§4.1). We then start by evaluating i10 performance against state-of-the-art NVMe-over-RDMA (NVMe-RDMA) [30] and NVMe-over-TCP (NVMe-TCP) [26] implementations over a variety of settings including varying loads, varying number of cores, varying read/write request ratios and varying number of target servers (§4.2). Next, we use CPU profiling to perform a deep dive into understanding how different aspects of i10 design contribute to its performance gains (§4.3). Finally, we evaluate i10 over real applications (§4.4) and compare its performance with state-of-the-art user-space stacks (§4.5). Several additional evaluation results (including sensitivity analysis against aggregation size and doorbell timeout values, performance with variable request sizes, scalability with multiple applications sharing the same core, benefits of syscall batching, etc.) can be found in the technical report [16].

Table 1: **Experimental setup used in our evaluation.**

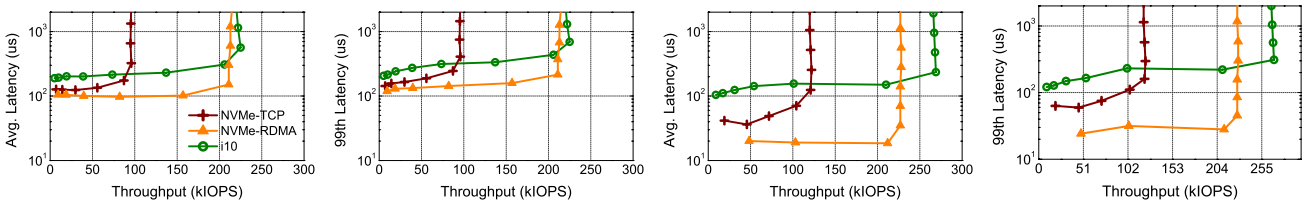
H/W configurations	
CPU	4-socket Intel Xeon Gold 6128 CPU @ 3.4GHz 6 cores per socket, NUMA enabled (4 nodes)
Memory	256GB of DRAM
NIC	Mellanox ConnectX-5 EX (100G) TSO/GRO=on, LRO=off, DIM disabled Jumbo frame enabled (9000B)
NVMe SSD	1.6TB of Samsung PM1725a
S/W configurations	
OS	Ubuntu 16.04 (kernel 4.20.0)
IRQ	`irqbalance` enabled
FIO	Block size=4KB, Direct I/O=on I/O engine=`libaio`, gtod_reduce=off CPU affinity enabled

4.1 Evaluation Setup

We use a testbed with two servers, each with 100Gbps links, directly connected without any intervening switches; while simple, this testbed allows us to ensure that bottlenecks are at the server-side thus allowing us to stress test i10. Both servers have the same hardware/software configurations (Table 1). Our NICs have one Ethernet port and one InfiniBand port, allowing us to evaluate both NVMe-TCP and NVMe-RDMA.

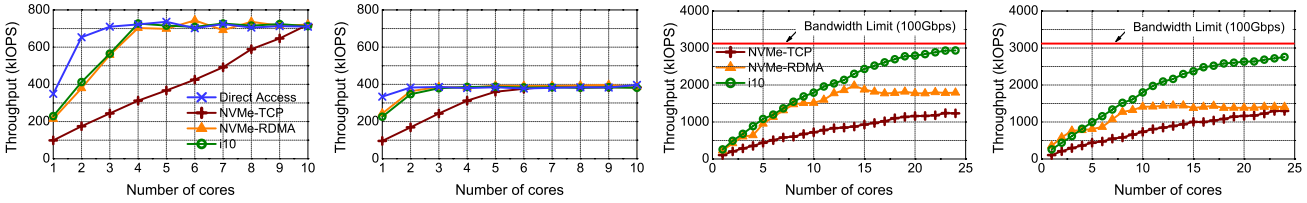
Our NICs support dynamically-tuned interrupt moderation feature that controls the network RX interrupt rate [41], which helps achieving maximum network throughput with minimum interrupts under heavy workloads; however, we disable it to show that i10 does not rely on special hardware features. Similarly, we do not optimize the Interrupt Request (IRQ) affinity configuration, but simply use the `irqbalance` provided by the OS. Finally, we use FIO [5] application for our microbenchmarks with a default I/O depth of 128. All I/O requests are submitted via the asynchronous I/O library (`libaio`) and direct I/O is enabled so as to bypass the kernel page cache and to make sure that I/O requests always go through the network to reach the target device.

i10 and NVMe-RDMA saturate our NVMe solid state



(a) Average Latency (SSD) (b) Tail Latency (SSD) (c) Average Latency (RAM) (d) Tail Latency (RAM)

Figure 6: **Single core performance (4KB random read)**: when compared to NVMe-TCP, i10 achieves significantly higher throughput-per-core with comparable latency; when compared to NVMe-RDMA, i10 achieves comparable throughput while achieving average and tail latency within 1.4× and 1.7×, respectively, for the case of SSDs.



