Fast, General State Machine Replication via RDMA-accelerated PAXOS

Paper #83

Abstract

State machine replication (SMR) runs replicas of the same program and uses a distributed consensus protocol (e.g., PAXOS) to enforce same program inputs among replicas, tolerating various faults. Recent SMR systems have shown to greatly improved the availablity of server programs. Unfortunately, consensus latency is often too high to make SMR widely adopted. This paper presents FALCON, a fast SMR system for general server programs by leveraging Remote Direct Memory Access (RDMA). FALCON intercepts a server program's socket calls and runs a new RDMA-accelerated PAXOS protocol that needs only two fastest RDMA write operations per input. This protocol addresses a scalability challenge by tightly integrating RDMA features within the fault-tolerant nature of PAXOS, making replicas reach consensus rapidly in parallel. On top of this protocol, FALCON presents a fast output checking protocol to improve assurance on whether replicas run in sync.

Evaluation on nine widely used, diverse server programs (e.g., `Memcached` and `MySQL`) shows that FALCON is: (1) general, it ran these servers without modifications except one program; (2) fast, it incurred merely a 4.28% mean overhead in response time and 4.16% in throughput, and its consensus latency was 40.1X faster than a prior SMR system built on `ZooKeeper`; (3) scalable, it achieved similar consensus latency from three to seven replicas; and (4) robust, its output protocol efficiently detected and recovered replicas from divergence.

1 Introduction

State machine replicaion (SMR) runs the same program on a number of replicas and uses a distributed consensus protocol (e.g., PAXOS [16]) to enforce the same inputs among replicas. Typically, PAXOS assigns a replica as the leader to propose consensus requests, while the other replicas agree or reject requests. Consensus on a new input can be achieved as long as a majority of replicas agree, thus SMR can tolerate various faults such as minor replica failures. Attracted by this strong fault-tolerance, recently, several SMR systems [12, 3, 16, 34, 27] have been built to greatly improve the availability of various server programs, because these programs tend to serve client requests at all time.

Unfortunately, despite these recent advances, SMR remains difficult to be widely adopted due to the high consensus latency in PAXOS. To agree on an input, traditional consensus protocols invoke at least one message round-trip between two replicas. Given that a `ping` in Ethernet takes hundreds of μs, a server program running in an SMR system with only three replicas must wait at least this time before processing an input. This latency may be acceptable for maintaining global configuration [12, 3] or processing SQL transactions [16, 34], but prohibitive for key-value stores. To mitigate this challenge, some recent SMR systems [60, 27] batch requests into one consensus round. However, batching can only mitigate a server's throught lost; it may aggravate response time.

As the replica group size grows, the consensus latency of PAXOS is even more difficult to scale, because a majority now involves more replicas. One scaling implementation approach may be using asynchrous IO (e.g., Libevent [41]) to improve the parallilism of replicas'consensus round-trips. However, the high latency of a each round-trip still exists, and the event loop in this approach may involve frequent context switches (each may take more time than a `ping` in Ethernet). To make SMR scales better, novel replication models are proposed, including multi-leader models [49, 46], clustering or nested consensus models [25, 33]. However, the core of these models, PAXOS itself, still faces a scalability challenge.

Remote Direct Access Memory (RDMA) is a promising technique to mitigate consensus latency because recently it becomes cheaper and increasingly pervasive in datacenters. RDMA allows a process to directly write to the user space memory of a remote process, completely bypassing the remote OS kernel or CPU (the so called "one-sided" operations). As a common RDMA practice, to ensure that such a write successfully resides in the memory of a remote process, the local process should wait until the remote NIC to send an ACK to the local host's NIC. An evaluation [48] shows that such a write round-trip takes only ~3 μs in the Infiniband architecture [1].

In addition to mitigating the one-hop consensus latency, RDMA also opens new opportunity to address the PAXOS scalability problem, because we can integrate RDMA features *tightly* within the fault-tolerant nature of PAXOS. In essense, PAXOS has already tolerated various faults, including machine failures and process crashes.

Therefore, we can safely ignore the ACKs in RDMA writes and rely on PAXOS to enforce the reliability of these writes.

This integration of PAXOS and RDMA features looks simple, but it leads to a fast, scalable PAXOS consensus algorithm with three steps. First, the leader stores a consensus request in local stable storage. Second, it does parallel RDMA writes to write this request to the remote memory of the other replicas without waiting any RDMA ACKs. Remote replicas also work in parallel: they poll from their local memory, store this request in local storage, and send consensus replies to the leader rapidly with RDMA writes, without waiting any RDMA ACKs either. Third, once the leader sees a majority of replies in its local memory, a consensus is reached.

In the second step of this algorithm, both the leader and remote replicas work in parallel, so a whole consensus latency aproximately consists of three operations: a leader's write to stable storage, a remote replica's write to local storage, and a single RDMA write round-trip. This consensus latency is no longer firmly correlated with replica group size (confirmed in our evaluation); its scalability is now mainly bounded by the capacity of outbound RDMA writes in the NIC hardware. By making the core of PAXOS scalable, other novel replication models [49, 46, 25, 33] can scale even better.

This paper present FALCON, an SMR system that replicates general server programs efficiently. With FALCON, a server program just runs as is, and FALCON automatically deploys this program on replicas of machines. FALCON intercepts inputs from the inbound socket calls (e.g., `recv()`) of a server program and invokes our PAXOS consensus algorithm to efficiently enforce same inputs across replicas. In a consensus protocol level, FALCON carefully tackles several technical challenges, including handling an atomicy challenge (§4.1) concurrent connections (§9).

However, to practically replicate general server programs, only enforcing same inputs is often not enough. An automated, efficient output checking mechanism that can improve the assurance on "replicas run in sync" is still missing in existing SMR systems [60, 27, 16, 56]. Server programs now already use multithreading to harness the power of multi-core hardware. Nondeterminism [36, 21, 5, 20, 51, 42, 19, 18, 4] caused by contentions in inter-thread resources (e.g., global memory and locks) and systems resources (e.g., network ports) can easily cause program execution states to diverge across replicas and compute wrong outputs to clients.

To handle nondeterminism, SMR systems either use deterministic multithreading and replay approaches [27, 16, 29] or rely on manually annotating share states in program code to detect execution divergence [34]. These approaches fall short in either performance or automation.

On top of the input consensus protocol, FALCON builds an efficient network output checking protocol that occasionally checks output divergence across replicas. This checking protocol works by first computing a accumulated hash by intercepting outbound socke calls (e.g., `send()`) in a server program, it then occasionally invokes an output consensus by collecting the hashes in replicas' consensus replies for the leader. FALCON then provides an optional rollback/restore mechanism to make an effort to restore the diverged replicas. Because hash computation is efficient and output consensus is invoked rarely, this output checking protocol introduced neglible performance impact in our evaluation.

