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, even if minor replicas fail. This strong fault-tolerance is particularly attractive for building a general SMR system for server programs. Unfortunately, to agree on inputs, traditional consensus protocols often incur prohibitive network latency. We present FALCON, a simple, deployable SMR system for general server programs by leveraging Remote Direct Memory Access (RDMA). FALCON intercepts inputs in a server's socket API, and it runs a new PAXOS consensus protocol that needs only two fast RDMA one-sided write operations for each input in normal case. FALCON addresses a pervasive challenge, avoiding a server' execution state divergence in active replicas, by presenting a fast, application-agnostic output checking mechanism on top of our consensus protocol.

Evaluation on 9 widely used, diverse server programs (e.g., `Memcached`, `MySQL`, and `ClamAV`) shows that FALCON is: (1) general, it ran these servers without any modification; (2) fast, it incured merely 4.28% mean overhead in latency and 4.16% in throughtput on these programs on popular benchmarks, and it ran 6X∼8X faster than three replication systems; and (3) robust, it detected and recovered execution divergence caused by a software bug in Redis, while Redis's own replication service missed the bug.

1 Introduction

State machine replicaion (SMR) runs the same program on a number of replicas and uses a distributed consensus protocol (e.g., PAXOS [10]) to enforce the same inputs among replicas. Consensus on a new input can be achieved if a majority of replicas agree, thus SMR can tolerate various faults such as minor replica failures. SMR's strong fault-tolerance mainly stems from its *synchronized replication*: before processing an input, each replica must stores this input in local stage storage and receives a majority of replica aggreements. Due to this strong fault-tolerance, recent SMR systems have shown promising results on greatly improving the availability of various programs, including systems that target both specific programs [8, 1] and general programs [10, 18, 14].

Unfortunately, despite these promising advances, two practical challenges still prevent SMR from being widely deployed for general programs, especially server programs that naturally demand high availability. The first challenge is performance. To reach consensus on input requests, traditional consensus protocols invoke TCP/UDP messages, which go through software network layers and OS kernels and often cause prohibitive latency. This network latency is especially severe in modern server storage servers (e.g., key-value stores) that tend to move computation and data into memory. To mitigate this challenge, Some SMR systems [**?**, **?**] batch requests into one consensus round, but this could only mitigate throught lost but not latency. As a possible consequence, although many recent storage systems [**?**] explicitly stated that they needed a replication system for high availability, they finely didn't adopt the batching approach.

The second challenge is that an automated, fine-grained approach is needed to avoid execution divergence of active (i.e., alive) replicas. Even in the absence of replica failures or network partitions, the executions of different replicas can still diverge due to contention of inter-thread resources [3] (e.g., shared memory) and systems resources [19] (e.g., files and network ports). This challenge not only lies in standard SMR systems which require deterministic executions, but it is also pervasive in commodity replication systems (e.g., Redis, `Memcached`, and `MySQL`) that seek for fault-tolerance in some degree.

Recently, Remote Direct Access Memory (RDMA) has become increasingly cost-efficient to be adopted in data center networking. RDMA's advantages such as make opens new opportunity to build fast consensus protocols. For instance, one-sided write operations allow one machine to directly writes to a remote machine without the involement of remote machine's software network layers, OS kernels, or CPU. Although these software layers play important roles in the reliability of traditional network communications, we argue that these layers are not inherent to distributed consensus protocols. For instance, the PAXOS consensus protocol has theoretically proven in the tolerance of of various faults (e.g., NIC failures and OS kernel crashes) and provided high availability in practice.

This paper present FALCON, an SMR systems that replicates general server programs efficiently by ex-

ploiting the fastest RDMA operations. With FALCON, a server program just runs as if it is the single copy, and FALCON automatically deploys this program on replicas of machines. FALCON enforces same network inputs and verifies network outputs for replicas. If FALCON finds that a replica produces an different output from what other replicas agree on, FALCON recovers this replica to a previous program checkpoint and re-executes inputs that have been agreed on from the checkpoint.

