CliqueMap: Productionizing an RMA-Based Distributed Caching System

Arjun Singhvi[†‡] Aditya Akella[†‡] Maggie Anderson[‡] Rob Cauble[‡] Harshad Deshmukh[‡]
Dan Gibson[‡] Milo M. K. Martin[‡] Amanda Strominger[‡] Thomas F. Wenisch[‡]
Amin Vahdat[‡]
[†]*University of Wisconsin - Madison* [‡]*Google Inc*

Abstract

Distributed in-memory caching is a key component of modern Internet services. Such caches are often accessed via remote procedure call (RPC), as RPC frameworks provide rich support for productionization, including protocol versioning, memory efficiency, auto-scaling, and hitless upgrades. However, full-featured RPC limits performance and scalability as it incurs high latencies and CPU overheads. Remote Memory Access (RMA) offers a promising alternative, but meeting productionization requirements can be a significant challenge with RMA-based systems due to limited programmability and narrow RMA primitives.

This paper describes the design, implementation, and experience derived from CliqueMap, a hybrid RMA/RPC caching system. CliqueMap has been in production use in Google's datacenters for over three years, currently serves more than 1PB of DRAM, and underlies several end-user visible services. CliqueMap makes use of performant and efficient RMAs on the critical serving path and judiciously applies RPCs toward other functionality. The design embraces lightweight replication, client-based quoruming, self-validating server responses, per-operation client-side retries, and co-design with the network layers. These foci lead to a system resilient to the rigors of production and frequent post-deployment evolution.

CCS Concepts

• **Computer systems organization** → *Distributed Systems; Key-Value Stores*; • **Networks** → *Datacenter networks*.

Keywords

Remote Memory Access; Remote Procedure Call; Key-Value Caching System

ACM Reference Format:
Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, Amin Vahdat. 2021. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452296.3472934

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472934

1 Introduction

Remote Procedure Call (RPC) frameworks are the backbone of modern Internet services as they provide a familiar programming abstraction for building distributed systems. The production needs of hyperscale distributed systems have layered requirements on RPC frameworks including protocol versioning, memory management, auto-scaling, logging, and support for binary upgrades. These features entail considerable CPU overhead at both the client and server, limiting operation (op) rate, bandwidth, and efficiency; at Google, even an empty RPC often costs >50 CPU-μs in framework and transport code across client and server. Per-RPC overheads easily dominate remote operations with little server-side computation, such as in-memory retrieval applications like distributed caches and key-value stores.

To reduce CPU overheads and to increase peak op rate for such applications, recent efforts use remote memory access (RMA) technologies [17, 18, 23, 33, 36, 40]. RMA has become ubiquitous in high-end networking; modern NICs natively support RMA [1, 3, 5, 34] and software implementations [8, 31, 35] enable continuous innovation in a hardware-independent manner. The essential element of RMA is that no server-side application code needs to run to complete an operation.

Whereas RMA raises distributed systems' performance and efficiency ceiling by using simple primitives, the features that make RPC-based systems robust—but sap CPU efficiency [25]—remain critical to production operation (see Table 1 and §2). Hybrid systems [33, 35, 40] leverage the strengths of RMA and RPC through the use of RMA-accessible data structures to accelerate performance-critical communication, while using RPCs to ease programming burden on less performance-critical paths. In practice, building such hybrid systems faces two key challenges: (1) designing performant data structures and protocols to accommodate concurrency between RMA and server-side execution (RPC handlers), and (2) meeting productionization expectations, such as availability, support for planned maintenance, memory management/efficiency, evolution over time, and software interoperability.

In this paper, we present the design, implementation, and experiences derived from CliqueMap, a hybrid RMA/RPC in-memory key-value caching system (KVCS) that overcomes the above two challenges. Like MICA [29], CliqueMap uses an associative hash table to enable remote access. Like Pilaf [33], each KV pair is guarded by a checksum that is used to validate responses. The core of CliqueMap's design is centered around (1) providing performant lookups, (2) a careful division of responsibilities between RPCs and RMAs across dataplane, control, and management operations, and

	Challenge	Description	Solution
1.	**Memory efficiency**	While RMA optimizes to reduce CPU cost, a KVCS must aggressively optimize for memory footprint, too. The number of backends and memory size per backend must grow/shrink without disruption.	CliqueMap uses the server-side RPC handlers for memory allocation/defragmentation and supports multiple eviction algorithms based on client-side recording of RMA access.
2.	**Enable agile evolution post deployment**	Production services undergo requirement changes and other upgrades throughout their lifetimes and must be easy to evolve to meet the new requirements.	CliqueMap leverages RPCs to introduce new features while ensuring that clients are resilient to such additions by embracing self-validating server responses and client retries.
3.	**Increased availability with minimal overheads**	Server failures should not lead to data unavailability or performance drops. Systems must tolerate frequent planned and unplanned restarts without loss of data.	CliqueMap uses an uncoordinated replication protocol with a load-aware client-based quoruming approach leading to increased availability with minimal performance overheads.
4.	**Software interoperability**	Corpora stored in the KVCS must be accessible by any authenticated production system, regardless of programming language.	CliqueMap uses named pipes and sub-processes to provide support for Go, Java, and Python.
5.	**Optimizing to heterogeneous networking hardware**	Our data centers operate across several generations of networking technology and RMA protocols, which vary over two orders of magnitude in available bandwidth per host.	CliqueMap operates over multiple RMA protocols, integrates tightly with software-defined NIC Pony Express [31], and provides WAN access via RPC.

Table 1: Productionization challenges and CliqueMap's solutions.

(3) support for key productionization expectations, such as evolution over time, high availability, interoperability, and ease of deployment.

CliqueMap accelerates the common-case read path via RMA primitives and uses RPC for mutations [33], specifically optimizing for serving workloads where lookup (GET) performance is critical and write (e.g., SET) performance is less so, a pattern common in Google's applications. An RMA-based read path substantially reduces CPU cost and increases peak op rate relative to (fully) RPC-based systems. Nonetheless, RPCs significantly simplify CliqueMap's implementation, because state mutation, mutual exclusion, and race resolution can all be solved with server-side code, and thereby sidestep the challenges of data structures that tolerate slow/racy RMA mutations [7]. To aggressively optimize server memory footprint, CliqueMap leverages server-side RPC handlers for memory allocation/defragmentation, to trigger automatic memory resizing, and to support various cache management algorithms.

To address production availability expectations, CliqueMap offers several replication modes, including "R=3.2", wherein each key/value-pair is replicated across three server backends and accessed via a client-side quoruming scheme [19], delivering consistent reads and writes at high performance and low overhead. Quoruming offers performance benefits: (1) it automatically mitigates tail GET latency, fetching data from the least loaded/nearest replica; and (2) it resolves races among concurrent GETs and mutations, avoiding distributed/global locking for replicated writes (see §5.3). During weekly binary upgrades, CliqueMap bolsters availability through explicit, proactive server data migration to warm spares.

CliqueMap combines the strengths of software datapaths, transports, and dataplanes in other ways (beyond simply leveraging RPCs) to address key interoperability goals. Notably, CliqueMap supports corpora accessed by systems (or a subset of their internal components) written in several high-level languages, even though these languages have no native support for RMA, by running a C++ CliqueMap client in a subprocess. By embracing the programmability of software-defined NICs (Pony Express [31]), CliqueMap takes advantage of RMA-like primitives that enable GET operations to complete in fewer round-trips than when using hardware-supported RMAs (§3), delivering better efficiency (§6.3) and overall latencies in most cases (§7). Lastly, by capitalizing client-side retries and self-validating responses, we have seamlessly evolved CliqueMap over time (§6), including over a hundred changes to CliqueMap's protocol definitions of varying complexity.

CliqueMap has been in production for more than three years, during which it has grown to serve more than 1PB of DRAM. CliqueMap now underlies production stacks for end-user-facing ads, maps, and other serving systems at Google, and is deployed across some 50 production clusters distributed among 20 warehouse-scale datacenters throughout the world. Use cases vary by corpus size, key-value geometry, batching factors, and other considerations, and comprise roughly 150M queries per second (QPS) globally.

Our experience with CliqueMap has highlighted several lessons that we believe are valuable to the research community, including: (1) the fundamental need to embrace RPCs even in performance-critical systems; (2) considering multi-language interoperability in RMA, a space that we see as open to exploration; (3) making memory efficiency a strong requirement in RMA-based systems; and (4) embracing software-based processing (e.g., programmable/software NICs) in system design.