We implemented FALCON in Linux. FALCON intercepts POSIX inbound socket calls (e.g., `accept()` and `recv()`) to coordinate inputs using the Infiniband RDMA architecture. FALCON intercepts POSIX outbound socket operations (e.g., `send()`) to invoke the output checking protocol. This simple, deployable interface design makes FALCON support general server programs without modifying them. To recover or add new replicas, FALCON leverages CRIU [15] to perform checkpoint and restore on general server programs.

We evaluated FALCON on nine widely used or studied server programs, including four key value stores (Redis, `Memcached`, `SSDB`, and `MongoDB`), one SQL server `MySQL`, one anti-virus server `ClamAV`, one multimedia storage server `MediaTomb`, one LDAP server OpenLDAP, and one advanced transactional database `Calvin` (with `ZooKeeper` as its SMR protocol). Our evaluation shows that

1. FALCON is general. For all evaluated programs, FALCON ran them without any modification except `Calvin` (we added a 23 patch to make its client and server programs communicate with sockets).
2. FALCON is fast. Compared to the nine servers' unreplicated executions, FALCON incurred merely 4.16% overhead on throughtput and 4.28% on response time in average. FALCON's consensus latency is 40.1X faster than `Calvin`'s `ZooKeeper`-based SMR service running on IPoIB.
3. FALCON is scalable to replica group size. FALCON achieved a scalable consensus latency of 9.9∼11.0 μs from 3 to 7 replicas.
4. FALCON is robust. Among XXX repeated executions, FALCON detected and recovered execution divergence caused by a software bug in Redis, while Redis's own replication service missed the bug.

Our major conceptual contribution is a fast, scalable PAXOS consensus algorithm by tightly integrating RDMA write operations with the fault-tolerant nature of PAXOS. FALCON and its implementation have the potential to largely increase the adoption of SMR. FALCON can also be applied to broad areas, including other replication protocols (e.g., byzantine fault-tolerance [35, 13]), distributed program analyses, and future datacenter operating systems (§6.2).

The remaining of this paper is organized as follows. §2 introduces background on PAXOS and RDMA features. §3 gives an overview of our FALCON system. §4 presents our input consensus protocol. §5 describes the output checking protocol. §6 FALCON's discusses current limitations and its broad applications in other areas. §7 presents evaluation results, §8 discusses related work, and §9 concludes.

2 Background

This section introduces the background of two key techniques in FALCON, the PAXOS consensus protocol (§2.1) and RDMA features (§2.2).

2.1 PAXOS

An SMR system runs the same program and its data on a set of machines (replicas), and it uses a distributed consensus protocol (typically, PAXOS [61, 39, 37, 14, 40, 47]) to coordinates inputs across replicas. For efficiency, in normal case, PAXOS often lets one replica work as the leader which invoke consensus requests, and the other replicas work as backups to agree on or reject these requests. If the leader fails, PAXOS elects a new leader from the backups.

When a new input comes, PAXOS starts a new consensus round, which invokes a consensus request on this input to backups. As long as a majority of replicas agree on this input (i.e., this input is *committed*), PAXOS guarantees that all replicas consistently agree to process this input This quorum based consensus makes PAXOS tolerate various faults such as machine failures and network crashes. Before a replica agrees on an input, PAXOS logs this input in the replica's stable storage for durability. As consensus rounds move on, PAXOS consistently enforce the same sequence of inputs and execution states across replicas without divergence, if a program behaves as a deterministic state machine (i.e., always produces the same output on the same input).

Netork latency of consensus messages is one key problem to make general server programs adopt SMR. For instance, in an efficient, practical PAXOS implementation [47], each input in normal case takes one consensus round-trip between every two replicas (one request from the leader and one reply from a backup).

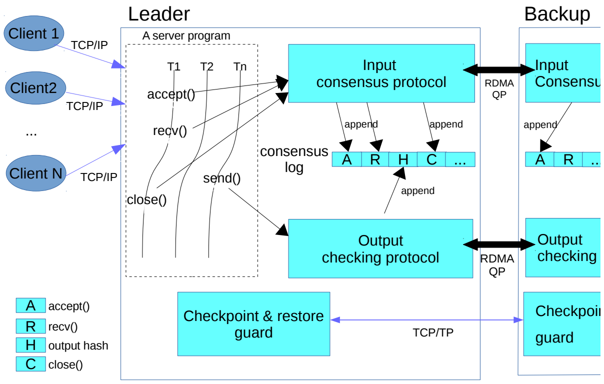


Figure 1: ***The*** **FALCON** ***Architecture.*** **FALCON components are shaded (and in blue).**

2.2 RDMA

RDMA hardware recently becomes commonplace in datacenters due to its extreme low latency, high throughput, and its decreasing prices. RDMA communications between a local network inteface card (NIC) and remote NIC requires setting up a Queue Pair (QP), including a send queue and a receive queue. Each QP associates with a Completion Queue (CQ) to store ACKs. A QP belongs to a type of XY: X can be R (reliable) or U (unreliable), and Y can be C (connected) or U (unconnected). FALCON and DARE's implementations mainly use RC QPs, because such a reliable, connected QP guarantees in-order, non-corrupted delivery in normal case. However, RC QPs may still lose packets in typical PAXOS exceptional cases (e.g., hardware failures, OS crashes, or server program restarts).

RDMA provides three types of communication primitives, from slowest to fastest: IPoIB, message verbs, and one-sided read/write operations. A one-sided RDMA read/write operation can directly write from one replica's memory to a remote replica's memory, completely bypassing OS kernel and CPU of the remote replica. For brevity, the rest of this paper denotes a one-sided RDMA write operation as a "WRITE".

To ensure a remote replica is alive and a WRITE succeeds, a common RDMA practice is that after a WRITE is pushed to a QP, the local replica polls for an ACK from the associated CQ before it continues (the so called *signaling*). However, depending on local algorithm logic, the local replica sometimes can do *selected signaling* [32]: it only checks for an ACK after pushing a number of WRITEs (previous WRITEs may already succeed before this ACK-checking starts).

3 Falcon Overview

Falcon runs replicas of a server program in a datacenter. Replicas connect with each other using RDMA QPs. Client programs located in LAN or WAN networks. The leader handles client requests and runs our RDMA-accelerated Paxos protocol to coordinate inputs among replicas.

Figure 1 shows Falcon's architecture. Falcon intercepts a server program's socket calls (e.g., `recv()`) using a Linux technique called LD_PRELOAD. Falcon involves four key components: a Paxos consensus protocol for input coordination (in short, the *coordinator*), an output checking protocol (the *checker*), a circular, reusable in-memory consensus log (the *log*), and a guard process that handles checkpointing and recovering a server's process state and file system state (the *guard*).

The coordinator is invoked when a server program thread calls an inbound socket call to manage a client socket connection (e.g., `accept()` and `close()`) or to receive inputs from the connection (e.g., `recv()`). On the leader side, Falcon executes the actual Libc socket call, extracts the returned value or inputs of this call, stores it in local SSD, appends a log entry to its local consensus log, and then invokes the coordinator for a new consensus request on "executing this socket call".