To coordinate inputs among replicas, FALCON intercepts a server program's socket APIs (e.g., `recv()`) to caputure inputs and introduces a new RDMA-accelerated PAXOS protocol to let replicas agree on these inputs. To ease understanding and checking tooks. this protocol complies with common style of popular paxos protocal [27]. In the normal case of this protocol, contrast to existing implementations which require one network round-trip (i.e., two messages for every two replicas), our protocal only requires two most efficient one-sided write operations.

However, input coordination is not sufficient to practically enforce same execution states for the same program across replicas. Nowadays most server programs already adopt multi-threading or multi-process models to harness the power of multi-core and improve performance. Contentions on inter-thread resources (e.g., global memory and Pthreads mutex locks) and systems resources (e.g., network ports and files) can easily cause program execution states among replicas to diverge an can never converge again.

To address this challenge, recent SMR systems leverage either deterministic multithreading techniques [14, 10] or detecting divergence of execution by manually annotating program states by threads, artificially trading off performance or automatacity.

Our key idea is that we don't need to a program's every (or every batch) network outputs because they most replicas's outputs indicate that this output is most likely the produced one. Either necessary or sufficient. Not necessary because most executions already produce same program behaviors (including outputs) even with concurrency bugs. Not sufficient because it could be all replicas producing the same buggy output and bypass consensus protocols. All we need is just lazily compare outputs and if a divergence is detected, we roll back programs and re-execute them.

To implement this idea, FALCON's output verification protocol first . network outputs on each individual replica, computes hash values incremental: compute the hash value of a union of last hash value and the output and periodically invoke our PAXOS consensus protocol to exchange the hash value. Then, if minor replicas' outputs diverge from the majority ones, we just roll back and re-execute these minor replicas without perturbing the others to agree on and process new inputs. If a majority can not reach, FALCON simply rolls back the XXX (leader?). Evaluation confirmed that XX.XX% cases.

In a conceptual level, to provide pratical SMR service for general programs, FALCON presents a new agree-execute-verify execution model, which contrasts from previous agree-execute models and execute-verify models. We argue that agree is essential to SMR due to its strong fault-tolerance on machine failures and packet losses (even RDMA networks have packet loss when machines fail or programs crash). Having a general input coordination protocol also mitigates the need of writing application-specific input mixer and manually code annotation. Moreover, a automatic, fast output verification protocol is essential to SMR because we aim to replicate general, diverse server programs that may diverge due to resource contentions. In sum, by coordinating inputs and verifying outputs among replicas, FALCON practically enforces same execution states and outputs among replicas.

We implemented FALCON in Linux. FALCON intercepts common POSIX incoming socket operations (e.g., `accept()` and `recv()`) to coordinate inputs using the Infiniband RDMA architecture. FALCON also intercepts outcoming 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 support practical checkpoint and restore on server programs, FALCON leverages CRIU.

We evaluated FALCON on 10 server programs, including 9 widely used server programs: 4 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, one FTP server `Open TFTP`. We have also evaluated `Calvin`, an advanced transactional database system that leverages `ZooKeeper` as its SMR service. Our evaluation shows that

1. FALCON is general. For the 9 widely used server programs, FALCON ran them without any modification. We only needed to modify `Calvin` because it integrates its sever and client in the same process; we wrote a 23 patch to make `Calvin`'s server use POSIX sockets to accept client requests.
2. FALCON is fast. Compared to the 10 servers' unreplicated executions, FALCON incurred merely 4.16% overhead on throughtput and 4.28% on response time in average. FALCON is 40.1X faster than `Calvin`'s zookeeper-based SMR service on response time.
3. 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.

4. FALCON is extensible. To extend optimization on read-only requests, XX lines of code in our two provided APIs, FALCON is able to avoid the read-only requests in Redis to do consensus and XX times faster than Redis's own replication system.

Our major conceptual contribution is leveraging RDMA to make synchronized, PAXOS-based replication adoptable. This new protocol incorporates fasted RDMA hardware features, while it still pertains same fault-tolerance guarantees as traditional PAXOS protocols. FALCON has the potential to serve as an effective research template for other replication areas (e.g., byzantine fault-tolerance). In addition, a fast, general SMR service has been long persued as a fundamental building block for the emerging datacenter operation system.

