Tailwind: Fast and Atomic RDMA-based Replication

Yacine Taleb, *Univ Rennes, Inria, CNRS, IRISA;* Ryan Stutsman, *University of Utah;*
Gabriel Antoniu, *Univ Rennes, Inria, CNRS, IRISA;* Toni Cortes, *BSC, UPC*

https://www.usenix.org/conference/atc18/presentation/taleb

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Tailwind: Fast and Atomic RDMA-based Replication

Yacine Taleb*, Ryan Stutsman†, Gabriel Antoniu*, Toni Cortes‡

*Univ Rennes, Inria, CNRS, IRISA †University of Utah, ‡BSC and UPC

Abstract

Replication is essential for fault-tolerance. However, in in-memory systems, it is a source of high overhead. Remote direct memory access (RDMA) is attractive to create redundant copies of data, since it is low-latency and has no CPU overhead at the target. However, existing approaches still result in redundant data copying and active receivers. To ensure atomic data transfers, receivers check and apply only fully received messages. Tailwind is a zero-copy recovery-log replication protocol for scale-out in-memory databases. Tailwind is the first replication protocol that eliminates *all* CPU-driven data copying and fully bypasses target server CPUs, thus leaving backups idle. Tailwind ensures all writes are atomic by leveraging a protocol that detects incomplete RDMA transfers. Tailwind substantially improves replication throughput and response latency compared with conventional RPC-based replication. In symmetric systems where servers both serve requests and act as replicas, Tailwind also improves normal-case throughput by freeing server CPU resources for request processing. We implemented and evaluated Tailwind on RAMCloud, a low-latency in-memory storage system. Experiments show Tailwind improves RAMCloud's normal-case request processing throughput by 1.7×. It also cuts down writes median and 99th percentile latencies by 2x and 3x respectively.

1 Introduction

In-memory key-value stores are an essential building block for large-scale data-intensive applications [3, 19]. Recent research has led to in-memory key-value stores that can perform millions of operations per second per machine with a few microseconds remote access times. Harvesting CPU power and eliminating conventional network overheads has been key to these gains. However, like many other systems, they must replicate data in order to survive failures.

As the core frequency scaling and multi-core architecture scaling are both slowing down, it becomes critical to reduce replication overheads to keep-up with shifting application workloads in key-value stores [13]. We show that replication can consume up to 80% of the CPU cycles for write-intensive workloads (§4.4), in strongly-consistent in-memory key-value stores. Techniques like remote-direct memory access (RDMA) are promising to improve overall CPU efficiency of replication and keep predictable tail latencies.

Existing RDMA-based approaches use message-passing interfaces: a sender remotely places a message into a receiver's DRAM; a receiver must actively poll and handle new RDMA messages. This approach guarantees the atomicity of RDMA transfers, since only fully received messages are applied by the receiver [4, 10, 30]. However, this approach defeats RDMA efficiency goals since it forces receivers to use their CPU to handle incoming RDMA messages and it incurs additional memory copies.

The main challenge of efficiently using RDMA for replication is that failures could result in partially applied writes. The reason is that receivers are not aware of data being written to their DRAM. Leaving receivers idle is challenging because there is no protocol to guarantee data consistency in the event of failures.

A second key limitation with RDMA is its low scalability. This limitation comes from the connection-oriented nature of RDMA transfers. Senders and receivers have to setup queue pairs (QP) to perform RDMA. Lots of recent work has observed the high cost of NIC connection cache misses [4, 11, 32]. Scalability is limited as it typically depends on the cluster size.

To address the above challenges, we developed Tailwind, a zero-copy primary-backup log replication protocol that completely bypasses CPUs on all target backup servers. In Tailwind, log records are transferred directly from the source server's DRAM to I/O buffers at target servers via RDMA writes. Backup servers are *completely passive* during replication, saving their CPUs for other purposes; they flush these buffers to solid-state drives (SSD) periodically when the source triggers it via remote procedure call (RPC) or when power is interrupted. Even though backups are idle during replication, Tailwind is strongly consistent: it has a protocol that allows backups to detect incomplete RDMA transfers.

Tailwind uses RDMA write operations for all data movement, but all control operations such as buffer allocation and setup, server failure notifications, buffer flushing and freeing are all handled through conventional RPCs. This simplifies such complex operations without slowing down data movement. In our implementation, RPCs only account for 10^{-5} of the replication requests. This also makes Tailwind easier to use in systems that use log replication over distributed blocks even if they

were not designed to exploit RDMA.

Since Tailwind needs only to maintain connections between a primary server and its backups, the number of connections scales with the size of a replica group, not with the cluster size, making Tailwind a scalable approach.

We implemented and evaluated Tailwind on RAMCloud, a scale-out in-memory key-value store that exploits fast kernel-bypass networking. Tailwind is suited to RAMCloud's focus on strong consistency and low latency. Tailwind significantly improves RAMCloud's throughput since each PUT operation in the cluster results in three remote replication operation that would otherwise consume server CPU resources.

Tailwind improves RAMCloud's throughput by 1.7× on the YCSB benchmark, and it reduces durable PUT median latency from 32 µs to 16 µs and 99th percentile latency from 78 µs to 28 µs. Theses results stem from the fact that Tailwind significantly reduces the CPU cycles used by the replication operations: Tailwind only needs 1/3 of the cores RAMCloud uses to achieve the same throughput.

This paper makes four key contributions;

- it analyzes and quantifies CPU related limitations in modern in-memory key-value stores;
- it presents Tailwind's design, it describes its implementation in the RAMCloud distributed in-memory key-value store, and it evaluates its impact on RAMCloud's normal-case and recovery performance;
- to our knowledge, Tailwind is the first log replication protocol that eliminates all superfluous data copying between the primary replica and its backups, and it is the first log replication protocol that leaves servers CPU idle while serving as replication targets; this allows servers to focus more resources on normal-case request processing;
- Tailwind separates the replication data path and control path and optimizes them individually; it uses RDMA for heavy transfer, but it retains the simplicity of RPC for rare operations that must deal with complex semantics like failure handling and resource exhaustion.

