Batching with End-to-End Performance Estimation

Avidan Borisov
Technion

Nadav Amit
Technion

Dan Tsafrir
Technion

ABSTRACT

Batching heuristics are used in multiple layers of the TCP/IP stack, aiming to improve performance by amortizing overheads. When performance is defined as average latency and throughput, optimal batching decisions can be infeasible if application-perceived end-to-end performance is unknown, which is commonly the case in general-purpose setups. We address this problem by occasionally adding a few easily maintained counters to TCP metadata exchanges and using them to estimate end-to-end performance via Little's law. We experimentally show that these estimates are accurate when application requests can be identified by the kernel (corresponding, for example, to send system calls, packets, or some fixed number of bytes). Had these estimates been used to dynamically toggle Nagle batching, they could have extended Redis's range of sustainable throughput at tolerable latencies by nearly 2x and improved latency within this range by as much as nearly 3x. When the kernel cannot identify requests on its own, we propose that applications use a simple new interface to enlighten it, thereby ensuring accuracy.

ACM Reference Format:
Avidan Borisov, Nadav Amit, and Dan Tsafrir. 2025. Batching with End-to-End Performance Estimation. In *Workshop in Hot Topics in Operating Systems (HOTOS 25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730372

1 INTRODUCTION

General-purpose network stacks use heuristics to batch multiple operations with the goal of improving performance by reducing overheads through amortization. For example, upon transmission, at the top of the kernel's TCP/IP stack, we can find such heuristics as Nagle's algorithm [37] and auto-corking [16], which may delay bytes transmission to form bigger packets. Deeper in the stack, towards the NIC driver, bytes may be combined into packets larger than the maximum transmission unit (MTU [39]) to make use of the NIC's TCP segmentation offload capability [11, 18]. Drivers may likewise delay informing NICs that there are packets to send, to amortize the cost of ringing the doorbell [3, 12, 17].

The problem we aim to alleviate is that batching decisions might inadvertently degrade end-to-end latency, throughput, or both. This problem is compounded by the fact that the effect of batching at the sender may entirely depend on timing information at the receiver of which the sender is unaware.

In §2, we exemplify this problem and propose that a solution is to make both sides aware of end-to-end performance through a lightweight exchange of TCP metadata. We hypothesize that this information can enable better batching decisions and facilitate informed tradeoffs between throughput and latency. We explain why existing TCP information (round-trip time etc.) is insufficient and briefly survey the relevant state of the art, highlighting that existing batching approaches disregard end-to-end performance.

In §3, we describe our idea to readily estimate end-to-end latency using Little's law [33], by monitoring three kernel queues within the two communicating parties and maintaining three easily-calculated counters per queue. By Little's law, the average queuing delay of a given queue is Q/λ, such that Q and λ are the average queue size and the rate of elements that enter the queue, respectively. The three TCP queues we monitor are (1) sent, unacknowledged messages; (2) received, unread messages; and (3) delayed acknowledgments. We show that average end-to-end latency can be trivially derived from the queuing delay estimates associated with these three queues. The three per-queue counters are two accumulators and a timestamp used to deduce Q and λ. (Average throughput is computed from λ.)

Applications experience end-to-end performance at a granularity of "request" and "response" units, such that latency is the time between the two and throughput is the number or responses per time unit. The kernel is typically unaware of these units, and so we should bridge this semantic gap to be able to accurately measure end-to-end performance.

Based on past storage-related experience [46], we hypothesize that client and server invocations of the send system call (or equivalent) may reasonably approximate requests and responses, respectively, for some workloads. We later show that even packets or bytes may sometimes achieve this goal. For other workloads, we propose using a simple interface that eliminates the semantic gap (and obviates the

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05…$15.00
https://doi.org/10.1145/3713082.3730372

