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 inbound socket calls and runs a new RDMA-accelerated PAXOS protocol that needs only two RDMA write operations per input. This protocol addresses a scalability challenge by tightly integrating RDMA operations with 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 detected and recovered a real-world bug in `MySQL`.

1 Introduction

State machine replicaion (SMR) runs the same program on a number of replicas and uses a distributed consensus protocol (e.g., PAXOS [11]) 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 [8, 1, 11, 22, 18] 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 [8, 1] or processing SQL transactions [11, 22], but prohibitive for key-value stores. To mitigate this challenge, some recent SMR systems [**?**, **?**] batch requests into one consensus round. However, batching can only mitigate a server's throught lost; it may aggravate response time.

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 host machine to directly write to the memory of a remote host. As a common RDMA practice to ensure this write resides in remote memory, the local host waits until its NIC receives an ACK sent from the remote NIC. This whole round-trip does not invole the OS kernel or CPU at the remote host (i.e., it is a one-sided write). An evaluation [**?**] shows that such a round-trip takes only $\sim$3 μs in the Infiniband architecture [**?**].

However, despite much effort, it is still quite challenging to fully exploit RDMA speed in PAXOS protocols due to the unrichness of RDMA features. We present this challenge in detail by elaborating two existing approaches below. One straightforward approach is IP over Infiniband (IPoIB). This approach emulates TCP/IP on RDMA hardware so that traditional PAXOS implementations can enjoy RDMA speedup without modifications. However, this loose combination of RDMA and PAXOS is still one order of magnitude slower than fastest RDMA operations because IPoIB goes through the OS kernel and copies network data between kernel and user space.

To further improve consensus speed, DARE [**?**] proposes a second approach by simply replacing message passing in PAXOS with one-sided RDMA operations. For speed, DARE lets the leader handle a whole consensus round with three steps. The leader first appends a consensus request to a consensus log in all remote replicas with RDMA writes. For the the successful writes with ACKs, it then updates the tail pointers in remote logs and wait ACKs of these updates. Finally, the leader knows that the

minimum tail pointer among at least a majority of replcias reach consensus.

To the best of our knowledege, DARE's approach achieves the fastest consensus speed in existing approaches because all communications are simply replaced with the fatest RDMA writes (although we argue that a stable storage for consensus requests should be added to ensure PAXOS durability). However, this approach faces a scalability challenge: to ensure a remote replica is alive, each step has to wait ACKs from the previous step before it starts, and each RDMA write has to wait for its own ACK. In this pure leader-based algorithm, ACKs are necessary for every next step to start. As the replica group size grows, the leader has to do RDMA writes to remote replicas one by one, making its consensus latency grows linearly to replica group size (confirmed in our evaluation).

Our key observation to address this scalability challenge is that simply replacing RDMA writes with PAXOS communications is not sufficient, and we can integrate RDMA writes even *more tightly* with the fault-tolerant nature of PAXOS. In essense, PAXOS has already tolerated various faults, including machine failures and packet losses (even reliable connections in RDMA may lose packets during hardware and program failovers). Therefore, we can safely ignore the ACKs in RDMA writes and rely on PAXOS to handle the reliability issues of these writes.

This tight integration of PAXOS and RDMA writes looks simple, but it leads to a new fast, scalable PAXOS consensus algorithm with three steps in normal case. First, the leader stores a consensus request in local stable storage (SSD). 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 linear to replica group size (confirmed in our evaluation); its scalability is now mainly bounded by the max number of outbound RDMA writes in the RDMA NIC hardware.

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 new consensus algorithm to enforce same inputs across replicas.

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 [**?**, 18, 11, **?**]. Server programs now already use multithreading to harness the power of multi-core hardware. Nondeterminism [23, 16, 3, 15, 33, 28, 14, 13, 2] 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. Existing replication approaches either use only `ping` to check the liveness of replicas (e.g., Redis [37]) or rely on manually annotating share states in program code to detect execution divergence [22].

On top of the input consensus protocol, FALCON builds an efficient output checking protocol that occasionally checks output divergence across replicas. This checking protocol works by first computing a accumulated hash for 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 and restore mechanism for program deployers to make an effort to restore the diverged replicas.