Our major engineering contribution includes the FALCON implemention and its evaluation on 10 diverse, widely used server programs. Due to the lack of a general SMR system, industrial developers have spent tremondous efforts on building specific replication systems for their own programs and "invent the wheels again and again". Note that understanding, building, and maintaining a usable SMR systems requires extreme expert knowledege, burdens, and are extremely challenging (so many PAXOS papers). For example, the Redis or `Memcached` lag bug. Our FALCON system and evaluation has shown promising results on building a fast, general, and extendible SMR system and help developers greatly release these burdens. We have released all FALCON's source code, benchmarks, and raw evaluation results at `github.com/osdi16-p83/falcon`.

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 with key components. §4 presents our input coordination protocol. §5 output checking protocol. §**??** introduces implementation details. §6 FALCON's discusses limitations and applications. §7 presents evaluation results, §8 discusses related work, and §9 concludes.

2 FALCON 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) to coordinates inputs across replicas. For efficiency, in normal case, PAXOS often let 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.

When a new input comes, PAXOS starts a new consensus round, which invokes a consensus request on this input to the other replicas. PAXOS guarantees that all replicas consistently agree to process this input as long as a majority of replicas' agreement. This quorum based consensus makes PAXOS tolerate various faults such as machine failures and network crashes. Before a replica agrees on a input, PAXOS logs this input in the replica's persistent storage for durability. As rounds move on, PAXOS consistently enforce the same sequence of inputs among replicas. If a program runs as a deterministic state machine (i.e., given the same input, the program always produces the same output), PAXOS guarantees that programs on active replicas never diverge.

Netork latency of consensus messages is one key challenge to make SMR support general server programs which demand high perforrmance, espeically in-memory storage servers. Because each input input is processed by a server program, existing consensus protocols involve consensus messages on TCP or UDP, which go through software network layers and the OS kernel, causing hundreds of μs latency in LAN. For instance, even in an efficient PAXOS protocol, each input in normal case takes two consensus messages between every two replicas (one a request from leader to backup and the other a reply from backup to leader).

2.2 RDMA

RDMA recently has become commonplace in Datacenter networking due to its high performance and its decreashing price. For instance, a machine with 40Gbps RDMA NIC and 24-cores costs 3.8K US $, and a RDMA switch with 40Gbps costs about 16K US $. RDMA provides three types of communication primitives, including IPoIB, message verbs, and one-sided read/write operations, from slow to fast. To perform RDMA operations, the process and the remote process establishes a communication end point called Queue Pairs (QP). The remote memory access is fully operated by hardware without involing software network layers, OS kernel, or CPU of the remote machine. QP are lossless in normal case, but packet losses may happen during machine or software (e.g., the server program) restarts.

One-side RDMA operations can totally write from one machine's memory to a remote machine's memory directly. However, for one-sided operations, the remote machine's memory is not aware of the write either, so a careful protocol design is necessary when one-sided operations are used.

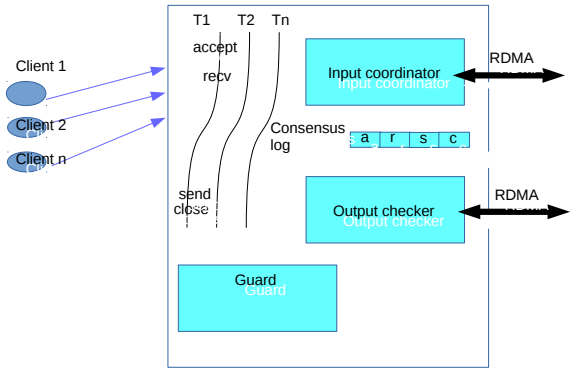

Figure 1: *The* FALCON *Architecture.* FALCON components are shaded (and in green).

3 FALCON Overview

This section presents an overview of FALCON's architecture with its deployment model and key components (§3.1), and it gives an example to show how it works (§3.2).

3.1 Architecture

To replicate a server program, FALCON is deployed in a datacenter, with a set of three or five replicas connecting with RDMA architecture, InfiniBand. Client programs located in LAN or WAN network and send client requests to the leader machine as if only one server program is running. If clients send requests to the wrong machines, FALCON's backup machines deny the requests and reply the leader's IP.