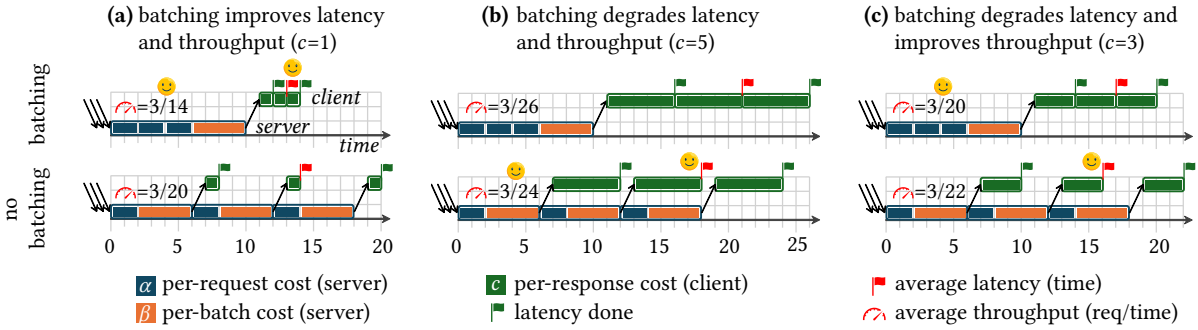

Figure 1: *Without the client's help, any server on/off batching decision can result in a suboptimal outcome, depending on c (the client's processing cost), even in an idealized scenario where message processing is never explicitly delayed and per-request and per-batch costs are fixed (α=2, β=4).*

need to monitor network queues): the application invokes "create" and "complete" routines when issuing a request and receiving a response, respectively. Applying Little's law as described above to the associated counters would then yield accurate application-perceived end-to-end performance.

We demonstrate the potential benefit of our proposal in §4 by running the Redis [42] key-value store twice—with and without Nagle batching—under various load conditions, logging the aforementioned queue counters, and conducting an offline analysis that shows: (1) that our estimates are accurate; (2) that, had they been used to dynamically toggle Nagle batching, they could have increased the maximal load that Redis may serve by nearly 2x while meeting a commonly used latency SLO of under 500 µs [4, 40]; and (3) that they would have improved this latency by as much as nearly 3x.

We discuss the challenges of fully implementing our proposal and integrating it into a live system in §5.

2 MOTIVATION

The Problem. Network stacks commonly batch multiple operations, processing them as a single unit rather than individually. Some perceive this optimization as trading off latency for throughput, amortizing processing costs by waiting for operations to accumulate and then handling them en masse. But the effects of batching are more nuanced.

Consider, for example, a simple scenario where batching never intentionally delays message processing. Instead, messages may accumulate due to system congestion, and the batching scheme need only decide whether to process them individually or as a group [4]. Even in such a straightforward case, batching may improve or degrade average throughput, latency, or both, depending on subtle timing differences.

Figure 1 illustrates such a scenario, where $n = 3$ client requests are waiting for server processing at time 0. We assume a fixed cost $\alpha + \beta$ for serving one request and generating a response, such that α and β are the per-request and per-batch (amortizable) cost, respectively. Overall processing time is therefore $n \cdot \alpha + \beta$ with batching or $n \cdot (\alpha + \beta)$ without.

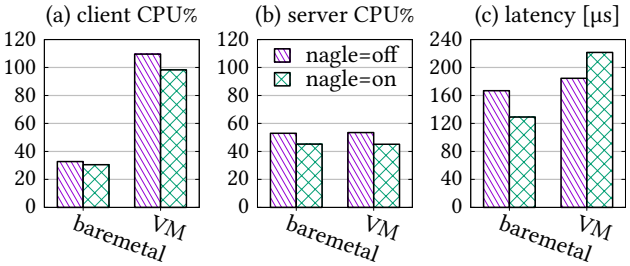

Figure 2: *When running the Redis client inside a VM, it uses significantly more CPU (a), whereas the server's CPU usage remains about the same because it experiences the same workload (b). The client change flips the outcome of Nagle batching (c), thus providing a real example analogous to the artificial one shown in Figure 1.*

We likewise assume a fixed processing time c per response at the client. As c increases from 1 to 5, batching shifts from improving latency and throughput averages (Figure 1a) to degrading them (Figure 1b). Setting c to 3 yields a mixed result: improved throughput but degraded latency (Figure 1c). Throughout these different end-to-end outcomes, the activity from the server's perspective remains identical. Client-side batching would expose the client to the same problem.

We have repeatedly encountered and been frustrated by variants of the above problem when attempting to improve batching policies across the kernel TCP/IP network stack. Figure 2 exemplifies this problem in a real setup resembling the artificial scenario just described. We run a single Redis client (as detailed in §4) on bare metal and within a virtual machine (VM). The client generates a fixed load of 20,000 requests per second handled by a bare metal Redis server. Unsurprisingly, the VM configuration consumes much more CPU than its bare metal counterpart (Figure 2a), which means c (client-side processing cost) is significantly increased. In contrast, the server's behavior remains similar under this fixed load, as evident by the similar CPU usage (Figure 2b). As in Figure 1, we see that different c values can lead to different batching outcomes (Figure 2c): Nagle is advantageous for the bare metal client but not for the VM client.