2 Motivation and Background

Replication and redundancy are fundamental to fault tolerance, but at the same time they are costly. Primary-backup replication (PBR) is popular in fault-tolerant storage systems like file systems and key-value stores, since it tolerates f stop-failures with $f+1$ replicas. Note that, we refer to a primary replica server as *primary*, and secondary replica server as *secondary* or *backup*. In some systems, backup servers don't process user-facing requests, but in many systems each node acts as both a primary for some data items and as a backup for other data items. In some systems this is implicit: for example, a key-value store may store its state on HDFS [28],

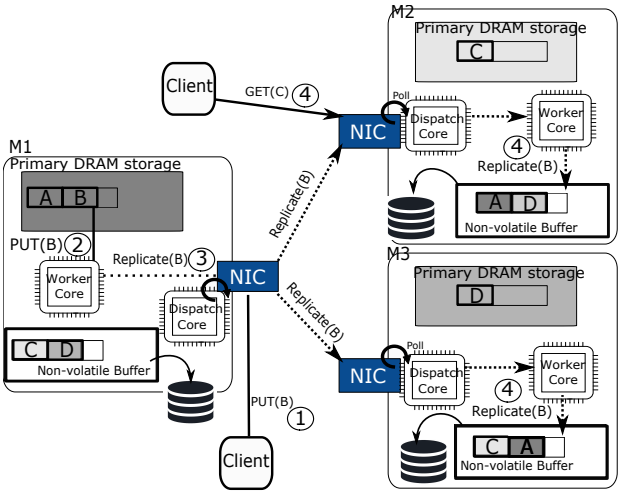


Figure 1: Flow of primary-backup replication

and a single physical machine might run both a key-value store frontend and an HDFS chunkserver.

Replication is expensive for three reasons. First, it is inherently redundant and, hence, brings overhead: the act of replication itself requires moving data over the network. Second, replication in strongly consistent systems is usually synchronous, so a primary must stall while holding resources while waiting for acknowledgements from backups (often spinning a CPU core in low-latency stores). Third, in systems, where servers (either explicitly or implicitly) serve both client-facing requests and replication operations, those operations contend.

Figure 1 shows this in more detail. Low-latency, high-throughput stores use kernel-bypass to directly poll NIC control (with a dispatch core) rings to avoid kernel code paths and interrupt latency and throughput costs. Even so, a CPU on a primary node processing an update operation must receive the request, hand the request off to a core (worker core) to be processed, send remote messages, and then wait for multiple nodes acting as backup to process these requests. Batching can improve the number of backup request messages each server must receive, but at the cost of increased latency. Inherently, though, replication can double, triple, or quadruple the number of messages and the amount of data generated by client-issued write requests. It also causes expensive stalls at the primary while it waits for responses. In these systems, responses take a few microseconds which is too short a time for the primary to context switch to another thread, yet its long enough that the worker core spends a large fraction of its time waiting.

2.1 The Promise of RDMA

Recently, remote-direct memory access (RDMA) has been used in several systems to avoid kernel overhead and to reduce CPU load. Though the above kernel-bypass-based request processing is sometimes called (*two-sided*) RDMA, it still incurs request dispatching

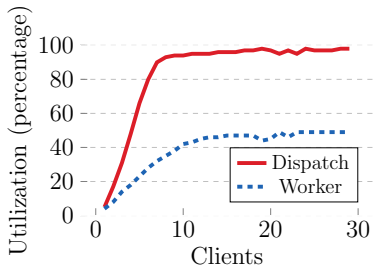


Figure 2: Dispatch and worker cores utilization percentage of a single RAMCloud server. Requests consist of 95/5 GET/PUT ratio.

and processing overhead because a CPU, on the destination node, must poll for the message and process it. RDMA-capable NICs also support so called *one-sided* RDMA operations that directly access the remote host's memory, bypassing its CPU altogether. Remote NICs directly service RDMA operations without interrupting the CPU (neither via explicit interrupt nor by enqueuing an operation that the remote CPU must process). One-sided operations are only possible through reliable-connected queue pairs (QP) that ensure in-order and reliable message delivery, similar to the guarantees TCP provides.

2.1.1 Opportunities

One-sided RDMA operations are attractive for replication; replication inherently requires expensive, redundant data movement. Backups are (mostly) passive; they often act as dumb storage, so they may not need CPU involvement. Figure 2 shows that RAMCloud, an in-memory low-latency kernel-bypass-based key-value store, is often bottlenecked on CPU (see §4 for experimental settings). For read-heavy workloads, the cost of polling network and dispatching requests to idle worker cores dominates. Only 8 clients are enough to saturate a single server dispatch core. Because of that, worker cores cannot be fully utilized. One-sided operations for replicating PUT operations would reduce the number of requests each server handles in RAMCloud, which would indirectly but significantly improve read throughput. For workloads with a significant fraction of writes or where a large amount of data is transferred, write throughput can be improved, since remote CPUs needn't copy data between NIC receive buffers and I/O or non-volatile storage buffers.

2.1.2 Challenges

The key challenge in using one-sided RDMA operations is that they have simple semantics which offer little control on the remote side. This is by design; the remote NIC executes RDMA operations directly, so they lack the generality that a conventional CPU-based RPC handlers would have. A host can issue a remote *read* of a single, sequential region of the remote processes virtual address space (the region to read must be *registered* first, but a process could register its whole virtual address space). Or, a host can issue a remote *write* of a single, sequential region of the remote processes virtual address space (again, the region must be registered with the NIC). NICs support a few more complex operations (compare-and-swap, atomic add), but these operations are currently much slower than issuing an equivalent two-sided operation that is serviced by the remote CPU [11, 30]. These simple, restricted semantics make RDMA operations efficient, but they also make them hard to use safely and correctly. Some existing systems use one-sided RDMA operations for replication (and some also even use them for normal case operations [4, 5]).

However, no existing primary-backup replication scheme reaps the full benefits of one-sided operations. In existing approaches, source nodes send replication operations using RDMA writes to push data into ring buffers. CPUs at backups poll for these operations and apply them to replicas. In practice, this is is effectively emulating two-sided operations [4]. RDMA reads don't work well for replication, because they would require backup CPUs to schedule operations and "pull" data, and primaries wouldn't immediately know when data was safely replicated.

Two key, interrelated issues make it hard to use RDMA writes for replication that fully avoids the remote CPUs at backups. First, a primary can crash when replicating data to a backup. Because RDMA writes (inherently) don't buffer all of the data to be written to remote memory, its possible that an RDMA write could be partially applied when the primary crashes. If a primary crashes while updating state on the backup, the backup's replica wouldn't be in the "before" or "after" state, which could result in a corrupted replica. Worse, since the primary was likely mutating all replicas concurrently, it is possible for all replicas to be corrupted. Interestingly, backup crashes during RDMA writes don't create new challenges for replication, since protocols must deal with that case with conventional two-sided operations too. Well-known techniques like log-structured backups [18, 23, 25] or shadow paging [35] can be used to prevent update-in-place and loss of atomicity. Traditional log implementations enforce a total ordering of log entries [9]. In database systems, for instance, the order is used to recreate a consistent state during recovery.

