Performance Evaluation of i10 Linux I/O Scheduler

Jaehyun Hwang, Qizhe Cai, Midhul Vuppalapati, Rachit Agarwal
Cornell University

1 Evaluation Setup

We use a testbed with two servers (host and target), each with 100Gbps links, directly connected without any intervening switches. Both servers have the same hardware/software configurations as shown in Table 1.

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 DRAM
NIC	Mellanox ConnectX-5 Ex VPI (100G) TSO/GRO=on, LRO=off, DIM disabled Jumbo frame enabled (9000B)
NVMe SSD	1.6TB Samsung PM1725a
S/W configurations	
OS	Ubuntu 16.04 (Linux kernel 5.8.0)
Applied patches	Batching dispatch [1] nvme-tcp optimzations [2–4]
IRQ	`irqbalance` enabled
FIO	Block size=4KB, Direct I/O=on I/O engine=`libaio`, gtod_reduce=off CPU affinity enabled

We compare our i10 I/O Scheduler with "Noop" and "Kyber" I/O schedulers. We use the default values for all parameters (*e.g.*, for i10 batching thresholds, we use 16 for the number of requests and 50μs for timeout). An early version of the i10 idea is described and evaluated in [5].

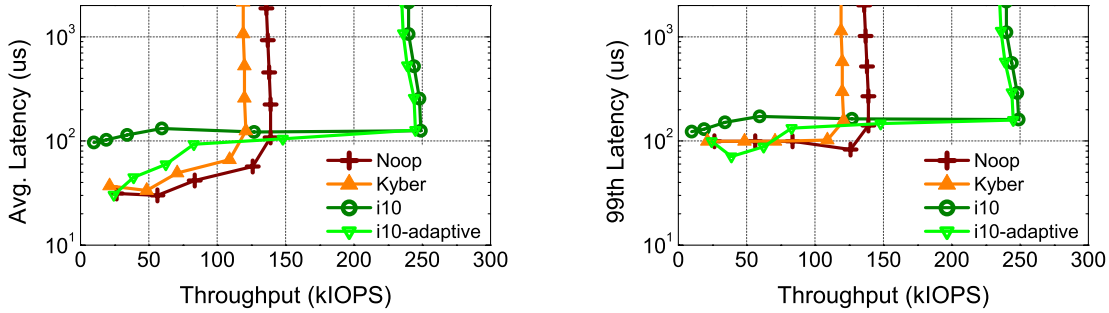
We also evaluate a new version of i10 — referred to as i10-adaptive — that adaptively sets the batching threshold based on the number of outstanding requests and number of consecutive timeouts, as measured at the block layer. While we believe that better algorithms could be designed for adaptive batching, our current algorithm is based on simple observations and additive-increase multiplicative-decrease mechanism. More specifically, if a batch dispatch is triggered without a timeout and if number of outstanding requests is greater than the current batch size, the batch size is increased by 1 — this is because i10 has the "ability" to batch larger number of requests without triggering the timeout. On other other hand, upon two consecutive timeouts, we reduce the batch size by a factor of 2 — this is because timeouts indicate that the system load is not high enough, and i10 can operate at a better operating point on the latency-throughput tradeoff curve by reducing the batch size.

2 Experimental Results

2.1 Remote Storage Access

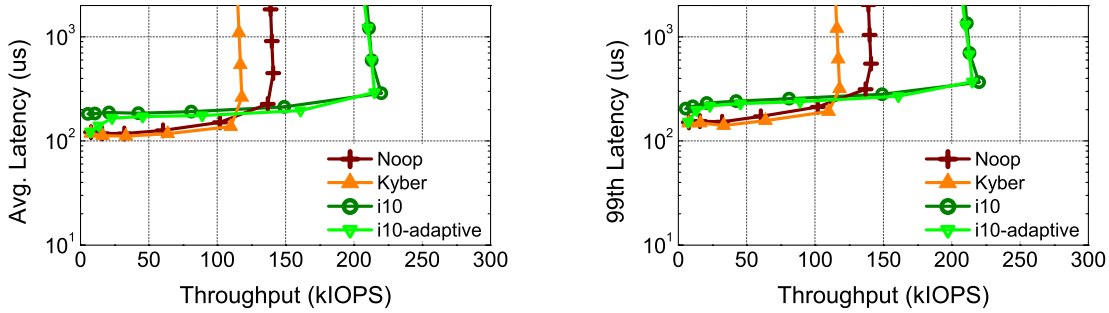
In this subsection, we measure the performance of remote storage access; the host-side applications (FIO) access the target-side storage devices (NVMe SSD or RAM block device) over NVMe-over-TCP.