Goal. We hypothesize that the problem can be alleviated by making the stack aware of end-to-end throughput and latency, and by adopting batching decisions accordingly. Since TCP/IP communicating parties already exchange metadata, it seems both reasonable and straightforward to enrich this information with a few end-to-end performance counters to enhance batching quality. We further hypothesize that the availability of such information will allow systems to deploy policies like "maximize throughput as long as latency remains below a specified threshold." We focus on average performance in this work and defer metrics like tail latency to future studies.

Latency Background. We are by no means the first to estimate end-to-end latency. We would have preferred to use some readily available mechanism that provides this functionality to guide batching decisions, which serves as our primary motivation. We resorted to developing a new mechanism because we were unable to identify a suitable preexisting approach that we can use.

The first alternative we explored and ruled out was using TCP-maintained metrics that are related to latency. Notably, we found that round trip time (RTT) [27] performs poorly, as it does not account for application read delays (encapsulated in c in Figure 1), which are responsible for a substantial portion of end-to-end latency [41]. It is also significantly inflated due to delayed acks [6].

Subsequently, we considered and ruled out relevant state-of-the-art alternatives. For example, Swift [31] and Fathom [15, 41] are unsuitable because they are unavailable for public use. (Also, they rely on NIC timestamping or NTP clock synchronization, which are not necessarily available to arbitrary clients outside the server's organization—clients which we would like to support.)

Batching Background. We are unaware of existing batching approaches that use end-to-end performance to guide their decisions. We hypothesize that this implies that they are susceptible to the problem highlighted in Figure 1. Considering the state of the practice, we systematically reviewed the batching mechanisms of the Linux TCP/IP stack. They are ad-hoc and statically make assumptions that may or may not hold. For example, Nagle's algorithm [37] is enabled by default in the TCP/IP stack. It delays the transmission of small packets and buffers data until a TCP ack is received or enough bytes accumulate to fill the MTU (or 200ms elapse). Many argue that it hampers performance and recommend to disable it [7, 44] due to undesirable interactions with delayed acks [9] or other reasons [35, 36]. Auto-corking [16] is likewise enabled by default, buffering bytes until previous packets are freed from the NIC's transmit ring after a completion interrupt. Being always on, it too introduces latency issues [20]. TSO [11] has a more sophisticated set of rules that it employs while accumulating bytes beyond MTU. But like its simpler counterparts, it can degrade performance as well [47]. Other mechanisms exist [8, 12] and they too suffer from similar problems [29].

Many previous studies explored approaches to improve batching [4, 23, 26, 34, 38, 40, 43, 49]. We do not have space to survey them here except saying none utilize end-to-end performance to guide their actions. Of these studies, IX [4] should be mentioned because it employs "adaptive batching" of requests in response to receive-side "congestion," and these two quoted terms correspond directly to our motivating example: In Figure 1 (top), the server handles multiple in-flight requests ("congestion") by processing them together ("adaptive batching"), refraining from attempting to increase the batch size by means of waiting for additional requests to accumulate. The IX authors say that "when applied adaptively, batching also decreases latency because [it reduces] head-of-line blocking," which we showed is not always the case (as some of them subsequently noticed [40]). In our terminology, IX's adaptive batching is static, as it is a predetermined policy oblivious to end-to-end performance.

3 END-TO-END ESTIMATION

Our goal is to estimate application-perceived end-to-end throughput and latency, which we contend will help make better batching decisions. Next, we describe the queuing theory that we use to achieve this goal (§3.1), how we apply the theory to the kernel's TCP/IP stack (§3.2), how we propose to bridge the semantic gap between the stack and the applications that use it (§3.3), and our prototype implementation used to evaluate the idea (§3.4).

3.1 Utilizing Little's Law

Assume an average of λ travelers check into a hotel each day. If they stay for an average of D days, Little's law states that the average number of hotel guests is $Q = D \times \lambda$ [33]. The same principle applies to packets in a queue. Restating the theorem by switching sides in the equation, $D = Q/\lambda$, namely, the queuing delay (D) is the ratio of the queue's average occupancy (Q) and packet arrival rate (λ), both of which are easy to maintain on the fly. (Little's law is commonly used in this way; also in contexts other than networking [13, 22, 48].)

When considering only packets admitted to the queue (i.e., excluding dropped packets), queuing theory implies that the arrival rate equals the departure rate, namely, λ is also the throughput of the queue. Additionally, when disregarding the time packets spend in the network external to the two communicating machines, latency can be expressed as the sum of queuing delays like D.