Unfortunately, a second key issue with RDMA operations makes this hard: each operation can only affect a single, contiguous region of remote memory. To be efficient, one-sided writes must replicate data in its final, stable form, otherwise backup CPU must be involved, which defeats the purpose. For stable storage, this generally requires some metadata. For example, when a backup uses data found in memory or storage it must know which portions of memory contain valid objects, and it must be able to verify that the objects and the markers that delineate them haven't been corrupted. As a result, backups need some metadata about the objects that they host in addition to the data items themselves. However, RDMA writes make this hard. Metadata must

inherently be intermixed with data objects, since RDMA writes are contiguous. Otherwise, multiple round trips would be needed, again defeating the efficiency gains.

Tailwind solves these challenges through a form of low-overhead redundancy in log metadata. Primaries incrementally log data items and metadata updates to remote memory on backups via RDMA writes. Backups remain unaware of the contents of the buffers and blindly flush them to storage. In the rare event when a primary fails, all backups work in parallel scanning log data to reconstruct metadata so data integrity can be checked. The next section describes its design in detail.

3 Design

Tailwind is a strongly-consistent RDMA-based replication protocol. It was designed to meet four requirements:

Zero-copy, Zero-CPU on Backups for Data Path. In order to relieve backups CPUs from processing replication requests, Tailwind relies on one-sided RDMA writes for all data movement. In addition, it is zero-copy at primary and secondary replicas; the sender uses kernel-bypass and scatter/gather DMA for data transfer; on the backup side, data is directly placed to its final storage location via DMA transfer without CPU involvement.

Strong Consistency. For every object write Tailwind synchronously waits for its replication on all backups before notifying the client. Although RDMA writes are one-sided, reliable-connected QPs generate work completion to notify the sender once a message has been correctly sent and acknowledged by the receiver NIC (i.e. written to remote memory) [8]. One-sided operation raise many issues, Tailwind is designed to cover *all* corner cases that may challenge correctness (§3.4).

Overhead-free Fault-Tolerance. Backups are unaware of replication as it happens, which can be unsafe in case of failures. To address this, Tailwind appends a piece of metadata in the log after every object update. Backups use this metadata to check integrity and locate valid objects during recovery. Although a few backups have to do little extra work during crash recovery, that work has no impact on recovery performance (§4.6).

Preserves Client-facing RPC Interface. Tailwind has no requirement on the client side; all logic is implemented between primaries and backups. Clients observe the same consistency guarantees. However, for write operations, Tailwind highly improves end-to-end latency and throughput from the client perspective (§4.2).

3.1 The Metadata Challenge

Metadata is crucial for backups to be able to use replicated data. For instance, a backup needs to know which portions of the log contain valid data. In RPC-based systems, metadata is usually piggybacked within a replication request [11, 21]. However, it is challenging to update both data and metadata with a single RDMA write

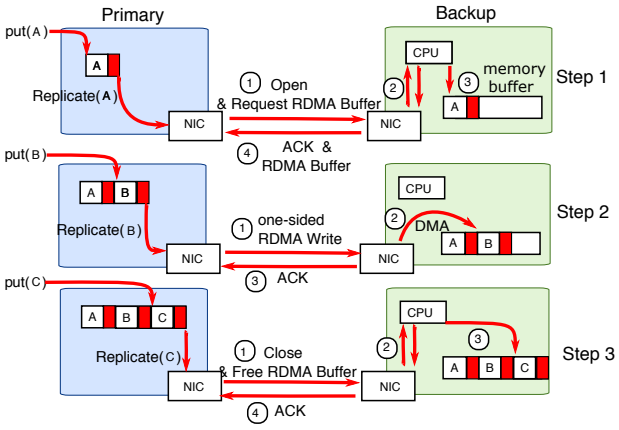


Figure 3: The three replication steps in Tailwind. During the first (open) and third (close) steps, the communication is done through RPCs. While the second step involves one-sided writes only.

since it can only affect a contiguous memory region. In this case, updating both data and metadata would require sending two messages which would nullify one-sided RDMA benefits. Moreover, this is risky: in case of failures a primary may start updating the metadata and fail before finishing, thereby invalidating all replicated objects.

For log-structured data, backups need two pieces of information: (1) the *offset* through which data in the buffer is valid. This is needed to guarantee the atomicity of each update. An outdated offset may lead the backup to use old and inconsistent data during crash recovery. (2) A *checksum* used to check the integrity of the length fields of each log record during recovery. Checksums are critical for ensuring log entry headers are not corrupted while in buffers or on storage. These checksums ensure iterating over the buffer is safe; that is, a corrupted length field does not "point" into the middle of another object, out of buffer, or indicate an early end to the buffer.

The protocol assumes that each object has a *header* next to it [4, 12, 26]. Implementation-agnostic information in headers should include: (1) the size of the object next to it to allow log traversal; (2) an integrity check that ensures the integrity of the contents of the log entry.

Tailwind checksums are 32-bit CRCs computed over log entry headers. The last checksum in the buffer covers all previous headers in the buffer. For maximum protection, checksums are end-to-end: they should cover the data while it is in transit over the network and while it occupies storage.

To be able to perform atomic updates with one-sided RDMAs in backups, the last checksum and the current offset in the buffer must be present and consistent in the backup after *every* update. A simple solution is to append the checksum and the offset before or after every object update. A single RDMA write would suffice then for both data and metadata. The checksum *must* necessarily be sent to the backup. Interestingly, this is not

the case for the offset. The nature of log-structured data and the properties of one-sided RDMA make it possible, with careful design, for the backup to compute this value at recovery time without hurting consistency. This is possible because RDMA writes are performed (at the receiver side) in an increasing address order [8]. In addition, reliable-connected QPs ensure that updates are applied in the order they were sent.

Based on these observations, Tailwind appends a checksum in the log after every object update; at any point of time a checksum guarantees, with high probability, the integrity of all previous headers preceding it in the buffer. During failure-free time, a backup is ensured to always have the latest checksum, at the end of the log. On the other hand, backups have to compute the offset themselves during crash recovery.

3.2 Non-volatile Buffers

In Tailwind, at start up, each backup machine allocates a pool of in-memory I/O buffers (8 MB each, by default) and registers them with the NIC. To guarantee safety, each backup limits the number of buffers outstanding unflushed buffers it allows. This limit is a function of its local, durable storage speed. A backup denies opening a new replication buffer for a primary if it would exceed the amount of data it could flush safely to storage on backup power. Buffers are pre-zeroed. Servers require power supplies that allow buffers to be flushed to stable storage in the case of a power failure [4, 5, 20]. This avoids the cost of a synchronous disk write on the fast path of PUT operations.

Initiatives such as the OpenCompute Project propose standards where racks are backed by batteries backup, that could provide a minimum of 45 seconds of power supply [1] at full load, including network switches. Battery-backed DIMMs could have been another option, but they require more careful attention. Because we use RDMA, batteries need to back the CPU, the PCIe controller, and the memory itself. Moreover, there exists no clear way to flush data that could still be residing in NIC cache or in PCIe controller, which would lead to firmware modifications and to a non-portable solution.