(a) Read (SSD) (b) Write (SSD) (c) Read (RAM) (d) Write (RAM)

Figure 7: **Multi-core performance (4KB random read/write)**: i10 achieves throughput significantly better than NVMe-TCP and comparable to NVMe-RDMA while operating on commodity hardware; i10 and NVMe-RDMA also achieve near-perfect scalability with number of cores.

drives (SSD) with only 4 cores; hence, we also use RAM block device to evaluate multi-core scalability. In addition, our RAM-based experiments allow us to emulate i10 performance for the emerging Non-Volatile Main Memory devices as their performance is close to that of DRAM [45]. Unless otherwise stated, we use 4 cores for SSD-based and 16 cores for RAM-based evaluation.

4.2 Performance Evaluation

We now evaluate i10 performance across a variety of settings. All experiments in this subsection focus on host-side CPU utilization since target CPU was never the bottleneck (§4.3.2).

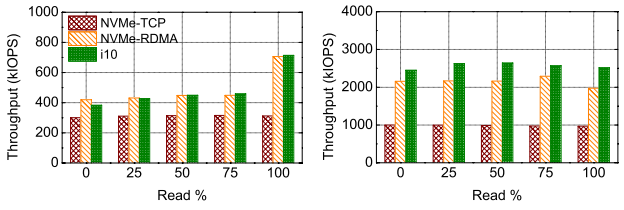
4.2.1 Single core performance

Figure 6 presents the single-core performance for all the evaluation schemes for both NVMe SSD and RAM block devices. The key takeaway here is that, when compared to NVMe-RDMA, i10 offers better throughput at high loads, but slightly higher latency (still at hundred-microsecond granularity) at low loads. Intuitively, when the load is high, i10 works with full-sized caravans without delayed doorbell timeouts, thus achieving high throughput. For low loads, i10 would wait for more requests to arrive in the i10 queue until the doorbell timer expires, which slightly increases its average end-to-end latency. However, we note that even for low loads, i10 average latency is comparable to that of NVMe-RDMA — *e.g.*, for the case of SSDs, where SSD access latency becomes the bottleneck, i10 achieves an average latency of 189μs while NVMe-RDMA achieves 105μs (Figure 6(a)). We observe similar trends for the tail (99th percentile) latency for the case of SSDs — as shown in Figure 6(b), i10's tail latency of 206μs is within 1.7× of NVMe-RDMA tail latency of 119μs, again since SSD access latency is the main bottleneck. For the case of RAM block device, i10 has higher average and tail latency compared to NVMe-RDMA since kernel overheads start to dominate; however, i10 still achieves comparable or better throughput per core. Finally, we also observe that all the three schemes perform better with RAM block device when compared to SSD since the former significantly reduces the access latency, not requiring any interrupt handling between device driver and RAM block device.

4.2.2 Scalability with number of cores

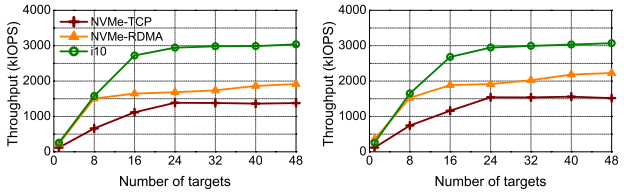
To understand i10 performance with increasing number of cores, we extend the previous single-core measurements to use up to 24 cores. Figure 7 presents the results. We observe that, for the case of random reads, i10 and NVMe-RDMA saturate the SSD with 4 cores, which is a factor 2.5× improvement over NVMe-TCP (Figure 7(a)), while direct access (where the requests go to local SSD) saturates the SSD using 3 cores. The case of random writes in Figure 7(b) is similar to Figure 7(a); both NVMe-RDMA and i10 saturate the maximum random write performance with 3 cores whereas NVMe-TCP requires 6 cores.

With RAM block device, i10 mostly outperforms both NVMe-TCP and NVMe-RDMA as shown in Figures 7(c) and Figures 7(d). Perhaps most interestingly, i10 is able to saturate the 100Gbps link, achieving 2.8M IOPS with ∼20 cores. We observe that NVMe-RDMA stays around 1.5–2M IOPS after 10 cores for both random read and write workloads. While we believe that this performance saturation is not fundamental to NVMe-RDMA and is merely a hardware issue, we have not yet been able to localize the core problem; we note, however, that similar observations have been made



(a) SSD (4 cores) (b) RAM block device (16 cores)

Figure 8: **i10 maintains its performance against NVMe-TCP and NVMe-RDMA with workloads comprising varying read/write ratios (4KB mixed random read/write)**.