Accordingly, given a queue q, we track the information we need to calculate Q and λ using a 4-tuple queue state

Algorithm 1 Update the given queue state q_s with nitemes, which can be positive (items added) or negative (removed).

```
1: Initialize q_s = (time: now, size: 0, total: 0, integral: 0)
2: procedure TRACK(q_s, nitems)
3:     t ← now; dt ← (t - q_s.time); q_s.time ← t
4:     q_s.integral ← q_s.integral + q_s.size · dt
5:     q_s.size ← q_s.size + nitems
6:     if nitems < 0 then
7:         q_s.total ← q_s.total + (-nitems)
```

Algorithm 2 Given two successive states, compute the average latency and throughput during the time between them.

```
1: procedure GETAVGS(q_prev, q_now)
2:     Δq ← q_now − q_prev
3:     return (Q: Δq.integral / Δq.time, tput (=λ): Δq.total / Δq.time, latency: Q/λ)
```

(q_s) defined in the first line of Algorithm 1. Whenever the size of q changes, the network stack updates the state by calling the TRACK procedure with the number of items that are added (positive) or removed (negative). The state records its time, the number of items in q at that time ("size"), the cumulative number of items that left q until that time ("total"), and a corresponding time-weighted accumulator ("integral"), which is updated in Line 4 of Algorithm 1.

Subtracting successive state instances provides the data needed to compute Q and λ (and hence latency and throughput) with the GETAVGS procedure, as shown in Algorithm 2. To illustrate, assume that initially q holds one item for 10 µs and subsequently four items for 20 µs. The associated "integral" is therefore $1\times10+4\times20 = 90$, such that when dividing it by the elapsed time (as specified in Line 3 of GETAVGS), it gives the average size of q ($\frac{90}{10+20} = 3$ items), which corresponds to Q. Similarly, subtracting successive "total" values and dividing them by the elapsed time gives the departure rate λ, which, as explained above, is the throughput.

GETAVGS does not use "size," so each two 3-tuples (integral, total, time) contain all the information needed to estimate a queue's average latency and throughput between the corresponding times. Such tuples can be occasionally exchanged between peers, providing GETAVGS with the information it needs to compute remote queue performance estimations.

3.2 Combining Delays into Latency

Next, we identify the kernel TCP/IP stack queues contributing to the end-to-end latency L and explain how the individual queuing delays are combined into L. For simplicity, for now, we assume that each message (request or response) is sent in a single packet. The L we compute accounts for network stack latency and excludes application processing time. However, if such processing time delays subsequent messages (as in Figure 1), it will be captured as a queuing delay and reflected in the measured latency of those messages.

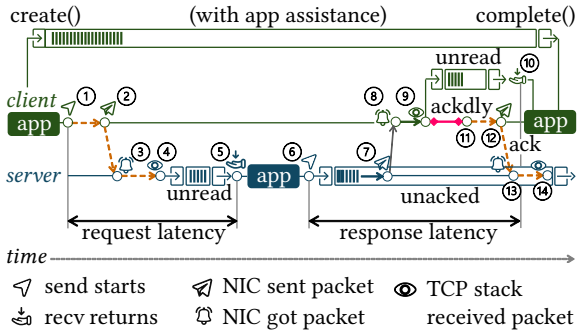

Figure 3: *The request latency (1→5) begins with client send (1) and ends with server recv (5). The response latency (6→10) begins with server send (6) and ends with client recv (10). Their sum L = (1→5) + (6→10) is the end-to-end latency we aim to estimate. The events along the way are detailed using the legend. Let* $L_0 = L^{server}_{unacked} - L^{client}_{ackdelay}$ = (6→14) - (9→11) = (6→9) + (11→14), *and observe that, on average, the left dashed journey (message from client to server between 1→4) can be approximated by the right dashed journey (ack from client to server between 11→14, especially when it is piggybacked on a future request). Thus,* $L_0 \approx$ (6→9) + (1→4) *and so* $L \approx L_0$ + (4→5) + (9→10) *by its above definition, giving* $L \approx L^{server}_{unacked} - L^{client}_{ackdelay} + L^{server}_{unread} + L^{client}_{unread}$, *as claimed. A more accurate estimate can be obtained more easily if clients provide hints (top of figure; see §3.3).*

We contend that L can be accurately estimated by measuring TCP/IP queuing delays of only three queues. Let $L_{unacked}$ be the delay of the queue of messages sent by the application that are not yet acknowledged. Let L_{unread} be the delay of the queue of messages received by the TCP/IP stack but not yet read by the destination application. Let $L_{ackdelay}$ be the delay of the queue of messages received by the TCP/IP stack but not yet acknowledged to the peer. (Ack delays are not typically queued, but treating them as such and deducing their average delay with Little's law greatly simplifies the associated calculation in our implementation.)

