Beyond Simple Request Processing with RAMCloud

Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman
University of Utah

Abstract

RAMCloud is a scale-out data center key-value store designed to aggregate the DRAM of thousands of machines into petabytes of storage with low 5 µs access times. RAMCloud was an early system in the space of low-latency RDMA-based storage systems. Today, it stands as one of the most complete examples of a scalable low-latency store; it has features like distributed recovery, low fragmentation through a specialized parallel garbage collector, migration, secondary indexes, and transactions.

This article examines RAMCloud's key networking and dispatch decisions, and it explains how they differ from other systems. For example, unlike many systems, RAMCloud does not use one-sided RDMA operations, and it does not partition data across cores to eliminate dispatch and locking overheads.

We explain how major RAMCloud functionality still exploits modern hardware while retaining a loose coupling both between a server's internal modules and between servers and clients by explaining our recent work on Rocksteady, a live data migration system for RAMCloud. Rocksteady uses scatter/gather DMA and multicore parallelism without changes to RAMCloud's RPC system and with minimal disruption to normal request processing.

1 Introduction

Remote direct memory access (RDMA) has been a recent hot topic in systems and database research. By eliminating the kernel from the network fast path and allowing the database to interact directly with the network card (NIC), researchers have built systems that are orders of magnitude faster than conventional systems both in terms of throughput and latency.

RAMCloud is an example of early research in this space. RAMCloud [15] is a key-value store that keeps all data in DRAM at all times and is designed to scale across thousands of commodity data center nodes. In RAMCloud, each node can service millions of operations per second, but its focus is on low access latency. End-to-end read and durable write operations take just 5 µs and 14.5 µs respectively on our hardware (commodity hardware using DPDK). RAMCloud supports distributed unordered tables, transactions [10], and ordered secondary indexes [9].

Internally, RAMCloud servers leverage modern NIC features for high performance, but, in many ways, its internal networking and dispatch structure is somewhat conventional. Other research systems have more aggressive approaches that use techniques like simpler data models (for example, distributed shared memory), strict workload partitioning across cores, or one-sided RDMA operations to eliminate server CPU involvement

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

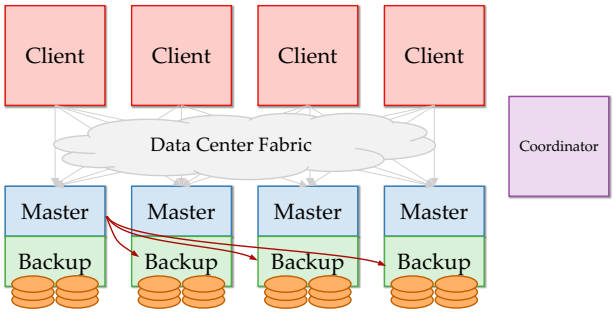

Figure 1: RAMCloud Architecture.

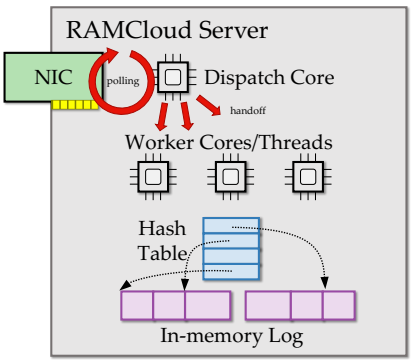

Figure 2: Internal Server RPC Dispatch.

and improve access latency [3, 4, 8, 6, 12]. Today, other systems achieve inter-machine latencies of 2.5 µs and serve hundreds of times more operations per second per machine [11, 16].

Despite this, RAMCloud's internal design has advantages. This article considers RAMCloud's internal server networking, dispatching, and synchronization; extracts lessons from its design choices; and discusses its tradeoffs. RAMCloud's design results in loose coupling between servers and clients and between the internal modules of servers; it also helps support many of the "extra" things that servers in a large cluster must do. To be useful, low-latency scale-out stores must evolve from micro-optimized key-value stores to more robust systems that deal with fault-tolerance, (re-)replication, load balancing, data migration, and memory fragmentation.