2 Background and Motivation

In-memory key-value caching systems (KVCS) are crucial to user-facing services throughout the industry [10, 41]. As the name suggests, a KVCS exposes a KV interface to the clients and typically supports per-key operations such as GETs and mutations (e.g., SET and ERASE).

While a KVCS should be performant and efficient (i.e., the data-plane operations should be fast and have minimal CPU overhead), these aspects alone do not make a KVCS viable in production. Table 1 outlines key practical requirements that a production KVCS should meet, pertaining to availability, memory efficiency, evolution, and interoperability.

2.1 RPC or RMA?

System designers must address what network transport(s) to use in their design, as this choice has a fundamental bearing on performance and the requirements in Table 1.

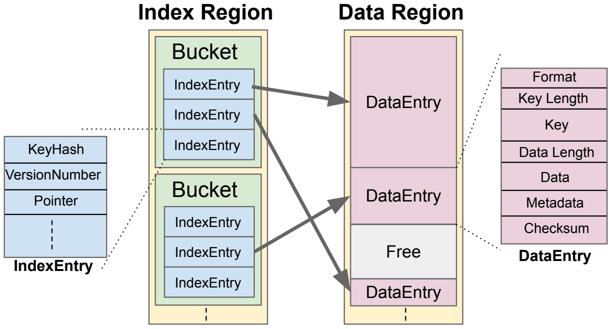


Figure 1: CliqueMap index region and data region layouts.

RPC KVCS. Memcached [4] is a well-known KVCS, built upon conventional kernel networking primitives. Twitter uses a forked version of Memcached, known as Twemcache [41], to serve a majority of its caching traffic. Google, too, has its own internal version, known as *MemcacheG*, a translation of Memcached, using Stubby RPC—Google's production-grade RPC—as its transport. By building on Stubby RPC, MemcacheG inherits critical productionization features from its RPC framework, such as application-to-application authentication (ALTS [2]), integrity protection, forward- and backward-versioning assistance, per-RPC ACLs, and interoperability across multiple languages.

This feature wealth comes at a CPU cost; even a well-optimized Stubby RPC incurs >50 CPU-μs across client and server—far higher overheads than those of state-of-the-art academic RPC prototypes (e.g., eRPC [22]) which tend to focus on performance to the exclusion of production requirements such as authentication, privacy/encryption, and privilege model. In contrast, Stubby strongly favors feature richness to support the needs of tens of thousands of non-network-expert professional software engineers. For most use cases, Stubby's benefits far outweigh its costs, but for an in-memory KVCS, a minimum CPU cost of 50μs per op eclipses the CPU cost of simply accessing memory. Such a cost limits the usefulness of distributed caching, especially for systems with large working sets for which distributed storage is a critical means to husband expensive DRAM (§6.5).

RMA KVCS. A number of systems advocate using RMA as the network transport for KVCS dataplane primitives [17, 23]. Challenges arise when intersecting the core ideas of these systems with the requirements of Table 1. Systems built entirely atop RMA require careful coordination between client and server binaries, making post-deployment evolution a slow process. Simplifying assumptions around failures, hardware homogeneity, or viability of memory pre-allocation may simply not hold in practice, and can lead to performant but otherwise complex/impractical systems.

RMA/RPC hybrid KVCS systems. Hybrids [33, 35, 40] offer a middle ground, and can more obviously accommodate the aforementioned requirements around tolerance for heterogeneity and post-deployment agility. Such hybrid designs refute *RMA or RPC* as a false dichotomy, observing that both are useful building blocks for higher-level systems, and use of one can complement, rather than exclude, the other. We embrace this philosophy throughout the CliqueMap design.

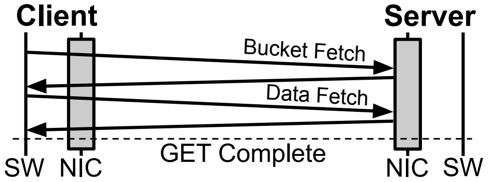


Figure 2: Sequence diagram of a 2×R GET in CliqueMap.

3 Overview and Productionization Ideas

CliqueMap is Google's production RMA/RPC hybrid KVCS, offering state-of-the-art performance and efficiency while meeting the production requirements outlined in Table 1. We first describe the basic design, then build on it in subsequent sections.

CliqueMap uses performant and CPU-efficient RMAs for common-case GETs, but RPCs for mutations and other traffic, thereby making it simpler to ensure consistency, enact subtle memory allocation and management techniques, and deliver new features over time.

Self-Validating Responses. Inspired by Pilaf [33], each KV pair in CliqueMap is guarded by a checksum across its key, value, and metadata. Since RMAs are not atomic, clients performing lookups always verify this checksum end-to-end (per KV pair). Checksum validation failures are attributed to *torn reads*, that is, an RMA read that observes intermediate state of a concurrent mutation of the underlying datum on the server—such failures are rare, but normal. Augmenting the checksum, additional metadata accompanies responses that ensures clients and servers agree on configuration, memory layout, and version. Clients retry lookups that fail validation steps at an appropriate level of the stack (§9).

Backend Data Structure (Figure 1). To support wide-ranging key and value sizes while handling key hash collisions, CliqueMap uses an associative hash table [29]. The backend data structure is composed of two logically-distinct RMA accessible regions—the *index region* and *data region*. The index region consists of fixed-size Buckets, and each Bucket contains a number of fixed-size tuples, known as IndexEntries. An IndexEntry is tagged with a key hash and has a pointer (a memory region identifier, offset, size) that indicates a position in the data region, wherein the actual KV pair is stored. Multiple DataEntries reside in the data region, which is managed by backend RPC handlers (§4).

2×R GETs (Figure 2). The baseline CliqueMap GET operation, which we refer to as *2xR*, relies on two RMA reads in sequence, operating on the index and data regions, respectively. (1) The client computes a hash mapping the Key (an arbitrary string) to a fixed-size KeyHash, which uniquely identifies a backend and Bucket associated with the Key. Then, (2) the client fetches the associated Bucket via an RMA read. (3) The client scans the Bucket for an IndexEntry with the desired KeyHash. If there is no match, then the client declares a miss. Otherwise (4), the client issues a second RMA read to fetch the potentially-matching DataEntry. (5) On completion of the second RMA read, the client validates the response by: (a) validating the checksum end-to-end, to guard against tearing, and (b) verifying that the DataEntry contains the desired Key (not merely KeyHash), guarding against a (very) rare 128-bit hash collision.

SETs. The client issues an RPC, including the KV pair to be SET, to the appropriate backend. On receiving a SET RPC, the backend allocates capacity for the new DataEntry in the data region, prepares an RMA-friendly pointer and scans the relevant Bucket in the index region for an existing mapping. If such an entry is found, the backend overwrites it with the new pointer, and reclaims the old DataEntry as free space. Backends apply SETs only when doing so monotonically increases a particular KV pair's *version* (see §5.2).

Clients. The CliqueMap client library transparently retries GET/SET operations to overcome transient failures, such as checksum validation failures that can arise under a race, subject to both a user-specified deadline and retry count. Retries occur at a layer appropriate to the error (e.g., checksum failures may retry RMA operations, but failed RMA operations may retry on new connections, etc.).

This basic design delivers RMA's intrinsic performance advantages for common-case read operations, and use of RPCs for SETs/ mutations eases the complexity of ensuring consistency, but not all key requirements are met. The challenges outlined in Table 1 call for further augmentation:

1. Memory efficiency via RPC-based mutations. CliqueMap allocates and manages memory capacity locally, triggered by RPCs that run entirely in the backends. Using straightforward code, these backends implement rich replacement and allocation policies, and can restructure the index and data regions on demand (e.g., via defragmention/replacement) or even trigger background processes (e.g., memory reshaping in §4.1). CliqueMap's self-validating lookup mechanisms ensure detection and retry of any resulting races. Importantly, these mechanisms make it possible for CliqueMap backends to resize their memory footprint on demand, rather than wastefully pre-allocate/pre-register for peak memory capacity on startup.

2. Lightweight replication with client-based quoruming. To deal with slow or failed servers, CliqueMap offers deployment modes in which data is replicated across multiple backends. To realize availability benefits with minimal performance cost, CliqueMap adopts an uncoordinated replication approach in which replicas do not synchronize in the serving path, and clients use load-aware quoruming to resolve data consistency.