(a) Read (RAM) (b) Write (RAM)

Figure 9: **i10 maintains its performance with increasing number of target devices (4KB random read and write).** The trend is similar to the multi-core/single-target case in Figure 7.

in other recent papers [31].

4.2.3 Performance with varying read/write ratios

Figure 8 presents results for workloads comprising varying ratio of read/write requests, varying from 0:100 to 100:0, for both SSD and RAM block device. For the case of SSD (Figure 8(a)), we observe that throughput in each case is limited by random write performance except that of the 100% read ratio case. This observation is consistent with the previous study that shows random write can interfere with random read because of wear leveling and garbage collection, where 75% read shows a similar IOPS with 50% read [24]. Consistent to results in previous subsections, both i10 and NVMe-RDMA saturate the SSD while requiring fewer cores than NVMe-TCP. With RAM block device (Figure 8(b)), the throughput changes with marginal fluctuation regardless of the read ratio; nevertheless, i10 continues to achieve comparable or better throughput than NVMe-RDMA across all workloads.

4.2.4 Scalability with multiple targets

We now evaluate i10 performance with increasing number of target devices, using up to 48 target RAM block devices. Here, we only focus on RAM devices since our testbed has a limited number of SSD devices. The setting here is that of each core running applications that access data from targets assigned to the application. The assignment is done in a round-robin manner — for up to 24 targets, each core will serve one target and for more than 24 targets, we assign the additional targets to the cores starting with core0 (*e.g.*, for 36 targets, the first 12 cores serve two targets each and the remaining 12 cores serve one target each).

In Figure 9, we observe that i10 outperforms both NVMe-TCP and NVMe-RDMA, saturating the 100Gbps link with 16 or more targets. The 24-target throughput is kept after 24 targets for all schemes as every host core is fully used (for i10, the 100Gbps link is already saturated). The overall trend is similar or even slightly better when compared to the RAM cases of the multi-core scalability scenario (Figure 7) as the I/O requests are processed in parallel across different i10-ℓanes. This result confirms that i10 maintains its performance benefits with increasing number of targets incurring little CPU contention across the various i10-ℓanes. For the scenarios where such CPU contention is severe (assuming an extremely large number of targets), the single-core/multi-target throughput is further studied in Appendix B in [16].

4.3 Understanding Performance Gains

We now evaluate how various design aspects of i10 contribute to its end-to-end performance, and then use CPU profiling to build a deeper understanding of i10 performance gains.

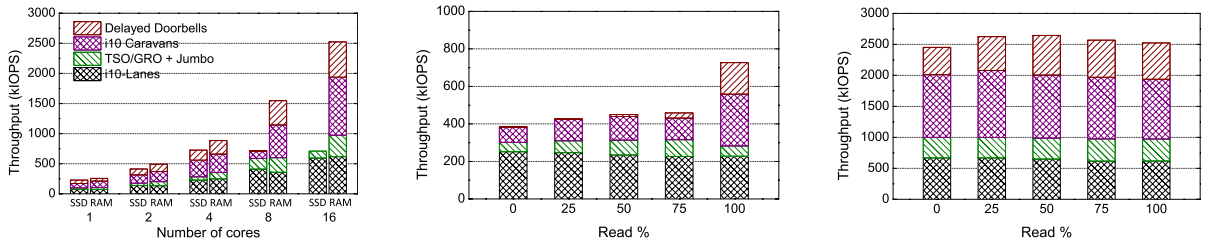
4.3.1 Performance contribution

Figure 10 shows that each of the design aspects of i10— i10-ℓane, i10 caravans, and delayed doorbells — are essential to achieve the end-to-end i10 performance. In Figure 10(a), we measure 4KB random read throughput increasing the number of cores from 1 to 16. The baseline is our i10-ℓane performance, which scales well with multiple cores. Enabling TSO/GRO and jumbo frames makes slightly further improvement over the i10-ℓane throughput. With NVMe SSD, i10 contribution is limited by the SSD performance (Figure 10(b)), but it is clear that with RAM block device, i10 caravans and delayed doorbells contribute to the performance improvement significantly, by 38.2% and 23.2% of the total throughput with 16 cores as shown in Figure 10(c). We also confirm that these improvement trends are maintained regardless of the read ratio for both SSD and RAM devices. The above results therefore indicate that all of the three building blocks are indeed necessary to design an end-to-end remote storage I/O stack that achieves the aforementioned benefits.