Finally, we exemplify RAMCloud's principles by describing the design of a new live migration system in RAMCloud called Rocksteady. Rocksteady uses scatter/gather DMA while building only on simple RPCs, and it runs tightly pipelined and in parallel at both the source and destination while minimizing impact on tail latency for normal request processing.

2 RAMCloud Networking, Transports and Dispatch

In RAMCloud, each node (Figure 1) operates as a *master*, which manages RAMCloud records in its DRAM and services client requests, and a *backup*, which stores redundant copies of records from other masters on local SSDs. Each cluster has one quorum-replicated *coordinator* that manages cluster membership and table-partition-to-master mappings.

RAMCloud only keeps one copy of each record in memory to avoid replication in expensive DRAM; redundant copies are logged to (remote) flash. It provides high availability with a fast distributed recovery that sprays the records of a failed server across the cluster in 1 to 2 seconds [14], restoring access to them. RAMCloud manages in-memory storage using an approach similar to that of log-structured filesystems, which allows it to sustain 80-90% memory utilization with high performance [17].

Request processing in each server is simple (Figure 2); its key design choices are explained here.

Kernel Bypass and Polling. RAMCloud processes all requests with no system calls. It relies on kernel bypass to interact directly with the NIC. Interrupt processing would introduce extra request processing latency and would hurt throughput, so RAMCloud employs a dispatch thread pinned to a single CPU core that polls NIC structures to fetch incoming requests. When a RAMCloud process is completely idle, it still fully consumes a single core. As the number of cores per socket increases, the relative efficiency of polling improves. Aside from keeping data in DRAM, these two optimizations alone account for most of RAMCloud's latency improvements.

Request Dispatching. The remaining cores on the CPU socket are worker cores; they run a single, pinned thread each. The dispatch core takes incoming requests from the NIC and hands each of them off to a worker core. Dispatch comes at a significant cost in RAMCloud; for example, workers complete small read operations in

less than 1 µs, but dispatch costs (handoff and extra cache misses) account for about 40% of request processing latency on the server. Furthermore, data is not partitioned within a machine, and each worker core can access any record; hence, workers need additional synchronization for safety. Dispatching is also the key throughput bottleneck for workloads consisting of small requests and responses. Section 2.1 below discusses how this seemingly costly decision improves efficiency when considering the system as a whole.

NUMA Oblivious. RAMCloud is intentionally non-uniform memory access (NUMA) oblivious. Cross-socket data access and even cross-socket access to reach the NIC is expensive. The expectation is that RAMCloud servers should always be restricted to a single CPU socket. Making RAMCloud NUMA-aware would only provide small gains. For example, with two socket machines, some RAMCloud tables and indexes could see up to a 2× speed up if their data fit within what two sockets could service rather than one. Unfortunately, nearly all tables are likely either to be most efficient on just one socket or to require the resources many sockets [2]. Tables that require the resources of exactly two sockets are likely rare. Servers already need some form of load balancing that works over the network, and intersocket interconnects do not offer significant enough bandwidth improvements (2-3× compared to network links, and they only reach one or three other sockets) to make them worth the added complexity. If machines with many sockets (8 or more, for example) become commonplace, then this decision will need to be revisited.

Remote Procedure Call (RPC). *All* interactions between RAMCloud servers and between clients and servers use RPC. Some systems use one-sided RDMA operations to allow clients to directly access server-side state without involving the server's CPU. RAMCloud explicitly avoids this model, since it tightly infuses knowledge of server structures into client logic, it complicates concurrency control and synchronization, and it prevents the server from reorganizing records and structures. One-sided operations suffer from other problems as well: they are hardware and transport specific, and they do not (yet) scale to large clusters [7].

Zero-copy Networking. Even though clients are not allowed to directly access server state via one-sided RDMA operations, servers do use scatter/gather DMA to transmit records without intervening copies. RAMCloud's RPC layer allows regions of memory to be registered with NICs that support userlevel DMA, including its in-memory log, which holds all records. This can have a significant performance impact on bandwidth-intensive operations like retrieving large values or gathering recovery data.

Transport Agnosticism. RAMCloud supports a wide variety of transport protocols and hardware platforms, and it applies the same consistent RPC interface over all of them. Initially, the lowest latency protocol was built over Infiniband with Mellanox network cards, but, today, a custom, packet-based transport protocol running atop Ethernet via DPDK is the preferred deployment. For transports and hardware that support it, RAMCloud uses zero-copy DMA to transmit records without extra configuration. Some transports supported by RAMCloud, like Infiniband's Reliable Connected mode are fully implemented in hardware on the NIC; the NIC provides fragmentation, retransmission, and more. Overall, though, they provide decreased transparency compared to software transport protocols, and they are harder to debug.

2.1 Interface and Threading Tradeoffs

RAMCloud's choice of strict RPC interfaces and its use of concurrent worker threads over a shared database may seem simplistic compared to other systems that carefully avoid server CPU use and cross-core synchronization. On an operation-by-operation basis these choices are costly, but not when the system is considered as a whole.

Using RPC allows semantically rich operations on the server and helps in synchronizing server background work with normal request processing. For example, a RAMCloud read operation fetches a value based on a string key, which involves two server-side DRAM accesses: one for a hash table bucket, and another to access the value and check its full key. One-sided operations would require two round-trips to fetch the value, destroying their latency advantage. Clever techniques like speculatively fetching extra buckets, caching chunks of remote structures, or inlining values in buckets if they are fixed size can help, but they make assumptions about locality

and index metadata size. Reads are the simplest case, and more complicated operations only make exploiting one-sided operations harder.

Initially, RAMCloud servers were single-threaded. Each server directly polled the NIC for incoming requests and executed each operation in a simple run-to-completion loop. Seemingly, this fit RAMCloud's latency goal; *median* access times for small reads were 400 ns lower than what RAMCloud's dispatch gives today. This model is similar to partitioned systems; each machine would need to run a separate server on each core. Unfortunately, the system was fragile. Fast recovery required fast failure detection, and even small hiccups in request processing latency would result in timeouts. The timeouts would result in failure detection pings, which would get stuck behind the same hiccups. The resulting false recovery would put additional load on the servers, causing cascading failures. Responding to pings is a trivial background task; since then, RAMCloud has evolved to include several heavier, but essential background operations that would only exacerbate the issue.

RAMCloud's primary value comes from its low and *consistent* access latency. Its 99.9^{th} percentile read access time is just 100 µs. Its flexible dispatch is a key reason for that. When a database is strictly partitioned across cores, requests are statistically likely to collide even if they are uniformly distributed across cores [13]. To combat this, partitioned systems must significantly underutilize hardware if access latency distribution is a concern. Partitioning also requires more careful and aggressive load redistribution to combat transient workload changes. There is also evidence that real-world large-scale in-memory stores are DRAM-capacity limited, not throughput limited [1], eliminating the main argument against RAMCloud's flexible dispatch.

RAMCloud servers aren't passive containers; they actively perform tasks in the background. For example, when servers crash in RAMCloud, every server assists in the recovery of the crashed server; similarly, every server must re-replicate some of its backup data to ensure crashes do not eventually lead to data loss. One of the most important background tasks in RAMCloud is log cleaning, a form of generational garbage collection that constantly defragments live data in memory [17]. Cleaning is key to long-term high memory utilization (and RAMCloud's cost effectiveness, since DRAM is its primary cost). Log cleaning runs with little impact on normal case performance, and it automatically scales to more cores when memory utilization is high and performance is limited by garbage collection.

Strictly partitioning the database among cores and eliminating synchronization would make log cleaning impractical. Cleaning would have to be interleaved with normal operations, which would hurt latency (head-of-line blocking) and throughput (interleaving tasks would thrash core caches). It would also make log cleaning less effective. The cleaner tries to globally separate low-churn records from high-churn records to reduce write amplification due to cleaning [1, 17]; partitioning data among cores would restrict its analysis to partition boundaries.

Aside from handoff and synchronization costs, RAMCloud's worker model leaves room for improvement. Partitioned systems effectively rely on clients on one machine to route requests to a specific core on a server to eliminate request handoff costs. Strictly mapping requests to cores can hurt tail latency, but an optimistic approach where clients issue requests directly to server cores and fallback to general dispatch could improve latency and throughput. Ideally, NIC hardware support like Intel's Flow Director and Mellanox's flow steering could help, but the latency and overhead to reconfigure the NIC-level steering tables makes them hard to use to combat transient contention.

A second serious issue is managing client threads. Synchronous requests to a remote server (both client-to-server or server-to-server) block a thread for 5 µs. This leaves too little time to efficiently (OS) context switch to another task and back. As a result, RPC callers must juggle asynchronous RPC requests for efficiency, especially on servers where remote replication delays would otherwise dominate worker CPU time. Better userlevel threading primitives could help and simplify code.

A good model for stable and low-latency request dispatching remains an open research question today.

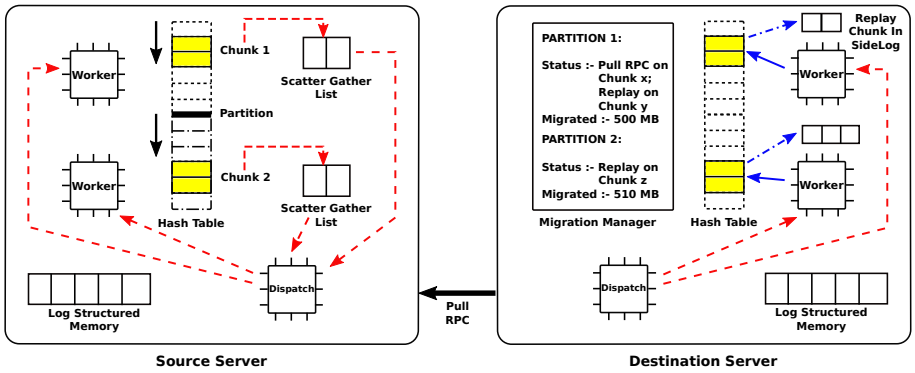

Figure 3: Low-impact Pipelined and Parallel Data Migration.

3 Rocksteady: Fast, Low-impact Data Migration

Data migration demonstrates the effectiveness of RAMCloud's dispatch model. RAMCloud must be able to quickly adapt to changing workloads. This requires scaling up, scaling down, and rebalancing data and load in response to changing workloads and server failures. Ideally, since all data lives in memory and RAMCloud employs fast networking, data migration should be fast. However, moving data rapidly without impacting access latency is challenging. RAMCloud's dispatch model helps with this.

Rocksteady is our ongoing effort to build a fast live-migration mechanism that exploits RAMCloud's unique properties while retaining 99.9th percentile access latencies of 100 µs. It is fully asynchronous at both the migration source and destination; it uses modern scatter/gather DMA for zero-copy data transfer; and it uses deep pipelining and fine-grained adaptive parallelism at both the source and destination to accelerate transfer while instantly yielding to normal-case request processing.

3.1 Rocksteady Design

Rocksteady instantly transfers ownership of all involved records to the destination at migration start, similar to some other approaches [5]. The destination subsequently handles all requests for records; writes can be serviced the moment ownership is transferred over, while reads can be serviced only after the requested records have been migrated over from the source. If demanded records are missing at the destination, they are given migration priority. This approach favors instant load reduction at the source.

All data transfer in Rocksteady is initiated by the destination server; a *pull* RPC request sent to the source returns a specified amount of data which is then replayed to populate the destination's hash table. There are two benefits to employing a destination driven protocol. First, the source server is stateless; a migration manager at the destination keeps track of all progress made by the protocol. Second, the loose coupling between source and the destination helps pipelining and parallelization; the source and the destination can proceed more independently to avoid stalls.

Rocksteady is heavily influenced by RAMCloud's core principles. Figure 3 gives an overview of how the destination pulls data and controls parallelism at both the source and destination. Its key features are explained below in more detail.

Asynchronous and Adaptive. Rocksteady is integrated into RAMCloud's dispatch mechanism. All pull RPC requests sent to the source are scheduled onto workers by the source's dispatch thread. Similarly, all data received at the destination is scheduled for replay by the destination's dispatch thread. Leveraging the dispatch gives four benefits. First, Rocksteady blends in with background system tasks like garbage collection and (re-)replication. Second, Rocksteady can adapt to system load ensuring minimal disruption to normal request processing (reads

and writes) while migrating data as fast as possible; to help with this, pull and replay tasks are fine-grained, so workers can be redirected quickly when needed. Third, since the source and destination are decoupled, workers on the source can always be busy collecting data for migration, while workers on the destination can always make progress by replaying the earlier responses. Finally, Rocksteady makes no assumptions of locality or affinity; a migration related task (pull or replay) can be dispatched to *any* worker, so any idle capacity on either end can be put to use.

Zero-copy Networking with RPCs. *All* data transfer in Rocksteady takes place through RAMCloud's RPC layer, allowing the protocol to be both transport and hardware agnostic. Destination initiated one-sided RDMA reads may seem to promise fast transfers without the source's involvement, but they breakdown because the records under migration are scattered across the source's in-memory log. RDMA reads do support scatter/gather DMA, but reads can only fetch a single contiguous chunk of memory from the remote server (that is, a single RDMA read *scatters* the fetched value locally; it cannot *gather* multiple remote locations with a single request). As a result, an RDMA read from the destination could only return a single data record per remote operation unless the source pre-aggregated all records for migration beforehand, which would undo its benefits. Additionally, one-sided RDMA would require the destination to be aware of the structure and memory addresses of the source's log. This would complicate synchronization, for example, with RAMCloud's log cleaner. Epoch-based protection can help (normal-case RPC operations like read and write synchronize with the local log cleaner this way), but extending epoch protection across machines would couple the source and destination more tightly. Even without one-sided RDMA, Rocksteady does use scatter/gather DMA [15] when supported by the transport and the NIC to transfer data from the source without *any* intervening copies.

Parallel and Pipelined. Rocksteady's *migration manager* runs at the destination's dispatch thread. At migration start, the manager logically divides the source server's key hash space into *partitions*; these partitions effectively divide up the source's hash table, which contains a pointer to each record that needs to be migrated. Partitioning is key to allowing parallel pulls on the source. A different pull request can be issued concurrently for each partition of the hash space, since they operate on entirely different portions of the source's hash table and records. When dispatched at the source, a worker linearly scans through the partition of the hash table to create a scatter/gather list of pointers to the records in memory, avoiding *any* intervening copies.

Records from completed pull requests are replayed in parallel into the destination's hash table on idle worker threads. Pull requests from distinct partitions of the hash table naturally correspond to different partitions of the destination's hash table as well. As a result, parallel replay tasks for different partitions proceed without contention. Depending on the destination server's load, the manager can adaptively scale the number of in-progress pull RPCs, as well as the number of in-progress replay tasks. The source server prioritizes normal operations over pull requests, which allows it to similarly adapt when it detects load locally.

Performing work at the granularity of distinct hash table chunks also lets Rocksteady pipeline work within a hash table partition. Whenever a pull RPC completes, the migration manager first issues a new, asynchronous pull RPC for the next chunk of records on the same partition and immediately starts replay for the returned chunk.

Eliminating Synchronous Re-replication Overhead. During normal operation, all write operations processed by a server are synchronously replicated to three other servers acting as backups. This adds about 10 µs to the operation, though concurrent updates are amortized by batching. Migration creates a challenge; if records are appended to this log and replicated, then migration will be limited by log replication speed.

Rocksteady avoids this overhead by making re-replication lazy. The records under migration are already recorded in the source's recovery log, so immediate re-replication is not strictly necessary for safety. All updates that the destination makes to records that have been migrated to it are logged in the normal way as part of its recovery log, but record data coming from the source is handled differently to eliminate replication from the migration fast path.

To do this, each server's recovery log can fork *side logs*. A side log is just like a normal log, except that it

is not part of the server's recovery log until a special log record is added to the recovery log to merge it in. This makes merging side logs into the main recovery log atomic and cheap. Rocksteady uses them in two ways. First, all records migrated from the source are appended to an in-memory side log; the data in side logs is replicated as migration proceeds, but lazily and at low priority. The side log is only merged into the server's recovery log when all of the data has been fully replicated. The second purpose of the side logs is to avoid contention between parallel replay workers; each worker uses a separate side log, and all of them are merged into the main log at the end of re-replication. During migration, the destination serves requests for records that are only in memory in a side log and have not been replicated or merged into the main log; next, we show how to make this process safe.

Lineage-based Fault Tolerance for Safe Lazy Re-replication. Avoiding synchronous re-replication of migrated data creates a serious challenge for fault tolerance. If the destination crashes in the middle of a migration, then neither the source nor the destination would have all of the records needed to recover correctly; the destination may have serviced writes for some of the records under migration, since ownership is transferred instantly at the start of migration. Rocksteady takes a unique approach to solving this problem that relies on RAMCloud's fast recovery. RAMCloud's distributed recovery works to restore a crashed server's records back into memory in 1 to 2 seconds.

To avoid synchronous re-replication of all of the records as they are transmitted from the source to the destination, the migration manager registers a dependency for the source server on the tail of the destination's recovery log at the cluster coordinator. If either the source or the destination crashes during migration, Rocksteady transfers ownership of the data back to the source. To ensure the source has all of the destination's updates, the coordinator induces a recovery of the source server which logically forces replay of the destination's recovery log tail along with the source's recovery log. This approach keeps things simple by reusing RAMCloud's recovery at the expense of extra effort in the rare case that a machine involved in migration crashes.

Rocksteady shows the benefits of RAMCloud's design. It uses simple (asynchronous) RPC, but it benefits from zero-copy scatter/gather DMA. It uses multicore parallelism, but it avoids latency inducing head of line blocking. As RAMCloud is extended with more and more functionality, its simple networking and dispatch model becomes increasingly important.

4 Conclusion

Userlevel NIC access, RDMA, and fast dispatch have been hot topics driven in large part by the elimination of I/O in favor of DRAM as a storage medium. RAMCloud was an early system in this space, but many models have emerged for exposing large clusters of DRAM-based storage over the network and for building systems on top.

Hardware capabilities significantly influence RAMCloud's networking and dispatch design choices, yet, in many ways, its choices are rather conventional. Sticking to a server-controlled dispatch model and strict RPC interfaces has helped RAMCloud remain modular by insulating servers from one another and insulating clients from server implementation details. These choices have enabled several key features that are likely to prove to be essential in a real, large-scale system while preserving low and predictable access latency.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Cidon, D. Rushton, and R. Stutsman. Memshare: A Dynamic Multi-tenant Memory Key-value Cache. Arxiv, October, 2016.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *USNIX OSDI '12*, pages 261–264.

[3] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *USENIX NSDI '14*, pages 401–414.

[4] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance . In *ACM SOSP '15*, pages 85–100.

[5] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *ACM SIGMOD '15*, pages 299–313.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC '16*, pages 437–450.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *USENIX OSDI '16*, pages 185–201.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM '14*, pages 295–306.

[9] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *USENIX ATC '16*, pages 57–70.

[10] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *ACM SOSP '15*, pages 71–86.

[11] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-value Store Server Platform. In *ACM ISCA '15*, pages 476–488.

[12] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USNIX NSDI '14*, pages 429–444.

[13] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.

[14] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *ACM SOSP '11*, pages 29–41.

[15] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[16] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX ATC '15*, pages 291–305.

[17] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *USENIX FAST '14*.