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 [10] 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, while that of DARE is 7.0∼28.2 μs from 3 to 5 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 distributed protocols, 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 [41, 26, 24, 9, 27, 31]) 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. PAXOS guarantees that all replicas consistently agree to process this input as long as a majority of replicas agree. 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 (a program always produces the same output given 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 [31], 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).

2.2 RDMA

RDMA hardware recently becomes commonplace in datacenters due to its extreme low latency, high throughput, and its decreashing 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* [**?**]: 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-

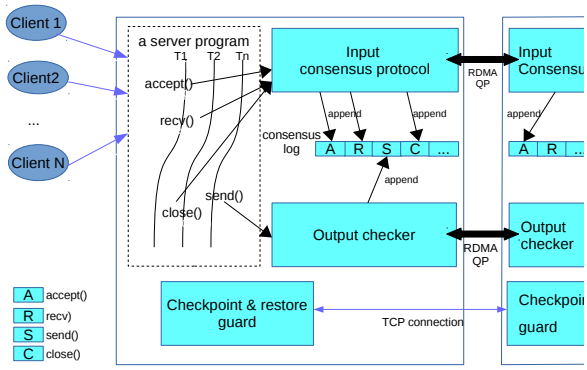

Figure 1: *The* FALCON *Architecture.* FALCON components are shaded (and in green).