3. Self-validating responses coupled with retries as a key building block. While self-validating responses and retries resolve race resolution in the basic design, we also find them to be key enablers in supporting seamless binary upgrades and recovering from failures. Self-validating responses from the server make the client aware of transient changes occurring at backends, and trigger the clients to retry at the appropriate layer of the stack. Due to the number of backends globally and weekly upgrade schedule, upgrades are the norm and this approach simplifies delivery of "hitless" upgrades.

4. Decoupled design for non–C-family clients. CliqueMap provides support for Java, Go, and Python clients via language-specific shims, enabling non–C-family internal components of a broader system to access the corpora stored in CliqueMap and thereby easing adoption of CliqueMap in established multi-language serving ecosystems. Each language shim launches a subprocess, which in turn contains the primary C++ client library, thereby side-stepping error-prone native-code invocation mechanisms, maximizing code reuse, and unifying debugging processes among all languages.

5. Leverage software transports for heterogeneity and performance. Because our datacenters operate across several generations of networks, CliqueMap tightly integrates with Google's software-defined NIC, Pony Express [31]. This integration includes a custom RMA operation that matches precisely the semantics of the CliqueMap GET operation, enabling most GETs to complete with a single network round trip (see §6.3). Further, CliqueMap offers RPCs as a seamless fallback for lookups in scenarios wherein RMA protocols are not applicable/available (e.g., lookup over WAN).

4 Backend Responsibilities

CliqueMap's backend memory layout is designed to accommodate 2×R-style GETs while also maintaining significant freedom to relocate data and change protocol over time, a balance relying heavily on the self-verifying properties realized by client-side validation and retry (§3). Critically, client-side validation ensures that server-side modification of a KV pair or associated metadata implicitly poisons the operation. Although rarely triggered in practice (less than 0.01% of all ops), such retries grant the backend code significant freedom when adjusting memory layout, simplifying both defragmentation and, later in the design's evolution, dynamic resizing. Ultimately, server-side logic need only be concerned about making retryable conditions transient, detectable, and rare, rather than entirely invisible.

4.1 Memory Allocation and Reshaping

Index Region Allocation and Reshaping. Index region memory allocation is straightforward and is initially provisioned on backend restart based on the expected key count in the underlying corpus. Crucially, indexes can be upsized at runtime when they surpass a target load factor. During such reshaping, the backend creates a new, second index, populates it, and then revokes remote access to the original index. At this point, client-initiated RMA operations fail; clients enacting retries for failed RMAs contact backends via RPC as part of their retry procedure for such errors, at which time the client also learns the new per-backend index size. For simplicity, mutations stall during an index resize.

Data Region Allocation. Because the *data region* is random-access in nature, the memory pool for DataEntries is governed by a slab-based allocator [11] and tuned to the deployment's workload. Slabs can be repurposed to different size classes as values come and go in the lifetime of the backend task. Because all allocations occur within an RPC, allocation logic can freely use the familiar programming abstraction provided by RPCs.

Data Region Reshaping. *Memory registration* for RMA is widely recognized to be expensive, as it requires the operating system to communicate with the RMA-capable NIC, to pin memory, and to manipulate translation tables. Naive designs that pre-allocate all backend memory at startup—and thereby avoid memory registration at runtime—are tempting, but we found this approach strands DRAM and thereby risks high operating costs. Taking advantage of freely-relocatable DataEntries, CliqueMap's DataEntry pool resizes on demand, so that deployments can provision for common-case, rather than peak, DRAM usage. When nearing current capacity, an individual backend task can asynchronously grow (reshape) its data region. Reshaping works by pre-allocating the maximum possible *virtual* address range needed to serve the entirety of a machine's memory capacity[1], but only ever populating a subset of the address

[1] via `mmap(PROT_NONE)` of a very large virtual range.

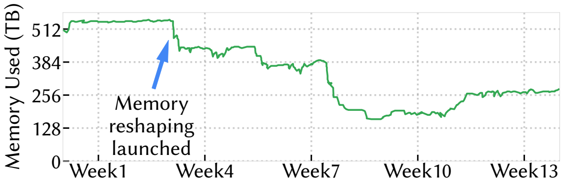


Figure 3: Memory reshaping in CliqueMap and subsequent DRAM savings.

range. That is, the data pool is always *virtually contiguous*, but not fully backed by DRAM. During expansion, CliqueMap establishes a second, larger, overlapping RMA memory window[2], and begins advertising this new window to clients as the data region. Clients converge over time to using the second window exclusively, perhaps after a retry. Because kernel memory management operations have unpredictable duration, CliqueMap initiates growth according to a high-watermark policy, performing such work off the critical path, but triggered by some RPC-based operation. As with the index region, data region downsizing occurs with non-disruptive restart.

Rollout of the reshaping feature saved 10% (50TB) of customer DRAM at launch (see Figure 3). Shortly thereafter the underlying corpus itself shrank, and without further human intervention CliqueMap dropped its DRAM usage by 50% (200TB). Since each individual backend makes an independent scaling decision, the aggregate savings is derived from the sum of many independent scaling decisions, and fluctuates in time.

4.2 Cache Eviction

A mutation (via RPC) triggers cache eviction when it encounters one of two conflict conditions:

- **Capacity Conflict.** No spare capacity in the data region. An eviction anywhere in the data pool suffices.
- **Associativity Conflict.** No spare IndexEntry in the key's Bucket. For the newly-installed KV to be RMA-accessible, an existing KV pair must be evicted from the Bucket.

The latter is a consequence of the set-associative but RMA-friendly data structure, which we mitigate by dynamically scaling the index (§4.1) to make associativity conflicts rare. CliqueMap also offers an optional RPC fallback in the case of a Bucket overflow, indicated by a bit in the Bucket. Clients encountering an overflowed bucket may optionally send an RPC to the backend [29], incurring higher latency but still serving a hit—a tradeoff appropriate when the downstream cost of a cache miss is high relative to the RPC cost.

Because CliqueMap uses RMAs for GETs, backends have no direct record of access information to facilitate recency-based eviction algorithms, such as LRU. Instead, clients inform backends of data touches via RPC, as a batched background process, to amortize RPC overheads. Backends ingest access records *en masse* to implement configurable eviction policies—LRU, ARC [32], and others.

Eviction Procedure. Like reshaping, the eviction mechanism again relies on the self-verifying properties inherent in the design. Because a checksum covers the IndexEntry and DataEntry in combination, the RPC handler processing an eviction can nullify IndexEntry pointers and modify DataEntry contents. Subtly, this admits interleavings in which 2×R GETs already in progress might still complete; this is acceptable as such GETs are considered ordered-before the eviction.

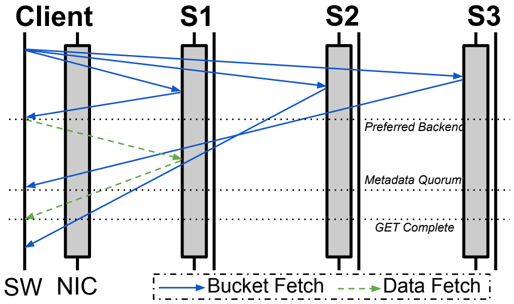


Figure 4: Sequence diagram of GETs in CliqueMap R=3.2 mode.

Since the new inbound KV may differ in size from the evicted entry, multiple evictions in an appropriate size class may be needed. Once sufficient space is available, a new DataEntry is written, followed by the relevant IndexEntry's pointer into the data region, which establishes an ordering point after which the new value is visible. In practice, evictions occur at roughly half the rate of SETs.

5 Availability

CliqueMap offers *replication* to ensure read/write availability in the face of unplanned failures and to provide some tolerance for slow backends. The replication scheme is designed to avoid inter-replica coordination to keep overheads low. Again, self-validating responses and retries play a role, assisting race resolution without the need for remote or global locking.

5.1 Quorumed GETs Under Three Replicas

When operating with three replicas, copies of each KV pair are assigned to adjacent backend tasks. I.e., for each key, CliqueMap uses a consistent key hash to first determine the backend number for a logical primary replica *i*–as if no replication existed–and then assigns copies of KV pairs to physical backends *i*, *i+1* and *i+2* (all mod *N*).

Next, we augment 2×R-style lookups to accommodate replication by performing an index fetch from all replicas (Figure 4). Although all three backends will respond, due to load differences or network proximity, one backend's response will arrive at the client first. The client then fetches the datum from the first responding backend, termed the *preferred backend*. Upon receipt of its second index response, the client can attempt a *quorum*—a per-KV-pair majority vote—to ensure consistency between replicas [19]. Deployments with three replicas and a quorum of two are known as *R=3.2*. R=3.2 CliqueMap cells are resilient against single failures[3].

