Cooperative Concurrency Control for Write-Intensive Key-Value Workloads

Mark Sutherland
EcoCloud, EPFL
Lausanne, Switzerland
mark.sutherland@alumni.epfl.ch

Babak Falsafi
EcoCloud, EPFL
Lausanne, Switzerland
babak.falsafi@epfl.ch

Alexandros Daglis
Georgia Institute of Technology
Atlanta, Georgia, USA
alexandros.daglis@cc.gatech.edu

ABSTRACT

Key-Value Stores (KVS) are foundational infrastructure components for online services. Due to their latency-critical nature, today's best-performing KVS contain a plethora of full-stack optimizations commonly targeting read-mostly, popularity-skewed workloads. Motivated by production studies showing the increased prevalence of write-intensive workloads, we break down the KVS workload space into four distinct classes, and argue that current designs are only sufficient for two of them. The reason is that KVS concurrency control protocols expose a fundamental tradeoff: avoiding synchronization by partitioning writes across threads is mandatory for high throughput, but necessarily creates load imbalance that grows with core count and write fraction. We break this tradeoff with C-4, a co-design between NIC hardware and KVS software that judiciously separates write requests into two classes: independent ones that can be balanced across threads, and dependent ones which must be queued. C-4 dynamically partitions independent writes with the NIC to increase the load balancing flexibility of current KVS designs, and adds a software layer to the KVS to compact dependent writes into batches. Our evaluation shows that for write-intensive workloads, C-4 reduces 99th% tail latency by $1.3 - 5\times$ and improves throughput by up to $1.7\times$.

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Computer systems organization** → **Multicore architectures**; *Client-server architectures*; • **Networks** → *Network servers.*

KEYWORDS

key-value stores, load balancing, NIC architecture, tail latency, synchronization, concurrency, linearizability

ACM Reference Format:
Mark Sutherland, Babak Falsafi, and Alexandros Daglis. 2023. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3567955.3567957

1 INTRODUCTION

Key-Value Stores (KVS) are a backbone for online services because of the broad applicability of their elegant interface. Due to their widespread use, production KVS must meet extremely demanding performance constraints, because their latency directly impacts the response time of user queries that must meet real-time interactivity requirements. This combination of ubiquity and stringent performance needs has driven major research and development efforts to optimize underlying system layers [7, 54, 61, 71], or even employ dedicated hardware [44, 58] to boost KVS performance.

Despite the expansion of KVS into new deployment contexts [11], today's state-of-the-art KVS are still architected for yesterday's workloads. Emerging use cases such as distributed machine learning [90], publish-subscribe systems [1], or message queues [82] often have drastically different characteristics than canonical read-dominated workloads [11, 90]. To quantify, recent studies of production KVS traces reveal that >30% of workloads at Twitter have more writes than reads [90]. Furthermore, the popularity skew between items is drastically greater than typically assumed, with 50% of workloads exhibiting data popularity skew coefficients between 1.4–2.5 [90], compared to the vast majority of prior work considering coefficients near unity [12, 21, 58, 60, 61, 75].

The diversification of KVS workloads has direct performance implications for server systems: today's KVS designs drastically and necessarily under-utilize server hardware due to load imbalance between threads. The underlying cause for such imbalance is that current KVS statically partition writes among threads, a necessary design decision for high throughput when deploying KVS on manycore servers [60, 61, 91]. Write partitioning precludes any load-balancing framework from operating on writes by design, which we show can result in up to $1.7\times$ throughput loss, or up to $5\times$ higher tail latency than what should be possible. Therefore, we conclude that in order to justify the steep cost of keeping data in memory [25] and fully utilize the capabilities of powerful server hardware, it is necessary to revisit today's KVS designs to effectively handle write-intensive workloads.

We analyze these emerging workloads via a KVS workload taxonomy, and isolate the following two distinct yet common workloads that create load imbalance. Workloads having light popularity skew and large fractions of write requests suffer increased tail latency because of transient queueing events created by the inability to load-balance writes. In contrast, under extreme popularity skew, the thread handling writes to the hottest data item is overloaded and caps the KVS' throughput despite other threads being idle.

We posit that addressing both problematic workload characteristics requires the system's load balancer to distinguish independent from dependent writes, and make decisions accordingly. When the workload has many independent writes, static partitioning can be relaxed because true write conflicts are exceedingly rare. In contrast, when extreme skew creates frequent dependent writes, foregoing load balancing and assigning all the writes to a single thread implicitly enables that thread to perform a single batched update comprised of the many dependent writes.

Following these insights, we introduce **C-4**, a **C**ooperative **C**oncurrency **C**ontrol **C**o-design of NIC hardware and KVS software that breaks the tradeoff between write partitioning and load imbalance. C-4's NIC hardware additions apply inter-thread load balancing for independent writes, while still allowing threads to elide expensive locking primitives. When a thread handles multiple dependent writes, C-4 compacts these writes in software into a single batched update, boosting all threads' throughputs via improved locality and fewer reader-writer interactions. C-4 performs both such operations while preserving strong consistency guarantees. To summarize, we make the following contributions:

- We construct a KVS workload taxonomy comprising four distinct regions, and show that existing designs only sufficiently accommodate half of the workload space.
- We show that the common issue for both problematic regions is a concurrency control policy relying on static write partitioning. Using discrete-event simulation, we show such policies result in up to 5× worse tail latency, and up to 45% reduced throughput on modern manycore CPUs.
- We introduce C-4, a NIC-software co-design, to address both problematic workload regions. C-4 outperforms the state-of-the-art KVS design by reducing 99th percentile latency by 1.3 – 5× and increasing throughput by 1.3 – 1.7×.

We begin by presenting our motivation and KVS workload taxonomy in Sec. 2. Then, Sec. 3 quantifies the impacts of the tradeoff between write partitioning and load imbalance. Sec. 4 and Sec. 5 introduce C-4's design and implementation. We describe our methodology in Sec. 6 and evaluate C-4 in Sec. 7. We discuss related work in Sec. 8 and conclude in Sec. 9.

2 MOTIVATION

In today's datacenters, KVS have transcended their traditional use-cases such as in-memory caching of disk-resident data [10, 29, 73]. Due to the fact that online services demand near real-time response latencies, KVS must operate with stringent limits on the latency of their *slowest* responses, known as the "tail latency" [18]. Therefore, it is logical for KVS to adopt state-of-the-art proposals promising reduced network delay, kernel-bypass protocols, and first-class load balancing [7, 17, 54, 71, 83].

Write-intensive workloads are a critical open challenge in KVS research, because state-of-the-art KVS systems are often expressly designed around read-mostly workloads with considerable item popularity skew [44, 58, 60, 61, 75]. Emerging studies of production KVS data have revealed the prevalence of workloads that are more write-intensive and/or more severely skewed than previously assumed [11, 90], leading to a natural question: what bottlenecks prevent today's KVS from performing ideally on such workloads?

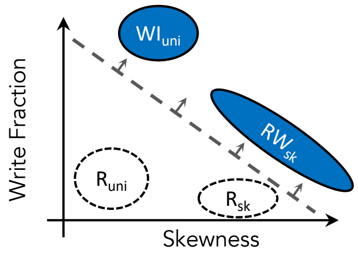


Figure 1: KVS workload taxonomy, with the blue regions targeted by C-4.

2.1 KVS Workload Taxonomy

A given KVS workload can be represented by its location on the cross-product of two axes representing its skewness and write fraction—knowing a workload's location is enough to intuitively describe its characteristics and associated bottlenecks. Fig. 1 shows these two axes and identifies four regions that serve as a "spanning set" of the space. Regions are identified using the nomenclature $\{R|RW|WI\}_{(sk|uni)}$, where the leading letter(s) represent(s) whether the workload is read-mostly, read-write, or write-intensive, and the subscript represents whether the data popularity distribution is skewed or pseudo-uniform.

The two regions lying beneath Fig. 1's dashed line are well-studied in KVS literature. R_{uni} workloads are the least challenging—providing a high-throughput KVS is possible with existing systems such as `memcached` [69]. Adding popularity skew moves the workload to the R_{sk} region, where performance is primarily determined by the maximum throughput to read the hottest items. Absorbing the workload's skew requires the KVS to use a concurrency control policy supporting *concurrent lock-free readers*, such as MICA [60, 61], Masstree [64], RackOut [75], KV-Direct [58], and ccKVS [31].

In contrast, workloads above the dashed line are largely unaddressed by current research. The fundamental challenge in both workload regions is that the increased prevalence of writes creates a tradeoff between high system throughput and load imbalance across threads. When writes are plentiful, achieving high throughput mandates a concurrency control policy that partitions writes across threads to avoid synchronization overheads, known as Concurrent Read, Exclusive Write (CREW [61]). However, such partitioning necessarily creates load imbalance that leads to either increased tail latency or throughput bottlenecks.

WI_{uni} workloads (e.g., those with ≥ 50% writes) will experience inflated tail latencies on today's KVS' simply because write partitioning precludes any load balancing framework [17, 54, 83] from acting on 50% or more of the requests. A recent characterization from Twitter has shown that more than 35% of their production workloads can be classified as WI_{uni} [90], motivating additional work to handle them with improved tail latency. Facebook also reports the prevalence of extremely write-dominated workloads (i.e., having 92% writes) that store ML statistics [11, §4.1].

RW_{sk} workloads also lack write load balancing, and face two additional challenges. First, the write load on a small group of items can be high enough to overwhelm the hottest thread. Although this challenge has been identified in prior work, the issue has been understood to apply to RW_{sk} workloads with $\sim$ 50% writes [44, 60, 75]. However, in light of recent work showing that skew coefficients are far higher than previously studied [90], static write imbalance becomes a far more common bottleneck. We demonstrate that such imbalance can occur even with single-digit write fractions. Second, the workload's read-write nature creates high contention on the hottest cache lines, further decreasing per-thread throughput.

We specifically identify concurrency control policies using static write partitioning as the underlying cause of both problems. A theoretically optimal system would allow any thread to serve reads and writes, leading to completely balanced load. However, enabling concurrent writers has been shown to reduce KVS throughput by 2× due to the need for atomic instructions for synchronization, and assumed costs in cache coherence [61].

Furthermore, although emerging frameworks exist that would enable write load balancing [53, 66], they are unlikely to yield benefits for in-memory KVS with µs-scale service times. Despite drastically improving scalability, these frameworks' use of software transactional memory techniques (e.g., version-chaining and garbage collection) translate to limited performance gains because of the additional overheads from version management. We confirmed this by porting a hash table-based KVS to the Multi-version Read-Log-Update (MV-RLU) framework [53], and observed that read and write latencies increased by 1.75× and $2-40\times$ respectively. Therefore, instead of pursuing write concurrency, we contribute two insights allowing write partitioning to serve as an enabling factor rather than an impediment.