3.3 Replication Protocol

3.3.1 Write Path

To be able to perform replication through RDMA, a primary has to has to reserve an RDMA-registered memory buffer from a secondary replica. The first step in Figure 3 depicts this operation: a primary sends an `open` RPC to a backup ①. Tailwind does not enforce any replica placement policy, instead it leaves backup selection up to the storage system. Once the `open` processed ② + ③, the backup sends an acknowledgement to the primary and piggybacks necessary information to perform RDMA writes ④ (i.e. `remote_key` and `remote_address` [8]). The open call fails if there are no available buffers. The primary has then to retry.

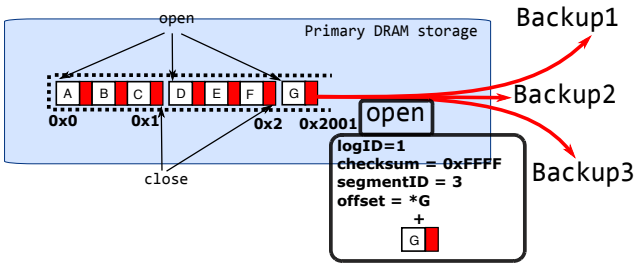


Figure 4: Primary DRAM storage consists of a monotonically growing log. It is logically split into fixed-size segments.

At the second step in Figure 3, the primary is able to perform all subsequent replication requests with RDMA writes ①. Backup NIC directly put objects to memory buffers via DMA transfer ② without involving the CPU. The primary gets work completion notification from its corresponding QP ③.

The primary keeps track of the last written offset in the backup buffer. When the next object would exceed the buffer capacity, the primary proceeds to the third step in Figure 3. The last replication request is called `close` and is performed through an RPC ①. The `close` notifies the backup ② + ③ that the buffer is full and thus can be flushed to durable storage. This eventually allows Tailwind to reclaim buffers and make them available to other primaries. Buffers are zeroed again when flushed.

We use RPCs for `open` and `close` operations because it simplifies the design of the protocol without hurting latency. As an example of complication, a primary may choose a secondary that has no buffers left. This can be challenging to handle with RDMA. Moreover, these operations are not frequent. If we consider 8 MB buffers and objects of 100 B, which corresponds to real workloads object size [19], `open` and `close` RPCs would account for 2.38×10^{-5} of the replication requests. Larger buffers imply less RPCs but longer times to flush backup data to secondary storage.

Thanks to all these steps, Tailwind can effectively replicate data using one-sided RDMA. However, without taking care of failure scenarios the protocol would not be correct. Next, we define essential notions Tailwind relies on for its correctness.

3.3.2 Primary Memory Storage

The primary DRAM log-based storage is logically sliced into equal *segments* (Figure 4). For every `open` and `close` RPC the primary sends a metadata information about current state: *logID*, latest checksum, *segmentID*, and current offset in the last segment. In case of failures, this information helps the backup in finding backup-data it stores for the crashed server.

At any given time, a primary can only replicate a single segment to its corresponding backups. This means a backup has to do very little work during recovery; if a primary replicates to r replicas then only r segments would be open, in case the primary fails.

```
   input  : Pointer to a memory buffer rdmaBuf
   output: Size of durably replicated data offset
1  currPosition = rdmaBuf ;
2  offset = currPosition ;
3  while currPosition < MAX_BUFFER_SIZE do
       /* Create a header in the current position  */
4      header = (Header)currPosition;
5      currPosition += sizeof(header);
       /* Not Corrupted or incomplete header       */
6      if header→type != INVALID then
7          if header→type == checksumType then
8              checksum = (Checksum)currPosition;
9              if checksum != currChecksum then
                   /* Primaries never append a zero
                      checksum, check if it is 1.   */
10                 if currChecksum == 0 and checksum == 1 then
11                     offset = currPosition + sizeOf(checksum);
12                 else
13                     return offset;
14             else
                   /* Move the offset at the end of
                      current checksum              */
15                 offset = currPosition + sizeOf(checksum);
16         else
17             currChecksum = crc32(currChecksum, header);
18     else
19         return offset;
       /* Move forward to next entry               */
20     currPosition += header→objectSize;
   /* We should only reach this line if a primary
      crashed before sending close                  */
21 return offset;
```

Algorithm 1: Updating RDMA buffer metadata

3.4 Failure Scenarios

When a primary or secondary replica fail the protocol must recover from the failure and rebuild lost data. Primary and secondary replicas failure scenarios require different actions to recover.

3.4.1 Primary-replica Failure

Primary replica crashes are one of the major concerns in the design. In case of such failures, Tailwind has to: (1) locate backup-data (of crashed primary) on secondary replicas; (2) rebuild up-to-date metadata information on secondary replicas; (3) ensure backup-data integrity and consistency; (4) start recovery.

Locating Secondary Replicas. After detecting a primary replica crash, Tailwind sends a query to all secondary replicas to identify the ones storing data belonging to the crashed primary. Since every primary has a unique logID it is easy for backups to identify which buffers belong to that logID.

Building Up-to-date Metadata. Backup buffers can either be in `open` or `close` states. Buffers that are `closed` do not pose any issue, they already have up-to-date metadata. If they are in disk or SSD they will be loaded to memory to get ready for recovery. However, for `open` buffers, the backup has to compute the offset and find the last checksum. Secondary replicas have to **scan** their `open` buffers to update their respective checksum and offset. To do so, they iterate over all entries as depicted in Algorithm 1.

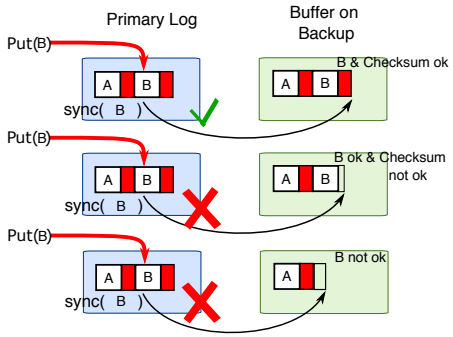


Figure 5: From top to bottom are three scenarios that can happen when a primary replica crashes while writing an object (B in this case) then synchronizing with backups. In the first scenario the primary replica fully writes the message to the backup leaving the backup in a correct state. B can be recovered in this case. In the second scenario, the object B is written but the checksum is partially written. Therefore, B is discarded. Similarly for the third scenario where B is partially written.

Basically, the algorithm takes an `open` buffer and tries to iterate over its entries. It moves forward thanks to the size of the entry which should be in the header. For every entry the backup computes a checksum over the header. When reaching a checksum in the buffer it is compared with the most recently computed checksum: the algorithm stops in case of checksum mismatch. There are three stop conditions: (1) integrity check failure; (2) invalid object found; (3) end of buffer reached.