CliqueMap augments each IndexEntry with a VersionNumber, which is globally unique and monotonic within a KV pair (§5.2). A GET under R=3.2 reports a cache hit if and only if (1) the DataEntry and its corresponding IndexEntry pass checksum validation (e.g., no torn value was observed); (2) at least two IndexEntries agree on VersionNumber and KeyHash (i.e., a version quorum exists); (3) the full Key in the DataEntry matches the requested Key (i.e., no hash collision); and (4) the DataEntry was fetched from a backend

[2] via subsequent `mmap()` calls to populate memory on demand.

[3] We proved single failure tolerance using TLA+ [27], a formal verification language.

with the quorumed VersionNumber (i.e., data came from a quorum member), again capitalizing on the tenet of self-verifying responses.

Under this protocol, a successful GET quorum thus requires responses from only two replicas, a property useful for both failure/race tolerance and performance, because a slow backend's response can be ignored when the remaining two agree. CliqueMap's first responder preference leverages this property by *speculating* that the preferred backend will form a quorum with at least one subsequent response; this is likely whenever the mutation rate is sufficiently rare, relative to the overall size of the corpus. When this speculation fails, i.e., when the first responder isn't part of the quorum, the client retries, preferentially fetching the datum from a distinct backend.

5.2 Multi-Replica Mutations

Clients perform mutations by sending RPCs to all replicas for a particular key. Each such mutation proposes a client-nominated VersionNumber, a tuple comprised of {TrueTime [12], ClientId, SequenceNumber}, such that each VersionNumber is globally unique and the VersionNumbers emitted by a particular client ascend monotonically. By using TrueTime [12]—a globally-consistent coordinated clock—for the uppermost bits, each client eventually nominates the highest VersionNumber for retried mutation operations, which is crucial for per-client forward progress. Specifically, a mutation proceeds at a backend only when the client's proposed VersionNumber is higher than the VersionNumber stored for each datum. As a result, each KV pair has a monotonically increasing VersionNumber, and all backends can independently agree on final mutation order, without requiring a common RPC arrival order.

Erases. ERASE operations are a special case of mutations. Like SETs, they are performed via RPC and make forward progress even when a replica is down. ERASEs also bear a client-nominated VersionNumber, retained so that late-arriving SETs cannot restore affirmatively-erased values. But unlike other operations, VersionNumbers for ERASEd elements cannot reside in the index region, since such a design untenably spends DRAM capacity for erased elements. However, VersionNumbers for erased elements need not be RMA-accessible, and so they are stored in a per-backend sideband data structure—a fully associative, fixed-size *tombstone cache* on the backend's heap. Further, a summary VersionNumber tracks the largest VersionNumber ever evicted from the tombstone cache. When reasoning about VersionNumber monotonicity, mutations consult the tombstone cache, its summary, *and* the contents of the index region. Because some erased elements' VersionNumbers are approximated (bounded above) by the summary VersionNumber, reasoning about ERASEd VersionNumbers is sometimes coarse-grained but never inconsistent.

Compare-And-Set (CAS). CAS operations are another special case of mutation. Like a SET, they install a new value, but *only* if the stored VersionNumber matches a provided VersionNumber. CAS provides a limited means of implementing conditional updates and reasoning about their success, with the provision that the VersionNumber is known a priori (e.g., memoized from a previous operation on the same key).

5.3 Race Conditions

Clients do not coordinate mutations, and mutations of the same key may occur near-simultaneously. These mutations interact without explicit synchronization with RMA-based GETs; race resolution hinges on the self-verifying properties of GETs and associated retries performed by CliqueMap clients. This strategy has the significant advantage that no expensive RMA-based synchronization is needed (e.g., remote locking), but the notable downside that forward progress is not guaranteed. With R=3.2, our design objective was to provide *obstruction free* [20] forward progress for GETs, notably, that they will succeed when they don't compete against a SET of the same key or encounter a failure condition reducing replica count below quorum. As such, it is possible for repeated mutations to starve GETs causing them to time out and report an error, once their retry count and/or deadline is exhausted. In practice, speed differential between RMA and RPC makes this a non-concern.

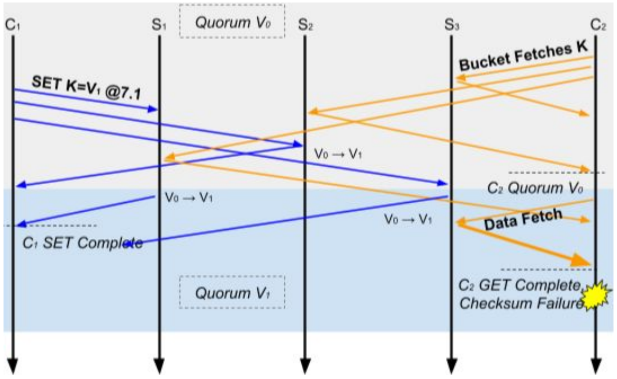


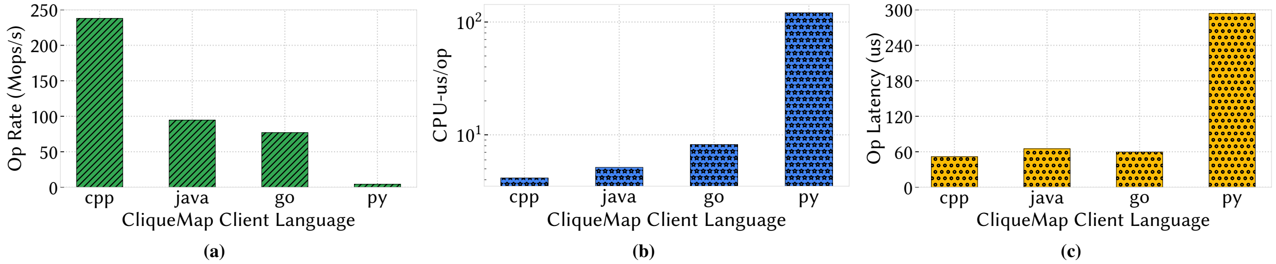
Figure 5: An example race condition, in which a GET initiated by client C_2 races against a SET initiated by client C_1, detected in CliqueMap R=3.2 mode by a self-validating response. C_2's GET attempt leads to a potential version quorum when it verifies that two fetched indices contain the same VersionNumber V_0, but the final data fetch can still collide against C_1's ongoing SET, since it is not atomic with respect to earlier accesses.

To exemplify race conditions that can arise in R=3.2, consider a race in which Client C_1 attempts SET $K = V_1$ while Client C_2 attempts a GET of K (see Figure 5), where initially $K \rightarrow V_0$. Depending on the precise timing and interleaving of operations at server replicas S_1 through S_3, C_2's GET attempt can result in quorum on value V_0 (V_1), wherein the GET is logically ordered before (after) the SET, or can result in a retryable checksum failure; in the example, checksum failure occurs when the slowest of the three data array mutations, namely the one at S_3, occurs during C_2's RMA data fetch operation, resulting in metadata quorum for V_0, but a torn read nonetheless.

Note that an *inquorate* outcome—wherein an operation cannot arrive at a quorum—is not possible when GETs race against single SETs. In contrast, a GET that races against multiple concurrent SETs, or experiences a failure condition (e.g., backend failure, torn read, etc.), may subsequently fail to achieve quorum. CliqueMap overcomes such races by retrying operations that fail due to a potential race.

5.4 Quorum Repairs

A key with a quorum of only two backends—instead of all three—is called a *dirty quorum*. In a dirty quorum, not all backends agree on



(a)

(b)

(c)

Figure 6: CliqueMap performance by client language - (a) Op rate; (b) CPU cost and (c) Median Op latency.

a key's existence or its VersionNumber. Dirty quorums arise due to backend task failures, uncoordinated eviction (~1 in 7M GETs), and RPC failures.

A second failure, such as the loss of an additional backend, causes the dirty quorum to degrade to an inquorate state, which is treated as a cache miss. However, because the cost of a cache miss can be high (e.g., may require reading data from persistent storage), CliqueMap supports *quorum repair* in which a backend triggers an explicit on-demand recovery, sourcing from the remaining two healthy replicas.