Insight 1: Static write partitioning is overly conservative, and can be relaxed in the vast majority of cases. The only time write partitioning is strictly mandatory is when two writes to the same partition are outstanding at the same time—there is no reason for a write to partition P_X to queue behind one to P_Y when $X \neq Y$. Load balancing of independent writes would allow WI_{uni} workloads to have nearly identical tail latency to R_{uni} ones.

Insight 2: Write partitioning naturally creates batches of queued writes that can be compacted into a single update. In RW_{sk} workloads at high load, mandatory write partitioning is the common case, creating an optimization opportunity at the writer: all queued writes can be compacted into a single update, reducing reader-writer synchronization and cache line access latency. Write compaction has been previously applied to kernel data structures [9], LSM-based KVS [2], and commutative transactions [72], but never in KVS with µs-scale tail-latency Service Level Objectives (SLOs).

By leveraging these two insights, C-4 is the first mechanism that enables KVS to effectively handle write-heavy workloads. As top-performing in-memory KVS use some form of static write partitioning [12, 21, 31, 60, 61, 75, 91], C-4 is applicable to all such designs. C-4's performance improvements reduce the resources required for a given load (i.e., fewer servers and/or allocated cores), and therefore grant significant cost savings in the datacenter because thousands of servers are dedicated to KVS in production [73, 90]. Having described the challenges created by common concurrency control choices, we now present a high-level model estimating the benefits of improved concurrency control for emerging workloads.

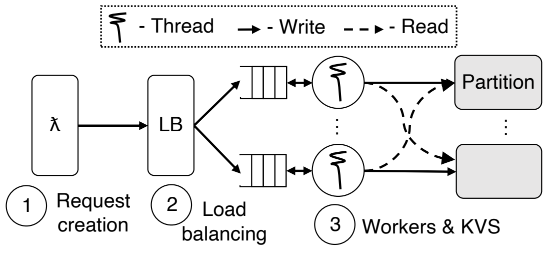


Figure 2: Queueing model for a server running a KVS.

3 KVS CONCURRENCY CONTROL MODEL

To quantify concurrency control's performance impacts, we use discrete-event simulation of a queueing system modelling a many-core server running a KVS, whose design is shown in Fig. 2. In Step ①, the load generator component creates new requests according to a Poisson process, with configurable inter-arrival times. We model workloads from Fig. 1's regions by varying the percentage of writes and the popularity skew according to a Zipfian distribution with configurable γ, as is standard in KVS literature [58, 60, 61, 75].

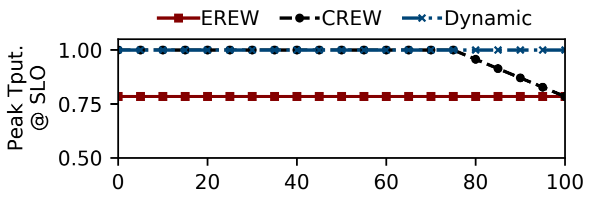
Step ② models a load balancer (LB) component which assigns requests to threads. If the KVS' concurrency control policy allows load balancing for this request type (e.g., for reads in CREW), the LB assigns the request to a thread according to the JBSQ(2) policy, which approximates a single-queue system [54]. If not, the LB assigns the request to a thread by statically hashing the key.

Step ③ concerns the worker threads and the KVS itself. Our KVS model targets a system deployed on a shared-memory multiprocessor, using any memory consistency model desired. We model the KVS as a set of abstract "partitions", each containing a version number that is used for reader-writer synchronization with optimistic concurrency control. Writers atomically increment the partition's version at the beginning and end of each update, and readers retry requests when their version checks fail. Our analysis applies regardless of memory consistency model or reader-writer channel. Each request's service time is modelled as $\bar{S} = T_{kvs} + T_{fixed}$, where T_{kvs} represents the KVS' service time, and T_{fixed} represents interacting with the load balancer and network stack. We choose T_{kvs} as uniformly distributed in the interval $[400, 800]ns$ to model in-memory datastores, and $T_{fixed} = 100ns$ to model a hardware-terminated protocol. See Sec. 6 for methodology details.

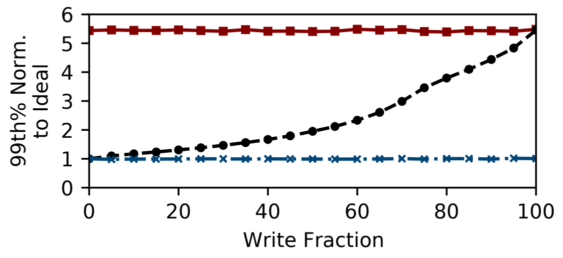
3.1 Dynamic Write Partitioning

To demonstrate the effects of Insight 1 for WI_{uni} workloads, we parameterize the model with 64 worker threads, a uniform key popularity distribution, and vary the write fraction f_{wr} from $0-100\%$. Borrowing terminology from MICA [61], we study three concurrency control policies:

- *EREW*: A fully partitioned system without load balancing. All requests are directed with static key-to-partition hashing.
- *CREW*: Allows concurrent readers and hence load balancing of reads, but uses partitioned writes. The state-of-the-art for in-memory KVS [58, 60, 61, 75].
- *Dynamic*: Maintains concurrent readers, and only treats a partition P_X as being in "exclusive write" mode as long as a



(a) Throughput under SLO.



(b) Tail latency.

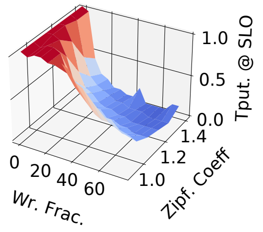
Figure 3: Throughput and tail latency of modelled concurrency control policies with a WI_{uni} workload.

write is still outstanding to it. After the write is completed, future writes to the same partition can be served by any thread. Allows load balancing of independent writes disallowed by EREW and CREW.

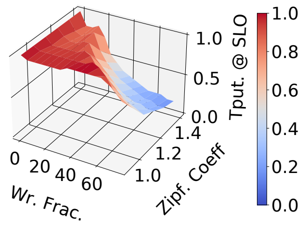
Fig. 3a shows the maximum throughput each policy can attain under SLO, and Fig. 3b shows the resulting excess 99th% latency compared to an ideal system that allows concurrent reads and writes without any synchronization overheads. *In the rest of the paper, any mention to SLO refers to a 99th% latency target of* $10 \times \bar{S}$. Each point in Fig. 3b is calculated by taking the maximum throughput under SLO of that policy, and normalizing its 99th% to the ideal system at the same load. We use this methodology to not artificially penalize policies that reach lower throughput than the ideal system.

The additional load balancing flexibility granted by the CREW and Dynamic policies impacts both achievable throughput and tail latency. Fig. 3a shows EREW's inability to balance load leads to saturation at 75% of the ideal system's throughput, regardless of f_{wr}. By allowing concurrent reads, CREW matches the ideal system's throughput for $f_{wr} < 75\%$. As f_{wr} approaches 100, CREW's performance converges to EREW due to write request queueing. Thus, even with CREW concurrency control, the KVS must be run at lower load to control tail latency. The Dynamic policy is not constrained by this tradeoff, and matches the performance of the ideal system regardless of f_{wr}.

Fig. 3b shows that CREW's tail latency matches the ideal system for a read-only workload, and increases proportionally with f_{wr}. For WI_{uni} workloads with $\geq 50\%$ writes, the static write partitioning of CREW and EREW inflates tail latency by $2 - 5.5\times$. In contrast, the Dynamic policy delivers near-identical tail latency to the ideal system because of its improved load balancing ability. Our model shows that a dynamic partitioning policy following Insight 1 can deliver the tail latency of an unattainable ideal system with full concurrency and no synchronization overheads.



(a) CREW baseline.



(b) Compaction enabled.

Figure 4: Modelled throughput of CREW and compaction-enabled KVS under RW_{sk} workloads.

3.2 Write Compaction

To show the impact of static write imbalance for RW_{sk} workloads and the benefit of write compaction (Insight 2), we enhance step ③ of our queueing system as follows: i) upon receiving a new write request, a worker scans its request queue for writes to the same key, ii) if found, the writes are buffered into a private log, and iii) the worker sets a timer to begin a "compaction window". When the timer exceeds the KVS' specified SLO, the worker closes the compaction window and sends replies to each compacted request. We defer discussion of the consistency implications to Sec. 4.3.1.

While a compaction window is open, any writes that are collected in a worker's private log have service time $\bar{S}_{comp} = T_{fixed} + T_{comp}$, where $T_{comp} = 100ns$. We set T_{comp} by measuring the latency of adding a small `struct` representing the compacted request to a pre-sized `std::vector` on a Xeon E5-2680 CPU.

An RW_{sk} workload can arise from various parameter combinations in Fig. 1's design space, as long as the write load on the hottest partition(s) is approximately equal to $1/N$ where N is the number of workers. Due to the fact that higher write load implies greater compaction opportunity, we study the entire design space defined by $\gamma \in [0.9, 1.4]$ and $f_{wr} \in [0, 80]$.

Fig. 4a shows the throughput under SLO of a CREW KVS, normalized to the same ideal system introduced in Sec. 3.1. Our results show that write partitioning becomes a clear bottleneck for the vast majority of workloads making up the RW_{sk} region, because the heavy skew simply overwhelms the KVS' hottest worker with writes. For example, a workload with $(\gamma, f_{wr}) = (0.99, 35\%)$ only attains 56% of its ideal maximum throughput. Our model shows that, for the highest skews, write partitioning leads to throughput bottlenecks even with f_{wr} in the low single digits. For example, our model shows that the KVS only attains 66% of its maximum theoretically sustainable load for $(\gamma, f_{wr}) = (1.4, 5\%)$. In comparison, Fig. 4b shows the throughput of a KVS with write compaction. Compaction allows a workload with $\gamma = 0.99$ to match the ideal system's performance up to $f_{wr} = 55\%$, and provides a $1.56\times$ speedup for a workload with $(\gamma, f_{wr}) = (1.4, 5\%)$.

The above opportunities are in fact conservative, because our queueing system does not consider cache or memory access times. In a real deployment, the CREW baseline would experience significant cache contention on the hottest partition(s), degrading performance faster than shown in Fig. 4a and granting compaction even greater relative improvements.

Summary. Our models show evidence that write partitioning is not necessarily a bottleneck for KVS performance. Following Insight 1, dynamic write partitioning makes it possible for WI_{uni} workloads to attain near-ideal tail latency. Furthermore, following Insight 2, write compaction can boost throughput under SLO by 1.5 – 2× for RW_{sk} workloads.