4.3.2 Understanding Bottlenecks

We now use CPU profiling for the 4-core case of SSD and the 16-core case of RAM block device in Figure 10(a), with the goal of understanding the bottlenecks for each of the three schemes. Our profiling results in Figure 12, Table 3 and Table 4 divide the entire remote access process into 7 components as described in Table 2. Our key findings are:

(1) i10 spends fewer CPU cycles in network processing: i10 improves upon NVMe-TCP by a factor of 2.7× in terms of CPU usage reduction in network processing parts (Network Tx and Rx combined) for both NVMe SSD and RAM block device, while showing comparable CPU utilization to NVMe-RDMA in the same parts. The main problem of NVMe-TCP is that it underutilizes the network capacity even with such high



(a) Throughput with varying # cores (b) Throughput (SSD) with varying read % (c) Throughput (RAM) with varying read %

Figure 10: **Each of i10-ℓane, caravans, delayed doorbells are necessary to achieve the end-to-end i10 performance**.

Table 2: **Taxonomy of CPU usage.**

Component	Description
Applications	Submit and receive requests/responses via I/O system calls (host). Ideally, all cycles would be spent in this component.
Block TX	Process the requests at the blk-mq layer and ring the doorbells to the remote storage access layer (host) or the local storage device (target).
Block RX	Receive the requests/responses from the network Rx queues or the local storage device.
Net TX	Send the requests/responses from the I/O queues (NVMe-RDMA uses Queue Pairs in the NIC).
Net RX	Process packets and insert into the network Rx queues (by the network interrupt handler).
Idle	Enter the CPU *Idle* mode.
Others	Include all the remaining overheads such as task scheduling, IRQ handling, spin locks, and so on.

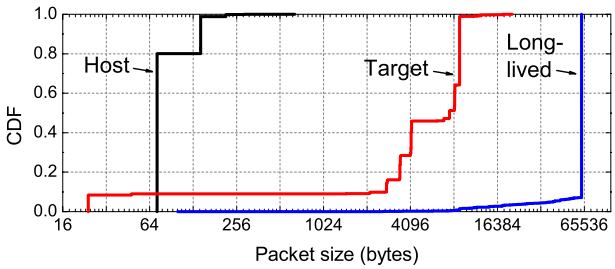


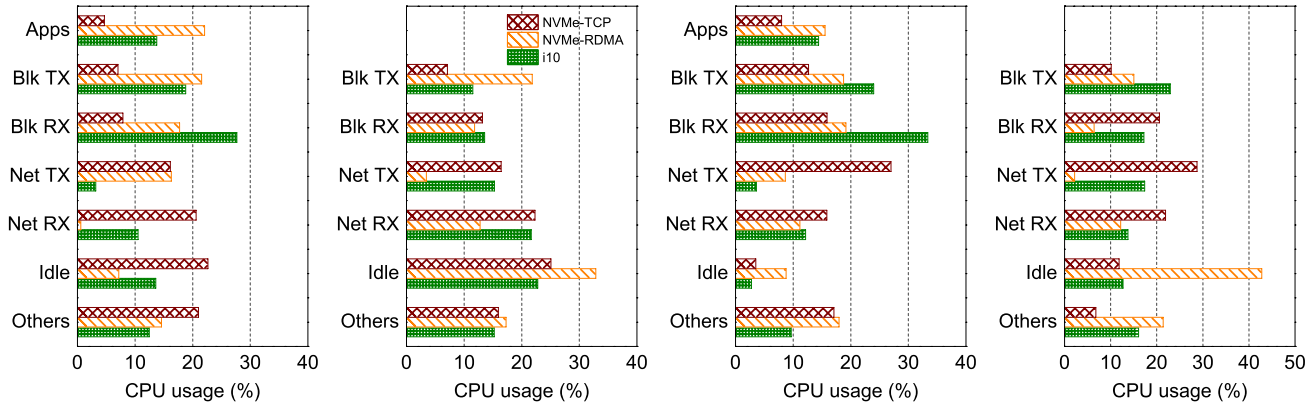
Figure 11: **For 4K random read, NVMe-TCP does not benefit from TSO** due to the packet sizes being mostly 72B at host and $<$ 9KB at target, much smaller than ideal 61KB packet size.