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: the PAXOS consensus protocol for input coordination (in short, the *coordinator*), the output checking protocol (the *checker*), the in-memory consensus log (the *log*), the 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 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 to poll its consensus log for new consensus requests. If the coordinator agrees the request, this thread 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 its coordinator will just follow the leader's consensus requests on executing what socket calls and then forward these calls to the local server program.

```
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.*

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.

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 introduces the basic workflow of FALCON's PAXOS-based input coordination protocol in normal case (§4.1), describes how it handles concurrent connections (§4.2), presents the leader election protocol (§4.3), and then discusses the reliability of our protocol (§4.4).

4.1 Normal Case Operations

FALCON' input coordination protocol contains three roles. The first role is a PAXOS consensus log (for short, *log*) which resides in each replica and contains the same sequence of socket calls called by a server program. The second role is a leader thread which invokes consensus request and writes to its local log as well as remote replicas' logs. In FALCON, leader threads are just a server

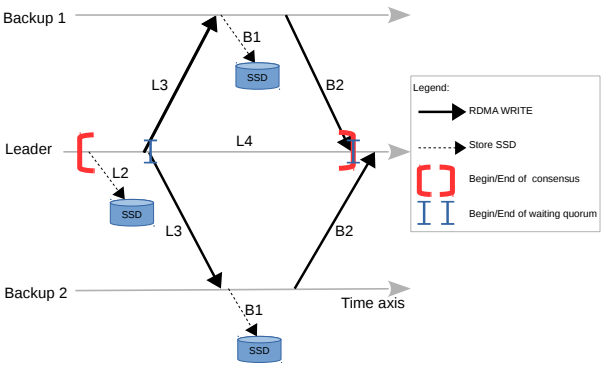

Figure 3: FALCON *consensus algorithm in normal case.*

program's worker threads that processes client requests. A leader replica can have multiple leader threads depending on how many threads Third, a follower thread which (dis)agrees consensus requests on a backup replica.

Figure 2 shows the data structure of a log entry in FALCON's consensus log. Most fields are regular as those in a typical PAXOS protocol 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 replicas to write back their consensus replies to the leader via a RDMA one-side write. The `conn_vs` is for identifying which connection this socket call belongs to (see 9). 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()`).

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 is executing the actual socket call, because FALCON needs to get the actual received data bytes and then replicates them in remote replicas' logs.

The second step is local preparation, including assigning a global, monotonically increashing 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 is sending consensus requests to replicas by using RDMA one-sided write to write this struct to remote replicas' logs. Note that these RDMA write operations do not need to wait for the struct actually be written to remote logs. Instead, FALCON's leader thread just return immediately after putting the structs to the RDMA QP (Queue Pair) between a backup replica, because PAXOS protocol FALCON implements has already considered packet losses (e.g., due to remote replica failures).

The fourth step is waiting for replicas' consensus replies. The leader does a busy loop to check the `ack` array until it sees that a majority of replicas, including itself, have aggreed on this log entry. In normal case, both the third and fourth steps will return immediately because no synchronization context switch is involved.

On a replica side, one tricky issue on replying consensus request is that, unlike traditional TCP/UDP messages, RDMA on-sided write operations do not guarantee atomicy. For instance, a leader thread can be doing a remote RDMA write on the viewstamp `vs`, while a follower thread can be reading this variable concurrently without knowing when the leader's write can finish, causing the replica to read a partial (wrong) value. Thus, in a RDMA-accelerated protocol, an extra mechanism is needed to gurantee that a leader's write has finished and then replicas can agree or disagree.

FALCON's follower thread tackles this issue by adding a canary value after the actual `data` array. Leveraging a RDMA feature that its writes are lossless in normal case, the follower thread always checks the canary value according to `data_size` and then starts the consensus reply decision. This check guarantees that a log entry is completely written in a local backup.

To ahchieve small consensus latency, FALCON's follower thread does a busy loop on a dedicated CPU core to agree on consensus requests from the leader. Each backup only needs one follower thread. This thread always busy reading the latest un-agreed log entry in its local log, and it it sees a log entry has completely written, it runs three steps. First, it does a regular PAXOS view ID checking to see whether the leader is up to date, and if so, it stores the log entry in its local SSD. Second, it does a RDMA write to send back a Yes/No ack to the leader's `ack` array element with its own node ID.