Figure 1 shows FALCON's architecture within one replica machine. To run a server's x86 binary in a replica, FALCON simply runs this command: "LD_PRELOAD=falcon.so ./server". FALCON invokes a server program's socket calls (e.g., `recv()`) and involve four key components: the input coordination protocol (for short, *coordinator*), the output checking protocol (the *checker*), the in-memory consensus log (the *log*), the guard process that handles checkpointing and recovering the process state and file system state of the server program (the *guard*).

The coordinator is invoke when a server's thread calls a socket call to accept/close a client socket connection or to receive input bytes from the connection. On the leader replica, FALCON executes the actual Libc socket call, extracts the returned contents of this call (e.g., the returned buffer from `recv()`), and invokes the coordinator for a new consensus request on this socket call with the contents. Sending consensus request is fast because the leader coordinator directly writes a socket call request to remote backup replicas' log with RDMA. In this request, in order tell the backups what socket calls they should execute, this request also carries the latest request viewstamp that has reached consensus (i.e., the latest committed request).

On a backup replica, FALCON's coordinator check its own log to receive new consensus requests, and it directly sends an Yes or No ACK by writing the remote leader's same struct for this socket call with RDMA. The backup coordinator then forwards all the requests up to the latest committed socket call to its local server process. FALCON does not need to intercept a server's socket calls in backups because these servers just follow the leader's consensus requests. To respond consensus requests rapidly, each backup coordinator spawns a dedicated thread. This thread runs a busy loop to pull consensus requests from the consensus log. This high-perfomance thread runs in a spare dedicated CPU core and elinimates context switches, unlike traditional TCP/IP that block threads socket calls to wait for requests.

A consensus log appending operation is invoked whenever a leader's server executes a socket call (except `send()` calls, handled by output checker). The leader coordinator just appends socket calls to the log and backups follow socket calls in this log.

Output checker is occasionally invoked as the leader's server program executes `send()` operations. For every 4K bytes (MTU size) of output in each connection, the checker unions the current hash with these bytes and computes a new hash. Then, for every *Tcheck* hash generations, the checker invokes a consensus request on this hash value. This output consensus is the same as the coordinator's input consensus except that backups' output checkers send ACKs with their own hash values with the same index, whenever the hash values are ready (some backups may run slowly). When a majority of hashes are back, the leader's output checker compares these values and then makes an effort to roll back divergent replicas to a previous checkpoint before the last matching hash. The leader's checker rolls back other replicas by sending roll-back requests to the local guard.

The leader's guard accepts roll-back requests from the output checker and forwards the requests to the corresponding guard on another, divergent repilca. The divergent replca's guard then rolls back the server program to a previous checkpoint before the last matching one, and then invokes the local input coordinator to re-forward the requests in stable storage to the server. There is no a second output hash comparisons between backups and leader until the leader invokes a new output checking consensus request next time.

```
1 : void main(int argc, char *argv[]) {
2 :   int nfds = 0;
3 :   struct pollfd fds[MAX];
4 :   int sock = socket(...);
5 :   fds[nfds].fd = sock;
6 :   fds[nfds].events = POLLIN;
7 :   nfds++;
8 :   ...; // Call bind() and listen().
9 :   while (poll(fds, nfds, 0) > 0) {
10:     for (int i=0; i<MAX; i++) {
11:       if(fds[i].revents == POLLIN) {
12:         if (fds[i].fd == sock) {
13:           fds[nfds].fd = accept(sock, ...);
14:           fds[nfds].events = POLLIN;
15:           nfds++;
16:         } else {
17:           char in[1024], out[1024];
18:           recv(fds[i].fd, in, ...);
19:           process_request(in, out);
20:           send(fds[i].fd, out, ...);
21:           close(fds[i].fd);
22:           fds[i].fd = fds[i].events = 0;
23:           nfds--;
24:         }
25:       }
26:     }
27:   }
28: }
```

Figure 2: ***A server example.***

```
1: void main(argc, char *argv) {
2:   ...; // Get server IP:port from argv[ ].
3:   int sock = socket(...);
4:   connect(sock, ...); // Connect to IP:port.
5:   send(sock, ...); // Send a http request.
6:   recv(sock, ...); // Wait for server's response.
7:   close(sock);
8: }
```

Figure 3: ***A client example.***

3.2 Example

Figure 2 shows a simplified server code based on the Redis key-value store, and Figure 3 shows a client cocde. The server accepts a new connection from a client, receives one input request, process the request, and then sends a reply. Suppose the client sends a "SET a b" request.