CPU usage in network processing. To investigate the reason further, we measure the packet sizes used in each TCP/IP processing for 4KB random read, comparing to a long-lived TCP connection that achieves ∼30Gbps using a single core. Figure 11 reveals that about 80% of packet sizes generated by the host are 72 bytes, the I/O request PDU size of NVMe-TCP. This implies that almost every single small-sized request consumes CPU cycles for TCP/IP processing, increasing per-byte CPU usage. The target can generate a larger size of packets as it sends 4KB response data back to the host, but still most of them (98%) are under the jumbo frame size (9000 bytes) while the long-lived TCP connection generates mostly 61KB packets with TSO. i10 caravans help mitigate this bottleneck by generating 1152B packets (that is, 72B×16) at the host and ∼61KB packets at the target; i10 caravans are thus able to help reduce CPU usage by 30.12% and 31.14% for SSD and RAM block device in network processing parts, compared to the baseline i10 that uses only i10-ℓane.

(2) i10 minimizes context switching overheads: i10 achieves 1.7× CPU usage reductions over NVMe-TCP in *Others* part that includes task scheduling overheads. NVMe-TCP involves three kernel threads at the host — one for `blk-mq` that corresponds to the application thread, another for the remote I/O and TCP/IP Tx processing, and the third for the packet interrupt handling and TCP/IP Rx processing. This model avoids (i) slow responsiveness to other threads and/or (ii) long bottom half procedure for the incoming packet interrupts, but can incur high context switching overhead; our measurement indicates that each context switch takes 1–3μs per request, consuming more CPU cycles at the host. i10 amortizes this switching overhead using the idea of delayed doorbells; when compared to i10 design that does not use delayed doorbells, we observe reduction of CPU usage by 14.2% and 14.15% for NVMe SSD and RAM block device in the Others part. While the previous work mainly focuses on the target architecture [24], this host-side optimization turns out to be essential to improve the remote I/O throughput given that all the three remote storage access technologies consume more CPU cycles at the host regardless of the device type.

(3) i10 improves CPU efficiency allowing more cycles for applications: CPU resources saved in network processing and in context switching (using caravans and delayed doorbells) can be utilized by the applications, resulting in improved throughput per core. For instance, i10 allows applications to use 2.9× and 1.8× more CPU cycles for SSD and RAM devices, when compared to NVMe-TCP. NVMe-RDMA also shows 1.9× more CPU usage than NVMe-TCP on applications with RAM block device.

(4) i10 pushes the performance bottlenecks to the block layer: Tables 3 and 4 show that i10 pushes the performance bottlenecks from network processing and other lower layers (scheduling, etc., including in Others) to upper layers, making



(a) Host (SSD) (b) Target (SSD) (c) Host (RAM) (d) Target (RAM)

Figure 12: **CPU consumption at various system components for i10, NVMe-TCP and NVMe-RDMA**. i10 and NVMe-RDMA use significantly fewer CPU cycles for network processing and task scheduling (in Others) at the host while allowing applications to consume more CPU cycles, when compared to NVMe-TCP.

Table 3: **CPU usage contribution for 4KB random read with NVMe SSD**: i10 reduces CPU usages in network processing (Net TX and RX) using i10 caravans and in task scheduling (Others) using delayed doorbells, which allows more CPU cycles for applications and block layer. Here, i10-ℓane contribution is measured with enabling TSO/GRO and jumbo frames.

	Applications	Block TX	Block RX	Net TX	Net RX	Idle	Others
i10-ℓane	4.67	7.02	7.88	16.14	20.61	22.66	21.02
i10 Caravans	+9.85	+14.54	+16.41	-13.7	-16.42	-16.34	+5.66
Delayed Doorbells	-0.75	-2.76	+3.38	+0.73	+6.34	+7.26	-14.2
i10	13.77	18.8	27.67	3.17	10.53	13.58	12.48

Table 4: **CPU usage contribution for 4KB random read with RAM block device**. The trends are similar to the SSD case above.

	Applications	Block TX	Block RX	Net TX	Net RX	Idle	Others
i10-ℓane	8.0	12.68	15.9	27.0	15.84	3.51	17.07
i10 Caravans	+5.77	+10.88	+8.83	-24.79	-6.35	-1.07	+6.73
Delayed Doorbells	+0.63	+0.43	+8.66	+1.42	+2.67	+0.34	-14.15
i10	14.4	23.99	33.39	3.63	12.16	2.78	9.65

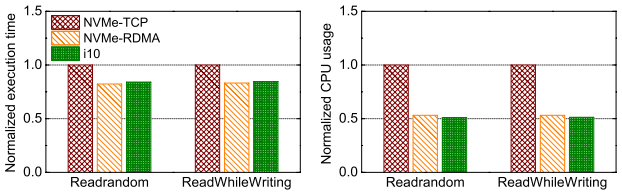
the block device layer a new bottleneck point in the kernel. i10 design did not attempt to perform changes in the block layer; however, it would be interesting to explore block layer optimizations to further improve end-to-end performance for remote (and even local) storage access.