4 C-4 DESIGN

4.1 System Prerequisites

Our design is tailored to systems featuring hardware-offloaded protocol stacks offering RPC semantics to applications [17, 39, 54, 57, 86], because they are a better fit for low-latency transports [35, 71]. Hardware offload is mandatory to remove network and RPC processing overheads, which still cost 1–5µs per RPC [48, 65] and are prohibitive for µs-scale requests [39, 57, 78]. Additionally, our work requires NIC-driven load balancing [17, 52], which we extend to realize dynamic write partitioning.

4.2 Write Balancing for WI_{uni} Workloads

Sec. 3.1 demonstrates the tail latency benefits of a concurrency control policy that enables load balancing for writes, under the invariant that a partition can only be written by a single thread at a time. The load balancer can maintain this invariant using a set of ephemeral partition-to-thread mappings, and dispatch requests accordingly. Partitions with an active mapping are in "exclusive mode", and all writes to them must be sent to the mapped thread, whereas requests to other partitions can be freely load-balanced. We call this policy "dynamic Concurrent Read, Exclusive Write" (d-CREW).

Unlike CREW, d-CREW is a stateful policy, and should be implemented server-side rather than on the clients or in the network. CREW can be trivially implemented wherever the KVS' static partitioning function can be computed, such as on the clients [61] or in programmable networking hardware [54, 93]. However, d-CREW allows partition-to-thread assignments to frequently change, thus maintaining the single-writer invariant would require constantly updating the mappings at every entity making request-to-thread assignment decisions. As KVS often receive requests from thousands of concurrent clients [73, §3.1], broadcasting partition updates is clearly not scalable.

d-CREW could be implemented in a centralized in-network load balancer such as RackSched [93], removing broadcasts for partition updates. However, the increased distance between the load balancer and KVS threads implies that partitions remain in exclusive mode for longer than necessary, because responses to write requests must traverse the network before partition mappings are released. As prior work has demonstrated that tail latencies increase with load balancer-to-server distance [54], we conclude that server-side d-CREW will yield superior performance to an in-network design.

Fig. 5 displays our proposed design to realize d-CREW in the NIC, by showing two writes W_1 and W_2 that target KVS partitions P_1 and P_N. We assume CREW's static partitioning maps both partitions to thread T_1, meaning the load balancer must assign both W_1 and W_2 to thread T_1. In contrast, d-CREW allows the requests to be balanced across threads T_1 and T_2, as long as the load balancer has knowledge of the partition being accessed by each write request.

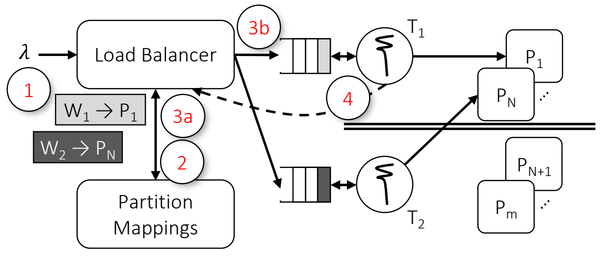


Figure 5: NIC additions for Dynamic CREW (d-CREW).

NIC extensions to extract application-level information (e.g., P4 [8] or FlexNIC [52]) are sufficient to enable d-CREW, and we expect them to be available in future products.

With d-CREW, the NIC in Fig. 5 unpacks each write's application-level header and determines the writes target partitions P_1 and P_N (Step ①). The load balancer then looks into its active partition-to-thread mappings for the requested partitions and finds no active match (Step ②), meaning that the NIC's load balancing policy can freely assign the requests to any thread. The NIC then creates two new partition-to-thread mappings for P_1 and P_N, placing them in exclusive mode (Step (3a)) and making them "sticky" to threads T_1 and T_2 as long as each write is outstanding. The writes are then sent to the threads (Step (3b)) for processing. When the threads send responses to the writes, the partitions are no longer in exclusive mode and the NIC can free the two respective partition mappings (Step ④), allowing incoming writes to P_1 and P_N to be assigned to any thread.

Introducing d-CREW into the NIC means that the state requirements for storing partition-to-thread mappings must be small enough to result in acceptable hardware provisioning, and low enough access times to meet the NIC's peak sustainable throughput. To illustrate, a server handling a workload with 75% writes at an aggressive 200$MRPS$, where each request takes 600ns (see Sec. 3), has roughly 90 writes outstanding at any given time. Even with over-provisioning to absorb transient fluctuations, storing a few hundred mappings only requires a few KBs, which today's NICs could easily accommodate.

Implementing "exact match" operations between incoming requests and existing mappings is more difficult, because it traditionally requires Content Addressable Memories (CAMs) [87]. However, our basic estimate above (confirmed in Sec. 7.1) reveals the number of mappings to be similar to the size of a CPU TLB. Therefore, we believe it is feasible to implement partition-mapping hardware using existing CAM technology. In the worst case where the system exhausts all the hardware's partition mappings, the drastic mismatch in request arrival and processing rates indicates a load spike requiring flow control mechanisms to throttle or drop requests [15, 85, 86].

4.3 Write Compaction for RW_{sk} Workloads

C-4's second mechanism, write compaction, addresses both bottlenecks in RW_{sk} workloads, namely: i) static load imbalance created by the combination of write fraction and skewness that overwhelms

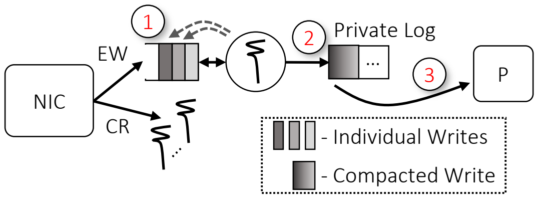


Figure 6: Design of write compaction in C-4.

a single writer, and ii) contention between writer and reader threads for the cache lines comprising the hottest partition(s). C-4 does so while maintaining the baseline KVS' consistency model.

To absorb the load created by highly skewed workloads, C-4 must achieve drastic speedups for the single thread serving the hottest partition. NetCache's authors similarly observed that a single unit serving requests on behalf of many workers must provide orders of magnitude higher throughput to handle skewed workloads [44, §2]. However, in our context the hottest thread is not required to handle both read and write load, because the NIC balances reads across all other threads. Therefore, the requisite speedup for the hottest thread drops significantly. Our design takes advantage of the NIC's ability to load-balance reads and independent writes, therefore the request queue of the hottest thread will be full of writes to *only* the hottest partition(s), creating opportunities for compacting many writes into a single batched update.

Intuitively, compacting a given write into a private batch is faster than writing into the KVS' underlying datastore, because it is unnecessary to write any shared cache lines. We estimate that the possible acceleration A of such compaction is:

$$A \simeq \frac{T_b + T_f}{\frac{T_b}{N} + T_c + T_f} \tag{1}$$

where T_b, T_f, T_c, and N represent the baseline service time, the fixed time to reply to a request (see Sec. 3), the time required to collect a single write into an ongoing batch, and the number of compacted requests, respectively. Using a software compaction microbenchmark, we measured values for each time parameter that indicate $A \simeq 4$ is attainable in an unloaded system. Furthermore, as read-write contention on hot partitions grows at high loads, the value of T_b will increase while T_c remains unaffected as the single thread dirties no shared cache lines as it gathers writes into a batch. Our evaluation studies these parameters in Sec. 7.2.

Fig. 6 shows the cooperation between the NIC and the KVS threads to achieve write compaction. The NIC's responsibility is to implement the d-CREW or CREW policy, spreading read load among all the threads and creating opportunities to compact dependent writes. When the KVS thread pulls a write request from its private queue, it scans a small number of extra queue slots to search for any matching writes to the same item (Step ①). If matches are found, the thread opens a new "compaction window", and collects all incoming writes to this item into a single update until the window closes (Step ②). Finally, the thread performs the single combined write to the KVS' data store (Step ③) and sends all the responses to the compacted writes (not shown). As such a technique requires only software modifications, implementations are possible on today's NICs that can enforce CREW.

The length of the compaction window permitted defines the achievable acceleration—the higher the number of compacted writes N, the greater the value of A. We observe that KVS' tail latency performance constraint creates an opportunity to finish all the compacted writes "just in time" before the SLO expires. Even with a stringent SLO of 10× the average service time, it is possible to compact upwards of 10 writes into a single window because the compacting thread is significantly accelerated. However, introducing compaction implies that all the writes are now invisible to the readers until the window closes and the final update is applied. We now discuss how C-4 maintains strong consistency guarantees in the presence of write compaction.

4.3.1 Maintaining Consistency Under Compaction. We target a baseline KVS providing *linearizability*, a strong consistency guarantee that requires a total write order and that the system's behavior respects the real-time ordering of requests [37]. Note that linearizability is a *local* property, which only applies to single objects (in our context, keys) in isolation. We do not consider consistency across multiple keys in this work for two reasons. First, the vast majority of KVS do not support multi-key updates [31, 58, 60, 61, 64, 75, 91], following the system design principle to "make it fast, rather than general or powerful" [56]. Such tasks are better implemented in higher layers such as transaction lock managers, which have their own set of related but different consistency models. Indeed, attaining atomically visible multi-key updates requires either serializability or *strict* serializability. We argue that because C-4 maintains linearizability, it is an equivalent candidate to an existing KVS for inclusion in a transactional system.

Fig. 7 displays three possible executions of a KVS with write compaction that demonstrate our key insight necessary to maintain linearizability. Each operation is expressed as $Op(Args) : Client$, where Op represents a KVS `get`, `set` or `response`, $Args$ contains the arguments passed to the KVS (e.g., the key being accessed, and its associated value for sets/responses), and $Client$ is the client thread requesting this operation from the KVS.

To argue that C-4 maintains linearizability, we focus on the specific scenario induced by the addition of write compaction—when a compaction window is open for key K with simultaneous outstanding read requests to K. Informally, a system's execution E can be *linearized* if it can be transformed into E' where two criteria are respected: (1) E' is sequential, meaning that each invocation is immediately followed by its corresponding response, and (2) the order of operations in E' respects all partial orderings in E [37]. A partial ordering is created when the response to an operation R_1 appears in the execution before another operation R_2's invocation—all operations that are not partially ordered are concurrent. Criterion (2) allows such concurrent operations to be re-ordered in E', because they are not constrained by orderings emanating from E.

To demonstrate, consider the execution E_1 in Fig. 7. In this execution, clients A and B both attempt to write key K with two different values, while client C reads K. E_1 is not linearizable because C's `get` could return the initial value $K = 0$ only if it executed *before* both A and B's `sets`; however, in E_1 C's `get` is partially ordered after A's `set`, therefore a linearizable system must return either

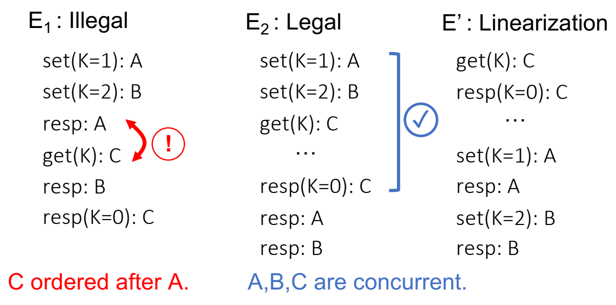


Figure 7: Executions showing linearizable compaction.