Third, the follower thread does a regular PAXOS check on `last_committed` and executes all socket calls that it has not executed before this viewstamp. It "executes" each log entry by sending the data in each entry to the local server program. On replicas, server programs run as is without being intercepted. In short, this follower thread runs a high performance loop to respond consensus requests and forward data to the local server program. Since each backup machine only has one follower thread and nowadays machines often have spare cores, we didn't find that the spin loop of this thread brought bring negative performance impact in our evaluation.

This protocol is highly optimized for minimizing consensus latency in normal case. In total, a consensus between the leader and one backup only requires two one-sided RDMA write operations (one from the leader to the backup and the other from the backup to the leader) and two SSD write operations (each in leader and backup). Although each RDMA one-sided operation takes about 3 μs, FALCON's protocol just puts the log entry to

the RDMA Queue Pair without needing to wait until the write succeeds on the remote backup, because the leader will have PAXOS's consensus reply (the RDMA write from the backup to leader). In addition, neither a leader thread or a follower thread does a synchronization context switch during this consensus (a synchronization context switch typically takes sub milli seconds, pretty slow).

4.2 Handling Concurrent Connections

Unlike traditional PAXOS protocols which mainly handle single-threaded programs due to their 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 match every concurrent client connection on the leader and to its matching connection on replica machines. A naive approach could be matching a leader's socket descriptor to the same one on a backup replica, but note that backups' servers may return different descriptors because such systems resources could be contended by other threads in the server's process and nondeterministic.

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 (aka, `accept()`) to as the connection ID for socket calls in the log. On a local replica, FALCON maintains a hash map between this connection ID to the actual local socket descriptor for each connection, then FALCON ensures that data bytes are forwarded from a leader's connection to the right matching connection on backups. FALCON also intercepts socket calls with the `close()` type to clean up this map. For servers that use UDP to serve requests, FALCON maps the viewstamp of a `recvfrom()` call to the socket descriptor returned from this call, and it cleans up the map on a corresponding `sendto()` call.

4.3 Leader Election

4.4 Reliability and Safety

To minimize protocol-level software bugs, FALCON's input coordination protocol design chooses the same replica behavior as a popular, traditional PAXOS protocol protocol [31], although FALCON uses RDMA operations to deliver consensus requests and replies and added some extra data structure fields. For instance, both FALCON and the protocol [31] involve two messages between a leader replica and a backup replica in normal case, and they both have the same four steps involved by same replica roles on leader election. We made this

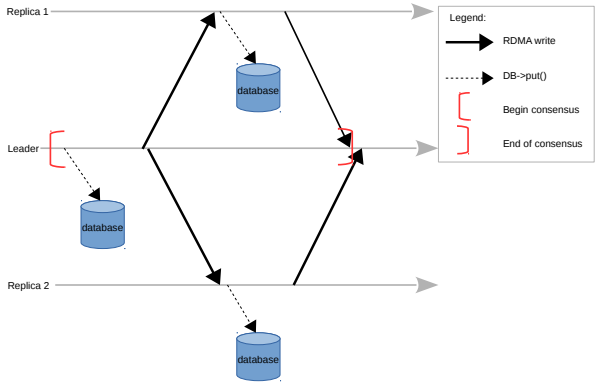

Figure 4: ***DARE consensus algorithm in normal case.***

design choice because PAXOS is notoriously difficult to understand, test, and verify; sticking with a traditional replica behavior helps us incorporate the readily mature understanding, engineering experience, and the theoretically verified reliability rules [**?**] into our new RDMA-accelerated PAXOS protocol.

A PAXOS protocol must ensure safety: the agreed operation must be an actually proposed one; if agreed, all active replicas must consistently enforce this operation. Safety is another important sweet spot that FALCON's input coordination protocol inherites from traditional PAXOS protocol by sticking with same replica behavior in traditional ones. As a traditional protocol, our input coordination protocol also uses view IDs and viewstamps to enforce a consistent, up-to-date leadership across replicas. To address the atomicy issue between remote RDMA writes and local memory reads, our protocol addes an completion check on log entries (§4.1).