The coordinator runs a consensus algorithm (§4), which WRITEs the local entry to backups' remote logs in parallel and polls the local log entry to wait quorum. When a quorum is reached, the leader thread simply finishes intercepting this call and continues with its server execution. As the leader's server threads execute more calls, Falcon enforces the same consensus log and thus the same socket call sequence across replicas.

On each backup side, the coordinator uses a Falcon internal thread called *follower* to poll its consensus log for new consensus requests. If the coordinator agrees the request, the follower stores the log entry in local SSD and then WRITEs a consensus reply to the remote leader's corresponding log entry. A backup does not need to intercept a server's socket calls because the follower will just follow the leader's consensus requests on executing what socket calls and then forward these calls to its local server program.

The output checker is occasionally invoked as the leader's server program executes outbound socket calls (e.g., `send()`). For every 1.5K bytes (MTU size) of accumulated outputs per connection, the checker unions the previous hash with current outputs and computes a new CRC64 hash. After a fixed number of hashes are generated, the checker then invokes consensus across replicas, which compares the hash at its global hash index on the leader side.

This output consensus is based on the input consensus algorithm (§4) except that backups carry their hash at the same hash index back to the leader. For this particular output consensus, the leader first waits quorum. It then waits for a few μs in order to collect more remote hashes. It then compares remote hashes it has.

```
struct log_entry_t {
  consensus_ack ack[MAX]; // Output hash
  viewstamp_t vs;
  viewstamp_t last_committed;
  int node_id;
  viewstamp_t conn_vs; // viewstamp when connection was accepted.
  int call_type; // socket call type.
  size_t data_sz; // data size in the call.
  char data[0]; // data, with canary value in last byte.
} log_entry;
```

Figure 2: Falcon*'s log entry for each socket call.*

If a hash divergence is detected, the leader optionally invokes the local guard to forward a "rollback" command to the diverged replica's guard. The diverged replica's guard then rolls back and restores the server program to a latest checkpoint before the last successful output check (§5). The replica then restores and re-executes socket calls to catch up. Because output hash generations are fast and an output consensus is invoked occasionally, our evaluation didn't observe performance impact on this checker.

4 Input Consensus Protocol

This section first presents Falcon's RDMA-accelerated Paxos consensus protocol, including the fast,scalable algorithm in normal case (§4.1), handling concurrent connections (§4.2), and the leader election protocol (§4.3). It then analyzes the reliability of our protocol (§4.4).

4.1 Normal Case Algorithm

Recall that Falcon' input consensus protocol contains three roles. First, the Paxos consensus log (§3). Second, a leader replica's server program thread (in short, a leader thread) which invokes consensus request. For efficiency, Falcon lets a server program's threads directly handle consensus requests whenever they call inbound socket calls (e.g., `recv()`). Third, a backup replica's Falcon internal follower thread (§3) which agrees on or rejects consensus requests.

Figure 2 shows the format of a log entry in Falcon's consensus log. Most fields are regular as those in a typical Paxos protocol [47] except three ones: the `ack` array, the client connection ID `conn_vs`, and the type ID of a socket call `call_type`. The `ack` array is for backups to WRITE their consensus replies to the leader. The `conn_vs` is for identifying which connection this socket call belongs to (see 4.2). The `call_type` identifies four types of socket calls in Falcon: the `accept()` type (e.g., `accept()`), the `recv()` type (e.g., `recv()` and `read()`), the `send()` type (e.g., `send()` and `write()`), and the `close()` type (e.g., `close()`).

Figure 2 shows the input consensus algorithm. Suppose a leader thread invokes a consensus request when it calls a socet call with the `recv()` type. A consensus request includes four steps. The first step (**L1**) is executing the actual Libc socket call, because Falcon needs to get the actual return values or received data bytes of this call and then replicates them in remote replicas' logs.

The second step (**L2**) is local preparation, including assigning a global, monotonically increasing viewstamp to locate this entry in the consensus log, building a log entry struct for this call, and writing this entry to its local SSD.

The third step (**L3**) is to WRITE a log entry to remote backups in parallel. Unlike a previous RDMA-based consensus algorithm [56] which has to wait for ACKs from remote NICs, our WRITE immediately returns after pushing the entry to its local QP between the leader and each backup, because Paxos has handled the reliability issues (e.g., packet losses) for our WRITEs. In our evaluation, pushing a log entry to local QP took no more than 0.1 μs. Therefore, the WRITEs to all backups are done in parallel (see **L3** in Figure 2).

The fourth step (**L4**) is the leader thread polling on its `ack` field (notably, different from an RDMA ACK) in its local log entry for backups' consensus replies. Once consensus is reached, the leader thread finishes intercepting this `recv()` socket call and continues with its server application logic.

On a backup side, one tricky synchronization issue is that an efficient way is needed to make the leader's RDMA WRITEs and backups' polls atomic. For instance, while a leader thread is doing a WRITE on `vs` to a remote backup, this backup's follower thread may be reading this variable concurrently, causing a incomplete (wrong) read.

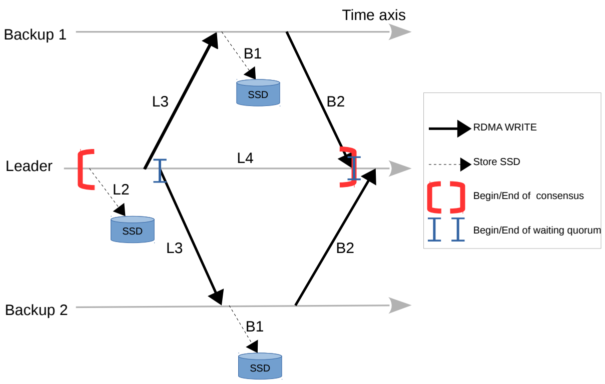


Figure 3: ***Falcon consensus algorithm in normal case.***

To address this issue, one existing approach [22, 32] leverages the left-to-right ordering of RDMA WRITEs and puts a special non-zero variable at the end of a fix-sized log entry because they mainly handle key-value stores with fixed value length. As long as this variable is non-zero, the RDMA WRITE ordering guarantees that the log entry WRITE is complete. However, because Falcon aims to support general server programs with largely variant received data lengths, this approach cannot be applied in Falcon.

An alternative approach is using atomic primitives provided by RDMA hardware, but a prior evaluation [63] has showned that RDMA atomic primivites are much slower than normal RDMA WRITEs and local memory reads.

Falcon tackles this issue by adding a canary value after the actual `data` array. Because Falcon uses a QP with the type of RC (reliable connection) (§2), the follower always first checks the canary value according to `data_size` and then starts a standard Paxos consensus reply decision [47]. Our efficient, synchronization-free approach guarantees that the follower always reads a complete log entry.