Single core performance in terms of latency and throughput for remote storage access:



(a) Average Latency (b) Tail Latency

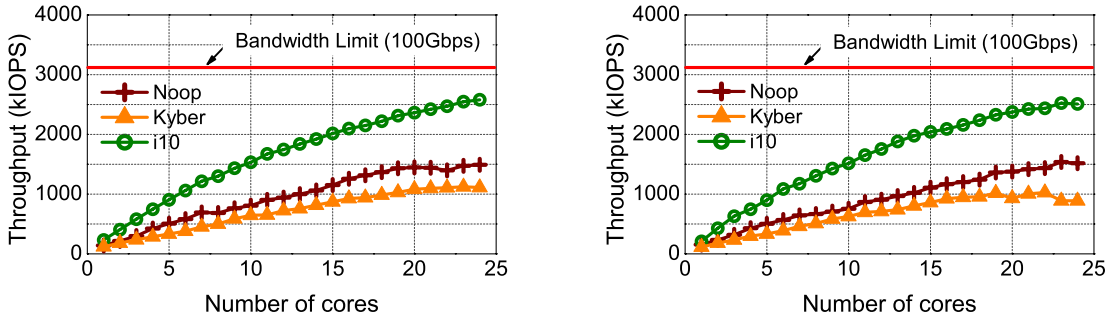
Figure 1: Target device: Remote RAM block device (4KB random read). The key take-away from this figure is that i10 achieves higher throughput when compared to Noop and Kyber by trading off a small amount of latency. Moreover, i10-adaptive is able to reduce the latency at lower loads by adaptively reducing the batch size.



(a) Average Latency (b) Tail Latency

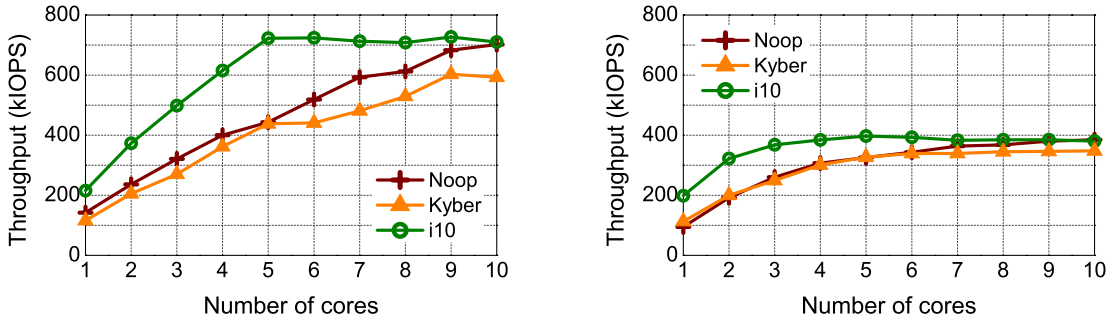
Figure 2: Target device: Remote NVMe SSD (4KB random read). The key take-away from this figure is that i10 achieves higher throughput when compared to Noop and Kyber by trading off a small amount of latency. Moreover, i10-adaptive is able to reduce the latency at lower loads by adaptively reducing the batch size.

Scalability of throughput for remote storage access with number of CPU cores:



(a) Random read (b) Random write

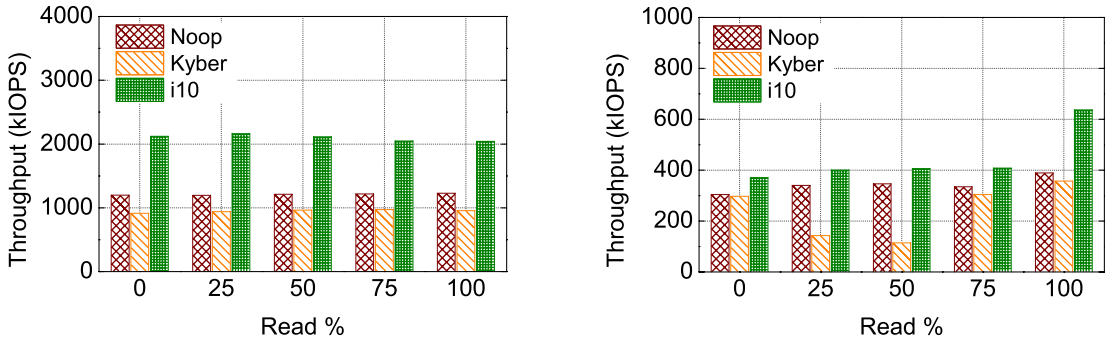
Figure 3: Target device: Remote RAM block device (4KB random read/write, I/O depth $= 64$**).** The key take-away from this figure is that, in terms of throughput for remote storage access, i10 achieves much better scalability when compared to Noop and Kyber with increasing number of cores. i10-adaptive performance is similar to i10 in this test, since the I/O depth size is fixed.