A combination of three factors guarantee the correctness of the algorithm: (1) **the last entry is always a checksum**; Tailwind implicitly uses this condition as an end-of-transmission marker. (2) **Checksums are not allowed to be zero**; the primary replica always verifies the checksum before appending it. If it is zero it sets it to 1 and then appends it to the log. Otherwise, an incomplete object could be interpreted as valid zero checksum. (3) **Buffers are pre-zeroed**; combined with condition (2), a backup has a means to correctly detect the last valid offset in a buffer by using Algorithm 1.

3.4.2 Corrupted and Incomplete Objects

Figure 5 shows the three states of a backup RDMA buffer in case a primary replica failure. The first scenario shows a successful transmission of an object B and the checksum ahead of it. If the primary crashes, the backup is able to safely recover all data (i.e. A and B).

In the second scenario. the backup received B, but the checksum was not completely received. In this case the integrity check will fail. Object A will be recovered and B will be ignored. This is safe, since the client's PUT of B could not have been acknowledged.

The third scenario is similar: B was not completely transmitted to the backup. However, there creates two possibilities. If B's header was fully transmitted, then the algorithm will look past the entry and find a 0-byte at the end of the log entry. This way it can tell that the RDMA

operation didn't complete in full, so it will discard the entry and treat the prefix of the buffer up through A as valid. If the checksum is partially written, it will still be discarded, since it will necessarily end in a 0-byte: something that is disallowed for all valid checksums that the primary creates. If B's header was only partially written, some of the bytes of the length field may be left zero. Imagine that o is the offset of the start of B. If the primary intended B to have length l and l' is the length actually written into the backup buffer. It is necessarily the case that $l' < l$, since lengths are unsigned and l' is a subset of the bits in l. As a result, $o + l'$ falls within the range where the original object data should have been replicated in the buffer. Furthermore, the data there consists entirely of zeroes, since an unsuccessful RDMA write halts replication to the buffer, and replication already halted before $o +$ `sizeof(Header)`. As a result, this case is handled identically to the incomplete checksum case, leaving the (unacknowledged) B off of the valid prefix of the buffer.

A key property maintained by Tailwind is that torn writes never depend on checksum checks for correctness. They can also be detected by careful construction of the log entries headers and checksums and the ordering guarantees that RDMA NICs provide.

Bit-flips The checksums, both covering the log entry headers and the individual objects themselves ensure that recovery is robust against bit-flips. The checksums ensure with high probability that bit-flip anywhere in any replica will be detected. In `closed` segments, whenever data corruption is detected, Tailwind declares the replica corrupted. The higher-level system will still successfully recover a failed primary, but it must rely on replicas from other backups to do so. In `open` segments data corruption is treated as partially transmitted buffers; as soon as Tailwind immediately stops iterating over the buffer and returns the last valid offset.

3.4.3 Secondary-replica Failure

When a server crashes the replicas it contained become unavailable. Tailwind must re-create new replicas on other backups in order to keep the same level of durability. Luckily, secondary-replica crashes are dealt with naturally in storage systems and do not suffer from one-sided RDMA complications. Tailwind takes several steps to allocate a new replica: (1) It suspends all operations on the corresponding primary replica; (2) It **atomically** creates a new secondary replica; (3) It resumes normal operations on the primary replica. Step (1) ensures that data will always have the same level of durability. Step (2) is crucial to avoid inconsistencies if a primary crashes while re-creating a secondary replica. In this case the newly created secondary replica would not have all data and cannot be used.

However, it can happen that a secondary replica stops and restarts after some time, which could lead to inconsistent states between secondary replicas. To cope with this, Tailwind keeps, at any point of time, a version number for replicas. If a secondary replica crashes, Tailwind updates the version number on the primary and secondaries. Since secondaries need to be notified, Tailwind uses an RPC instead of an RDMA for this operation. Tailwind updates the version number right after the step (2) when re-creating a secondary replica. This ensures that the primary and backups are synchronized. Replication can start again from a consistent state. Note that this RPC is rare and only occurs after the crash of a backup.

3.4.4 Network Partitioning

It can happen that a primary is considered crashed by a subset of servers. Tailwind would start locating its backups, then rebuilding metadata on those backups. While metadata is rebuilt, the primary could still perform one-sided RDMA to its backups, since they are always unaware of these type of operations. To remedy this, as soon as a primary or secondary failure is detected, all machines close their respective QPs with the crashed server. This allows Tailwind to ensure that servers that are alive but considered crashed by the environment do not interfere with work done during recovery.

4 Evaluation

We implemented Tailwind on RAMCloud a low-latency in-memory key-value store. Tailwind's design perfectly suits RAMCloud in many aspects:

Low latency. RAMCloud's main objective is to provide low-latency access to data. It relies on fast networking and kernel-bypass to provide a fast RPC layer. Tailwind can further improve RAMCloud (PUT) latency (§4.2) by employing one-sided RDMA without any additional complexity or resource usage.

Replication and Threading in RAMCloud. To achieve low latency, RAMCloud dedicates one core solely to poll network requests and dispatch them to worker cores (Figure 1). Worker cores execute all client and system tasks. They are never preempted to avoid context switches that may hurt latency. To provide strong consistency, RAMCloud always requests acknowledgements from all backups for every update. With the above threading-model, replication considerably slows down the overall performance of RAMCloud [31]. Hence Tailwind can greatly improve RAMCloud's CPU-efficiency and remove replication overheads.

Log-structured Memory. RAMCloud organizes its memory as an append-only log. Memory is sliced into smaller chunks called *segments* that also act as the unit of replication, i.e., for every segment a primary has to choose a backup. Such an abstraction makes it easy to replace RAMCloud's replication system with Tailwind. Tailwind checksums can be appended in the log-storage, with data, and replicated with minimal changes to the code. In addition, RAMCloud provides a log-cleaning mechanism which can efficiently clean old checksums and reclaim their storage space.

CPU	Xeon E5-2450 2.1 GHz 8 cores, 16 hw threads
RAM	16 GB 1600 MHz DDR3
NIC	Mellanox MX354A CX3 @ 56 Gbps
Switch	36 port Mellanox SX6036G
OS	Ubuntu 15.04, Linux 3.19.0-16, MLX4 3.4.0, libibverbs 1.2.1

Table 1: Experimental cluster configuration.

We compared Tailwind with RAMCloud replication protocol, focusing our analysis on three key questions:

Does Tailwind improve performance? Measurements show Tailwind reduces RAMCloud's median write latency by 2× and 99th percentile latency by 3× (Figure 7). Tailwind improves throughput by 70% for write-heavy workloads and by 27% for workloads that include just a small fraction of writes.