A follower thread in a backup replica polls from the latest un-agreed log entry and does three steps to agree on consensus requests, shown in Figure 3. First (**B1**), it does a regular Paxos view ID checking to see whether the leader is up-to-date, it then stores the log entry in its local SSD. Second (**B2**), it does a WRITE to send back a consensus reply to the leader's `ack` array element according to its own node ID. Backups perform these two steps in parallel (see Figure 2).

Third (**B3**, not shown in Figure 3), the follower does a regular Paxos check on `last_committed` and executes all socket calls that it has not executed before this viewstamp. It then "executes" each log entry by forwarding the socket calls to the local server program. This forwarding faithly builds and closes concurrent connections between the follower and the local server program according to the socket calls in the consensus log.

In a implementation level, Falcon stores log entries in local SSD using Berkely DB [9]. Although Falcon's algorithm does not wait every RDMA ACK in order to achieve high scalability, we use selective signaling (§2) to occasionally check and clear ACKs in the CQ associated with the QP, an essential ACK clearing step in RDMA implementations.

4.2 Handling Concurrent Connections

Unlike traditional Paxos protocols which mainly handle single-threaded programs due to SMR's deterministic state machine assumption, FALCON aims to support both single-threaded as well as multithreaded server programs running on multi-core machines. Therefore, a strongly consistent mechanism is needed to map every concurrent client connection on the leader and to its corresponding connection on backups. A naive approach could be matching a leader connection's socket descriptor to the same one on a backup, but backups' servers may return nondeterministic descriptors due contentions on systems resources.

Fortunately, PAXOS have already made viewstamps of socket calls strongly consistent across replicas. For TCP connections, FALCON addes the `conn_vs` field, the viewstamp of the the first socket call in each connection (i.e., `accept()`) as the connection ID for log entries. Then, FALCON maintains a hashmap on each local replica to map this connection ID to local socket descriptors.

4.3 Leader Election

Compared to traditional PAXOS leader election protocols, RDMA-based leader election poses one main issue caused by RDMA. Because backups do not communicate frequently with each other in normal case, thus a backup does not know the remote memory locations where the other backups are polling. Writing to a wrong remote memory location may cause the other backups to miss all leader election messages. An existing approach establishes a control QP with specific remote memory to handle leader election, but it poses complexity on the extra QPs and specific data structures.

For simplicity, FALCON handles election election with the same consensus log. It addresses this issue with a simple approach. The leader does WRITEs to remote logs as heartbeats with a period of T. Once backups have not received heartbeats from the leader for a period of 3*T, they start to elect new leader. Before starting a standard leader election phase as in tranditional PAXOS protocols [47], each backup always first does RDMA one-sided reads to get remote replicas' current polling location.

Then, backups start a standard PAXOS leader election algorithm [47] on consensus logs with three steps. First, backups propose a new view with a standard two-round PAXOS consensus [37] by including its current log entry offset. The backup with largest offset have a priority to win this proposal. Second, the winning proposer proposes itself as a primary candidate, another two-round PAXOS consensus. Third, if the second step reaches a quorum, the new leader notifies remote replicas itself as the new leader and it starts to WRITE periodic heartbeats.

4.4 Reliability

To minimize protocol-level bugs, FALCON's PAXOS protocol mostly sticks with a popular, practical implementation [47], espeically the behaviors of senders and receivers (§4.1 and §4.3). For instance, both FALCON's normal case algorithm and the popular implementation [47] involve two messages and same senders and receivers (although we use WRITEs and carefully make them run in parallel). We made this choice because PAXOS is PAXOS is notoriously difficult to understand [52, 37, 39, 61] or implement [14, 47] verify [65, 26]. Aligning with a practical PAXOS implementation [47] helps us incorporate these readily mature understanding, engineering experience, and the theoretically verified safety rules into our protocol design and implementation.

Although FALCON's PAXOS protocol works on a RDMA network, the reliability of this protocol does not rely on the lossless networking in RDMA. FALCON's protocol still complies with the standard PAXOS failure-handling model, where a stable storage exists, but hardware may fail, network may be partitioned, packets may be delayed or lost, and server programs may crash.

5 Output Checking Protocol

This section presents FALCON's output checking protocol for detecting and recovering from replicas' execution divergence. This section first introduces how FALCON computes and compares network outputs among replicas (§5.1), and then introduces its checkpoint and rollback mechanism to deal with divergence (§5.2).

5.1 Computing and Comparing Network Outputs

One main issue on checking network outputs is the physical timing when a server program produces an network output is usually miscellaneous. For example, when we ran Redis simply on pure SET workloads, we found that different replicas reply the numbers of "OK" replies for SET operations randomly: one replica may send four of them in one `send()` call, while another replica may only send one of them in each `send()` call. Therefore, comparing outputs on each `send()` call among replicas may not only yield wrong results, but may unnecesssarily slow down server programs among replicas.

To overcome this timing issue, FALCON presents a bucket-based hash computation mechanism. When a server calls a `send()` call, FALCON puts the sent bytes into a local, per-connection bucket with 1.5KB (same as MTU size). Whenever a bucket is full, FALCON computes a new CRC64 hash on a union of the current hash

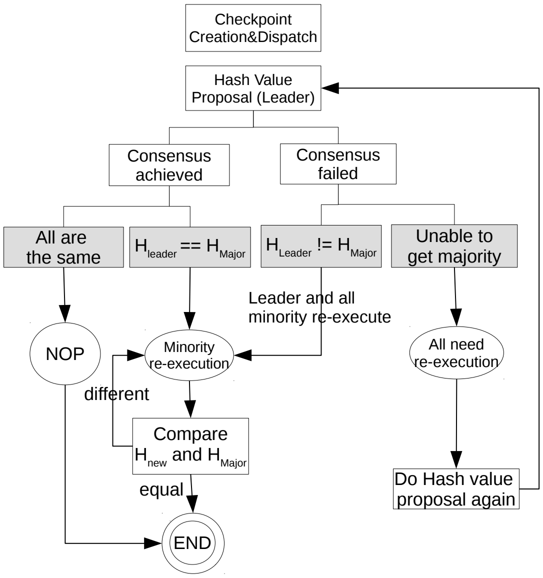

Checkpoint Creation&Dispatch
Hash Value Proposal (Leader)
Consensus achieved
Consensus failed
All are the same
$H_{leader} == H_{Major}$
$H_{Leader} != H_{Major}$
Unable to get majority
Leader and all minority re-execute
NOP
Minority re-execution
All need re-execution
different
Compare H_{new} and H_{Major}
equal
Do Hash value proposal again
END

Figure 4: ***Workflow to Handle Network Output Divergence.***

and this bucket. Such a hash computation mechanism encodes accumulated network outputs. Then, after every T_{comp} (by default, 1000 in FALCON) local hash values are generated, FALCON invokes a output checking protocol to check this hash across replicas. The index of this hash in the generated sequence is consistent across replicas because each replica runs the same mechanism to generate the hash sequence.