To manage the risk of a dirty quorum degrading to an inquorate state, backends independently scan their cohorts (i.e., the other two backends) for missing or stale KV pairs, detected via KeyHash exchange to minimize overhead. A backend observing a dirty quorum performs an on-demand repair on a key-by-key basis by: (1) issuing a SET to the dirty backend to install the missing key K at a new VersionNumber N and (2) updating the VersionNumber of the key K to N at the repairing backend as well as the other (clean) backend. This repair procedure ensures that all the three backends settle on a consistent view of Key K at VersionNumber N. We tune the inter-scan interval to suit the needs of the deployment; tens of seconds is typical.

A similar process operates *en masse* whenever a backend restarts after an unplanned failure, such as a crash. Specifically, restarted backends request repairs from the other two healthy backends in their cohort.

6 Evolution

We next discuss a variety of changes we made to CliqueMap after initial deployment, which reflect challenges we did not anticipate in initial design. We've found that CliqueMap's core design ideas—self-verification, retries, and use of RPC in general—substantially eased this evolution.

6.1 Warm Spares for Planned Maintenance

CliqueMap initially relied entirely on repairs (§5.4) to tolerate regular binary upgrades and server maintenance, as well as crashes. However, upgrades are extremely common–essentially *always* in progress–due to scale and staged rollout practices (§3). Such rollouts would effectively drop all data from R=1 deployments, since no repairs are possible without replication. Demand for less-disruptive rollouts for R=1 configurations motivated the addition of warm *spare backends*, which temporarily host the shard of a backend undergoing maintenance.

A backend task notified of planned maintenance, e.g., a new binary rollout, triggers a migration of its identity and data to a warm spare. In the course of normal RMA-based GETs, clients may discover in-progress backend migrations via a configuration ID stored in each Bucket. Specifically, during response validation, if a client observes that the configuration ID in the fetched Bucket does not match its expectation (established at connection-time alongside other RMA-relevant metadata), the client enacts a retry by refreshing its configuration from an external high-availability storage system [13, 15]. In doing so, the client discovers all migrations in flight and (temporary) roles of any spare backends.

Although initially motivated by R=1, sparing is effective for R=3.2 to avoid even transient dirty quorums. Maintaining three healthy replicas via sparing during planned maintenance ensures that quorums remain clean and the system still tolerates unplanned failures that occur during rollouts. We now deploy a small number of spares in almost all cells.

6.2 Extending Beyond C++

Google datacenters run software primarily written in C++, Java, Go, and Python, each representing pools of thousands of developers. Launched initially with only C++ support, CliqueMap was, for a time, unavailable to many of Google's developers, and corpora stored in CliqueMap were only accessible by systems composed entirely of C++. Because serving stacks atop RPC are built from components in many languages, we required a solution to *broaden access* across languages, even if not at the full performance envelope of our native C++ client.

Programming APIs for RMA are typically user-level libraries in C or C++ (e.g., *libibverbs* [6]). Supporting RMA in other languages is challenging, as RMA operates on essentially raw memory abstractions—pointers, offsets, etc. One might conclude the obvious approach for RMA from these languages is to leverage native code invocation mechanisms (e.g., JNI in Java) to directly invoke an underlying C library. We do not take this approach because it would require us to maintain language-specific implementations of the CliqueMap client library, which would greatly increase complexity (e.g., of changing the library functionality over time). Instead, we provide a lightweight shim for each language, which in turn launches the CliqueMap C++ client as a Linux subprocess. We communicate between these processes using *named pipes*, which are simple abstractions available in all these languages.[4] The subprocess approach is a tradeoff—it avoids per-language maintenance burden and complexity for each non-C++ language, while sacrificing some efficiency.

[4] We also developed a shared memory mechanism specifically to accelerate Java.

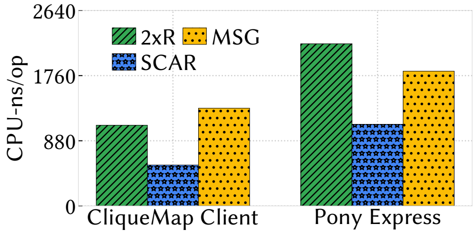


Figure 7: CliqueMap client and Pony Express (client and server-side) CPU efficiency under different lookup strategies.

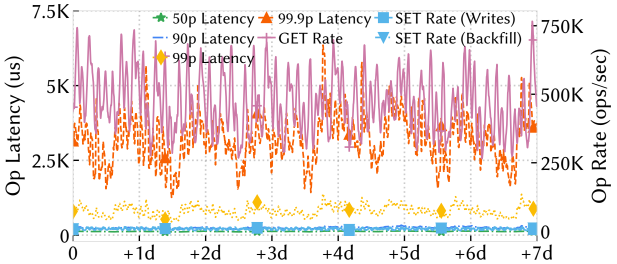


Figure 8: Ads Workload.

Figures 6a and 6b summarize performance of a large synthetic workload wherein 500 clients retrieve random keys from 500 backends. Figure 6a plots the maximum GET op rate achievable in each language, using 64B objects fetched at maximum possible rate, and Figure 6b plots the associated CPU cost per operation. Figure 6c plots lookup latency at 1K/GETs/sec/client (not peak). While overheads can be significant relative to the C++ baseline, we have found that the resulting implementations are still performance-competitive relative to RPC counterparts, though admittedly non-optimal. We anticipate further optimization in this area (§9).

6.3 From 2×R to Scan-and-Read

Google's datacenters are highly heterogeneous, i.e., not all deployments support all RMA protocols. The 2×R GET strategy is generic and viable on a variety of transports (Pony Express [31], 1RMA [34], and RDMA), but programmable transports like Pony Express and emerging SmartNICs offer opportunities to optimize specifically for KV stores [8]. When operating atop Pony Express, we leverage a custom-built RMA-like primitive, called *Scan-and-Read (SCAR)*, wherein we perform the "scan" of the Bucket server-side, in Pony Express, and return the combined Bucket and DataEntry to the initiating client. By performing a small computation in the server-side NIC, SCAR removes a full round trip from the critical path, and also reduces the fixed, per-op CPU overheads of a full second RMA operation from both Pony Express and the CliqueMap client. While small in the absolute sense, these overheads are large relative to the handful of memory accesses required to perform the SCAR operation.

Figure 7 shows CPU efficiency of the CliqueMap client and of Pony Express under three distinct lookup strategies: 2×R, SCAR, and a two-sided messaging/RPC approach (MSG). Overall, we find that an individual SCAR operation is about as costly as a normal Pony Express RMA read. Because it halves the total number of RMA operations per GET, SCAR is substantially more efficient on the whole than 2×R. As a further point of comparison, the overheads needed to wake server-side application threads to process and respond to inbound messages (as is in the two-sided messaging case, MSG) significantly exceed the CPU cost of simply performing SCAR's Bucket scan.

SCAR is now in widespread use, as it is especially helpful for corpora with typically-small object sizes. However, SCAR has an occasional downside: the combination of R=3.2, SCAR, large object size, and high lookup concurrency can transiently incast the CliqueMap client, as SCAR solicits three full copies of the datum. When combined with scarce downlink bandwidth (e.g., due to older or oversubscribed hardware), this incast can lead to higher tail latency (§7.2.2).

6.4 R=2/Immutable Mode

Google has a rich ecosystem of durable storage systems [14–16] which utilize persistent storage media to avoid data loss. To reduce lookup latencies relative to persistent storage and DRAM requirements relative to R=3.2, we added an R=2 mode to CliqueMap, in which an immutable corpus is loaded from an external system of record. Because the data is immutable, only one replica need be consulted for most operations, although the second replica still serves in the event of a failed backend task. As such, R=2/Immutable resembles CliqueMap R=1, in terms of its network behaviors, and has data availability properties that tolerate single-backend failures.

6.5 Disaggregating Local State

CliqueMap was initially intended to displace RPC-based KVCS for CPU efficiency improvement of applications that accessed a distributed corpus. However, we found that CliqueMap's latency was sufficiently low that some serving stacks that relied on serving data shards from local memory could instead access those corpora over the network from CliqueMap. Importantly, remote access allows these serving tasks to become *stateless*. Statelessness allows compute to scale independently from DRAM (holding data), leading to overall improved resource efficiency. Increased strategic focus on disaggregation occurred well after initial launch and minor features enabling such use cases were added, e.g., customizeable hash functions.