$K = 1$ or $K = 2$ to C. A naive write compaction implementation can result in the exact violation represented by E_1. If a thread opens a compaction window for K after A's `set` and responds to A, any future read operations will return the value in the KVS' underlying datastore, despite needing to return A's `set` value.

To preserve linearizability, C-4 delays responding to the writes in a compaction window until it closes, and therefore reads during the compaction window are considered concurrent and can legally return the value in the KVS' datastore. To illustrate, E_2 in Fig. 7 contains the same operations as E_1 with the response to A deferred until the compaction window closes. Therefore, C's `get` is concurrent with both A and B's `sets` and is not required to be ordered after either one. Then E' is a sequential execution meeting both criteria for linearizability: C's `get` executes "first", before A and B's `sets`, which close the compaction window and set $K = 2$. For reads that are ordered before or after a compaction window, C-4's design is no different from the baseline, so we omit these cases for brevity.

Summary. C-4 comprises two interrelated mechanisms to handle write-intensive workloads. d-CREW ensures that writes with true dependence (i.e., to the same partition) are the *only* requests which cannot be load balanced, and write compaction allows a thread to apply multiple dependent writes at once while maintaining consistency. In a sense, C-4 can be viewed as a hardware-supported delegation system [63, 84]; the NIC's support for d-CREW determines which threads currently have exclusive access to partitions, and "delegates" writes to those threads via its load balancing mechanism. Software then combines the dependent writes without the overhead of shuffling requests between threads in traditional delegation systems.

5 IMPLEMENTATION

We implement C-4 on top of the NeBuLa architecture [86], as it satisfies both design prerequisites in Sec. 4.1. Our implementation also applies to a variety of NIC architectures (e.g., the NanoPU [39] or the RISC-V RocketChip's integrated NIC [50]). NeBuLa's NIC is based on Scale-Out NUMA [74] enhanced with the NI_{split} architecture [16] to accelerate core-NIC interactions, which are particularly important for manycore servers.

Applications using NeBuLa preallocate a set of buffers for requests that are managed by the NIC and freed by the RPC layer, and schedule new RPCs or responses using memory-mapped Queue Pairs (QPs) similar to RDMA's Virtual Interface Architecture [23]. Upon RPC arrival, NeBuLa terminates the transport protocol and appends the RPC to a memory-mapped queue of RPCs to be assigned to threads. Threads signal their availability to NeBuLa after completing a previous request, triggering NeBuLa's load balancing stages to assign a new request based on the Join Bounded Shortest Queue (JBSQ) policy [54, 86]. Implementing C-4 requires hardware modifications to NeBuLa's load balancing stages, extensions to the NIC-software interface, and the software implementation of write compaction.

5.1 Interface Modifications

Both mechanisms comprising C-4 require the NIC-software interface to specify whether requests and their responses correspond to KVS reads or writes. The NIC requires such information for the d-CREW policy, to keep track of when a given partition enters and exits exclusive mode and update its set of partition-to-thread mappings. The KVS must communicate two additional pieces of information to the NIC to enable creation of exclusive mappings: first, how to identify the fields in the application header corresponding to the key and request type (i.e., read or write), and second, how to transform the application's key to a partition identifier.

The key and type field identification information (expressed as offsets and lengths within request packets' application-level headers), is communicated to the NIC during the NeBuLa stack's "setup phase" where the application's threads inform the NIC of their active queues and packet buffers using `ioctl` system calls. Our implementation works with a KVS that uses a simple fixed-format header where the offsets are known in advance, similar to prior work [48, 52, 61]. If the KVS is built on top of an RPC layer that uses different application and wire formats (e.g., Protocol Buffers [32]), extracting the correct offsets requires additional information for each application format. Prior work that proposes in-memory metadata for RPC data transformation [92] would be sufficient to enable C-4 in this context.

To convert the incoming request's key to a partition ID, C-4 assumes knowledge of the function $f()$ used by the KVS software to map keys into logical partitions. The granularity of partitions implicitly dictates C-4's load-balancing flexibility. Coarser-grained partitions create more load imbalance, because more unrelated keys are conservatively considered to be in exclusive mode [20]. C-4 chooses the $f()$ used by the KVS software to select hash buckets, meaning that the minimal load-balancing unit is a few tens of keys. C-4 communicates the number of buckets to the NIC during setup.

A partition mapped to a thread by C-4 can only be re-assigned after being released by the application. We therefore modify NeBuLa's RPC layer so that the function for sending an RPC response takes one more argument, indicating whether the application is releasing exclusive access. Our implementation *always* releases exclusive mappings upon completing a write to maximize load-balancing opportunities; retaining mappings longer than necessary to potentially increase locality is an interesting future direction.

5.2 Hardware Additions for Write Balancing

Fig. 8 shows the additions to NeBuLa's existing pipelines to implement C-4's d-CREW, using the example of two conflicting writes

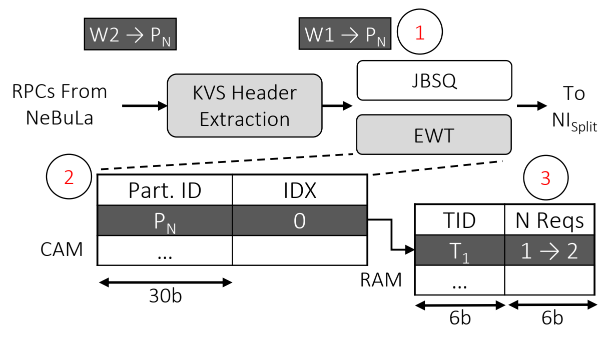


Figure 8: Modified NIC load balancing pipelines. Shaded portions are added in C-4.

that pass through the request assignment stage. All active mappings are stored in a small hardware table, the Exclusive Writer Table (EWT). Each active mapping in the EWT is associated with the thread holding this partition in exclusive mode, and a counter tracking the number of outstanding writes to this partition.

When W_1 arrives at the load-balancing stage (Step ①), dedicated logic extracts the KVS' headers to determine if this request is a write, and what partition must be looked up in the EWT. As there is no active entry for P_N, the partition is unassigned and C-4 assigns the request to T_1 using JBSQ. C-4 then allocates a new entry in the EWT for P_N, initializing its counter value to one outstanding write, and the owner thread to T_1 (Step ②). When W_2 arrives, its lookup of P_N hits the EWT and thus C-4 assigns W_2 to T_1, and increments the outstanding write counter (Step ③). After T_1 processes both writes, it releases exclusive access to P_N using the RPC response interface of Sec. 5.1. Each message triggers an access to the EWT for P_N to decrement the outstanding write counter (not shown). C-4 frees the EWT entry after W_2's response, enabling writes to P_N to be load balanced again.

Fig. 8 also shows the information stored in each field of the EWT. We provision a 30-bit partition ID and six bits for each of the outstanding request counter and thread identifier to support up to 64 threads, with each partition supporting up to 64 concurrently outstanding writes. To quantify the EWT's cost, we use CACTI 6.5 [59], configured with the following parameters: a 22nm process, built-in device projections, dynamic power optimization, and a 2GHz frequency (aggressively assuming a write balancing decision happens every 0.5ns). A 128-entry EWT with the partition and data widths above requires 0.004mm^2 of area and draws 6.85mW of dynamic power. In comparison, 64-core server chips can consume in excess of 280W [33], meaning that C-4's EWT imposes a negligible overhead of 0.002%, particularly because of the benefits in load balancing it enables.

The factor dictating the EWT's scale is the bandwidth-delay product between the load balancing hardware and the server's many cores. In future servers that could handle more outstanding writes concurrently, the EWT would need more entries as well as wider thread identifiers. More entries in the EWT would increase its power requirements accordingly with its CAM portion—in a deployment where power is prohibitive, any of the plethora of circuit techniques for low-power CAM devices could be applied [79]. The width of the request counter and thread identifiers can be increased without significant concerns for the EWT's cycle time or power, because they are both stored in direct-mapped RAM.

Although C-4 only uses its EWT for write balancing, its small CAM structure can be repurposed for other stateful per-RPC load balancing decisions, such as mapping specific *types* of RPCs to certain threads to increase instruction-cache locality [92], or to keep certain cores idle to ensure low tail latency [19]. The EWT can also implement software-installed packet filtering operations like those used by RSS and Flow Director [43], with the additional ability to create mappings on demand as traffic arrives.

Table 1: Parameters used for cycle-accurate simulation.

Cores	64× 4-wide issue OoO ARMv8 @ 2GHz 128-entry ROB, TSO
L1 Caches	64 KB 4-way L1-D and L1-I, 64 B blocks 2 ports, 32 MSHRs, 4-cycle lat. (tag+data)
LLC	Shared block-interleaved NUCA, 64 MB 16-way, 1 bank/tile, 11-cycle lat. (tag+data)
Coherence	Directory-based Non-Inclusive MESI
Memory	45 ns latency, 8×25.6 GB/s DDR4-3200
Interconnect	2D mesh, 16 B links, 3 cycles/hop

5.3 Software Support for Compaction

Our software implementation of write compaction is a single module residing between NeBuLa's RPC layer and the KVS itself. To allow this layer to scan incoming queues for compaction opportunities, we add an interface to NeBuLa's RPC layer allowing the caller to apply a lambda function on each valid incoming request in the queue, similar to BPF [67]. C-4 scans for compaction opportunities on every incoming write request. When an opportunity is detected, C-4 opens a compaction window, records the key being compacted, and begins buffering all incoming writes to it. Each compacted request buffers two pieces of information to allow C-4 to send a matching RPC response when the compaction window closes: i) the request's buffer address and size in memory, which NeBuLa's transport layer later reclaims, and ii) the sender's node ID to whom the response is sent.

To maintain the KVS' SLO guarantees, we use the processor's monotonic hardware clock, that has a known frequency and is readable from user mode with limited overhead [4, 40, 51]. Upon opening a compaction window, C-4 snapshots the current clock value and sets the window's expiration time as $T_{expiry} = T_{current} + \bar{S}(SLO-1)$, where we assume the SLO and average service time ($\bar{S}$) are known. C-4 checks the current clock value upon each request's completion to determine if the compaction window must be closed. When the expiration deadline is reached, C-4 applies the final compacted value to the datastore and then creates an RPC response for each of the buffered writes.

6 METHODOLOGY

System Organization. We evaluate C-4 with cycle-accurate full-system simulation of a 64-core processor running Ubuntu Linux

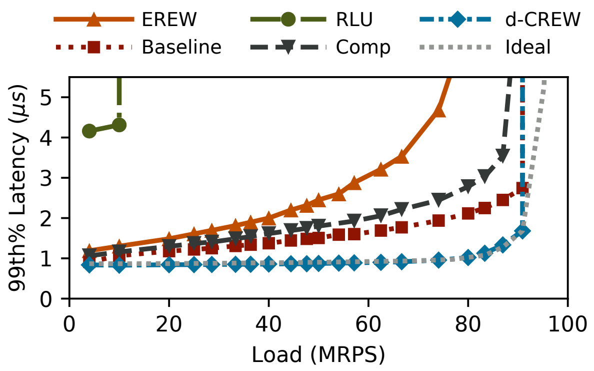


Figure 9: Comparison of KVS throughput under SLO, using uniform key popularity and f_{wr} = 50%.

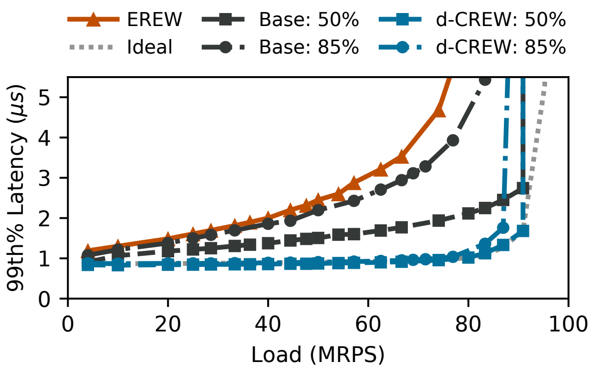


Figure 10: Comparison of KVS throughput under SLO with varied f_{wr}, using uniform key popularity.

18.04, using the QFlex simulator [80]. Sec. 6 summarizes our simulation parameters. Our manycore CPU is representative of today's server products such as Intel's Xeon Scalable [34, 49], AMD's EPYC [33], Amazon's Graviton [3, 88], and Huawei's Kunpeng [89]. C-4's performance benefits scale with core count, because load imbalance in a queueing system increases with more workers (cores).

We evaluate a NIC bandwidth up to 500Gbps, because our experiments showed C-4 was not able to reach its CPU IOPS bound before saturating NIC bandwidth for non-minimally sized KV items. Such per-server network bandwidth will be available in the near future, given the current availability of 400GigE products [76] and IO interconnect capabilities far exceeding NIC data rates (e.g., AMD EPYC 7763's 128× PCIe 4.0 lanes deliver 2Tbps of I/O bandwidth).

KVS Software. We use the MICA in-memory KVS [61], starting from the eRPC project's implementation [46, commit 1bfc7ec], porting it to NeBuLa and ARMv8 for simulator compatibility, and adding C-4's compaction layer, requiring only 105 lines of code. All experiments use a 64-thread instance with a 819MB dataset of 1.6M items, having 16B keys and 512B values in 1M hash buckets. We employ a workload generator which generates client requests with configurable rate, Zipfian popularity skew, and write fraction, using a Poisson arrival process. We measure all latencies server-side using cycle-accurate timestamps inside the simulator. Measurements begin when a request's first packet arrives at the NIC, and ends when its corresponding response's last packet leaves the NIC.

Evaluated Configurations. We study six different system configurations to show C-4's benefits:

(1) **Baseline**: The unmodified MICA KVS, running on the NeBuLa stack and using CREW concurrency control.
(2) **EREW**: Same as Baseline but using EREW.
(3) **Ideal**: Same as Baseline, but using a read-only workload, thus granting maximal load balancing flexibility with no reader-writer synchronization. Used as an upper performance bound for workloads containing writes.
(4) **RLU and MV-RLU**: Modifications of MICA that use the RLU and MV-RLU frameworks [53, 66] to provide concurrent readers *and* writers without read-side locking. RLU uses commit-deferral with degree 16, and MV-RLU adopts `ORDO` to remove global logical clock contention [51].
(5) **d-CREW**: Uses the same KVS as Baseline, and adds C-4's requisite extensions to support d-CREW. Targets WI_{uni} workloads.
(6) **Comp**: Enables C-4's software compaction support, while keeping static CREW concurrency control. Targets RW_{sk} workloads.

Performance Metrics. We evaluate C-4's performance in terms of maximum throughput under a 99th percentile latency SLO. We set a target SLO of 10× the average request service time, as is common in the literature [17, 45, 54, 83, 86]. As write compaction exposes a tradeoff between the duration of a compaction window and the performance benefit of C-4, we also show a slightly relaxed 20× SLO in the results for write compaction.

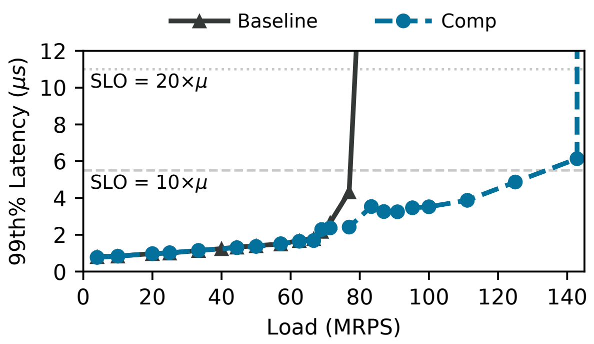
7 EVALUATION

We evaluate C-4 by focusing individually on the two challenging regions of the KVS workload space described in Sec. 2.1, and the C-4 mechanisms introduced for each one: d-CREW for WI_{uni} and write compaction for RW_{sk}. Therefore, we present experiments showing the impact of each technique on its respective region of applicability, and show that both techniques have limited impact outside their target region.

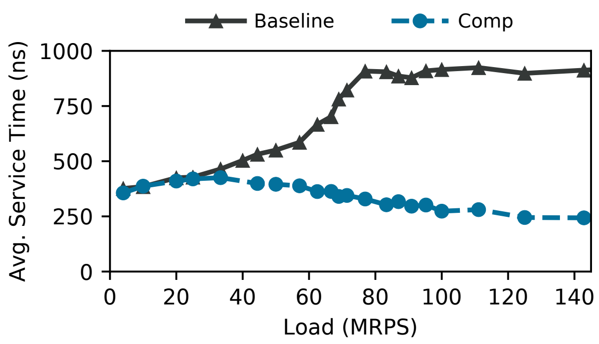
7.1 Dynamic Write Partitioning

We begin by evaluating the tail latency benefits of C-4's d-CREW policy. Fig. 9 compares all of Sec. 6's system configurations except MV-RLU, using a WI_{uni} workload with uniform key popularity and f_{wr} = 50%. Only d-CREW closely tracks Ideal up to 91MRPS, because it provides the greatest load balancing flexibility, only falling back to the baseline's behavior if there is a true write-write conflict. In contrast, all other systems incur excess queueing as load increases.

RLU can only support 10MRPS under a 10× SLO, due to its costly `commit` process to write-back logged values – writes that promote logs run for 10–20μs despite RLU's commit deferral, forming deep queues that lead to SLO violations. MV-RLU's series is not shown because it cannot even meet the 10× SLO at 4MRPS, the lowest load in Fig. 9. The reason for its seemingly poor performance is that versions are cleaned up using complex garbage collection, causing requests that block behind cleanup operations to stall for ~ 70μs.



(a) Throughput under SLO.



(b) Hottest thread's average service times.

Figure 11: System performance comparison for an RW_{sk} workload with $\gamma = 1.25$ and $f_{wr} = 5\%$.

However, MV-RLU's true multi-versioning shows benefits under a relaxed SLO of 100μs, attaining 2.5× higher throughput than RLU.

EREW fares better, delivering 76MRPS under SLO, but only reaches 80% of Ideal's throughput because it lacks load balancing support. Although EREW eschews all software synchronization, static partitioning results in excess queueing and sub-optimal performance compared to the systems supporting load balancing.

Our results confirm the intuition that write compaction is ineffective for WI_{uni} workloads. In fact, Comp performs 4MRPS worse than the baseline, because each thread incurs the additional cost of scanning its request queue for potential writes to compact, which is usually fruitless in WI_{uni} workloads. Such overheads grow at high loads—seen as a widening gap between the Comp and Baseline (CREW) series —due to more entries to scan in deeper queues. The best-performing systems, CREW and d-CREW, achieve the same throughput under SLO (91MRPS), with d-CREW achieving 1.3× 99th% latency reduction compared to CREW, by relaxing unnecessary queueing—essentially approaching the superior queueing model of an Ideal system without the associated costs of software synchronization. Our simulations closely track the expectations set by our queueing model in Sec. 3.1.

7.1.1 Increased Write Fractions. Fig. 10 shows the throughput under SLO of EREW, baseline CREW, and d-CREW, with a WI_{uni} workload as f_{wr} increases from 50% to 85%. We show a single line for EREW because it is insensitive to the workload's f_{wr}. As f_{wr} increases, the baseline policy loses load balancing potential and approaches EREW, creating increased 99th% latency and reduced throughput under SLO.

In contrast, d-CREW's benefits increase with higher f_{wr}. For 85% writes, baseline CREW cannot balance the vast majority of requests, leading to a reduced peak throughput of 83MRPS and 5× higher 99th% latency compared to Ideal, closely tracking our queueing model's prediction. In contrast, d-CREW nearly matches Ideal's performance until its CPU saturation point. Overall, d-CREW provides 87MRPS under SLO compared to CREW's 83MRPS, and 3.1× lower 99th% latency at a higher absolute load. At a load of 90MRPS, the Exclusive Writer Table (EWT) has an average of 30 active entries with $f_{wr} = 50\%$ and 52 with $f_{wr} = 85\%$, confirming our simple estimate in Sec. 5.2 for the EWT's limited size. The maximum EWT sizes were 64 and 90 for $f_{wr} = 50\%$ and 85% respectively.

7.2 Write Compaction

We now evaluate C-4's write compaction support, targeting RW_{sk} workloads. Fig. 11 compares baseline CREW and C-4 with write compaction enabled, for a highly skewed workload with $\gamma = 1.25$ and 5% writes. Despite the workload being read-dominated, Fig. 11a shows that the baseline saturates at 76MRPS, because a single thread is overloaded by writes to a single partition. In contrast, with write compaction, C-4 scales to 125MRPS under a 10× SLO, and 142MRPS with a relaxed 20× SLO.

To further explain these performance gains, Fig. 11b plots the average on-core service time of requests assigned to the hottest thread. As load increases, the baseline's service time grows exponentially because of increased time to invalidate and fetch cache lines for the partition's version number and corresponding data. At 76MRPS, the hottest thread's service time increases by 2.4× to 908ns, and remains roughly constant beyond 80MRPS because the network stack's flow control mechanism begins rejecting incoming requests as the KVS saturates. The same trend also occurs for the rest of the threads, whose average service time grows by 1.6× because the KVS' hottest items are repeatedly invalidated from their L1 caches due to frequent writes.

In contrast, C-4's write compaction policy inverts the aforementioned trend. As load increases, the hottest thread's service time decreases because compaction directly mitigates the overheads associated with contention between the single writer and multiple readers. Beyond 40MRPS, Fig. 11b shows that the hottest thread begins to open compaction windows and effectively batch writes, resulting in average service time reduction. Therefore, cache line contention does not increase despite higher offered load, allowing the thread's service time to drop to 243ns. Comparing these results to Eqn. (1)'s compaction performance model, we measured a 3.7× service time reduction for the hottest thread with write compaction, where our model predicts 3.9×. We attribute the difference to software overheads in managing compaction window metadata, such as reading the hardware system timer and checking for impending SLO violations, which are not captured by the model.

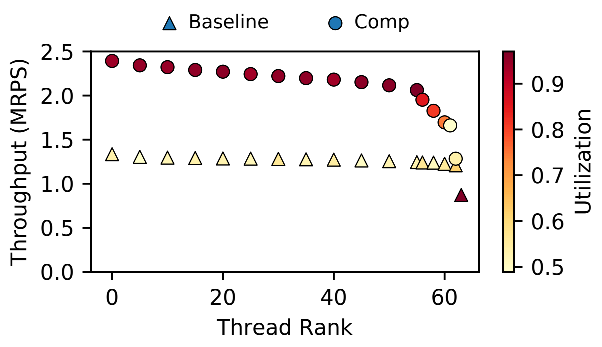


Figure 12: Per-thread throughput, using the same workload as Fig. 11 (γ = 1.25, f_{wr} = 5%).

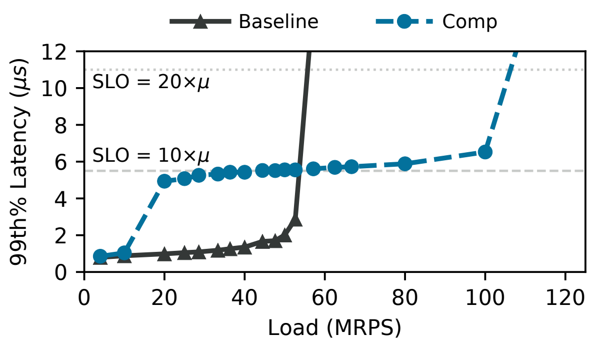


Figure 13: Performance for RW_{sk} with γ = 0.99, f_{wr} = 50%.

Fig. 12 compares the throughput and core utilization for a subset of the KVS threads, sorted by decreasing throughput, taken at 76MRPS for the baseline and 125MRPS for C-4.[1] In the baseline system, both throughput and utilization are roughly uniform across threads, with all threads except the overloaded writer attaining ∼ 1.28MRPS. The overloaded writer (Fig. 12's rightmost data point) attains < 1MRPS at near-maximal utilization due to its inflated service times (see Fig. 11b).

With C-4, both reader and writer threads attain drastically higher throughput. The 3.7× service time reduction for the hottest thread translates to lower utilization, in contrast to the baseline where only the hottest thread utilizes its core while others are idle. This trend is visible by comparing the utilization of the hottest writer thread in Fig. 12; we note that the thread moves from having the lowest throughput, to having the second lowest. With compaction enabled, the hottest writer improves from 0.92MRPS to 1.66MRPS and sees its utilization fall to 47% even at higher system load, because it handles writes more efficiently. Other threads with ranks between 55–63 show the same trend in C-4: rapid write compaction creates idle periods where other requests could be handled.

C-4 saturates not because of write overload, but due to reads saturating the threads, which are already serving > 2.3MRPS and operating near 100% utilization (Fig. 12). Therefore, our implementation could scale to modestly higher throughput by harvesting the extra cycles made available by C-4's write skew absorption via compaction. Modifications to the underlying NeBuLa stack for such purposes are possible, which we leave for future work.

Fig. 13 evaluates a workload with γ = 0.99 and 50% writes, used by prior work [44, 60, 61] due to its challenging nature. Due to less cache-resident hot data, the baseline only attains 56MRPS under SLO, compared to 76MRPS in Fig. 11a. The single overloaded writer thread is the bottleneck for both workloads. In contrast, C-4 attains 58MRPS under an identical 10× SLO, and 100MRPS under a 20× SLO. C-4's drastic jump in tail latency beyond 10MRPS is because compaction events immediately form the 99th% of this workload. However, increasing load from 20MRPS to 80MRPS only causes the 99th% to increase by ∼ 300*ns*, indicating that the compaction windows at lower load are often opened but collect very few writes. Such behavior is an artifact of our implementation, which only closes a window upon an impending SLO violation. A software modification to adaptively close compaction windows early under low load (e.g., when polling a thread's CQ yields no incoming requests) would resolve the early jump in 99th% latency.

[1]We show a subset of the threads for readability. Omitted threads fall between the displayed points.

7.3 Sensitivity Analysis

Finally, we evaluate the impact of varying MICA's key and value sizes on the performance of C-4's write compaction. We compare our previous results using γ = 1.25, 5% writes, and 16B/512B KV pairs, to a workload using 8B/8B KV pairs (called Tiny) and one using 16B/128B pairs (Medium). In all workloads, we scaled the number of KV pairs to keep the same total dataset size, so that the same overall data fraction is cache-resident. Sec. 7.3 compares the CREW baseline and C-4 in terms of maximum throughput under 10× SLO, as well as the speedups of the KVS' hottest compacting thread and the remaining threads.

As the item size shrinks, the baseline system's performance increases by 1.8× for Medium and by 3.5× for Tiny items, because threads spend less time per request accessing data. C-4 benefits from these service time reductions to a lesser degree (e.g., by 1.5× for Medium items) because threads only write the underlying data structure once per compaction window, and compact all other requests locally, which is less sensitive to item size.

The hottest-thread speedup granted by C-4 drops with smaller KV pairs, as the baseline system's service times approach the latency of appending a write to an open compaction window. However, as the speedup granted to the hottest thread reduces, the impact on readers becomes the critical performance driver. The hottest thread only attains 1.1× speedup with Tiny items because our compaction implementation trades off increased compute latency for reduced coherence activity. Compaction has higher compute requirements because it requires many instructions to create the requisite metadata and compact its private logs, whereas the baseline can write a single 8B KV item with a single `store` instruction. In summary, C-4 provides robust throughput benefits across item sizes, improving throughput under SLO by 1.4× and 1.33× for Tiny and Medium items, respectively.

Table 2: Impact of item size on write compaction.

	Throughput @ SLO		Speedup	
Key/Value Sizes (B)	Base (MRPS)	Comp (MRPS)	Hot Thread	Other Threads
8/8 (Tiny)	266	363	1.1×	1.3×
16/128 (Med)	142	190	1.3×	1.3×
16/512 (Lg)	76	125	1.6×	1.6×

8 RELATED WORK

KVS Designs. We group KVS systems into two categories: software-only or NIC-assisted. Representatives of the former include memcached [69], MemC3 [27], Masstree [64], and MICA [60, 61]. All of these systems reach their peak performance with R_{uni} or R_{sk} workloads—in fact, MICA's authors [60] propose using frequency scaling on the overloaded core to alleviate the static write imbalance bottleneck in RW_{sk} workloads. Frequency scaling alone is insufficient to absorb RW_{sk}'s load imbalance, but could be added to C-4 for further performance gains.

NIC-assisted designs enhance the KVS with RDMA, FPGA, or programmable switch support. However, the vast majority of such designs do not perform well on write-intensive workloads. The two best performers in this space are NetCache [44] and KV-Direct [58], as dedicated hardware performs the entire KVS' functionality instead of using CPU cores. RDMA-enabled designs include FaRM [22], HERD [47], ccKVS [31], RackOut [75], and DrTM [13, 14]. The write imbalances we identify in Sec. 2.1 in a single-node context would be strictly worse in the multi-node distributed KVS settings targeted by these RDMA-assisted designs.

Emerging SmartNIC frameworks propose a split-KVS design, where a small amount of NIC memory stores the KVS' hottest items [26, 81], embodying the "small cache, big effect" principle [28]. Like NetCache [44], such designs focus on reads and are inefficient for write-intensive workloads, because writes are handled on a slow path to keep the CPU's main memory up-to-date.

The Minos KVS targets queueing delays created when requests for small items queue behind those for extremely large items (e.g., 100s of KBs-MBs). The authors propose reserving dedicated cores for large-item requests, and re-balancing incoming writes in software. Therefore, Minos only supports a CRCW concurrency control policy and must fall back to using spinlocks on each KVS partition [21, §4.2]; although acceptable for read-dominated workloads, the overheads would grow for emerging write-heavy workloads. C-4's d-CREW policy could be adapted to accomplish size-aware load balancing using the EWT, maintaining lightweight concurrency control without spinlocks.

LSM-based KVS designs such as LevelDB [2] or RocksDB [70] have begun to encounter bottlenecks in their in-memory component because of the increased speed of storage devices and growing core counts of server CPUs. FloDB adds another level to the hierarchy of LSM data structures to support rapid random writes [6]. TRIAD improves LSM performance for skewed workloads by preventing popular keys from being compacted and flushed to storage [5]. C-4 targets orthogonal bottlenecks occurring in systems where the entire KVS is in memory. A hypothetical combination of FloDB and TRIAD with a single in-memory level would experience both bottlenecks presented in Sec. 2.1.

Concurrency and Synchronization. The most applicable works to our target workloads are those that can enable write load balancing (for WI_{uni}), and lock-free readers in the presence of frequent writes (for RW_{sk}). For WI_{uni}, Hardware Transactional Memory [42] or hardware-supported synchronization such as QOLB [55] should deliver similar performance to C-4 with the appropriate KVS software rewrite, as true data conflicts are rare. Both these designs come with far higher hardware implementation complexity. For example, consider that Intel RTM is still disabled due to memory ordering violations [41] and will be removed in many future products [30]. C-4 only requires a small set of NIC extensions and no changes to the caches or memory ordering hardware.

RCU [68], RLU [66] and MV-RLU [53] enable both write balancing and lock-free reads by forcing writes to create new copies of data and make them visible to readers after a quiescent period. Our evaluation shows that the cost of RLU and MV-RLU versioning is prohibitive for µs-scale KVS, even with very few true conflicts. Delegation-based approaches such as `ffwd` [84], Flat Combining [36], and Remote Core Locking [63] could enable NIC-driven write balancing, but software threads must rebalance requests so that critical sections execute on one core—essentially implementing CREW in software with the overheads of rebalancing.

Both OpLog [9] and Doppel [72] propose similar designs to C-4's write compaction. OpLog targets update-heavy kernel data structures by creating per-core update logs, which readers scan and apply lazily. Similarly, Doppel executes conflicting transactions by creating per-core values that are merged on commit. Applying OpLog to a KVS would move synchronization bottlenecks from the KVS' data structure to the per-core logs, and Doppel would incur similar overheads to RLU at commit time (see Sec. 7.1). In contrast, C-4's compaction logs are invisible until the compaction window closes, thus avoiding costly copies and reconciliations.

System Software for RPCs. Although we implement C-4 over the NeBuLa architecture, our design could be implemented over a variety of baselines (e.g., NanoPU [39], FlexNIC [52], or NICA [26]). Due to the unique challenges of µs-scale RPCs that characterize KVS, several proposals pursue synchronization-free RPC load balancing [17, 38, 45, 54, 62, 77, 83, 93]. C-4 is orthogonal to these works because none of them can relax the restrictions that writes impose on load balancing.

9 CONCLUSION

Key-Value Stores are a cornerstone for virtually every large-scale online service. Extensive research efforts over the past decade have resulted in KVS designs targeted towards read-dominated workloads, or read-write workloads without the extreme popularity skew in emerging workloads. As a result, modern KVS implementations are inefficient when handling write-heavy workloads. We identify two problematic workload regions and provide insight into the root cause of inefficiency in modern KVS implementations – static write partitioning. Therefore, we propose C-4, a hardware-software co-design that employs two distinct and orthogonal mechanisms: dynamic write balancing and write compaction. Our evaluation shows that C-4 yields significant improvements under stringent SLOs, up to 5× lower tail latency for write-heavy workloads and 1.7× higher throughput for skewed read-write workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and feedback, as well as the reviewers of our artifact who assisted in improving its correctness and ease of use. We also wish to thank Marios Kogias, Adrien Ghosn, and Rishabh Iyer for many discussions on consistency models which enhanced the work in this paper. Finally, we thank the entire PARSA group at EPFL for their support, and CloudLab [24] for providing us access to servers used in the evaluation. This work was supported by EPFL graduate student funding, and the National Science Foundation under the award NSF-CCF-2006602.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the discrete-event simulation framework and methodology used to model the performance of various concurrency control policies evaluated in the paper. Specifically, we provide infrastructure to reproduce Figs. 3 and 4, and documentation indicating how to use the same methodology to generate results that can be compared against the those derived from full-system simulation. The purpose of this artifact is to allow the community to reproduce our results coming from discrete-event simulation, and to provide our event simulation framework to others as infrastructure for future research because of how useful it has been for our work.

A.2 Artifact check-list (meta-information)

- **Program:** A set of performance models for server components and software libraries, which are instantiated to model the concurrency control techniques presented in the paper.
- **Run-time environment:** Depends on `python3` and the discrete-event simulation library `SimPy`. All packages are installed through Python virtual environments.
- **Metrics:** Latency percentiles for modeled requests (50th, 90th, 95th, 99th). Throughput (million requests/second).
- **Output:** CSV files with performance data for the evaluated concurrency control policies, and scripts to plot the associated graphs.
- **Experiments:** Scripts are included to reproduce Figs. 3 and 4 of the paper, and instructions are included to study other experiments using this tool.
- **How much disk space required (approximately)?:** Less than 10 MB.
- **How much time is needed to prepare workflow (approximately)?:** ∼ 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1–48 hours, depending on precision and number of CPU cores available for experiment parallelization.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Workflow framework used?:** Python and Bash scripts.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7182800

A.3 Description

A.3.1 How to access. The artifact can be downloaded from GitHub[2] or Zenodo.[3] Please note that the GitHub version is provided as a tagged release, which will accrue differences over time compared to the current software version in the repository.

[2]https://github.com/parsa-epfl/queue_flex/releases/tag/c4_ae_v1.1
[3]https://doi.org/10.5281/zenodo.7182800

A.3.2 Hardware dependencies. There are no special hardware requirements to run this artifact. However, we recommend running all data-gathering experiments on a machine with $\geq$ 16 cores in order to parallelize the discrete-event simulations.

A.3.3 Software dependencies. The artifact depends on a working installation of `python3`, as well as the libraries `SimPy`, `matplotlib`, and `pytest`. All dependencies are installed automatically by scripts when setting up the artifact. For exact versions of the software dependencies used, please consult the `requirements.txt` file.

A.4 Installation

First, ensure that your system has a working installation of `python3`, as well as the `virtualenv` tool.

After downloading and unpacking the artifact, follow the below commands for installation. The general workflow is:

- Create a python virtual environment for all dependencies.
- Install all packages using `pip`.
- Test the functionality of the various simulation components.

First, create a virtual environment with the following command (or one like it, where the argument following `-p` points to a working version of python3). We have tested this artifact against `python 3.7.3, 3.8.10, and 3.8.13`, and although other versions may work equally well, we recommend one of these versions be used.

```
$ virtualenv -p python3 venv
```

Next, activate the virtual environment, and use the provide setup script to install dependencies and set environment variables.

```
$ source venv/bin/activate
$ ./setup.sh
```

To leave the virtual environment when finished, use the command `deactivate`.

A.5 Basic Test

To ensure the components are functioning properly and that the underlying simulation library is generating and consuming events, run the provided unit tests as follows:

```
$ py.test
```

The expected output for the unit tests is 61 passes and 2 purposeful failures, which are marked as `xfailed`. If `PyTest` reports the aforementioned outcome, the artifact is functioning properly.

A.6 Experiment workflow

All experiments use the following workflow:

- The user invokes the top-level script to gather results, which launches all the underlying event simulations. We provide two top-level scripts, one each for Figs. 3 and 4 of the paper.
- A CSV file is generated with all of the latency percentiles and throughput values.
- Finally, the user invokes the relevant plotting script, which generates output graphs based on the input CSVs.

A.7 Evaluation and expected results

Excess Tail Latency – Fig. 3. To reproduce our results for excess tail latency, use the top-level script `compare_system_excess_tlat.py`. Launching the script as follows will perform the same experiment shown in the paper, with a reduced number of simulation points (200k). Set `num_threads` to the number of CPU threads that will run experiments in parallel.

```
$ python compare_system_excess_tlat.py --threads <num_threads> 0.0
```

With the default number of simulation points, this experiment should require roughly one hour of runtime with 16 threads. To reproduce the values in the paper, add the flag `--reqs_to_sim 5000000`. This experiment may take hours or days depending on the number of threads used. On the 64-core servers used for the paper, this experiment required approximately 24 hours of runtime.

Once the experiment finishes, the output files are written in a directory called `excess_tlat_comparison/`. Use regular Unix tools to compare against the expected output, or use the provided plotting script to generate the graphs in the paper as follows:

```
$ diff expected_outputs/excess_tlat_short.csv
↪  excess_tlat_comparison/excess_tlat_comparison.csv
$ python plot/plot_excess_tlat.py
↪  excess_tlat_comparison/excess_tlat_comparison.csv
↪  excess_tlat.pdf tput_comp.pdf
```

Our simulator is deterministic, therefore a matching value for this reduced-length experiment implies a matching value for the full 5 million point simulation. We include both "short" and "full" raw results files in the `expected_outputs/` directory.

Write Compaction – Fig. 4. Similarly to the evaluation for excess tail latency, use the provided script `crew_comp_sim.py` as follows.

```
$ python compaction_sim.py --threads <num_threads> --ofile
↪  baseline.csv CREW # for baseline
$ python compaction_sim.py --threads <num_threads> --use_compaction
↪  --ofile compaction.csv CREW # for comp
```

This experiment also requires many hours, as even more simulation points are needed to produce the surface plots in the paper. Outputs can be compared against the expected values and plotted as above:

```
$ diff expected_outputs/baseline_short.csv /path/to/baseline.csv
$ diff expected_outputs/compaction_short.csv
↪  /path/to/compaction.csv
$ python plot/plot_surfs.py /path/to/baseline.csv
↪  /path/to/compaction.csv
```

Due to the number of simulations required for this experiment, we only include the "short" results files.

A.8 Experiment customization

Experiments can be customized in two ways: first, different command-line parameters can be passed to the top-level scripts in order to change parameters such as the distribution of keys in the generated load, the number of hash buckets in the index, the service time of requests, and many more. Second, building one's own experiments can be done by connecting various simulation components (found in the folder `components/`), or modifying those components. General patterns for creating and connecting simulation components can be found in the top-level file `custom_exp.py`.

Table 3: Simulation parameters for comparing results obtained with discrete-event simulation to those of full-system simulation presented in Sec. 7.

Figure	Shared Parameters	Applicable Modes
Fig. 9	-s 0 –write_frac 50	EREW, CREW
Fig. 10	-s 0 –write_frac {50,85}	d-CREW, Ideal

A.9 Notes

All the results generated are deterministic across many runs, so they can be directly compared against the expected values, as well as each other.

The expected output for 200k simulation points shows two behaviors that are different from our displayed results at face value—for excess tail latency, one will observe that the 99th% tail latency comparisons are different from those shown in Fig. 3b. However, this is an artifact of fewer samples being run. By using 5M samples, the exact paper results are obtainable. For write compaction, the expected output is nearly identical to what is displayed in Fig. 4. To reproduce the paper's exact results, use 5M samples as before.

Additionally, our artifact allows comparing our queueing model's predictions to those generated by cycle-accurate simulation in Figs. 9 and 10, using the top-level script `detailed_loadlat.py`. Each invocation of this script runs a single load-latency curve, so the parameters must be adjusted to model each system. The exact parameters for each experiment are shown in Table 3. For example, to generate a data series to compare to "Baseline" in Fig. 9, use the following command:

```
$ python detailed_loadlat.py --threads <num_threads> -s 0
↪  --write_frac 50 --reqs_to_sim 5000000 --ofile <base.csv> CREW
```

We remark that because the impacts of data locality and cache coherence are not captured by the discrete-event simulator, the tool significantly underestimates performance gains achievable with write compaction, and therefore we explicitly do not invite such comparisons. Increasing the tool's fidelity would require enhancing it with an actual coherence protocol model, in order to accurately capture the impact of cache-line contention alleviated by compaction (see Sec. 3.2 and Sec. 7.2). Ultimately, our artifact accurately predicts the performance gains of C-4's d-CREW component, and motivates the performance potential of compaction which is confirmed by our full-system cycle-accurate evaluation.

A.10 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek, "Thialfi: a client notification service for internet-scale applications." in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 129–142, https://doi.org/10.1145/2043556.2043570.

[2] Alphabet Inc., "Leveldb," March 2022. [Online]. Available: https://github.com/google/leveldb

[3] Amazon announces graviton2 soc along with new aws instances. AnandTech. [Online]. Available: https://www.anandtech.com/show/15189/amazon-announces-graviton2-soc-along-with-new-aws-instances-64core-arm-with-large-performance-uplifts

[4] *ARM Architecture Reference Manual for A-profile architecture*, Version H.a ed., ARM Limited, February 2022.

[5] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores." in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 363–375, https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau.

[6] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "FloDB: Unlocking Memory in Persistent Key-Value Stores." in *Proceedings of the 2017 EuroSys Conference*, 2017, pp. 80–94, https://doi.org/10.1145/3064176.3064193.

[7] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane." *ACM Trans. Comput. Syst.*, vol. 34, no. 4, pp. 11:1–11:39, 2017, https://doi.org/10.1145/2997641.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors." *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014, https://doi.org/10.1145/2656877.2656890.

[9] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Oplog: a library for scaling update-heavy data structures," Massachussetts Institute of Technology, Tech. Rep., 2014. [Online]. Available: https://dspace.mit.edu/handle/1721.1/89653

[10] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph." in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 49–60, https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[11] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook." in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 209–223, https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[12] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A Concurrent Key-Value Store with In-Place Updates." in *SIGMOD Conference*, 2018, pp. 275–290, https://doi.org/10.1145/3183713.3196898.

[13] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan, "Fast In-Memory Transaction Processing Using RDMA and HTM." *ACM Trans. Comput. Syst.*, vol. 35, no. 1, pp. 3:1–3:37, 2017, https://doi.org/10.1145/3092701.

[14] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM." in *Proceedings of the 2016 EuroSys Conference*, 2016, pp. 26:1–26:17, https://doi.org/10.1145/2901318.2901349.

[15] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload Control for µs-scale RPCs with Breakwater." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 299–314, https://www.usenix.org/conference/osdi20/presentation/cho.

[16] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore network interfaces for in-memory rack-scale computing." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 567–579, https://doi.org/10.1145/2749469.2750415.

[17] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 35–48, https://doi.org/10.1145/3297858.3304070.

[18] J. Dean and L. A. Barroso, "The tail at scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013, https://doi.org/10.1145/2408776.2408794.

[19] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang, "When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone." in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 621–637, https://doi.org/10.1145/3477132.3483571.

[20] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins." in *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, 1992, pp. 27–40, http://www.vldb.org/conf/1992/P027.PDF.

[21] D. Didona and W. Zwaenepoel, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 79–94, https://www.usenix.org/conference/nsdi19/presentation/didona.

[22] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory." in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414, https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic.

[23] D. Dunning, G. J. Regnier, G. L. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture." *IEEE Micro*, vol. 18, no. 2, pp. 66–76, 1998, https://doi.org/10.1109/40.671404.

[24] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14, https://www.flux.utah.edu/paper/duplyakin-atc19.

[25] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM footprint with NVM in facebook." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 42:1–42:13, https://doi.org/10.1145/3190508.3190524.

[26] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An Infrastructure for Inline Acceleration of Network Applications." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 345–362, https://www.usenix.org/conference/atc19/presentation/eran.

[27] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing." in *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 371–384, https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[28] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: provable load balancing for randomly partitioned cluster services." in *Proceedings of the 2011 ACM Symposium on Cloud Computing (SOCC)*, 2011, p. 23, https://doi.org/10.1145/2038916.2038939.

[29] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18, https://doi.org/10.1145/3297858.3304013.

[30] Gareth Halfacree. The Register: Intel sticks another nail in the coffin of TSX with feature-disabling microcode update. [Online]. Available: https://www.theregister.com/2021/06/29/intel_tsx_disabled/

[31] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan, "Scale-out ccNUMA: exploiting skew with strongly consistent caching." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 21:1–21:15, https://doi.org/10.1145/3190508.3190550.

[32] Google. Protocol Buffers. [Online]. Available: https://developers.google.com/protocol-buffers/

[33] L. Gwennap, "AMD Milan Extends Server Lead," *Linley Group Microprocessor Report*, March 2021.

[34] L. Gwennap, "Sapphire Rapids Spans Tiles," *Linley Group Microprocessor Report*, September 2021.

[35] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance." in *Proceedings of the ACM SIGCOMM 2017 Conference*, 2017, pp. 29–42, https://doi.org/10.1145/3098822.3098825.

[36] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff." in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010, pp. 355–364, https://doi.org/10.1145/1810479.1810540.

[37] M. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990, https://doi.org/10.1145/78969.78972.

[38] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis, "Mind the Gap: A Case for Informed Request Scheduling at the NIC." in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets-XVIII)*, 2019, pp. 60–68, https://doi.org/10.1145/3365609.3365856.

[39] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A Nanosecond Network Stack for Datacenters." in *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 239–256, https://www.usenix.org/conference/osdi21/presentation/ibanez.

[40] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manuals. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[41] Intel Corp. Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/Performance-Monitoring-Impact-of-TSX-Memory-Ordering-Issue-604224.pdf

[42] Intel Corp. RTM Overview. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/intrinsics-for-restrict-transactional-mem-ops/restricted-transactional-memory-overview.html

[43] Intel Corp. (2014) Introduction to Intel Ethernet Flow Director and Memcached Performance. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf

[44] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 121–136, https://doi.org/10.1145/3132747.3132764.

[45] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 345–360, https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[46] eRPC Repository. [Online]. Available: https://github.com/erpc-io/eRPC

[47] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services." in *Proceedings of the ACM SIGCOMM 2014 Conference*, 2014, pp. 295–306, https://doi.org/10.1145/2619239.2626299.

[48] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be General and Fast." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 1–16, https://www.usenix.org/conference/nsdi19/presentation/kalia.

[49] D. Kanter, "Xeon Scalable Reshapes Server Line," *Linley Group Microprocessor Report*, July 2017.

[50] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. H. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42, https://doi.org/10.1109/ISCA.2018.00014.

[51] S. Kashyap, C. Min, K. Kim, and T. Kim, "A scalable ordering primitive for multicore machines." in *Proceedings of the 2018 EuroSys Conference*, 2018, pp. 34:1–34:15, https://doi.org/10.1145/3190508.3190510.

[52] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC." in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 67–81, https://doi.org/10.1145/2872362.2872367.

[53] J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min, "MV-RLU: Scaling Read-Log-Update with Multi-Versioning." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 779–792, https://doi.org/10.1145/3297858.3304040.

[54] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens." in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 863–880, https://www.usenix.org/conference/atc19/presentation/kogias-r2p2.

[55] A. Kägi, D. Burger, and J. R. Goodman, "Efficient Synchronization: Let Them Eat QOLB." in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 170–180, https://doi.org/10.1145/264107.264166.

[56] B. Lampson, "Hints and principles for computer system design," 2020, retrieved May 3, 2022. [Online]. Available: https://arxiv.org/abs/2011.02455

[57] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 36–51, https://doi.org/10.1145/3445814.3446696.

[58] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 137–152, https://doi.org/10.1145/3132747.3132756.

[59] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques." in *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 694–701, https://doi.org/10.1109/ICCAD.2011.6105405.

[60] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 476–488, https://doi.org/10.1145/2749469.2750416.

[61] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage." in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 429–444, https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[62] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe." in *Proceedings of the ACM SIGCOMM 2019 Conference*, 2019, pp. 318–333, https://doi.org/10.1145/3341302.3342079.

[63] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications." in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012, pp. 65–76, https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

[64] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage." in *Proceedings of the 2012 EuroSys Conference*, 2012, pp. 183–196, https://doi.org/10.1145/2168836.2168855.

[65] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. D. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking." in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 399–413, https://doi.org/10.1145/3341301.3359657.

[66] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: a lightweight synchronization mechanism for concurrent programming." in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 168–183, https://doi.org/10.1145/2815400.2815406.

[67] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX Winter*, vol. 46, 1993.

[68] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, vol. 509518, 1998.

[69] Memcached: A Distributed Memory Object Caching System. [Online]. Available: https://memcached.org/

[70] Meta Inc. RocksDB GitHub. [Online]. Available: https://github.com/facebook/rocksdb

[71] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities." in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018, pp. 221–235, https://doi.org/10.1145/3230543.3230564.

[72] N. Narula, C. Cutler, E. Kohler, and R. T. Morris, "Phase Reconciliation for Contended In-Memory Transactions." in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 511–524, https://www.usenix.org/conference/osdi14/technical-sessions/presentation/narula.

[73] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook." in *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398, https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[74] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 3–18, https://doi.org/10.1145/2541940.2541965.

[75] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems." in *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, 2016, pp. 182–195, https://doi.org/10.1145/2987550.2987577.

[76] NVIDIA ConnectX-7 400G Ethernet. NVIDIA Corp. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf

[77] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads." in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 361–378, https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[78] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System." *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, 2015, https://doi.org/10.1145/2806887.

[79] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey." *IEEE J. Solid State Circuits*, vol. 41, no. 3, pp. 712–727, 2006, https://doi.org/10.1109/JSSC.2005.864128.

[80] Parallel Systems Architecture Lab (PARSA), "QFlex," March 2020. [Online]. Available: https://qflex.epfl.ch

[81] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general-purpose on-NIC memory." in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 1130–1147, https://doi.org/10.1145/3503222.3507711.

[82] Pivotal Software, "RabbitMQ: Persistence Configuration," https://www.rabbitmq.com/persistence-conf.html.

[83] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 325–341, https://doi.org/10.114

5/3132747.3132780.

[84] S. Roghanchi, J. Eriksson, and N. Basu, "ffwd: delegation is (much) faster than you think." in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 342–358, https://doi.org/10.1145/3132747.3132771.

[85] L. Suresh, P. Bodík, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries." in *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*, 2017, pp. 611–623, https://doi.org/10.1145/3127479.3132020.

[86] M. Sutherland, S. Gupta, B. Falsafi, V. J. Marathe, D. N. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-Optimized Architecture." in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 199–212, https://doi.org/10.1109/ISCA45697.2020.00027.

[87] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[88] B. Wheeler, "Graviton3 Debuts Neoverse V1," *Linley Group Microprocessor Report*, January 2022.

[89] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services." *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021, https://doi.org/10.1109/MM.2021.3085578.

[90] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 191–208, https://www.usenix.org/conference/osdi20/presentation/yang.

[91] J. Yang, Y. Yue, and R. Vinayak, "Segcache: a memory-efficient and scalable in-memory key-value cache for small objects." in *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 503–518, https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng.

[92] A. P. Zarandi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters." in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 407–420, https://doi.org/10.1145/3466752.3480055.

[93] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, "RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers." in *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1225–1240, https://www.usenix.org/conference/osdi20/presentation/zhu.