To compare a hash across replicas, FALCON's output checking protocol runs the same as the input coordination protocol (§4.1) except that the follower thread on each backup replica carries this hash value in the `ack` written back into the leader's corresponding log entry. To make an effort to collect more remote hash values, FALCON waits for $T_{waitack}$ (by default, 20 μs) and then starts to detect divergence on hash values.

Figure 4 shows the workflow on how the leader checks present replies and handles divergence, which includes four shaded cases: (1) all hashes are the same; (2) leader's hash equals a majority of replicas, but minor replicas' hashes diverge; (3) leader's hash diverges from a majority of replicas'; and (4) no majority has the same hash value. The first three cases should be the normal case unless a program tends to frequently compute outputs on random functions (e.g., a scientific similator).

Note that FALCON does has incorporated an input replaning approach (e.g., Eve [34] sequentially re-execute requests after a rollback) because we found that simply re-execution can already avoid the divergence. Previous work [44, 55] also confirmed that it was extremely unlikely to trigger a concurrency bug in both of two consenecutive executions.

5.2 Checkpoint and Restore Implementations

A guard process is running on each replica to checkpoint and restore the local server program. The guard does two tasks. First, FALCON assigns one backup replica's guard to checkpoint the local server program's process state and file system state of current working directory within a physical duration T_{ckpt} (by default, one hour in FALCON).

Such a checkpoint operation and its duration are not sensitive to FALCON's performance because the other backups can still reach quorum rapidly. Each checkpoint is associate with a last committed socket call viewstamp of the server program. After each periodical checkpoint, the backup distributes the checkpoint zip file to the other replicas.

Specifically, FALCON leverages CRIU, a popular, open source process checkpoint tool, to checkpoint a server program's process state (e.g., CPU registers and memory). Since CRIU does not support checkpointing RDMA connections, FALCON's guard first sends a "close RDMA QP" request to an FALCON internal thread, lets this thread closes all remote RDMA QPs, and then invokes CRIU to do the checkpoint.

The second task for guards is that all guards in all alive replicas handle rollback requests once divergence is detected (§5.1). According to the rollback workflow, a backup guard which receives a rollback request from the leader guard will kill the local server process and roll back to a previous checkpoint before the last successfuly hash check.

6 Discussions

This section discusses FALCON's limitations (§6.1) and its applications in other research areas (§6.2).

6.1 Limitations

FALCON currently does not hook random functions such as `gettimeofday()` and `rand()` because these random results are often explicit and easy to examine in network outputs (e.g., a timestamp in an HTTP reply header). Existing approaches [34, 47] in PAXOS protocols can also be leveraged to intercept these functions and make them produce same results among replicas.

FALCON's output checking protocol may have false positives or false negatives, because it is just designed to try to improve assurance on whether replicas run in sync. A server program running in FALCON may have false positive when it uses multiple threads to serve the same client connection and uses these threads to concurrently append outputs to networks (e.g., `ClamAV` in our evaluation). Running a deterministic multithreading scheduler [5, 18] with the server program addresses this problem.

A server program may also have false negative when it triggers a software bug but the bug does not propogate to network outputs. On client programs' point of view, such bugs do not matter. Moreover, FALCON already checkpoints file system state to mitigate this issue.

6.2 FALCON Has Broad Applications

In addition to greatly improving the availability of general server programs, we envision that FALCON can be applied in broad areas, and here we elaborate three. First, FALCON's RDMA-accelerated PAXOS protocol and its implementation could be an effective template for other replication protocols (e.g., byzantine fault-tolerance [35, 13]).

Second, by efficiently constructing multiple, equivalent executions for the same program, FALCON can benefit distributed program analysis techniques. Bounded by the limited computing resources on single machine, recent advanced program analaysis frameworks become distributed [50, 17] in order to offload analyses on multiple machines. FALCON can be leveraged in these frameworks so that developers of analysis tools can just focus on their own analysis logic, while FALCON's general replication architecture efficiently handles the rest.

Moreover, program analyses developers can tightly integrate their tools with FALCON. For instance, they can proactively diversify the orders of socket calls in FALCON's consensus logs among replicas to improve replicas' tolerance on security attacks [66].

Third, FALCON can be a core building block in the emerging datacenter operating systems [68, 28, 2]. As a datacenter continuously emerges a computer, an OS may be increasingly needed for thus a giant computer. FALCON's fast, general coordination service is espeicially suitable for such an OS's scheduler to maintain a consistent, reliable view on both computing resources and data in a datacenter. For instance, FALCON's latency is largely between less than 30 μs, much smaller than a typical context swtich of a process (typically, a few hundreds μs).

7 Evaluation

Our evaluation used three Dell R430 servers as SMR replicas. Each server having Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. Each machine has a Mellanox ConnectX-3 Pro Dual Port 40 Gbps NIC. These NICs are connected using the Infiniband RDMA architecture through a Dell S6000 high-performance switch with 32 40Gpbs ports. The `ping` latency between every two replicas are 84 μs (the IPoIB round-trip latency).

To mitigate network latency of public network, all client benchmarks were ran in a Dell R320 server (the client machine), with Linux 3.16.0, 2.2GHz Intel Xeon with 12 hyper-threading cores, 32GB memory, and 160GB SSD. This server connects with the replica machines with 1Gbps bandwidth LAN. The average `ping` latency between the client machine and a replica machine is 301 μs. A larger network latency (e.g., sending client requests from WAN) will further mask FALCON's overhead.

We evaluated FALCON on nine widely used or studied server programs, including four key-value stores Redis, `Memcached`, `SSDB`, `MongoDB`; `MySQL`, a SQL server; `ClamAV`, a anti-virus server that scans files and delete malicious ones; `MediaTomb`, a multimedia storage server that stores and transcodes video and audeo files; OpenLDAP, an LDAP server; `Calvin`, a widely studied transactional database system that leverages `ZooKeeper` as its SMR service. All these programs are multithreaded except Redis (but it can serve concurrent requests via Libevent). These servers all update or store important data and files, thus the strong fault-tolerance of SMR is especially attractive to these programs.

Table 1 introduces the benchmarks and workloads we used. To evaluate FALCON's practicality, we used the server developers' own performance benchmarks or popular third-party. For benchmark workload settings, we used the benchmarks' default workloads whenever available. We spawned up to 16 concurrent connections, and then we measured both response time and throughput. We also measured FALCON's bare consensus latency. All evaluation results were done with a replica group size of three except the scalability evaluation (§7.4). Each performance data point in the evaluation is taken from the mean value of 10 repeated executions.