Locally, both communicating parties maintain a queue state for each of the above queues and occasionally share the associated 3-tuples with their remote peer as explained in §3.1. With this information available, we claim that

$$L \approx L^{local}_{unacked} - L^{remote}_{ackdelay} + L^{local}_{unread} + L^{remote}_{unread}.$$

The explanation is provided in Figure 3. Both parties share their three local queue states (unacked, ackdelay, and unread), so both can estimate the L value of the other side; we use the maximum between the two to account for possible underestimations. Each party thus shares 36 bytes with its peer per exchange (three 4-byte counters per queue).

The above provides per-connection estimates, which can be averaged if a batching policy simultaneously affects multiple connections.

3.3 Bridging the Semantic Gap

The network stack operates on packets and bytes, unaware of application-level messages. This semantic gap makes it difficult for network stacks to measure end-to-end latency as perceived by applications. In our prototype (§3.4), we opted to treat plain bytes as messages because the Linux kernel already maintains the queue sizes for the for the aforementioned three queues (§3.2) in byte units. This decision limits the effectiveness of our experimental evaluation to workloads with requests and responses of similar size.

Going forward, our next step will be to treat buffers handed to the send system call (or equivalent) as an approximation of application requests and responses. This approach will require a larger and more intrusive kernel patch. But we hypothesize that it may commonly provide the network stack a reasonable view of end-to-end performance from the application's perspective, as others have successfully used it to identify latency-sensitive I/O operations [46].

Still, bridging the semantic gap inherently requires application assistance, as system calls do not always correspond to application messages, e.g., when system calls are batched to reduce overhead. We thus envision a hybrid approach: end-to-end performance for uncooperative applications is estimated by tracking system calls, while cooperative applications provide hints.

Hints are provided at the client side by passing a userspace-maintained 4-tuple "queue state" structure (§3.1) to the send system call using a pointer in send's ancillary data [32]. Applications modify this structure via a minimalist API comprising two functions – create(n) and complete(n) – invoked upon creation and completion of n requests and acting as lightweight wrappers for a userspace implementation of Track (Algorithm 1). The client's stack shares this queue state with the server (as explained in §3.2), allowing it to estimate end-to-end performance using Little's law applied to this single (logical) queue; no additional queue monitoring is needed. Moreover, because the client's semantics determine the observed end-to-end performance, the server needs not monitor and share its own queue states, further simplifying the implementation (top of Figure 3).

The suggested API requires a minimal implementation effort and can easily be integrated into C runtime libraries, making little or no assumptions about application-specific semantics. It therefore seems suitable for adoption by popular request-response frameworks like gRPC [21] and Thrift [1].

3.4 Prototype

We conducted a minimal evaluation of our idea using Linux v6.3, introducing the simplest change we could think of to enable the experiment described in §4. As noted, the "message" unit we use is byte because the queue sizes needed for end-to-end estimation (§3.2) exist as socket-level variables,[1] effectively limiting our evaluation to workloads with requests and responses of similar size. We additionally prototyped using packets as units, and the results turned out similarly limited. Beyond implementing the Track procedure, we only added several lines of code to the TCP/IP stack to invoke Track whenever the existing queue sizes change.

Our prototype exports the 3-tuple queue states as ethtool counters, enabling a userspace GetAvgs implementation. We do not exchange states between peers and instead rely on offline analysis of counters collected from both ends. Notably, our prototype does not dynamically toggle batching; it only allows us to assess the accuracy of our estimates and their *potential* to improve performance, had they been used.

4 EVALUATION

Methodology. We present experimental evidence supporting our idea using the Redis key-value store [42] and Nagle's algorithm [37] (explained in §2), which we use as a representative batching policy. As noted, many argue that Nagle batching should be disabled [7, 44], a practice adopted by Redis. We demonstrate that this static policy is suboptimal and that dynamic on/off batching guided by end-to-end performance estimates would be preferable. We do so by comparing Redis's estimated and measured latency in two benchmark runs: Nagle enabled and disabled. We disregard throughput because it is trivial to measure when application requests are identifiable, necessitating only monitoring of any local queue through which the requests traverse.

Our setup consists of two Dell PowerEdge R730 servers equipped with dual 28-core Intel Xeon E5-2660 CPUs and 128 GiB DRAM. The machines communicate via 100 Gbps NVIDIA ConnectX-5 NICs. One machine runs Redis and the other runs the Lancet load generator [30], which measures latency across varying load rates. Both run Ubuntu with our modified Linux kernel v6.3. Each party has two concurrent execution contexts – application thread (Redis or Lancet) and network stack routines handling incoming packets (IRQ and softIRQ) – which we pin to dedicated cores to reduce runtime variance.