7 Evaluation

This section first presents measurements of CliqueMap production workloads, and then, using synthetic workloads, we present controlled experiments that highlight CliqueMap behaviors in specific scenarios, including the impact of previously discussed productionization features.

7.1 Production Workloads

We highlight two production serving workloads: *Ads* (Figure 8), from a datacenter in The Dalles, Oregon (USA), and *Geo* (Figure 9), serving road traffic predictions from a datacenter in Lenoir, North Carolina (USA).

The Ads workload is a portion of the serving pipeline for Google's advertising business on third-party Internet properties. Advertising data is keyed by topic and fetched on-demand from CliqueMap cells (R=3.2) when participating in an auction process; late responses to such auctions are discarded and hence response time is critical to revenue opportunity. The graph supports CliqueMap's design

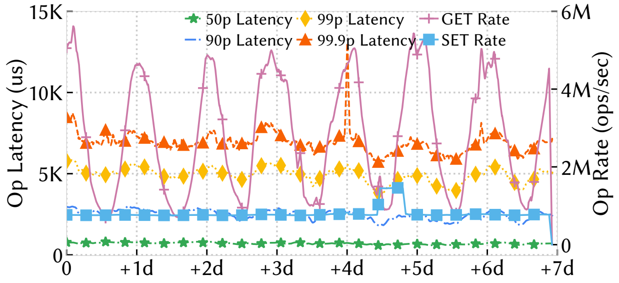


Figure 9: Geo Workload.

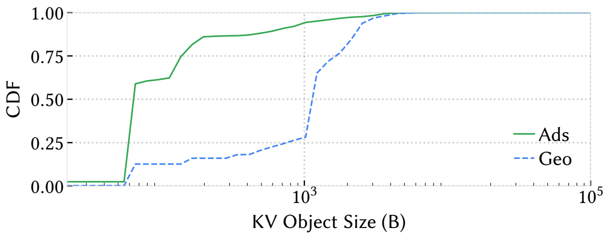


Figure 10: Ads and Geo Object Size Distribution.

decisions to optimize for a GET rate that greatly exceeds the SET rate. Ads fetch tends to be highly batched; batch sizes reach 30-300 KV pairs in the 99.9th percentile tail, which makes the client the bottleneck due to response incast, pushing 99.9% GET tail latency into the vicinity of 5ms.

Geo is a serving workload that provides estimates of traffic conditions on roadways throughout the world. It feeds into, e.g., driving directions suggested to end-users. The corpus is keyed by a road segment identifier, and stores a compact representation of utilization of the road segment in question. Like Ads, lookups are highly batched, usually consisting of tens of segments at a time. The underlying model experiences a high update rate. The Geo workload serves highly diurnal GET traffic intermixed with a background update rate for the corpus (SETs), originating from different client jobs. Despite the 3x variation in GET rate over the course of a day, 99.9% tail latency varies minimally. Like most CliqueMap workloads, GET performance is critical; less so for SETs.

These workloads are typified by values of different sizes; Figure 10 plots the CDF. For both workloads, objects tend to be small, typically at most a few KB (importantly, smaller than our typical MTU size), but there is a tail of larger objects.

7.2 Controlled Experiments

Next, we discuss the quantifiable behaviors of the CliqueMap design through a set of controlled experiments using synthetic workloads.

7.2.1 Preferred Backend Selection Benefits

To highlight the effect of quoruming to reduce tail latency in R=3.2, we set up a synthetic workload with a small, 3-backend R=3.2 CliqueMap cell, configured to use 2×R. Synthetic clients repeatedly GET the same 4KB-sized K/V pair. We then place one of three backends under load from an antagonist, which offers ~95Gbps of competing demand through its NIC. Figure 11 plots the resulting normalized median and tail latencies.

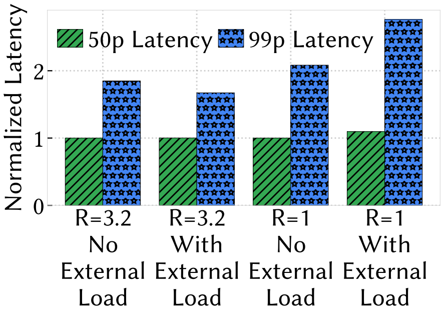


Figure 11: Preferred backend selection benefits under varied server host load; normalized to no-load.

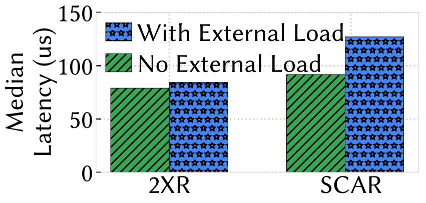


Figure 12: SCAR vs. 2×R performance under varied client load.

Takeaway: Preferred backend selection in R=3.2 tolerates a single slow server; there is almost no elevation in latency (within noise margins). In comparison, R=1 is obliged to rely on load to a slow server, and hence both median and tail suffer due to the overtaxed backend.

7.2.2 SCAR and Incast

§6.3 outlines our addition of SCAR to Pony Express, and because of its advantages we deploy SCAR to most production cells. But not all tail latency metrics improved when we introduced SCAR, as SCAR has a potential downside: when deployed with R=3.2, SCAR solicits *three full copies* of the datum, whereas 2×R solicits only one, plus three IndexEntries. That is, SCAR transiently incasts its client, which can be problematic when batch sizes or values are large.

Figure 12 plots the behavior of SCAR and 2×R when fetching relatively large (64KB) values, with and without competing load applied to the client (which exacerbates the incast condition). The difference in median GET duration is evident; because SCAR transfers 195KB per op (3× 64KB values and 3× 1KB Buckets), it begins to lag behind 2×R's 67KB transfer (1× 64KB value and 3× 1KB Buckets), *despite* SCAR's single round-trip advantage. The precise constants leading to this effect vary with technology generation; older, slower hardware observes this effect at smaller value/batch sizes.

Takeaway: Deploy SCAR when values/batch sizes are small relative to NIC speeds. It's acceptable to redundantly fetch data when individual KV sizes are small.

7.2.3 Maintenance

Maintenance, binary upgrades, and reconfigurations are ubiquitous, and hence CliqueMap performance during these events is critical. We next highlight the performance of CliqueMap GETs during repairs and data migrations.

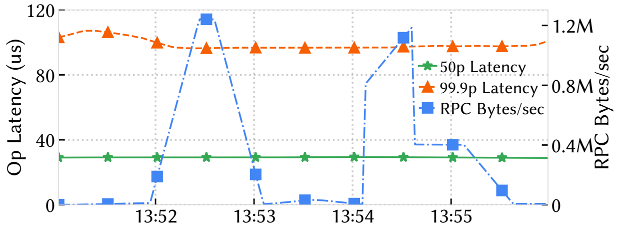


Figure 13: CliqueMap Planned Maintenance via Spares at consistent 100K GET/sec.

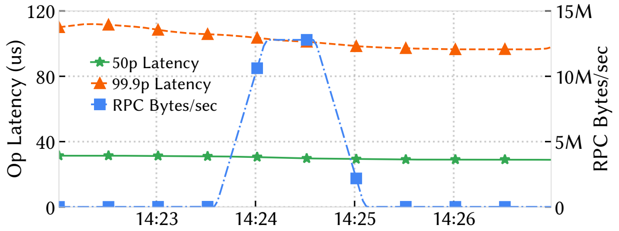


Figure 14: CliqueMap Unplanned Maintenance via Repairs at consistent 100K GET/sec.

In this scenario, we load an R=3.2 CliqueMap cell with 100K GET/sec from ten clients, and then inject artificial *planned* and *unplanned* maintenance events. In the former case, CliqueMap is notified of impending primary (non-spare) backend restart, and hence, can utilize warm spare backends gracefully; in the latter case, only post-restart repairs are possible. Figure 13 plots GET latency percentiles and RPC byte rates (from repairs and sparing) during the *planned* event. We inject the planned restart at 13:52:00. Immediately, the notified primary backend transfers its data to a spare (RPC traffic), and exits by 13:53:30. At 13:54:05, the spare returns the transferred data to the newly-restarted primary (RPC traffic again).

Takeaway: Warm sparing effectively hides planned maintenance from clients. Throughout the event, we see virtually no change in the client-observed latencies; fewer than 1 op in 1000 observes degraded performance, e.g., from retries.