Why does Tailwind improve performance? Tailwind improves per-server throughput by eliminating backup request processing (Figure 9), which allows servers to focus effort on processing user-facing requests.

What is the Overhead of Tailwind? We show that Tailwind's performance improvement comes at no cost. Specifically, we measure and find no overhead during crash recovery compared to RAMCloud.

4.1 Experimental Setup

Experiments were done on a 35 server Dell r320 cluster (Table 1) on the CloudLab [24] testbed.

We used three YCSB [2] workloads to evaluate Tailwind: update-heavy (50% PUTs, 50% GETs), read-heavy (5% PUTs, 95% GETs), and update-only (100% PUTs). We intitially inserted 20 million objects of 100 B plus 30 B for the key. Afterwards, we ran up to 30 client machines. Clients generated requests according to a Zipfian distribution ($\theta = 0.99$). Objects were uniformly inserted in active servers. The replication factor was set to 3 and RDMA buffers size was set to 8 MB. Every data point in the experiments is averaged over 3 runs.

RAMCloud's RPC-based replication protocol served as a baseline for comparison. Note that, in the comparison with Tailwind, we refer to RAMCloud's replication protocol as RAMCloud for simplicity.

4.2 Performance Improvement

The primary goal of Tailwind is to accelerate basic operations throughput and latency. To demonstrate how Tailwind improves performance we show Figure 6, i.e. throughput per server as we increase the number of clients. When client operations consist of 5% PUTs and 95% GETs, RAMCloud achieves 500 KOp/s per server while Tailwind reaches up to 635 KOp/s. Increasing the update load enables Tailwind to further improve the throughput. For instance with 50% PUTs Tailwind sustains 340 KOp/s against 200 KOp/s for RAMCloud, which is a 70% improvement. With update-only workload, improvement is not further increased: In this case Tailwind improves the throughput by 65%.

Tailwind can improve the number of read operations serviced by accelerating updates. CPU cycles saved allow servers (that are backups as well) to service more requests in general.

Figure 7 shows that update latency is also considerably improved by Tailwind. Under light load Tailwind reduces median and 99th percentile latency of an update from 16 µs to 11.8 µs and from 27 µs to 14 µs respectively. Under heavy load, i.e. 500 KOp/s Tailwind reduces median latency from 32 µs to 16 µs compared to RAMCloud. Under the same load tail latency is even further reduced from 78 µs to 28 µs.

Tailwind can effectively reduce end-to-end client latency. With reduced acknowledgements waiting time, and more CPU power to process requests faster, servers can sustain a very low latency even under heavy concurrent accesses.

4.3 Gains as Backup Load Varies

Since all servers in RAMCloud act as both backups and primaries, Tailwind accelerates normal-case request processing indirectly by eliminating the need for servers to actively process replication operations. Figure 8 shows the impact of this effect. In each trial load is directed at a subset of four RAMCloud storage nodes; "Active Primary Servers" indicates the number of storage nodes that process client requests. Nodes do not replicate data to themselves, so when only one primary is active it is receiving no backup operations. All of the active primary's backup operations are directed to the other three otherwise idle nodes. Note that, in this figure, throughput is per-active-primaries. So, as more primaries are added, the aggregate cluster throughput increases.

As client GET/PUT operations are directed to more nodes (more active primaries), each node slows down because it must serve a mix of client operations intermixed with an increasing number of incoming backup operations. Enough client load is offered (30 clients) so that storage nodes are the bottleneck at all points in the graphs. With four active primaries, every server node is saturated processing client requests and backup operations for all client-issued writes.

Even when only 5% of client-issued operations are writes (Figure 8a), Tailwind increasingly improves performance as backup load on nodes increases. When a primary doesn't perform backup operations Tailwind improves throughput 3%, but that increases to 27% when the primary services its fair share of backup operations. The situation is similar when client operations are a 50/50 mix of reads and writes (Figure 8b) and when clients only issue writes (Figure 8c).

As expected, Tailwind enables increasing gains over RAMCloud with increasing load, since RDMA elimi-

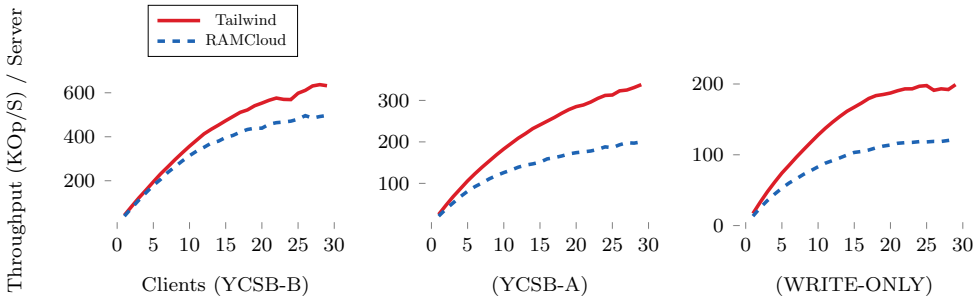
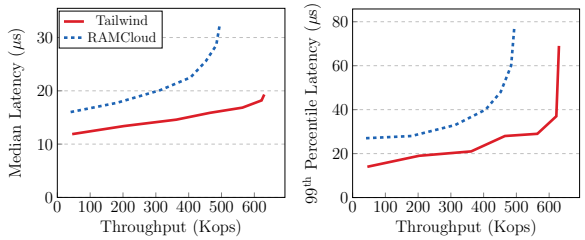


Figure 6: Throughput per server in a 4 server cluster.



(a) (b)

Figure 7: (a) Median latency and (b) 99th percentile latency of PUT operations when varying the load.

nates three RPCs that each server must handle for each client-issued write, which, in turn, eliminates worker core stalls on the node handling the write.

In short, the ability of Tailwind to eliminate replication work on backups translates into more availability for normal request processing, and, hence, better GET/PUT performance.

4.4 Resource Utilization

The improvements above have shown how Tailwind can improve RAMCloud's baseline replication normal-case. The main reason is that operations contend with backup operations for worker cores to process them. Figure 9a illustrates this: we vary the offered load (updates-only) to a 4-server cluster and report aggregated active worker cores. For example, to service 450 KOp/s, Tailwind uses 5.7 worker cores while RAMCloud requires 17.6 active cores, that is 3× more resources. For the same scenario, we also show Figure 9b that shows the aggregate active dispatch cores. Interestingly, gains are higher for dispatch, e.g., to achieve 450 KOp/s, Tailwind needs only 1/4 of dispatch cores used by RAMCloud.