Once the client calls its `connect()`, the leader's server program calls the `accept()` call intercepted by FALCON. The leader's input coordinator then invokes consensus on building a new connection by writing a struct for `accept()` to remote backups with RDMA. Once a majority of backups agree, the leader machine returns from the `accept()` call and continues to run.

When a client program calls its `send()`, leader's server traps into FALCON's `recv()` function call, which a invokes consensus on this new input "SET a b". If a consensus is made by a majority of replicas, the leader directly returns from `recv()` and processes the request, and each replica's dedicated thread sets up a new connection to the local server because the `recv()` consensus request notifies the backups that the last call, `accept()`, has reached consensus.

```
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 4: FALCON's *log entry for each socket call.*

When a request is processed and the server is about to send reply to the client, FALCON traps into the `send()` call, computes historical hash value on inputs when 4K bytes (MTU size). Since this request is short, no output check will be invoked at this time. All backups return consensus reply to the leader with their own hash values if this value is available on their replica. Similarly, backups do the earlier socket call: the backups' follower forwards "SET a b" request to the local server program.

4 Input Coordination 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 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 4 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 [27], 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 [27] 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 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 out-

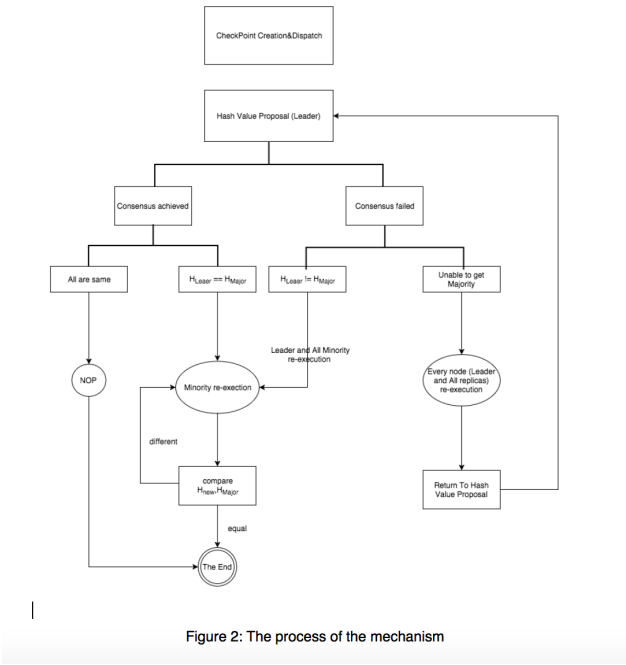

Figure 5: ***Workflow on Handling Network Output Divergence.***

puts on random functions (e.g., a scientific similator). Even so, we can leverage prior approaches to hook these random functions [27, 18] 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.

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 [18, 27] 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 [18] 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's Applications in Broad Areas