Finally, we forcibly crash a backend to simulate an *unplanned* maintenance event (Figure 14), at 14:22:00 (not evident from graph). The new backend restarts on another host by 14:23:30, and we observe a significant burst of RPC activity as repairs are performed. Latency fluctuates slightly during this interval, and even experiences a downward trend, as the clients perform less total work when the cell is degraded–after observing a connection failure, clients only send two out of three operations per GET, as they await reconnect.

Takeaway: Repairs augment warm sparing and have little performance impact under realistic load levels.

7.2.4 RMA Deployment Characterization

We next characterize the performance of moderate-sized CliqueMap cells under controlled load in homogeneous RMA deployments. We use a 950-host testbed with synthetic load generation, as the details of transport operation are difficult to isolate and study *in situ* in production workloads. Our testbed is equipped with Skylake-class CPUs and connected with a fabric capable of 50Gbps sustained and 100Gbps burst per host. To highlight the networking behaviors at scale, we operate a 500-backend R=1 CliqueMap cell; when using Pony Express [31] we enable SCAR, but use 2×R fetches when using 1RMA [34]. We use a fixed value size of 4KB, which, with framing and metadata, allows a GET response to fit within a single 5KB-MTU frame.

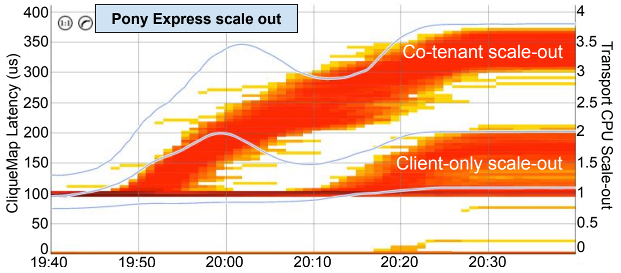


Figure 15: Pony Express scale out as a heatmap, wherein darker red indicates a larger fraction of machines scaled out to, on average, the number of cores reflected on the right axis. Lines demarcate 50th/90th/99th CliqueMap latencies on the left axis.

Pony Express Load Ramp. Pony Express can scale out to additional CPU for network Tx/Rx activities. We configure each client and backend to spread its load among four Pony Express *engines*. Engines are single-threaded and may time-multiplex a single core or each scale out to their own core in response to load. Figure 15 plots the GET latency percentiles as we ramp request rate to 400M GET/sec (among 10K client tasks, which is 800K ops/sec/backend). We overlay the degree of Pony Express scale-out on the right axis, as a heat map, wherein darker red indicates a larger fraction of machines scaled out to, on average, the number of cores reflected on the right axis.

The scale-out plot reveals two bands, respectively corresponding to hosts occupied by only CliqueMap clients (average 10.6 clients per host) and those also hosting a backend (500 such systems with one backend each). Hosts occupied by both CliqueMap backends *and* clients (co-tenant) are busier on average, and hence Pony Express scales out on these hosts first, reaching ~3.5 CPU/host. But as load continues to rise, client-only hosts also surpass scaling thresholds, and begin to scale out at 20:00, and *en masse* by 20:10, reaching ~1.5 CPU/host. The client-side scale-out process significantly reduces tail latency even as load continues to ramp up, because receive transfer parallelism is achieved within individual clients.

Takeaway: The combination of CliqueMap and Pony Express has significant capacity headroom. For near-term technologies, we don't expect significant design changes needed to realize further performance, because with tuning we can drive our system's op rate in a single cell well beyond the current demand.

1RMA Load Ramp. Figure 16 plots a similar experiment using 1RMA. 1RMA offers a different set of tradeoffs—in contrast to Pony Express, 1RMA's serving path is entirely hardware. However, 1RMA doesn't offer the SCAR primitive, and hence each lookup operation must use 2×R and incur two fabric round-trip times (RTTs) per operation. 1RMA also significantly optimizes interaction between the NIC and the server memory system via PCIe, so the application-visible RTT for 1RMA is lower than with more traditional packet-oriented systems.

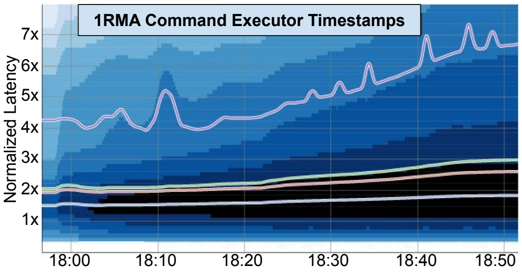


Figure 16: 1RMA Ramp: Fabric+PCIe timestamps during load ramp. Lines demarcate 50th/90th/99th/99.9th percentiles.

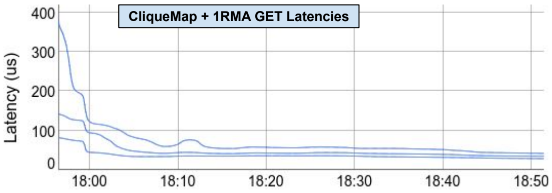


Figure 17: 1RMA Ramp - GET Latencies. Lines demarcate 50th/90th/99th percentiles.

We again ramp load from 0 to 400M GET/sec. The figure overlays a heatmap of the timestamps emitted by the 1RMA NIC, a hardware measurement of the combined latency of fabric and remote PCIe. At peak, this workload demands only ~32Gbps from server-side PCIe on average and hence we expect its latency to not be substantially elevated. Combined fabric/PCIe latency rises marginally with load, still well short of saturating the network. End-to-end GET latency, shown in Figure 17, is dominated by CPU time spent in the CliqueMap client, as depicted by the mostly unchanging latency distribution of CliqueMap GETs themselves. Perhaps surprisingly, the highest latency is observed at the *lowest* load, an effect we often see when our testbed is otherwise idle, due to power-saving C-state transitions at low load. By roughly 250K GET/sec/client, delays from C-state transitions have disappeared entirely and total latency remains insensitive to load.

Takeaway: RMA Infrastructure heterogeneity means there's no single optimal lookup method–the choice depends on the underlying infrastructure, and hence a system's ability to evolve over time matters. Counter-intuitively, despite requiring two fabric round-trips per GET, the simple and generic 2×R fetch strategy can outperform the SCAR-based strategy under load in this testbed, as the all-hardware 1RMA serving path incurs no software bottleneck on the serving side. Because CliqueMap can leverage a variety of transports and fetch algorithms atop them, CliqueMap can provide users a relatively uniform performance envelope, taking advantage of scale-out to do so in environments lacking significant offload.

7.2.5 Workload Variance

Figure 18 plots latency and Figure 19 plots CPU usage of CliqueMap backends under varying mixes of GETs and SETs. These are unlike our previous graphs, which differentiate GET from SET performance. It is no surprise that greater percentages of RPC-based SETs incur greater overheads and worse typical latency, as progressively more of the workload is unable to use RMA.

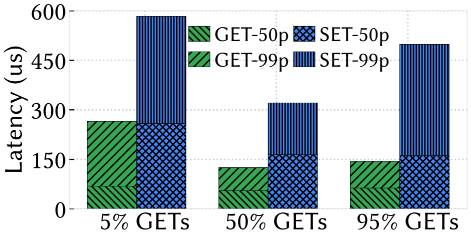


Figure 18: CliqueMap latencies under varying mixes of GETs and SETs, under fixed value size 4KB.

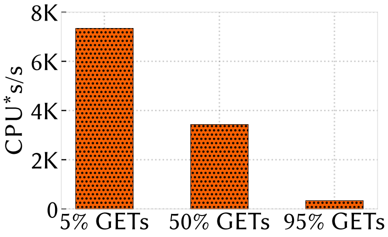


Figure 19: CliqueMap CPU cost under varying mixes of GETs and SETs, under fixed value size 4KB.

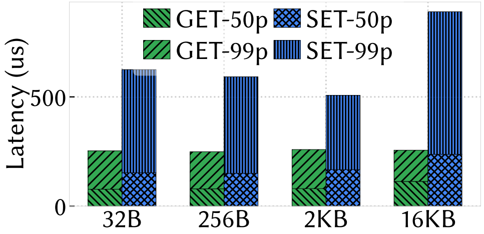


Figure 20: CliqueMap performance under varying value sizes.

We consider the effect of value size at fixed GET rate in Figure 20. For values sizes common in our production workloads, individual GET and SET performance are dominated by fixed costs–i.e., costs per op, not costs per byte–as our value sizes tend to be small (Figure 10).

Takeaway: CliqueMap delivers on its intent of providing nominal lookup latencies across diverse workloads.

8 Related Work

Since its inception, CliqueMap's goal has been to deliver the performance of state-of-the-art KVCSs in the literature (e.g., Pilaf [33], HERD [23], MICA [29], FaRM-KV [17, 18], and others [36, 39, 40]) to Google datacenters, adapting ideas as needed to meet the practical requirements of our environment. These requirements differ among hyperscale operators. For instance, the ubiquity of Stubby RPCs and ALTS [2] at Google means that third party solutions (e.g., *memcached*) aren't directly applicable. Rather, as with Twemcached [41], such solutions require a fair amount of investment to reach productionizeable feature sets. We note significant agreement between our productionization challenges and those discussed by Facebook [10], with differences in approach arising from the underlying technologies involved.

From Pilaf and other systems [36, 39], we take the key insight to compose RMA and RPC, as well as specific techniques to do so, such as self-validation. Elements of our design also resemble RPC-based systems. For example, when considering the combination of Pony Express and CliqueMap, SCAR (§6.3) bears similarity to message-oriented, rather than strictly RMA-oriented lookup strategies, akin to those in HERD [23]. Likewise, SCAR itself resembles a highly-specialized RPC [8, 22].

When evaluating solutions for availability and replication, we opted for quoruming over the primary/backup architecture adopted by HydraDB [40] and FaRM [18] so that we could take advantage of *preferred backends* (§5). To ensure that replication does not lead to significant overheads, we opted for client-side quoruming rather than indirection through a server to avoid serialization points (e.g., ZAB [21], CR [38]) and inter-replica communication (e.g., ABD [9, 30], Paxos [28] for reads; Hermes [26], CR [37, 38] for writes).

Lastly, we sought to build CliqueMap so that no potential user required special privilege to operate it, because such privileges impose undesirable adoption hurdles. This design choice ultimately made Unreliable Connected/Unreliable Datagram transports, as used by HERD [23] and FaSST [24], infeasible as sub-components. Instead, we rely on indirection through Pony Express to avoid binding to any high-privilege network APIs. Similarly, low default privilege levels make it difficult to realize predictable CPU and NIC siloing in our environments, despite the performance advantages demonstrated by MICA [29].

9 Experience and Conclusions

CliqueMap highlights the importance of complete system design that focuses on performance, robustness, and efficiency, by deploying high-performance/low-programmability RMA primitives on critical performance paths, and highly agile but less-efficient RPCs for other functionality. This division of labor capitalizes on the needs of a particular set of critical serving workloads—those with stringent demand for performance in serving paths. At the same time, it meets the expectations for productionization feature-richness of RPC based systems. To conclude, we summarize broad takeaways for the networking and systems-building communities based on our experiences derived from building CliqueMap.

Leverage RPC, in composition with RMA, to maintain post-deployment agility. Production services undergo requirement changes throughout their lifetimes. Ultimately, CliqueMap's lookup path is the only path heavily tailored for RMA, and the system is relatively easy to adapt to changes as a result. Throughout the design, we embrace the use of RPCs for control and management actions, and as options for dataplane operations, affording opportunities to refine the design and support new features: sparing for planned maintenance, diverse eviction algorithms, compression, and new mutation types. Systems maintainers for all-RMA designs tend to find even trivial changes (e.g., to memory layouts) a major challenge.

Enable multi-language software ecosystems. It is tempting to focus solely on performance, and hence, low-level languages such as C and C++. Early in CliqueMap's lifetime, we even turned away potential customers rooted outside these languages. In retrospect, a C++-only approach stunted growth and adoption, as our datacenters are vivid multilingual environments, grown out of the ease of development afforded by RPC. CliqueMap's language support makes it a viable option for many thousands of developers at Google. Whereas our current approach—named pipes to a subprocesses—meets our performance requirements well, it is not optimal. We believe that this area is ripe for the research community's future innovations in exploring new tradeoffs between maintenance burden, complexity, efficiency, and performance.

Don't compromise memory efficiency. We initially envisioned CliqueMap would provision for peak DRAM usage, to work around the notorious difficulties of *memory registration* [6]. As a result, early potential adopters faced a tradeoff: faster lookups for (perhaps) higher DRAM usage. Such trade-offs are difficult to analyze, as DRAM cost isn't uniform in time or geography. The right call in one datacenter might be wrong in another. By investing in memory efficiency while preserving lookup performance, CliqueMap became significantly more appealing.

Simplify design with self-validating server responses and client retries. We found the design pattern of combining self-validating responses with client-side retry greatly simplifying, as clients become resilient to a variety of hazards across all layers of the stack; self-validation can tolerate RMA operation failure, data races among competing mutations, backend configuration changes, and even wire protocol format changes. Through retries at the appropriate abstraction level, CliqueMap near-seamlessly handles these cases. A notable drawback of this approach is that GET forward progress is not guaranteed; nevertheless, we have found this drawback can be managed through (mostly-automated) tuning.

Programmable NICs offer advantage through specialization. Hardware implementations of RMA offer stunning performance envelopes, but software NICs offer continuous innovation and post-deployment customization. We could not have deployed Scan-and-Read (§6.3)—an optimization saving an entire RTT—without an underlying software NIC, Pony Express. The superior expressivity and reprogrammability of software NICs gives them a notable edge over faster-but-inflexible all-hardware designs, and helps bridge gaps caused by heterogeneous hardware deployments by deploying hardware-NIC-agnostic protocols. As so-called SmartNICs continue to emerge, opportunities to optimize them for serving systems will grow.

We recommend that designers of future infrastructure take advantage of these guidelines when building systems for hyperscale datacenter environments: maintain agility, sacrifice neither common-case performance nor DRAM efficiency, enable customers' practical needs, and adapt to the underlying technology landscape. These tenets underlie CliqueMap's design, execution, and evolution over time, and have led to a production-friendly and practically useful design point.

This work raises no ethical concerns.

Acknowledgments

We would like to thank early reviewers Jeff Mogul, Jason Hsueh, Jeff Hightower, and Philip Wells. Likewise we thank the anonymous SIGCOMM reviewers and our shepherd, Nathan Bronson, for providing valuable feedback. Lastly, we thank the production, serving, and support teams at Google for their partnership and contributions to the work—including but not limited to the Pony Express, 1RMA, Ads, Geo, and Travel teams.

References

[1] 2020. Chelsio Terminator 6 NICs. https://www.chelsio.com/terminator-6-asic/.

[2] 2020. Google's Application Layer Transport Security. https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security.

[3] 2020. Marvell FastLinQ 41000 Series Ethernet NICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/41000-ethernet-adapters.html.

[4] 2020. Memcached. http://memcached.org/.

[5] 2020. Nvidia Mellanox Connect-X NICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/.

[6] 2020. RDMA Core Userspace Libraries (libibverbs). https://github.com/linux-rdma/rdma-core.

[7] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*. 120–126.

[8] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Remote Memory Calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*. 38–44.

[9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.

[10] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 753–768.

[11] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel. In *USENIX Summer 1994 Technical Conference (USTC'94)*.

[12] Eric Brewer. 2017. *Spanner, TrueTime and the CAP Theorem*. Technical Report. https://research.google/pubs/pub45855/

[13] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*. 335–350.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[16] Jeffrey Dean. 2010. Evolution and future directions of large-scale storage and computation systems at Google. (2010). https://research.google/pubs/pub44877/

[17] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the Eleventh USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 54–70.

[19] David K Gifford. 1979. Weighted voting for replicated data. In *Proceedings of the seventh ACM Symposium on Operating Systems Principles (SOSP'79)*. 150–162.

[20] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* 522–529.

[21] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *41st International Conference on Dependable Systems & Networks (DSN'11)*. 245–256.

[22] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceeding of Sixteenth USENIX Symposium on Networked Systems Design and Implementation*. 1–16.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 Conference of ACM SIGCOMM*. 295–306.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 185–201.

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 158–169.

[26] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: a Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 201–217.

[27] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

[28] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[29] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 429–444.

[30] Nancy A Lynch and Alexander A Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. 399–413.

[32] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*. 115–130.

[33] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (ATC'13)*. 103–114.

[34] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. 708–721.

[35] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost memcached. In *In 2012 USENIX Annual Technical Conference (ATC'12)*. 347–353.

[36] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 1–15.

[37] Jeff Terrace and Michael J Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *2009 USENIX Annual Technical Conference*. 1–16.

[38] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*. 7.

[39] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*. 1–13.

[40] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 1–11.

[41] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 191–208.