We observe that RDMA still consumes a few CPU cycles to build and parse the command PDUs in network processing parts. In our profiling, one main difference between the SSD and RAM cases is that the RAM block device does not use IRQ to inform the I/O completion, but calls the the block layer functions immediately while in the SSD case, it still relies on the `nvme_irq` handling after the I/O is completed. This increases the IRQ handling overhead. Further, NVMe-RDMA generates another type of IRQ to call the block layer functions in the host after the network processing is done. This also slightly increases the IRQ handling overhead in the host.

4.4 i10 performance with RocksDB

To evaluate i10 performance over real applications, we use RocksDB, a popular key-value store deployed in several production clusters [17]. We install RocksDB in the host server with a remote SSD device mounted with XFS file system. The RocksDB database and write-ahead-log files are stored on the remote device. To minimize the effect of the kernel page cache, we clear the page cache every 1 second during the experiment. We use `db_bench`, a benchmarking tool of RocksDB, for generating the two workloads using default parameters [18]: *ReadRandom* and *ReadWhileWriting*. Before running the workloads, we populate a 55GB database using the *bulkload* workload.

We measure the end-to-end execution time and kernel-side CPU utilization using a single core. In this experiment, the doorbell timeout value is set to 1ms as RocksDB is not an



(a) Execution time (b) CPU usage in kernel

Figure 13: **i10 and NVMe-RDMA achieve** 1.2× **lower latency and** 2× **lower CPU utilization when compared to NVMe-TCP over SSD-based RocksDB.**

I/O bound application, and thus requires more time to aggregate appropriate number of requests. Figure 13 shows the performance of the three schemes, normalized by the NVMe-TCP performance. Again, we observe that i10 performs comparable to NVMe-RDMA while achieving almost 1.2× improvements over NVMe-TCP in terms of execution time. The reason why this improvement is different with that of FIO benchmarks is that, RocksDB itself is the main CPU cycle consumer (up to 70% CPU usage) to perform data compression, key matching, etc. However, i10 still allows RocksDB to utilize more CPU resources by requiring 2× lower CPU in the kernel across a fixed number of requests when compared to over NVMe-TCP. Our additional experiments with Filebench [39], presented in Appendix B in [16], indicate that if the application is I/O bound, i10 can achieve more than 2× per-core throughput improvements over NVMe-TCP in a similar setup.

4.5 Comparison with ReFlex

Now we compare i10 with ReFlex, a user-level remote Flash access stack [24] using FIO. Unfortunately, despite significant effort, we were unable to install the remote block device kernel module for ReFlex [8] in our system[3], which is required for ReFlex host to run legacy Linux applications such as FIO. Thus, we make an indirect comparison with ReFlex, measuring i10 throughput using the same 10Gbps NICs used in [24] (that is, Intel 82599ES 10GbE NICs) and the same FIO script [8]. We also use 1-core i10 target as ReFlex target server uses 1-core per tenant. In this setup, i10 saturates the 10Gbps link with ∼3 cores, whereas ReFlex-FIO requires 6 cores according to [24]. This result still suggests that i10 would be a good option for remote storage I/O when we use legacy throughput-bound applications.

We also note that using IX [6]-based ReFlex clients achieves higher throughput; for instance, ReFlex reports achieving 850 kIOPS for 1KB read-only request using a single core [24], thus requiring only 2 cores to saturate a 10Gbps link. However, this requires the client server to run IX, precluding integration with unmodified legacy applications.

[3]The kernel module is based on an old version of kernel (4.4.0) that does not include relevant device drivers for our SAS SSD where the OS is installed. We also failed to boot with our NVMe SSD even with the up-to-date BIOS.

5 Discussion

We discuss a number of possible extensions for i10.

Can we apply i10 techniques to improve iSCSI performance? We believe that many of our optimizations may be useful to improve iSCSI [35] performance; for instance, the current Linux iSCSI implementation consumes CPU cycles inefficiently when sending I/O requests and responses, not fully utilizing TSO/GRO; moreover, it also runs a dedicated kernel thread for TCP/IP processing. i10 caravans and delayed doorbells alleviate precisely these bottlenecks, and thus may be useful for iSCSI.

Integrations with Emerging Transport Designs. i10 design and implementation was originally motivated by the question whether state-of-the-art performance for remote storage stacks can be achieved using simple modifications in the kernel. Thus, i10 design naturally integrates with existing network stack within the kernel. An intriguing future work would be to integrate i10 with emerging CPU-efficient network transport designs that use hardware offload.