(a) Random read (b) Random write

Figure 4: Target device: Remote NVMe SSD (4KB random read/write, I/O depth $= 64$**).** The key take-away from this figure is that, in terms of throughput for remote storage access, i10 achieves much better scalability when compared to Noop and Kyber with increasing number of cores. i10-adaptive performance is similar to i10 in this test, since the I/O depth size is fixed.

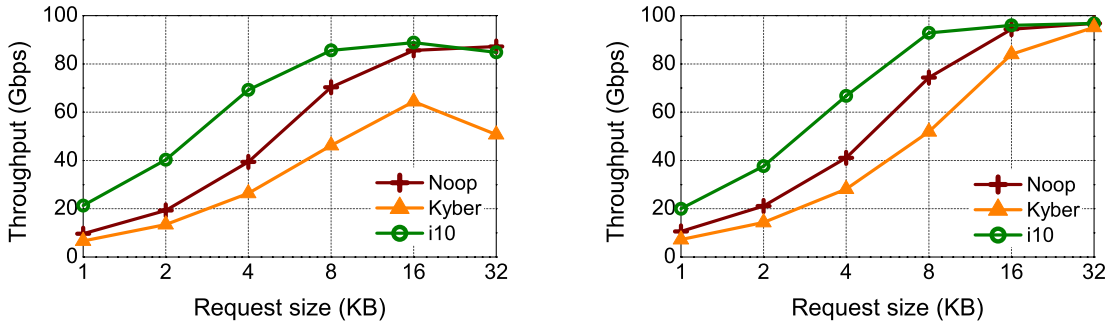
Performance for remote storage access with varying read/write ratios:



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

Figure 5: 4**KB mixed random read/write (I/O depth** $= 64$**).** The key take-away from this figure is that, in terms of throughput for remote storage access, i10 consistently achieves higher throughput (at the cost of slightly higher latency, as in previous results) when compared to Noop and Kyber for various read/write ratios. i10-adaptive performance is similar to i10 in this test, since the I/O depth size is fixed.

Performance for remote storage access with varying request sizes:



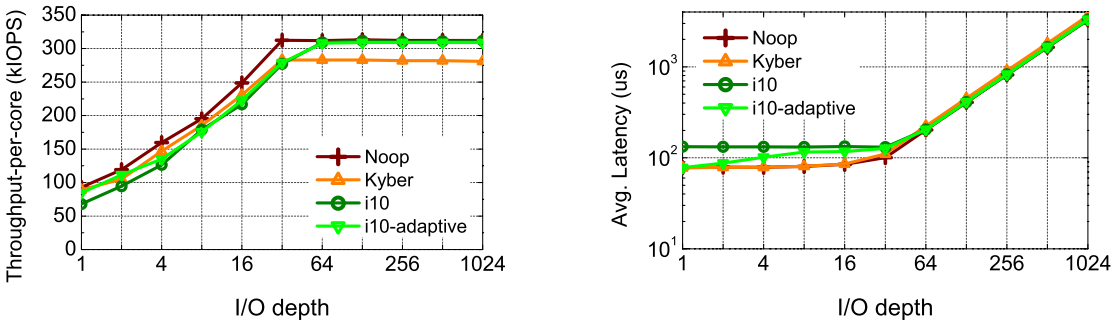
(a) Random write (b) Random read

Figure 6: Target device: Remote RAM block device (I/O depth $= 64$**, 16 cores).** The key take-away from this figure is that, in terms of throughput for remote storage access, i10 consistently achieves higher throughput (at the cost of slightly higher latency, as in previous results) when compared to Noop and Kyber for various request sizes. i10-adaptive performance is similar to i10 in this test, since the I/O depth size is fixed.

2.2 Performance for Local Storage Access

(This measurement is done with Linux kernel 5.10.0) We now measure the i10 performance with a local NVMe SSD to see how it works with a device that does not benefit from batching, while varying I/O depth from 1 to 1024.

Single core performance:



(a) Throughput-per-core

(b) Average latency

Figure 7: Target device: Local NVMe SSD (4KB random read). The key take-away from this figure is that, for high loads and for low loads, i10-adaptive can match the performance of existing I/O schedulers in terms of throughput-per-core and average latency (i10 performance is a little worse than i10-adaptive). However, better adaptation algorithms may be needed for medium loads.

References

[1] https://www.spinics.net/lists/linux-block/msg55860.html.

[2] http://git.infradead.org/nvme.git/commit/122e5b9f3d370ae11e1502d14ff5c7ea9b144a76.

[3] http://git.infradead.org/nvme.git/commit/86f0348ace1510d7ac25124b096fb88a6ab45270.

[4] http://git.infradead.org/nvme.git/commit/15ec928a65e0528ef4999e2947b4802b772f0891.

[5] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*, 2020.