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" results 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 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 is 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 log entry.

This output protocol starts by letting a leader thread invoke a consensue request on this hash comparison for its own client connection. The leader also writes its own hash value in the `ack` array with its own replica ID. Then, follower threads on backup replicas carry their local hash values in the `ack` array according to the connection ID `conn_vs`. Once a quorum of `ack` is ready, the leader simply detects divergence with the hash values in its local `ack` array. Note that at this moment, some replicas may not send back their reply yet because only a quorum is reached. To make an effort to collect a complete hash values, FALCON waits for $T_{waitack}$ (by default, 20 μs) and then starts to detect divergence on hash values.

Figure 5 shows the workflow on how the leader checks present replies and handles divergence, which include four possible 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). Even so, we can leverage prior approaches to hook these random functions [31, 22] and make them generate same return values among replicas.

Once the leader decides to roll back a diverged replica (including itself), it invokes a local guard process (§3) that handles checkpointing and rolling back the local server program. If a remote replica needs to be rolled back, the local guard forwards the roll back request to the guard in that remote replica.

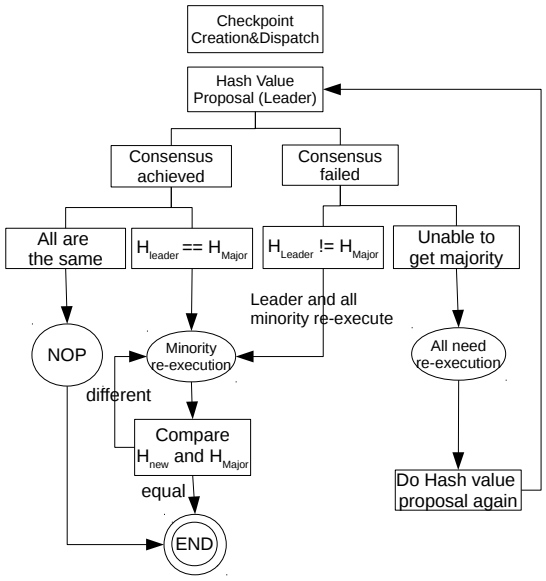

Figure 5: ***Workflow on Handling Network Output Divergence.***

We explicitly design this output checking protocol to be fast with two considrations: (1) because the hash comparison consensus happens occasionally (every T_{comp} hash generations), the performance penalty on this output consensus is negligible; and (2) all or most replicas' hash values are the same in normal case. Two parameters, T_{comp} and $T_{waitack}$, may be sensitive to FALCON's performance. We did an evaluation on diverse server programs and found that these default values are reasonable, general settings to these programs (§**??**).

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 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 leader and the other backups can still reach a consensus 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 tar bar to the other replicas to cope with replica failovers.

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 Queue Pair" request to an FALCON internal thread spawned in FALCON's LD_PRELOAD library, let this thread closes all RDMA Queue Pairs of this server's process with the other replicas' server processes, and then invokes CRIU to do the checkpoint.

After checkpointing process state, FALCON uses CRIU to stop this process and then checkpoints files in the process's current working directory recursively, including the socket call stable storage (§4). To avoid a server to modify files missed by FALCON, we only assign a server process write permissions to its current working directory and the "/tmp" directory (files in "/tmp" directory often do not matter). FALCON's file system checkpoint avoids the need of comparing program outputs to the file system.

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 guard which receives a rollback request kill the local server process and roll back to a previous checkpoint before the last matching hash comparison. Suppose the viewstamp associated with the previous checkpoint is vs_{ckpt}, and the viewstamp of the last matching hash comparison log entry is vs_{match}, FALCON ensures that vs_{ckpt} is smaller than vs_{match} so that this checkpoint is not diverged.

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 our output checkers have not detected network output divergence comming from these functions. Existing approaches [22, 31] in PAXOS protocols can 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 just makes an effort to practically indicate that replicas are running the same execution states. 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 produce outputs (e.g., `ClamAV`). Running a DMT scheduler in FALCON can address this problem. In our evaluation, all programs except `ClamAV` uses only one thread to process one client connection and they don't have such false positives.

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; FALCON already checkpoints file system state to mitigate this issue.