Results. Figure 4a shows the results obtained when the workload consists of a single client that sets 16 KiB values to 16B keys. We observe two key aspects. First, when comparing *measured* performance, Nagle batching proves counterproductive at lower throughput levels. Thus, for such conditions, the decision of Redis's developers to disable it is justified.

[1] Specifically, `sk_wmem_queued` and `sk_rmem_alloc` correspond to the queue sizes associated with $L_{unacked}$ and L_{unread}, respectively. No internal queue is associated with $L_{ackdelay}$, but its size is `rcv_nxt` minus `rcv_wup`, namely, the difference between the next ack sequence number to send (updated upon receiving packets) and the last sent.

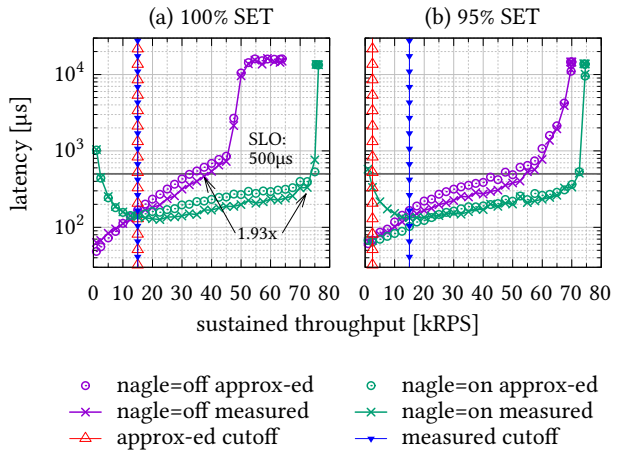

Figure 4: *Nagle's algorithm is disabled by Redis but proves beneficial under high SET load, suggesting that dynamic on/off decisions may be advantageous. Our estimates are accurate when messages have similar sizes (a) but fall short when they do not (b) due to our prototype's limitations. Accordingly, cutoff lines (which show when batching should be toggled) coincide on the left but not on the right.*

But as the load increases beyond the vertical "cutoff" lines, batching becomes advantageous, extending the sustainable range of tolerable latencies (defined here as under a commonly used SLO value of 500 µs [4, 40]) by 1.93x, from 37.5 to 72.5 kRPS. Also, while both configurations meet the SLO at 37.5 kRPS, Nagle batching reduces latency by 2.80x compared to the no-batching default, from 468 µs to 168 µs.

The second key aspect is observable when comparing measured performance to approximations from our prototype (§3.4), which are accurate and, notably, correctly identify the cutoff point where batching becomes worthwhile. These findings support the hypothesis about the potential benefit of dynamic on/off batching decisions guided by end-to-end performance approximations.

Our prototype is accurate despite estimating byte latency instead of request-response latency, as the workload solely consists of fixed-size SET requests triggering fixed-size success responses, tightly correlating with bytes. The difference is simply a matter of scaling by a constant.

Our prototype is less effective with heterogeneous workloads as shown in Figure 4b, which shows what happens if the SET-GET ratio changes from 100:0 to 95:5 percent. Now, 5% of the responses are large (16 KiB) and thus unharmed by Nagle batching under low load. And since the size of each GET response is roughly 34x larger than the combined size of 95 SET responses, our byte-based approximation incorrectly estimates that the latency of most "requests" is unaffected by Nagle batching in low load. Tracking system calls or getting hints from applications is therefore preferable (§3.3).

5 CHALLENGES AND FUTURE WORK

Metadata Exchange. As opposed to our offline prototype, our future implementation will exchange metadata between peers via TCP options (standard header extension). We expect the overhead to be small as states are small (§3.2) and we need only copy the state upon arrival. We maintain two states per connection: previous and current. Still, fast-path parsing of headers speculates no header extensions [14, 25]. We will thus reduce the frequency of the exchange as needed—Little's law estimates remain accurate regardless. In fact, instead of using some fixed exchange interval, we can do it on-demand.

Dynamic Toggling. The effect of enabling or disabling batching is unknown until tried, creating a classic exploration-exploitation tradeoff [5, 28], and so the system must occasionally try the other mode to decide which is better. We speculate a light method like ϵ-greedy [45] will suffice; an overly heavy approach might nullify the benefit of batching.

Orthogonally, because we simultaneously optimize potentially conflicting metrics—throughput and latency—toggling should ideally follow some system- or user-defined policy that balances between them, such as preferring latency, or maximizing throughput provided some latency SLO is met.