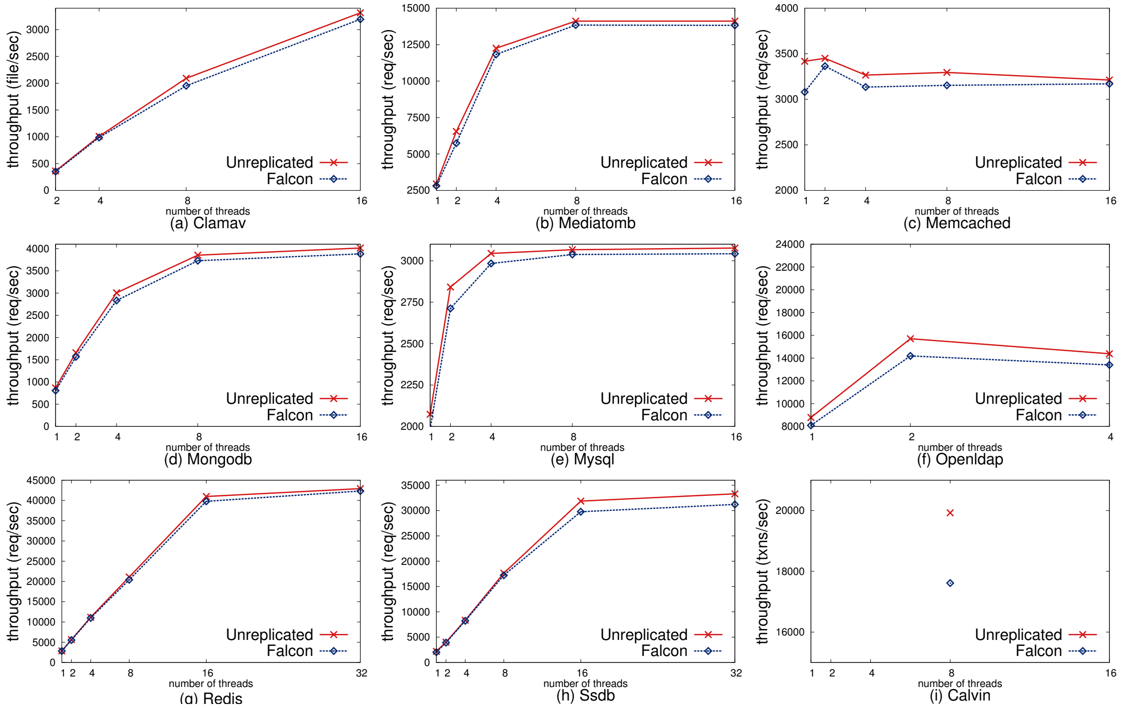
The rest of this section focuses on these questions:

§7.1: Is it easy to run general server programs in FALCON?

§7.2: What is FALCON's performance compared to the unreplicated executions? What is the consensus la-

Program	Benchmark	Workload/input description
`ClamAV`	clamscan	Files in `/lib` from same replica
`MediaTomb`	ApacheBench	Transcode video files in parallel
`Memcached`	mcperf	50% set and 50% put operations
`MongoDB`	YCSB	Workload C
`MySQL`	Sysbench	Concurrent SQL transactions
OpenLDAP	Self	Concurrent LDAP queries
Redis	Self	50% set and 50% put operations
`SSDB`	Self	50% set and 50% put operations
`Calvin`	Self	Concurrent SQL transactions

Table 1: *Benchmarks and workloads.* "Self" in the Benchmark column means we used a server program's own performance benchmark program.



(a) Clamav (b) Mediatomb (c) Memcached

(d) Mongodb (e) Mysql (f) Openldap

(g) Redis (h) Ssdb (i) Calvin

Figure 5: FALCON *throughput compared to the unreplicated execution.*

tency of FALCON's PAXOS protocol?

§7.4: How scalable is FALCON on different replica group sizes?

§7.3: What is FALCON's performance compared to existing SMR systems?

§7.5: How fast can FALCON checkpoint and recover replicas?

Program	# Calls	Input	SSD time	Quorum time
`ClamAV`	30,000	42.0	4.9 μs	7.2 μs
`MediaTomb`	30,000	140.0	4.4 μs	6.9 μs
`Memcached`	10,016	38.0	4.6 μs	6.3 μs
`MongoDB`	10,376	492.4	14.9 μs	16.4 μs
`MySQL`	10,009	26.0	5.0 μs	8.4 μs
OpenLDAP	10,016	27.3	6.4 μs	12.0 μs
Redis	10,016	107.0	3.6 μs	6.3 μs
`SSDB`	10,016	47.0	3.7 μs	10.9 μs
`Calvin`	10,002	93.0	2.5 μs	9.9 μs

Table 2: ***Leader's input consensus events per 10K requests.*** **The "# Calls" column means the number of socket calls that went through FALCON input concensus; "Input" means average bytes of a server's inputs received in these calls; "SSD time" means the average time spent on storing these calls to stable storage; and "Quorum time" means the average time spent on waiting quorum for these calls.**

7.1 Ease of Use

FALCON is able to run all nine evaluated programs without modifying them except `Calvin`. `Calvin` integrates its client program and server program within the same process and uses local memory to let these two programs communicate. To make `Calvin`'s client and server communicate with POSIX sockets so that FALCON can intercept the server's inputs, we wrote a 23 patch for `Calvin`.

7.2 Performance Overhead

Figure 5 shows FALCON's throughput and Figure 6 response time. We varied the number of concurrent client connections for each server program by from one to 16 threads. For `Calvin`, we only collected the 8-thread result because `Calvin` uses this constant thread count in their code to serve client requests. Overall, compared to these server programs' unreplicated executions, FALCON merely incurred a mean througput degrade by 4.16% (note that in Figure 5, the Y-axises of most programs start from a large number). FALCON's mean overhead on response time was merely 4.28%.

As the number of threads increases, all programs' unreplicated executions got a perforamnce improvement except `Memcached`. A prior evaluation [27] also observed a similar `Memcached` low scalability. FALCON scaled almost as well as the unreplicated executions.

FALCON achieves such low overhead in both throughput and response time due to two reasons. First, for each `recv()` call in a server program, FALCON's input coordination protocol only contains two one-sided RDMA writes and two SSD writes between each leader and backup. A parallel SSD write approach [10] may further improve FALCON's SSD performance. Second,

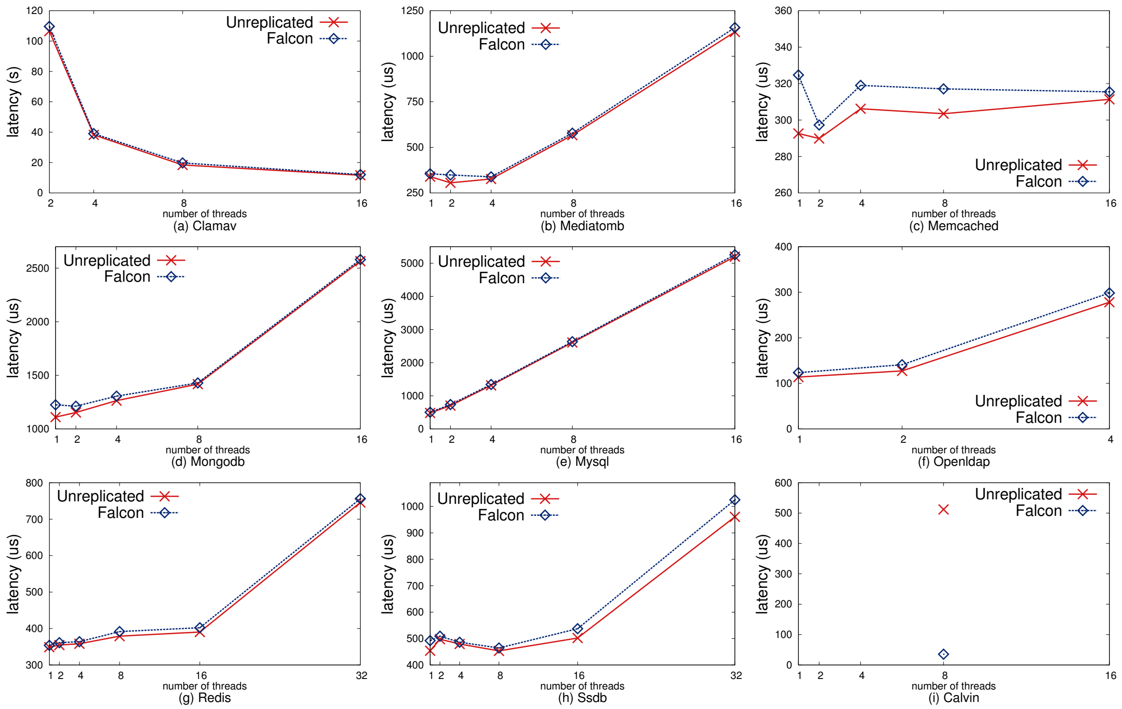


Figure 6: FALCON *response time (latency) compared to the unreplicated execution.*

FALCON's output checking protocol, which is based on the input coordination protocol, invokes occasionally, once for every T_{comp} output hash generations (§5.1).

To deeply understand FALCON's performance overhead, we collected the number of socket call events and consensus durations in the leader replica. Table 2 shows these statistics per 10K requests. For each socket call, FALCON's leader invokes a consensus, which includes an SSD write (the "SSD time" column in Table 2) and a quorum waiting phase (the "quorum time" column). The leader's quorum waiting phase implies backups' performance because each backup stores the proposed socket call in local SSD and then WRITEs a consensus reply to the leader.

By summing these two time columns, overall, a FALCON input consensus took only 9.9 μs (Redis) to 39.6 μs (MongoDB). This consensus latency mainly depends on the "Input" column: the average number of data bytes received in socket calls (e.g., MongoDB has the largest received bytes). FALCON's small consensus latency makes FALCON achieve reasonable throughputs in Figure 5 and response times Figure 6. This small latency suggests that, even if clients are deployed within the same datacenter network, FALCON may still achieve acceptable overhead on many programs (although FALCON's deployment model is running server replicas in a datacenter and clients in LAN or WAN).

7.3 Comparison with Traditional SMR systems

We compared FALCON with Calvin's SMR system because Calvin's inpur consensus uses ZooKeeper, one of the most widely used coordination service built on TCP/IP. To conduct a fair comparison, we ran Calvin's own transactional database server in FALCON as the server program, and we compared throughputs and the consensus latency with Calvin's ZooKeeper consensus.

As shown in Table 3, Calvin's ZooKeeper replication achieved 19.9K transactions/s with a 511.9 μs consensus latency. FALCON achieved 17.6K transactions/s with a 12.5 μs consensus latency. The throughput in Calvin was 13.1% higher than that in FALCON because Calvin puts transactions in a batch with a 10 ms timeout and then it invokes ZooKeeper for consensus on this batch. Batching helps Calvin achieves good throughput. FALCON currently has not incorporated a batching techinque because its latency is already reasonable (§7.2).

Notably, FALCON's consensus latency was 40.1X faster than ZooKeeper's mainly due to FALCON's RDMA-accelerated consensus protocol, although we ran Calvin's ZooKeeper consensus on IPoIB. A prior SMR evaluation [56] also reports a similar 320 μsZooKeeper consensus latency as we got. Two other recent SMR sys-

Performance metric	ZooKeeper	FALCON
Throughput (requests/s)	19,925	17,614
Consensus latency (μs)	511.9	12.5

Table 3: ***Comparison with Calvin's ZooKeeper replication.***

tems Crane [16] and Rex [27] may incur similar consensus latency as ZooKeeper's because both their consensus protocols are ran on TCP/IP. Overall, this FALCON-Calvin comparison suggests that Calvin will be better if clients prefer high throughput, and FALCON will be better if clients demand short latency.

7.4 Scalability on Replica Group Size

Because we had only three RDMA-enable machines in the evaluation time, we picked Redis as the testing program and ran five to seven FALCON instances on three machines: one machine held the leader instance, and each of the other two to three machines held two backups.

Given the same Redis workload, FALCON's five-replica setting had a 17.9K requests/s and 10.1 μs consensus latency, and its seven-replica setting had a 17.4K requests/s and 11.0 μs consensu latency. Compared to the three-replica setting, 18.0K throughput and 9.9 μs consensus latency, FALCON achieves a similar consensus latency from three to seven replicas, which mainly depends on the latency of two SSD stores and one WRITE round-trip (see Table 2).

We didn't consider these initial scalability results general; we found them promising. Given that each RDMA NIC hardware port supports up to 16 outbound RDMA WRITEs with peak performance [32], and our NIC has dual ports, we anticipate that our algorithm may scale up to around 32 replicas under current hardware techniques. We plan to buy more RDMA servers and verify whether FALCON's scalability is bounded by the capacity of RDMA hardware.

7.5 Checkpoint and Recovery

For all the nine programs, our output checking protocol found these servers produced identical results except ClamAV and MediaTomb. The output divergence of ClamAV is because its threading model. ClamAV uses multiple threads to serve a directory scan request, where the threads scan files in parallel and append the scanned files into a shared output buffer protected by a mutex lock. Therefore, sometimes ClamAV's output will diverge across replicas. Attaching a deterministic multithreading runtime with ClamAV avoided this divergence [16]. MediaTomb contained physical timestamp in its replies.

Each FALCON periodic checkpoint operation costed between 0.12s to 11.6s to the evaluated server programs. The checkpoint duration depends on whether a server's process has read files in its memory or modified files in its current working directory. ClamAV incurred the largest checkpoint time (11.6s) because we ran it to scan files with a total size of ~3GB in a /lib directory. We ran our recovery mechanism to recover ClamAV from its diverged timestamps, which rollbacks of all three replicas. A ClamAV rollback time varied from 0.81s to 2.72s for backups and the leader, including extracting a checkpoint ZIP, restoring process state with CRIU, and reconnecting RDMA QPs with remote replicas.