When execution divergence is detected in a replica, FALCON's rollback mechanism is not designed to guarantee that the re-executions of this replica will definitely avoids this divergence. We made this design choice because both our evaluation and a previous work Eve [22] found that divergence happens extremely rarely in evaluation. In our evaluation, we found that simply re-executing the log can practically make FALCON's replicas converge to same execution states. A similar finding is that although Eve provided a sequential re-execution approach to with divergence avoidance guarantee, which FALCON can leverage, but even Eve's evaluation didn't experience any divergence and thus this approach was not invoked.

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).

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 reliability and security analaysis frameworks are moving towards distributed 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, analyses developers can tightly integrate their tools with FALCON (e.g., they can proactively diversify the orders of socket calls in FALCON's consensus logs among replicas to improve replicas' tolerance on security attacks).

Third, FALCON can be a core building block in the emerging datacenter operating systems [**?**, **?**, **?**]. 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 15 to 20 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, 32GB memory, and 1T 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. This latency is achieved through IPoIB (IP over Infiniband), the optimal latency for a client to communicate with a server through traditional TCP/IP between replica machines.

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 160G 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 still serve requests concurrently using Libevent). These servers all update or store important data and files, thus the high 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 which made these servers approach peak throughput, and then we measured both response time (latency) 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.

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	TBD
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.

The rest of this section focuses on these questions:

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

§7.2: What is FALCON's performance compared to the unreplicated executions? What is FALCON's consensus latency on input coordination and output checking?

§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 recover replicas from output divergence?

7.1 Ease of Use

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

7.2 Performance Overhead

Figure 6 shows FALCON's throughput and Figure 7 response time. We varied the number of concurrent client connections for each server program by from one to 16 threads or until they reached peak performance. For `Calvin`, we only collected the 8-thread result because `Calvin` used this constant thread count 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 6, 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 [18] also observed

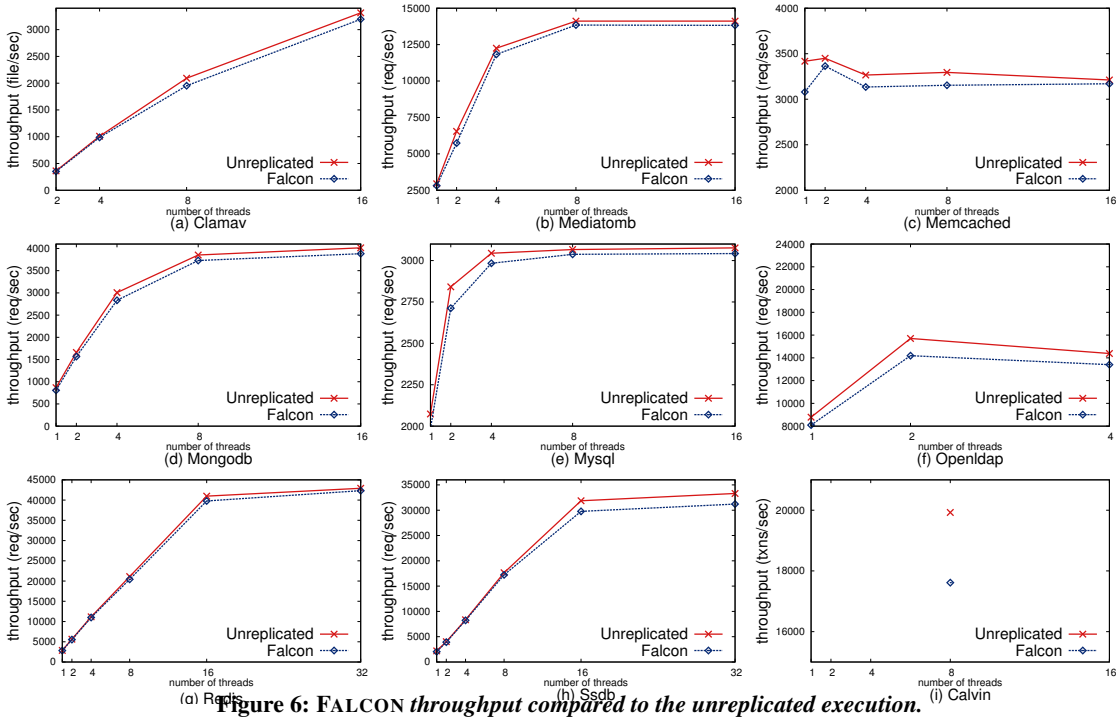

Figure 6: FALCON *throughput compared to the unreplicated execution.*

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

To deeply understand FALCON's performance overhread, we collected the number of socket call events and consensus durations in the leader replica. Table 2 shows these statistics for every 10K requests. For each socket call, FALCON's leader invokes a consensus, which includes an SSD write phase (the "SSD time" column in Table 2) and a quorum waiting phase (the "quorum time" column). The quorum waiting phase implies replicas' performance because each replica stores the proposed socket call into SSD and then sends a consensus reply by doing an RDMA write 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 6 and response times Figure 7. This small latency suggests that FALCON may still achieve acceptable overhead on some programs even if clients are deployed within the same datacenter network, although FALCON's default 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, both FALCON and `Calvin` SMR, we ran `Calvin`'s own transactional database as the server program, and we compared throughputs and the consensus latency. FALCON achieved 17.6K transactions/s with a 12.5 μs consensus latency. `Calvin` achieved 19.9K transactions/s with a 511.9 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` ahieves good throughput. FALCON currently has not incorporated a batching techinque because its latency is already reasonably fast (§7.2).

Notably, FALCON's consensus latency was 40.1X faster than `ZooKeeper`'s mainly due to FALCON's RDMA-accelerated consensus protocol. A prior SMR

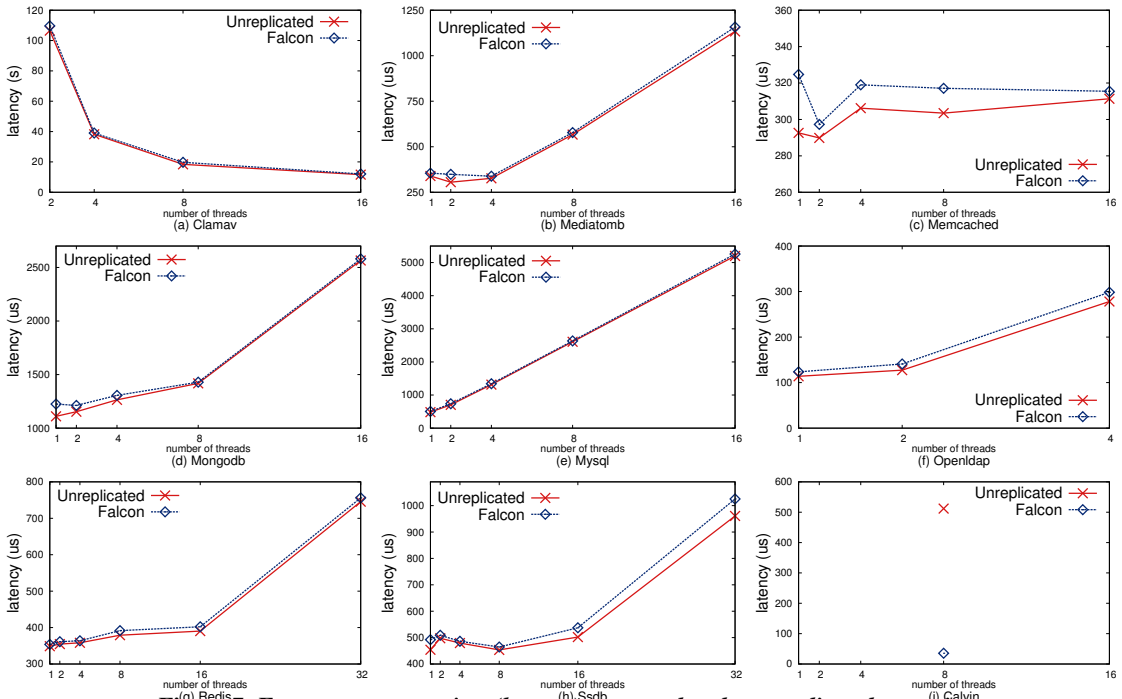

Figure 7: FALCON *response time (latency) compared to the unreplicated execution.*