Both observations confirm that, for updates, most of the resources are spent in processing replication requests. To get a better view on the impact when GET/PUT operations are mixed, we show Figure 10. It represents active worker and dispatch cores, respectively, when varying clients. When requests consist of updates only, Tailwind reduces worker cores utilization by 15% and dispatch core utilization by 50%. This is stems from the fact that a large fraction of dispatch load is due to replication requests in this case. With 50/50 reads and writes, worker utilization is slightly improved to 20% while it reaches 50% when the workload consists of 5% writes only.

Interestingly, dispatch utilization is not reduced when reducing the proportion of writes. With 5% writes Tailwind utilizes even more dispatch than RAMCloud. This is actually a good sign, since read workloads are dispatch-bound. Therefore, Tailwind allows RAMCloud to process even more reads by accelerating write operations. This is implicitly shown in Figure 10 with "Replication" graphs that represent worker utilization due to waiting for replication requests. For update-only workloads, RAMCloud spends 80% of the worker cycles in replication. With 5% writes RAMCloud spends 62% of worker cycles waiting for replication requests to complete against 49% with Tailwind. The worker load difference is spent on servicing read requests.

4.5 Scaling with Available Resources

We also investigated how Tailwind improves internal server parallelism (i.e. more cores). Figure 11 shows throughput and worker utilization with respect to available worker cores. Clients (fixed to 30) issue 50/50 reads and writes to 4 servers. Note that we do not count the dispatch core with available cores, as it is always available. With only a single worker core per machine, RAMCloud can serve 430 KOp/s compared to 660 KOp/s for Tailwind with respectively 4.5 and 3.5 worker cores utilization. RAMCloud can over-allocate resources to avoid deadlocks, which explains why it can go above the limit of available cores. Interestingly, when increasing the available worker cores, Tailwind enables better scaling. RAMCloud does not achieve more throughput with more than 5 available cores. Tailwind continues to improve throughput up to all 7 available cores per machine.

Even though both RAMCloud and Tailwind exhibit a plateau, this is actually due to the dispatch thread limit that cannot take more requests in. This suggests that Tailwind allows RAMCloud to better take advantage of per-machine parallelism. In fact, by eliminating the replication requests from dispatch, Tailwind allows more client-

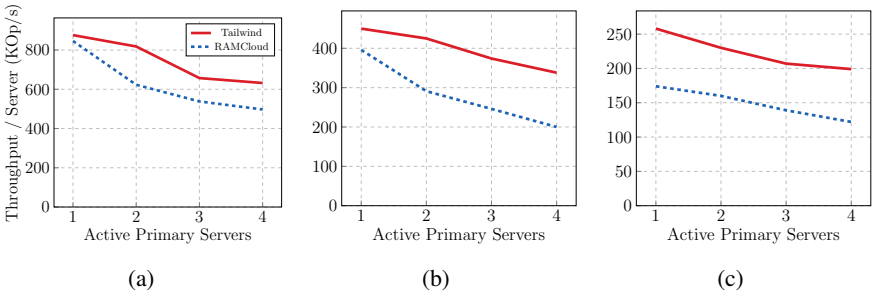


Figure 8: Throughput per active primary servers when running (a) YCSB-B (b) YCSB-A (c) WRITE-ONLY with 30 clients.

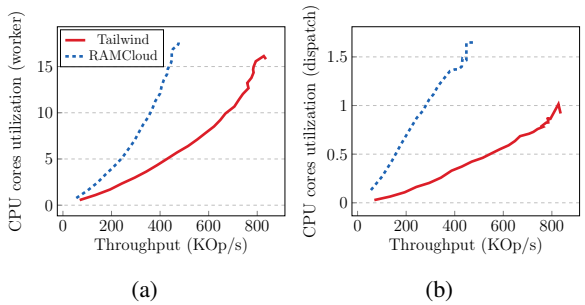


Figure 9: Total (a) worker and (b) dispatch CPU core utilization on a 4-server cluster. Clients use the WRITE-ONLY workload.

issued requests in the system.

4.6 Impact on Crash Recovery

Tailwind aims to accelerate replication while keeping strong consistency guarantees and without impacting recovery performance. Figure 12 shows Tailwind's recovery performance against RAMCloud. In this setup data is inserted into a primary replica with possibility to replicate to 10 other backups. RAMCloud's random backup selection makes it so that all backups will end up with approximately equal share of backup data. After inserting all data, the primary kills itself, triggering crash recovery.

As expected, Tailwind almost introduces no overhead. For instance, to recover 1 million 100 B objects, it takes half a second for Tailwind and 0.48 s for RAMCloud. To recover 10 million 100 B objects, Both Tailwind and RAMCloud take roughly 2.5 s.

Tailwind must reconstruct metadata during recovery (§3.4.1), but this only accounts for a small fraction of the total work of recovery. Moreover, reconstructing metadata is only necessary for `open` buffers, i.e. still in memory. This can be orders of magnitude faster than loading a buffer previously flushed on SSD, for example.

5 Discussion

5.1 Metadata Space Overhead

In its current implementation, Tailwind appends metadata after every write to guarantee RDMA writes atomicity (§3.1). Although this approach appears to introduce space overhead, RAMCloud's log-cleaning mechanism efficiently removes old checksums without performance impact [26]. In general, Tailwind adds only 4 bytes per object which is much smaller than, for example, RAMCloud headers (30 bytes).

5.2 Applicability

Tailwind can be used in many systems that leverage distributed logging [4, 12, 18, 20, 22, 33] provided they have access to RDMA-based networks. Recently, RDMA is supported in Ethernet in the form of RoCE or iWARP [8] and is becoming prevalent in datacenters [17, 39]. To be properly integrated in any system, Tailwind needs: (1) appending a checksum after each write; (2) implementing algorithm 1 during recovery. Aspects such as memory/buffer management do not impact Tailwind's core design nor performance gains because Tailwind reclaims replication-processing CPU cycles at backups.

6 Related Work

One-sided RDMA-based Systems. There is a wide range of systems recently introduced that leverage RDMA [4, 5, 10, 15, 16, 27, 30, 33, 34, 36, 38]. For instance, many of them use RDMA for normal-case operations. Pilaf [15] implements client-lookup operations with one-sided RDMA reads. In contrast, with HERD [10] clients use one-sided RDMA writes to send GET and PUT requests to servers, that poll their receive RDMA buffers to process requests. In RFP [10] clients use RDMA writes to send requests, and RDMA reads to poll (remotely) replies. Crail [29] uses one-sided RDMA to transfer I/O blocks, but it is not designed for availability or fault-tolerance. LITE [32] is a kernel module providing efficient one-sided operations and could be used to implement Tailwind.