Performance metric	Three	Five	Seven
Throughput (requests/s)	18,054	17,890	17,410
Consensus latency (μs)	9.9	10.9	11.0

Table 4: FALCON*'s scalability on replica group size.*

8 Related Work

State machine replication. State machine replication (SMR). SMR has been studied by the literature for decades, and it is recognized by both industry and academia as a powerful fault-tolerance technique in clouds and distributed systems [38, 58]. As a common practice, SMR uses PAXOS [39, 37, 61] and its popular engineering approaches [14, 47] as the consensus protocol to ensure that all replicas see the same input request sequence. Since consensus protocols are the core of SMR, a variety of study improve different aspects of consensus protocols, including performance [49, 40] and understandability [52]. Although FALCON's current implementation takes a popular engineering approach [47] for practicality, it can also leverage other consensus protocols and approaches.

Five systems aim to provide SMR or similar services to server programs and thus they are the most relevant to FALCON. They can be divided into two categories depending on whether they use RDMA to accelerate their services. Tet first category includes Eve, Rex, Calvin, and Crane, which leverage traditional PAXOS protocols (or similar synchronized replication protocols, e.g., Zookeeper in Calvin) running on TCP/IP as the coordination service. The evaluation in these systems have shown that SMR services incur modest overhead on server programs' throughput compared to their unreplicated executions. However, for key-value stores that are extremely critical on latency, their consensus latency one order of magnitude higher than that in FALCON (§7.3). These systems can leverage FALCON's general, RDMA-accelerated protocol to improve latency.

Notably, Eve presents a execution state checking approach based on their coordination service. Eve's completeness on detecting execution divergence relies on whether developers have manually annotated all thread-shared states in program code. FALCON's output checking approach is automated (no manuall code annotation is needed), and its completeness depends on whether the diverged execution states propogate to network outputs.

These two approaches are complementary and can be integrated.

The second category includes DARE, a coordination protocol that also uses RDMA to reduce latency. FALCON differs from DARE in two aspects. First, the reliability model in DARE is different from that in PAXOS: DARE assumes that a replica's memory is still accessible to remote replicas even if CPU fails, so that the leader still writes to remote backups. With this reliability model, DARE requires four one-sided RDMA writes on a consensus round between the leader and a backup. DARE's paper shows that the MTTF (mean time to failure) of memory and CPU is similar. In contrast, FALCON's reliability model aligns with PAXOS's: memory and CPU may fail, thus consensus requests must be written to stable storage. Thus, FALCON requires two one-sided RDMA writes and two SSD writes on a consensus round between the leader and a backup. Our evaluation shows that FALCON's latency is compatible with DARE's. The second difference between DARE and FALCON is that FALCON aims to support general, diverse server programs, while DARE's evaluation uses a XX-line key-value store.

RDMA techniques. RDMA techniques have been leveraged in many online storage systems to improve latency and throughput within datacenters. These systems include key-value stores [48, 32, 22], transactional processing systems [63, 23], and XXX. These systems mainly aim to use RDMA to speedup specific communications, where both their storage and client access have RDMA enabled. FALCON's deployment model is to provide SMR fault-tolerance to general server programs deployed in datacenters, and the client programs access these server programs in LAN or WAN. It would be interesting to investigate whether FALCON can improve the availability for both the client side and server side of these advanced systems within a datacenter, and we leave it for future work.

Determinism techniques. In order to make multithreading easier to understand, test, analyze, and replicate, researchers have built two types of reliable multithreading systems: (1) stable multi-threading systems (or StableMT) [8, 42, 4] that aim to reduce the number of possible thread interleavings for program all inputs, and (2) deterministic multi-threading systems (or DMT) [11, 21, 51, 5, 7, 29, 6] that aim to reduce the number of possible thread interleavings on each program input. Typically, these systems use deterministic logical clocks instead of nondeterministic physical clocks to make sure inter-thread communications (e.g., `pthread_mutex_lock()` and accesses to global variables) can only happen at some specific logical clocks. Therefore, given the same or similar inputs, these systems can enforce the same thread interleavings and eventually the same executions. These systems have shown to greatly improve software reliability, including coverage of testing inputs [6] and speed of recording executions[7] for debugging.

Concurrency. FALCON are mutually beneficial with much prior work on concurrency error detection [67, 57, 24, 43, 45, 69], diagnosis [59, 54, 53], and correction [31, 62, 64, 30]. On one hand, these techniques can be deployed in FALCON's backups and help FALCON detect data races. On the other hand, FALCON's asynchronous replication architecture can mitigate the performance overhead of these powerful analyses [17].

9 Conclusion

We have presented FALCON, a fast, general state machine replication system through building an RDMA-accelerated PAXOS protocol to efficiently coordinate replica inputs. On top of this protocol, FALCON introduces an automated, efficient output checking protocol to detect and recover execution divergence among replicas and improve the assurance of sync in replicas. Evaluation on widely used, diverse server programs showt that FALCON supports these unmodified programs with reasonable performance overhead, and can recover from replica divergence caused by real-world software bugs. FALCON has the potential to greatly increase the adoption of state machine replication and improve the reliability of general programs.

References

[1] An Introduction to the InfiniBand Architecture. `http://buyya.com/superstorage/chap42.pdf`.

[2] Why the data center needs an operating system. `http://radar.oreilly.com/2014/12/why-the-data-center-needs-an-operating-system.html`.

[3] ZooKeeper. `https://zookeeper.apache.org/`.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution.

In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[6] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[8] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[9] `http://www.sleepycat.com`.

[10] A. Bessani, M. Santos, J. a. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, 2013.

[11] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[12] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, Oct. 1999.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[15] Criu. `http://criu.org`, 2015.

[16] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[17] H. Cui, R. Gu, C. Liu, and J. Yang. Repframe: An efficient and transparent framework for dynamic program analysis. In *Proceedings of 6th Asia-Pacific Workshop on Systems (APSys '15)*, July 2015.

[18] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[19] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[20] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[21] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[22] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[23] 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 *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[24] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[25] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[26] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, Oct. 2011.

[27] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multicore. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[29] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[30] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[31] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. Aug. 2014.

[33] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, 2010.

[34] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[36] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[37] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[39] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[40] L. Lamport. Fast paxos. Fast Paxos, Aug. 2006.

[41] libevent. `libevent.org/`, 2015.

[42] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[43] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[44] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[45] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[46] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[47] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[48] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2013.

[49] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[50] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 308–318, 2008.

[51] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[52] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[53] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[54] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[55] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[56] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[57] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[58] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[59] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[60] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. May 2014.

[61] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[62] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[63] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, SOSP '15, Oct. 2015.

[64] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[65] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the*

Sixth Symposium on Networked Systems Design and Implementation (NSDI '09), pages 213–228, Apr. 2009.

[66] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[67] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

[68] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.

[69] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.