evaluation [**?**] also reports a similar 320 μs`ZooKeeper` consensus latency as we got. 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 FALCON instances on three machines: one machine held the leader instance, and each of the other two machines held two backups.

Given the same Redis workload, FALCON had a 17.9K throughput and 10.0 μs consensus latency. Compared to the three-node setting, 18.0K throughput and 9.9 μs consensus latency, FALCON is quite scalable. For seven nodes XXX TBD. We plan to buy more server machines for a larger scale of scalability evaluation.

7.5 Recovering from Output Divergence

Once an output divergence is detected, FALCON will roll bacak the divergent replica, including leader and backups. For all the nine programs, our output checking protocol found these servers produced identical results except `ClamAV` and `SSDB`.

`ClamAV`'s divergent output is caused by its special threading model: unlike the other evaluated programs which use one thread to server one client connection, `ClamAV` uses multiple worker threads to serve a client request (e.g., scanning a directory path recursively). FALCON's worker threads concurrently contend for a global mutex lock and then append the scanned files into the output buffer for the client, causing a nondeterministic order of scanned files among all replicas. We manually compared `ClamAV`'s output across replicas, and we found the files were indeed the same except their order.

We ran `SSDB` on the same workload as in §7.2 and triggered a previous unkown concurrency bug in the QPUSH operations. This concurrency bug was triggered by concurrent client connections to push elements to a queue in `SSDB`. In our evaluation, this bug was first triggered in a backup machine and caused a divergent output hash. the hashes of the other two replicas were still the same. FALCON's leader detected this divergence; it sent a rollback request to the divergent replica's guard, which took 1.2 ms; the guard killed the `SSDB` server and restored it from a prior checkpoint with 0.913 ms, and the recovered replica reconnected to the other replicas and started to server requests in 0.09 ms. Each process and file system checkpoint operation for `SSDB` took 954 ms.

This minor divergence suggests that we hitted a bug, for two reasons. First, FALCON has enfored strongly consistent inputs across replicas, if only replicas diverge, this divergence must be caused by the server program itself, not different inputs. Second, only one replica diverged, this suggests that the output does not contain randome values (e.g., physical times, or the `ClamAV` case), other-

wise all replicas should diverged.

We then carefully looked into the SSDB source code and identified the bug. It was caused by incorrect synchronization to a global queue in SSDB. We have reported this bug to the SSDB developers with a suggested bug fixing patch [**?**]. Although FALCON does not intend to find bugs, this promising finding reflects that FALCON could be extended as an advanced testing tool: its fast, general SMR service lets it easily support general program, and its consistently enforce inputs and efficient output checking makes the minor replica output divergence a strong software bug indicator.

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 [25, 39]. As a common practice, SMR uses PAXOS [26, 24, 41] and its popular engineering approaches [9, 31] 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 [32, 27] and understandability [34]. Although FALCON's current implementation takes a popular engineering approach [31] 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.

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	3.7 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.**

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 [**?**, **?**, **?**], transactional processing systems [**?**, **?**], 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 multi-threading easier to understand, test, analyze, and replicate, researchers have built two types of reliable multi-threading systems: (1) stable multi-threading systems (or StableMT) [6, 28, 2] that aim to reduce the number of possible thread interleavings for program all inputs, and (2) deterministic multi-threading systems (or DMT) [7, 16, 33, 3, 5, 19, 4] 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 [4] and speed of recording executions[5] for debugging.

Concurrency. FALCON are mutually beneficial with much prior work on concurrency error detection [44, 38, 17, 29, 30, 45], diagnosis [40, 36, 35], and correction [21, 42, 43, 20]. 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 [12].

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] `http://redis.io/`.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.