FALCON's a fast, general SMR service can be applied broadly, and here we elaborate three. First, FALCON's RDMA-accelerated PAXOS protocol and its implementation could be an effective template for many other distributed protocols in datacenters. For instance, an immediate extension of FALCON is to make it tolerate byzantine faults. Another promising direction is three-phase commit (3PC): 3PC is often blamed by its intolerance on network partitions and asynchronour communications, and its high latency caused by the three round-trips. Fortunately, within the RDMA-enabled datacenter context, people may leverage FALCON's techniques and experience to build a significantly faster and more reliable 3PC protocols.

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 10 widely used or studied server programs, including 4 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 impor-

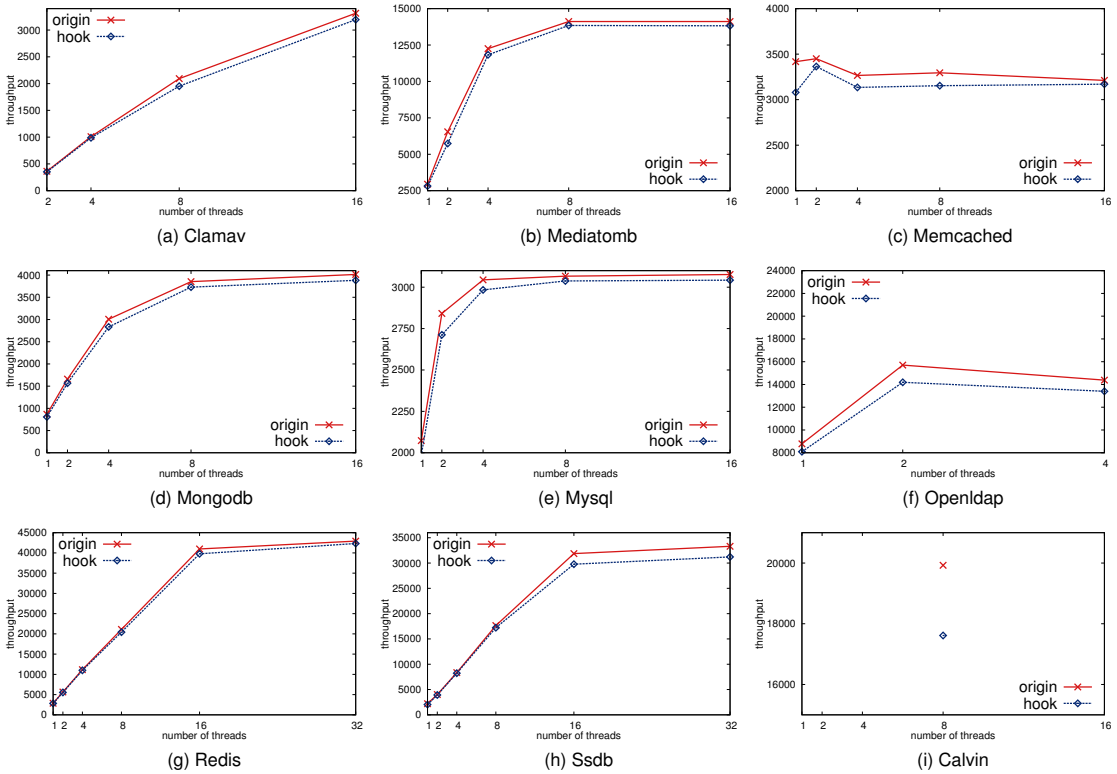

Figure 6: FALCON *throughput compared to the unreplicated execution.*

tant 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.

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 10 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. 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 an average througput degrade by 4.16% (note that in Figure 6, the Y-axises of most programs start from a large number). As the number of threads increases, FALCON still scales almost as well as the unreplicated executions. Figure 6 shows FALCON's response time, a 4.28% overhead compared to the unrepli-

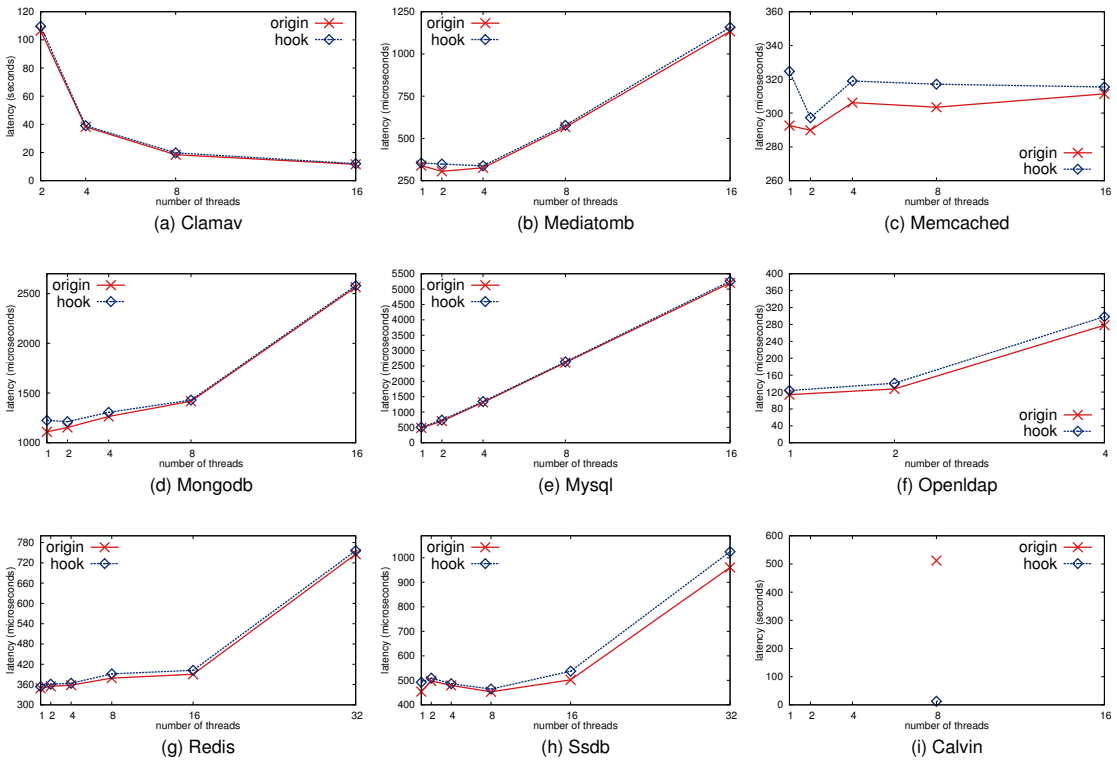

Figure 7: FALCON *response time (latency) compared to the unreplicated execution.*

cated 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" column in Table 2) and a quorum waiting phase (the "quorum" 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 column, overall, a FALCON input consensus took only 9.9 μs (Redis) to 39.6 μs (`MongoDB`).

This consensus latency mainly depends on the number of bytes received in a socket call (`MongoDB` has the largest received bytes), and it also depends on the server program's own IO (Redis does little IO). FALCON's small consensus latency makes FALCON achieve reasonable throughputs in Figure 6 and response times Figure 7.