Overheads of maintaining i10-ℓanes. Our i10-ℓane design does not introduce any additional overhead at either `blk-mq` or TCP/IP protocol layer, but simply exploits the existing per-core `blk-mq` that every request goes through regardless of whether it is remote access or not. The new overheads introduced by i10-ℓane are: i10 caravans, i10 queues, and TCP socket instances. To minimize memory usage and de-/allocation overheads of i10 caravan, each i10-ℓane maintains a single full-sized caravan instance (an array of kernel vectors that cover the aggregation size of requests and 64KB data) and reuse it whenever a new doorbell is rung. Memory requirement for maintaining per-(core, target) i10 I/O queues would be comparable to the current NVMe implementation, which creates per-(core, device) NVMe queues. Lastly, a dedicated socket requires small amount of state and thus, adds a minor memory overhead. We believe that such minor overheads are well worth the performance benefits achieved by i10.

Setting the right doorbell timeout value. In some extreme cases where a single host core needs to use many i10-ℓanes (that is, requests from a single core going to a large number of target servers), i10 may expose a throughput-latency tradeoff. Assuming the core generates requests at full rate, increasing number of i10-ℓanes means that the number of pending requests for each individual i10-ℓane will be reduced triggering doorbell timeout more often (see Appendix B in [16]). The tradeoff here is that to achieve throughput similar to the results in the previous section, we will now need larger delayed doorbell timeout value, resulting in relatively larger latency. This observation suggests that operators can set a higher timeout value when (1) applications are throughput bound; and/or (2) the number of targets per core increases. An interesting future work here is to explore setting the timeout value dynamically, depending on the "observed" load on the system.

Table 5: **Comparison of design decisions made in i10 with those in several prior works.**

	Storage stack	Network stack	API	App. event handling	Remote I/O event handling	Domain protection	Modification
Linux kernel	Kernel	Kernel	BSD socket	Syscalls	Per event	Native	No
MegaPipe [13]	Kernel	Kernel	lwsocket	Batched	per event	Native	App., Kernel
mTCP [19]	N/A	User-level	New API	Batched	N/A	Vulnerable	App., NIC driver
IX [6]	N/A	User-level	New API	Batched	N/A	Virtualization H/W	App., NIC driver
StackMap [44]	Kernel	Kernel	Ext. netmap	Batched	per event	Vulnerable	App., Kernel, NIC driver
ReFlex [24]	User-level	User-level	New API	Batched	Batched	Virtualization H/W	App., NIC driver
i10	Kernel	Kernel	BSD socket	Syscalls	Batched	Native	Kernel-only

6 Related Work

Table 5 compares i10 with several of the prior designs on optimizing the storage and network stacks. We briefly discuss some of the key related work below.

Existing remote storage I/O stacks. The fundamental performance bottlenecks of traditional remote storage stacks are well-understood [24]. For instance, iSCSI protocol [35] was designed to access remote HDDs over 1Gbps networks, achieving merely ∼70K IOPS per CPU core [12, 23, 24]. Similarly, a light-weight server for remote storage access based on Linux `libevent` and `libaio` achieves ∼75K IOPS per core [24]. Distributed file systems (*e.g.*, HDFS, GFS, etc.) are generally optimized for large data transfers, but are not so efficient for small-sized random read/write requests over high-throughput storage devices [10,36]. i10 significantly improves throughput-per-core when compared to existing kernel-based remote storage stacks, achieving performance close to state-of-the-art user-space and RDMA-based products.

CPU-efficient network stacks within the kernel. Motivated by the fact that existing kernel *network* stacks were not designed for high network bandwidth links, there has been a significant amount of recent work on designing CPU-efficient network stacks [13, 27, 38, 42]. These stacks focus primarily on network stacks and are complementary to optimizing remote storage stacks; nevertheless, several optimization techniques (*e.g.*, syscalls batching/scheduling, per-core accept queue, etc.) introduced in these works may be useful for i10 as well. For instance, we demonstrate in Appendix B in [16] that syscall-level batching from [13, 38, 42], when integrated with i10, helps further improve the performance by 1.2×.

CPU-efficient user-space network stacks. The main motivation of this approach is that the kernel is extremely complex and high-overhead due to various overheads in system calls, process/thread scheduling, context switching, and so on. Prior studies reveal that the current kernel stack has limited network processing power in terms of the number of messages per second, so it is difficult to saturate network link capacity if the applications generate only small-sized messages, *e.g.*, under tens to hundreds of bytes [6, 19, 44].