Many systems also use one-sided RDMA for replication. For instance, FaRM [4, 5], HydraDB [33], and DARE [22] use one-sided RDMA writes to build a message-passing interface. Replication uses this interface to place messages in remote ring buffers. Servers have to poll these ring buffers in order to fetch messages, process them, and apply changes. In [6] authors use one-sided RDMA for VM migration. The sender asynchronously replicate data and notifies the receiver at

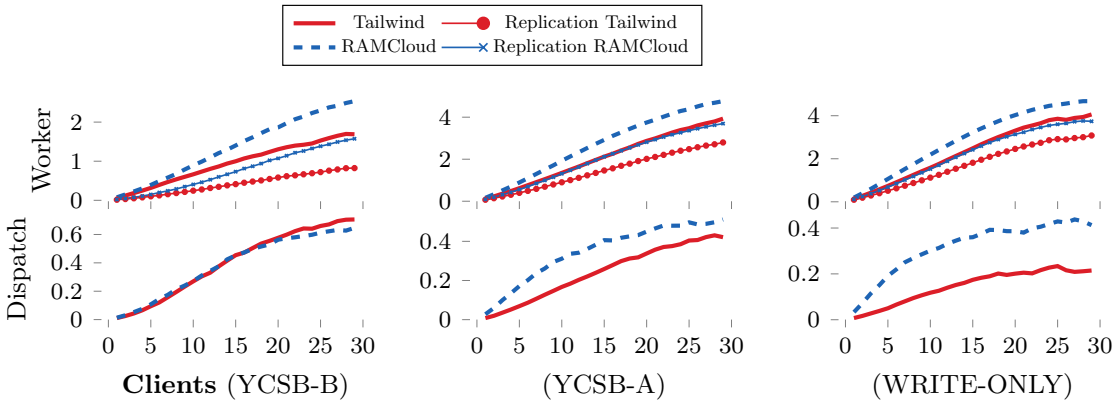


Figure 10: Total dispatch and worker cores utilization per server in a 4-server cluster. "Replication" in worker graphs represent the fraction of worker load spent on processing replication requests on primary servers.

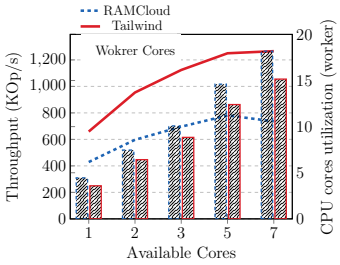


Figure 11: Throughput (lines) and total worker cores (bars) as a function of available cores per machine. Values are aggregated over 4 servers.

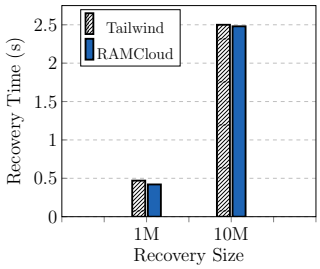


Figure 12: Time to recover 1 and 10 million objects with 10 backups.

the end of transfer then the backup applies changes in a transactional way.

No system that uses RDMA writes for replication leaves the receiver CPU completely idle. Instead, the receiver must poll receive buffers and process requests, which defeats one-sided RDMA efficiency purposes. Tailwind frees the receiver from processing requests by directly placing data to its final storage location.

Reducing Replication Overheads. Many systems try to reduce replication overheads either by relaxing/tuning consistency guarantees [3, 7, 14] or using different approaches for fault-tolerance [37]. Mojim [38] is a replication framework intended for NVMM systems. For each server it considers a mirror (backup) machine to which it will replicate all data through (two-sided) RDMA. It supports multiple levels of consistency and durability. RedBlue [14] and Correctables [7] provide different consistency levels to the applications and allows them to trade consistency for performance. Tailwind does not sacrifice consistency to improve normal-case system performance.

7 Conclusion

Tailwind is the first replication protocol that fully exploits one-sided RDMA; it improves performance without sacrificing durability, availability, or consistency. Tailwind leaves backups unaware of RDMA writes as they happen, but it provides them with a protocol to rebuild metadata in case of failures. When implemented in RAMCloud, Tailwind substantially improves throughput and latency with only a small fraction of resources originally needed by RAMCloud.

Acknowledgments

This work has been supported by the BigStorage project, funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963), by the Spanish Ministry of Science and Innovation (contract TIN2015- 65316), by Generalitat de Catalunya (contract 2014-SGR-1051). This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1566175 and CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by Facebook and VMware.

References

[1] The OpenCompute Project. http://www.opencompute.org/.

[2] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with

ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154.

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[4] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[5] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 54–70.

[6] Gerofi, B., and Ishikawa, Y. Rdma based replication of multiprocessor virtual machines over high-performance interconnects. In *2011 IEEE International Conference on Cluster Computing* (Sept 2011), pp. 35–44.

[7] Guerraoui, R., Pavlovic, M., and Seredinschi, D.-A. Incremental consistency guarantees for replicated objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 169–184.

[8] InfiniBand Trade Association. *IB Specification Vol 1*, 03 2015. Release-1.3.

[9] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal 21*, 2 (Apr 2012), 239–263.

[10] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[11] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 185–201.

[12] Kreps, J., Narkhede, N., and Rao, J. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece* (2011).

[13] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[14] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 265–278.

[15] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[16] Mitchell, C., Montgomery, K., Nelson, L., Sen, S., and Li, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 451–464.

[17] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 537–550.

[18] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[19] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[20] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[21] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The ramcloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[22] Poke, M., and Hoefler, T. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 107–118.

[23] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 8–8.

[24] Robert, R., and Eric, E. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *;login: 39*, 6 (2014), 36–38.

[25] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[26] Rumble, S. M., Kejriwal, A., and Ousterhout, J. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2014), FAST'14, USENIX Association, pp. 1–16.

[27] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 317–332.

[28] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[29] STUEDI, P., TRIVEDI, A., PFEFFERLE, J., STOICA, R., METZLER, B., IOANNOU, N., AND KOLTSIDAS, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull. 40* (2017), 38–49.

[30] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 1–15.

[31] TALEB, Y., IBRAHIM, S., ANTONIU, G., AND CORTES, T. Characterizing performance and energy-efficiency of the ramcloud storage system. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 1488–1498.

[32] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.

[33] WANG, Y., ZHANG, L., TAN, J., LI, M., GAO, Y., GUERIN, X., MENG, X., AND MENG, S. 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* (New York, NY, USA, 2015), SC '15, ACM, pp. 22:1–22:11.

[34] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[35] YLÖNEN, T. Concurrent shadow paging: A new direction for database research.

[36] YU, C., XU, C.-Z., LIAO, X., JIN, H., AND LIU, H. Live virtual machine migration via asynchronous replication and state synchronization. *IEEE Transactions on Parallel Distributed Systems 22* (2011), 1986–1999.

[37] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 15–28.

[38] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 3–18.

[39] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536.