7.3 Comparison with Traditional SMR systems

TBD.

7.4 Scalability on Replica Group Size

TBD.

7.5 Recovering from Output Divergence

TBD.

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 [21, 34]. As a common practice, SMR uses PAXOS [22, 20, 36] and its popular engineering approaches [9, 27] as the consensus protocol to ensure that all replicas see the same input re-

quest sequence. Since consensus protocols are the core of SMR, a variety of study improve different aspects of consensus protocols, including performance [28, 23] and understandability [30]. Although FALCON's current implementation takes a popular engineering approach [27] 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 [**?**, **?**, **?**], 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, 24, 2] that aim to reduce the number of possible thread interleavings for program all inputs, and (2) deterministic multi-threading systems (or DMT) [7, 12, 29, 3, 5, 15, 4] that aim to reduce the

Program	Benchmark	Benchmark workload description
`ClamAV`	Self	Scan files in `/usr/lib` directory
`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 benchnmark.

Program	# calls	input	SSD	quorum	diff
`ClamAV`	30	42.0	5.1 μs	6.1 μs	F
`MediaTomb`	3,000	140.0	4.7 μs	5.8 μs	F?
`Memcached`	10,016	38.0	4.9 μs	6.9 μs	F?
`MongoDB`	25,665	492.4	19.1 μs	20.4 μs	F?
`MySQL`	13,111	26.0	5.0 μs	15.7 μs	W?
OpenLDAP	8,040	27.3	5.7 μs	7.1 μs	T
Redis	10,016	107.0	3.6 μs	6.3 μs	T
`SSDB`	9,916	47.0	3.7 μs	10.9 μs	F*
`Calvin`	TBD	TBD	TBD	TBD	TBD

Table 2: FALCON *micro events.* 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" means the average latency on storing these calls to stable storage; "quorum" means the average latency on waiting quorum for these calls; and "diff" means whether the output checker found output divergence.

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 [39, 33, 13, 25, 26, 40], diagnosis [35, 32, 31], and correction [17, 37, 38, 16]. 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 [11].

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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.