Toggling Granularity. The decision to turn batching on or off inevitably occurs at some granularity. Finer granularities offer faster reaction. Coarser granularities are less sensitive to noise. Exponentially weighted moving averages have been used to smooth such noise in dynamic environments [2, 50] and can be computed online with low overhead [19]. Our initial results suggest that a granularity of a kernel tick may be suitable, but this must be evaluated experimentally, like all other aspects discussed in this section.

Better Batching Heuristics. We theorize that knowing end-to-end performance can be used not just to enable/disable existing ad-hoc policies (§2) but also to replace them with a more principled approach that gradually adjusts batching limits based on observed performance, using algorithms such as AIMD, which has been successfully utilized in the past to adapt to changing network conditions [10, 24].

6 CONCLUSIONS

The success of batching depends on end-to-end performance. We contend that it may be feasible to enlighten batching with a reasonable approximation of this information and thus achieve a better outcome.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Shivaram Venkatraman, for their helpful feedback.

REFERENCES

[1] Apache thrift: A framework for scalable cross-language services development. https://thrift.apache.org. Accessed: 2025-04-19.

[2] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer Dy, Javed Aslam, and David Kaeli. Virtual machine monitor-based lightweight intrusion detection. *ACM SIGOPS Operating Systems Review (OSR)*, 45(2):38–53, 2011. https://doi.org/10.1145/2007183.2007189.

[3] Intiyaz Basha, Satanand Burla, Felix Manlunas, and David S. Miller. liquidio: xmit_more support. https://github.com/torvalds/linux/commit/c859e21a35ce5604dde0b618169680aa3c7e3bdb, October 2017. Accessed: 2025-04-19.

[4] 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 Symposium on Operating Systems Design & Implementation (OSDI)*, pages 49–65, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[5] Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. The exploration-exploitation dilemma: a multidisciplinary framework. *PloS one*, 9(4):e95693, 2014. https://doi.org/10.1371/journal.pone.0095693.

[6] Robert T. Braden. RFC 1122: Requirements for internet hosts – communication layers, October 1989. https://doi.org/10.17487/RFC1122.

[7] Marc Brooker. It's always TCP_NODELAY. Every damn time. https://brooker.co.za/blog/2024/05/09/nagle.html, May 2024. Accessed: 2025-04-19.

[8] Jesper Dangaard Brouer. Achievement unlocked, 10Gbps full TX wirespeed smallest packet size on a single CPU core. https://netoptimizer.blogspot.com/2014/10/unlocked-10gbps-tx-wirespeed-smallest.html, 2014. Accessed: 2025-04-19.

[9] Stuart Cheshire. TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK. http://www.stuartcheshire.org/papers/NagleDelayedAck/, 2005. Accessed: 2025-04-19.

[10] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989. https://doi.org/10.1016/0169-7552(89)90019-6.

[11] Jonathan Corbet. TCP Segmentation Offloading (TSO). https://lwn.net/Articles/9123, 2002. Accessed: 2025-04-19.

[12] Jonathan Corbet. Bulk network packet transmission. https://lwn.net/Articles/615238/, 2014. Accessed: 2025-04-19.

[13] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, 2015. https://doi.org/10.1145/2815400.2815409.

[14] Linux Kernel Developers. TCP fast path. https://www.kernel.org/doc/html/latest/networking/snmp_counter.html#tcp-fast-path. Linux Kernel Documentation. Accessed: 2025-04-19.

[15] Linux Kernel Developers. Timestamping. https://docs.kernel.org/networking/timestamping.html. Linux Kernel Documentation. Accessed: 2025-04-19.

[16] Eric Dumazet. [PATCH net-next] tcp: auto corking. https://lore.kernel.org/all/1386311765.30495.246.camel@edumazet-glaptop2.roam.corp.google.com/, December 2013. Linux Kernel Mailing List. Accessed: 2025-04-19.

[17] Eric Dumazet. net/mlx4_en: use __netdev_tx_sent_queue(). https://lore.kernel.org/all/20181031153914.132127-4-edumazet@google.com/, October 2018. Linux Kernel Mailing List. Accessed: 2025-04-19.

[18] Alexander Duyck. Segmentation offloads in the Linux networking stack. https://docs.kernel.org/networking/segmentation-offloads.html. Linux Kernel Documentation. Accessed: 2025-04-19.

[19] Tony Finch. Incremental calculation of weighted mean and variance. Tony Finch's ex-work homepage. https://fanf2.user.srcf.net/hermes/doc/antiforgery/stats.pdf, February 2009. Accessed: 2025-04-19.