To avoid these overheads, user-space solutions place the entire network protocol stack in the user space and directly access the NIC through the user-level packet I/O engines such as DPDK [1] and netmap [34], while bypassing the kernel [6, 19, 21, 22, 29, 32, 33]. By putting small-sized packets onto the NIC buffer directly in a batched manner, they significantly improve the messages per second performance. ReFlex [24] also implements the remote Flash access stack in the user space, on top of IX [6]. We note that modifying applications for using the user-level stacks might not be fundamental as recent new systems support the POSIX interfaces (*e.g.*, TAS [22], Strata [25], and SplitFS [20]). As we have discussed throughout the paper, i10's goals are not to beat the performance of user-space solutions but rather to explore whether similar performance can be achieved without modifications in the application and/or network infrastructure.

7 Conclusion

This paper presents design, implementation and evaluation of i10, a new in-kernel remote storage stack for high-performance network and storage hardware. i10 requires no modifications outside the kernel, and operates directly on top of kernel's TCP/IP network stack. We have demonstrated that i10 is still able to achieve throughput-per-core comparable to state-of-the-art user-space and RDMA-based solutions. i10 thus represents a new operating point for remote storage stacks, allowing state-of-the-art performance without any modifications in applications and/or network infrastructure.

Acknowledgements

We would like to thank our shepherd, Simon Peter, and the anonymous NSDI reviewers for their insightful feedback. We would also like to thank Amin Vahdat, Anurag Khandelwal, Saksham Agarwal and Midhul Vuppalapati for many helpful discussions during this work. This work was supported in part by NSF 1704742, ONR Grant N00014-17-1-2419, a Google Faculty Research Award, and gifts from Snowflake and Cornell-Princeton Network Programming Initiative.

References

[1] Intel DPDK: Data Plane Development Kit. https://www.dpdk.org/.

[2] NVM Express 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2019.

[3] NVM Express over Fabrics 1.1. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf, 2019.

[4] Amazon. Introducing Amazon EC2 C5n Instances Featuring 100 Gbps of Network Bandwidth. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/.

[5] Jens Axboe. Flexible IO Tester (FIO) ver 3.13. https://github.com/axboe/fio, 2019.

[6] 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 *USENIX OSDI*, 2014.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *ACM SYSTOR*, 2013.

[8] Ana Klimovic et al. ReFlex github. https://github.com/stanford-mast/reflex, 2017.

[9] Peter Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SOSP*, 2003.

[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*, 2017.

[12] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *ACM SYSTOR*, 2017.

[13] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*, 2012.

[14] Christoph Hellwig. High Performance Storage with blk-mq and scsi-mq. https://events.static.linuxfound.org/sites/events/files/slides/scsi.pdf.

[15] HGST. LinkedIn Scales to 200 Million Users with PCIe Flash Storage from HGST. http://pcieflash.virident.com/rs/virident1/images/Case_Study_LinkedIn_PCIe_CS008_EN_US.pdf, 2014.

[16] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. Technical Report, https://github.com/i10-kernel/, 2020.

[17] Facebook Inc. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/, 2015.

[18] Facebook Inc. RocksDB benchmark script. https://github.com/facebook/rocksdb/blob/master/tools/benchmark.sh, 2019.

[19] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, 2014.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *ACM SOSP*, 2019.

[21] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[22] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *ACM Eurosys*, 2016.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *ACM ASPLOS*, 2017.

[25] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *ACM SOSP*, 2017.

[26] Lightbits Labs. The Linux Kernel NVMe/TCP support. http://git.infradead.org/nvme.git/shortlog/refs/heads/nvme-tcp, 2018.

[27] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ACM ASPLOS*, 2016.

[28] Robert Love. *Linux Kernel Development*. Addison-Wesley, 3 edition, 2010.

[29] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *ACM SIGCOMM*, 2014.

[30] Kazan Networks. ACHIEVING 2.8M IOPS WITH 100GB NVME-OF. https://kazan-networks.com/blog / achieving-2-8m-iops-with-100gb-nvme-of, 2019.

[31] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: a fast transactional dataplane for remote data structures. In *ACM SYSTOR*, 2019.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[33] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[34] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[35] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). https://www.ietf.org/rfc/rfc3720.txt, 2004.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE MSST*, 2010.

[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[38] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX OSDI*, 2010.

[39] Vasily Tarasov. Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench, 2018.

[40] Jason Taylor. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *OFC*, 2015.

[41] Mellanox Technologies. Dynamically-Tuned Interrupt Moderation (DIM). https://community.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2019.

[42] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The Case for VOS: The Vector Operating System. In *USENIX HotOS*, 2011.

[43] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *USENIX NSDI*, 2020.

[44] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX ATC*, 2016.

[45] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *USENIX FAST*, 2019.