[20] Thomas Glanzmann and Eric Dumazet. TCP auto corking slows down iSCSI file system creation by factor of 70. https://lore.kernel.org/netdev/1391867614.10160.89.camel@edumazet-glaptop2.roam.corp.google.com/, February 2014. Linux Kernel Mailing List. Accessed: 2025-04-19.

[21] gRPC: A high performance, open source universal RPC framework. https://grpc.io. Accessed: 2025-04-19.

[22] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: handling throughput variability for hypervisor IO scheduling. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 437–450, USA, 2010. https://www.usenix.org/conference/osdi10/mclock-handling-throughput-variability-hypervisor-io-scheduling.

[23] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 135–148, October 2012. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[24] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 314–329, 1988. https://doi.org/10.1145/52324.52356.

[25] Van Jacobson. TCP in 30 instructions. https://www.pdl.cmu.edu/mailinglists/ips/mail/msg00133.html, September 1993. IP Storage (IPS) Mailing List. Accessed: 2025-04-19.

[26] Eun Young 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 Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[27] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 2–7, 1987. https://doi.org/10.1145/55482.55484.

[28] Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987. https://www.jstor.org/stable/3689689.

[29] Jacob E. Keller. [RFC PATCH] net: limit maximum number of packets to mark with xmit_more. https://lore.kernel.org/all/20170825152449.29790-1-jacob.e.keller@intel.com/#t, August 2017. Linux Kernel Mailing List. Accessed: 2025-04-19.

[30] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *USENIX Annual Technical Conference (ATC)*, pages 881–896, 2019. https://www.usenix.org/conference/atc19/presentation/kogias-lancet.

[31] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM Conference on Applications,*

Technologies, Architectures, and Protocols for Computer Communication, pages 514–528. ACM, 2020. https://doi.org/10.1145/3387514.3406591.

[32] send(2) - Linux manual page. https://man7.org/linux/man-pages/man2/send.2.html. Accessed: 2025-04-19.

[33] John D. C. Little. A proof for the queuing formula: L = λw. *Oper. Res.*, 9(3):383–387, June 1961. https://doi.org/10.1287/opre.9.3.383.

[34] Mao Miao, Wenxue Cheng, Fengyuan Ren, and Jing Xie. Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 726–733, 2016. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0106.

[35] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application performance pitfalls and TCP's Nagle algorithm. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 27(4):36–44, March 2000. https://doi.org/10.1145/346000.346012.

[36] J. C. Mogul and G. Minshall. Rethinking the TCP Nagle algorithm. *ACM SIGCOMM Computer Communication Review (CCR)*, 31(1):6–20, January 2001. https://doi.org/10.1145/382176.382177.

[37] John Nagle. Congestion control in IP/TCP internetworks. *ACM SIGCOMM Computer Communication Review (CCR)*, 14(4):11–17, 1984. https://doi.org/10.1145/1024908.1024910.

[38] Peter Okelmann, Leonardo Linguaglossa, Fabien Geyer, Paul Emmerich, and Georg Carle. Adaptive batching for fast packet processing in software routers using machine learning. In *IEEE International Conference on Network Softwarization (NetSoft)*, pages 206–210. IEEE, 2021. https://doi.org/10.1109/NetSoft51509.2021.9492668.

[39] J. Postel. RFC 791: Internet Protocol, September 1981. https://doi.org/10.17487/RFC0791.

[40] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017. https://doi.org/10.1145/3132747.3132780.

[41] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, and Amin Vahdat. Fathom: Understanding Datacenter Application Network Performance. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 394–405. ACM, September 2023. https://doi.org/10.1145/3603269.3604815.

[42] Redis Ltd. Redis: The real-time data platform. https://redis.io, 2025. Accessed: 2025-04-19.

[43] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[44] Job Snijders and Peter N. M. Hansteen. Demise of Nagle's algorithm (RFC 896 - congestion control) predicted via sysctl. https://undeadly.org/cgi?action=article;sid=20240514075024, May 2024. OpenBSD Journal. Accessed: 2025-04-19.

[45] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[46] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 129–145, July 2021. https://www.usenix.org/conference/osdi21/presentation/tai.

[47] Turn off TCP segmentation. Volt Archive Data Administrator's Guide (V14). https://docs.voltdb.com/AdminGuide/adminservertcpseg.php, 2024. Accessed: 2025-04-19.

[48] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–94, 2024. https://doi.org/10.1145/3694715.3695968.

[49] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. Stackmap: Low-latency networking with the OS stack and dedicated NICs. In *USENIX Annual Technical Conference (ATC)*, pages 43–56, 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

[50] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. Managing memory tiers with CXL in virtualized environments. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 37–56, 2024. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong.