RESEARCH-ARTICLE

Efficient Fault Tolerance for Stateful Serverless Computing with Asymmetric Logging

SHENG QI, Peking University, Beijing, China

HAOYU FENG, Peking University, Beijing, China

XUANZHE LIU, Peking University, Beijing, China

XIN JIN, Peking University, Beijing, China

Published: 09 June 2025
Online AM: 28 March 2025
Accepted: 17 January 2025
Revised: 29 October 2024
Received: 19 May 2024

Efficient Fault Tolerance for Stateful Serverless Computing with Asymmetric Logging

SHENG QI, Computer Science, Peking University, Beijing, China
HAOYU FENG, Computer Science, Peking University, Beijing, China
XUANZHE LIU, Computer Science, Peking University, Beijing, China
XIN JIN, Computer Science, Peking University, Beijing, China

Serverless computing separates function execution from state management. Simple retry-based fault tolerance might corrupt the shared state with duplicate updates. Existing solutions employ log-based fault tolerance to achieve exactly-once semantics, where every single read or write to the external state is associated

This work is an extended version of the paper that appeared in SOSP 2023 [95]. The additional content includes:

(1) A full proof of idempotence for the Halfmoon-read protocol (Section 4.1.2)
(2) A full proof of idempotence for the Halfmoon-write protocol (Section 4.2.2)
(3) A full proof of consistency for the Halfmoon-read and Halfmoon-write protocol (Section 4.4)
(4) A full proof of idempotence and consistency for the hybrid protocol that combines Halfmoon-read and Halfmoon-write at object granularity (Section 4.5)
(5) A complete description of the switching procedure between the Halfmoon-read and Halfmoon-write protocols (Section 4.8), including:
  (a) A locking primitive to atomically commit single- and multi-version writes, and the implementation of the transitional read and write operations based on the locking primitive (Section 4.8.2)
  (b) A discussion on the idempotence and consistency of the switching procedure (Section 4.8.3)
  (c) A mechanism to ensure that the external state is up-to-date after the switching (Section 4.8.4).
(6) An extension to the Halfmoon-write protocol that enforces strict ordering of consecutive writes (Section 6.1).
(7) An extension to the the Halfmoon-read and Halfmoon-write protocols using a partially ordered log (Section 6.2).
(8) An extended evaluation of Halfmoon, including:
  (a) An evaluation of Halfmoon's read and write primitives under various request rates (Section 7.1).
  (b) An evaluation of Halfmoon's end-to-end performance in the presence of function failures and retries (Section 7.5).
  (c) An evaluation of Halfmoon's end-to-end performance when overlapping logging with program execution (Section 7.6).
  (d) An evaluation of the extended Halfmoon-write protocol that enforces strict ordering of consecutive writes (Section 7.7).

This work was supported in part by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant numbers 62325201 and 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Sheng Qi, Haoyu Feng, Xuanzhe Liu, and Xin Jin are affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, School of Computer Science at Peking University, and Center for Data Space Technology and System at Peking University.
Authors' Contact Information: Sheng Qi, Computer Science, Peking University, Beijing, China; e-mail: shengqi2018@pku.edu.cn; Haoyu Feng, Computer Science, Peking University, Beijing, China; e-mail: 2100012967@stu.pku.edu.cn; Xuanzhe Liu, Computer Science, Peking University, Beijing, China; e-mail: xzl@pku.edu.cn; Xin Jin (Corresponding author), Computer Science, Peking University, Beijing, China; e-mail: xinjinpku@pku.edu.cn.

ACM 0734-2071/2025/06-ART3
https://doi.org/10.1145/3725985

with a log for deterministic replay. However, logging is not a free lunch, which introduces considerable overhead to stateful serverless applications.

We present Halfmoon, a serverless runtime system for fault-tolerant stateful serverless computing. Our key insight is that it is unnecessary to symmetrically log *both* reads and writes. Instead, it suffices to log *either* reads *or* writes, i.e., asymmetrically. We design two logging protocols that enforce exactly-once semantics while providing log-free reads and writes, which are suitable for read- and write-intensive workloads, respectively. We theoretically prove that the two protocols are *log-optimal*, i.e., no other protocols can achieve lower logging overhead than our protocols. We provide a criterion for choosing the right protocol for a given workload, and a pauseless switching mechanism to switch protocols for dynamic workloads. We implement a prototype of Halfmoon. Experiments show that Halfmoon achieves 20%–40% lower latency and 1.5–4.0× lower logging overhead than the state-of-the-art solution Boki.

CCS Concepts: • **Information systems** → **Information storage systems**; • **Computer systems organization** → **Reliability**; **Availability;**

Additional Key Words and Phrases: Serverless computing, FaaS, logging, exactly-once semantics

ACM Reference Format:
Sheng Qi, Haoyu Feng, Xuanzhe Liu, and Xin Jin. 2025. Efficient Fault Tolerance for Stateful Serverless Computing with Asymmetric Logging. *ACM Trans. Comput. Syst.* 43, 1-2, Article 3 (June 2025), 43 pages. https://doi.org/10.1145/3725985

1 Introduction

Recent years have seen an increasing popularity of serverless computing [18, 35, 36, 39, 40, 57, 101, 104, 113, 127] in building cloud applications [54, 78, 87, 98, 106, 119, 125]. It features the **Function-as-a-Service (FaaS)** paradigm [99, 114, 130], where developers break down an application into a set of functions and their dependencies [79], and the cloud automates the deployment. FaaS enjoys elastic scaling and pay-per-use billing, which dramatically reduces resource management overhead.

Serverless platforms enable autoscaling by disaggregating compute and storage [45, 70]. Function-local state is not guaranteed to persist across invocations due to load balancing and elastic scaling of resources. To share state across multiple functions, applications typically rely on external storage for state management [12].

Extracting the state from **stateful serverless functions** (**SSFs**) [53, 123] brings challenges to application-level consistency in the presence of failures. While SSFs are decomposed into stateless functions and the external state, achieving fault tolerance of SSFs is not as simple as achieving fault tolerance for each component individually. Specifically, the fault tolerance of stateless functions can be achieved by retrying crashed functions, and that of the external state can be achieved by using a fault-tolerant external storage service. However, naively combining the two introduces anomalies upon failures. Consider an SSF that writes to the external state and then crashes. Retrying this function would duplicate the write that has already been applied.

Serverless runtimes should avoid such anomalies with exactly-once semantics [100, 123]. That is, no matter how many times an SSF crashes and gets re-executed, the effect on the external state should be equivalent to that produced by running the SSF exactly once, without crashing.

Log-based fault tolerance is a common approach to realizing exactly-once semantics [27, 100]. The idea is to enhance the retry-based at-least-once semantics with idempotence, i.e., at-most-once semantics. To achieve this, existing solutions associate every read or write to the external state with a log record. During re-execution, the SSF replays the log, recovering read results and skipping completed writes. Beldi proposes to atomically perform writing and logging in the external storage [123]. The state-of-the-art solution Boki decouples the log to a more efficient logging layer [53].

However, logging is not a free lunch. Beldi reports that reads and writes with logging are 2-4× more expensive than their raw counterparts. Even for Boki's optimized implementation, logging still accounts for 30%–50% overhead compared to raw operations (Section 2). Thus the primary goal of this article is to provide idempotence while minimizing logging overhead.

We present Halfmoon, a serverless runtime system for fault-tolerant SSFs. Halfmoon provides two logging protocols that enforce exactly-once semantics while providing log-free reads and writes, respectively. Our intuition is that there is no need to symmetrically log *both* reads and writes. Instead, it suffices to log *either* reads *or* writes, i.e., asymmetrically. Consider the stream of events that take place in the system. Our key insight is that reads and writes are *parameterized* by their timestamps in the stream, and idempotence boils down to the stability of timestamps [51]. An idempotent write should always be applied at the same point in the stream, i.e., at the same timestamp. Similarly, an idempotent read should always seek backward from its timestamp and observe the latest preceding write in the stream.

The key challenge of Halfmoon's asymmetric design is persisting events without logging. While a logged event (and its timestamp) is persistent on its own, an unlogged event must be recovered from other logged events after failure. The problem is that the assignment of timestamps is inherently non-deterministic. Halfmoon addresses this problem by leveraging the fact that many serverless applications do not require real-time consistency [102, 103, 115]. Instead of assigning the non-recoverable real time to log-free reads or writes, we generate their timestamps based on those of previous logged operations, in a *deterministic* and *recoverable* fashion. This allows us to eliminate the logging overhead for one type of operation, and rely on the log records of the other type to achieve both persistence and idempotence. We discuss Halfmoon's consistency guarantees in Section 4.4.

We prove that the two logging protocols are *log-optimal* for exactly-once semantics, i.e., no other protocols can achieve exactly-once semantics with lower worst-case logging overhead than our protocols. Intuitively, the two protocols are suitable for read- and write-intensive workloads, respectively. We provide theoretical and empirical analysis of Halfmoon under different read/write intensities, and propose a criterion for choosing the right protocol for a given workload.

Halfmoon also supports dynamic workloads that change their read/write intensity over time. We design a switching procedure that allows the runtime to change between Halfmoon's log-free read protocol and log-free write protocol. The switching is *pauseless*, i.e., the system remains operational and fault-tolerant during this process.

In summary, we make the following contributions.

- We design two logging protocols that enable exactly-once log-free reads and writes for SSFs, respectively.
- We theoretically prove that our protocols are log-optimal. Therefore, Halfmoon pushes the overhead of log-based fault tolerance to its lower bound.
- We provide a criterion for choosing the right protocol for a given workload, and a pauseless switching procedure to switch protocols for dynamic workloads.
- We implement a prototype of Halfmoon. Our experiments show that Halfmoon achieves 20%–40% lower latency and 1.5–4.0× lower logging overhead than the state-of-the-art solution Boki.

2 Motivation

Problem: logging overhead. To achieve exactly-once semantics for SSFs, existing solutions [53, 123] require logging for *both* reads and writes to the external state. Note that reads should be idempotent because writes and the branching of SSF logic may arbitrarily depend on read results.

Table 1. Latency of Log, Read, and Write Operations in Boki

	Log	Read	Write
median	1.18ms	1.88ms	2.47ms
99%-tile	1.91ms	4.60ms	5.86ms

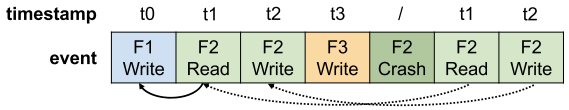


Fig. 1. Parameterizing reads and writes with timestamps.

We refer to these fault-tolerant logging protocols as *symmetric*. Boki [53] is the state-of-the-art symmetric approach that specifically optimizes its logging implementation. Nevertheless, logging still incurs substantial overhead. We benchmark Boki's logging latency under the setup of Section 7, using Amazon DynamoDB as the external storage [37]. Table 1 shows that logging accounts for 63% (48%) overhead at the median and 42% (33%) at the 99%-tile for reads (writes). This motivates us to design minimally fault-tolerant protocols for SSFs that perform logging only if necessary.

Opportunity: parameterizing reads and writes. The event stream is the key enabler of Halfmoon's design. The ordering of events in the stream can be obtained through sequencers [20, 32, 53, 72], synchronized clocks [29, 61, 75], or **state machine replication** (**SMR**) [15, 30, 38, 63]. Without loss of generality, we assume for now that each operation is associated with a timestamp and the event stream follows the timestamp order. We further discuss the availability of the event stream in Section 8.

Our key insight is that reads and writes are *parameterized* by their timestamps in the event stream. To achieve idempotence, a read should consistently seek backward from the same timestamp. Figure 1 shows a real-time interleaving of SSFs and the event timestamps. The first time F2 executes read at t_1, it sees the latest write from F1 at t_0. During re-execution, instead of seeing F3's write at t_3, it should also seek backward from t_1 and recover the previous result, i.e., the value written at t_0. Similarly, for a write to be idempotent, it should always take effect at the same point in the stream regardless of re-execution. As shown in Figure 1, when F2 recovers from failure, it should avoid overwriting F3's write because F2's write has been parameterized at t_2, i.e., it should take effect before F3's write.

By parameterizing reads and writes, idempotence boils down to the stability of read and write timestamps [51]. This motivates us to rethink the necessity of existing logging protocols. Our intuition is that there is no need to symmetrically log *both* reads and writes. Instead, by leveraging the inherent dependencies between events, it suffices to log *either* reads *or* writes, i.e., asymmetrically.

Challenge: persistence without logging. Parameterizing reads and writes does not provide fault tolerance on its own. The critical part is to persist the event stream in the first place, such that timestamps and the ordering of events remain stable across failures. This property is implicitly satisfied by symmetric logging protocols, as logged events are persistent by themselves. However, Halfmoon's design goal of being log-free presents a new challenge to persisting the event stream. For an unlogged event, the only way to persist it is to make sure that it can be recovered from other logged events. Consequently, the dependencies of this event must be deterministic. By requiring that SSFs be deterministic [123], the input parameters to reads or writes can be stably inferred from

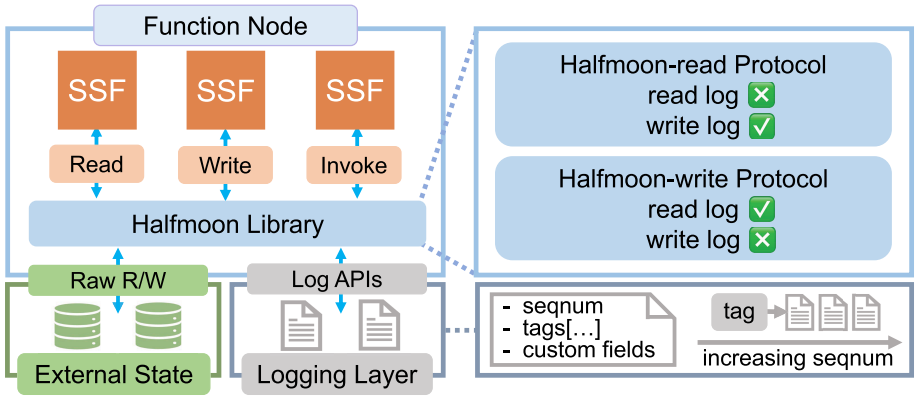


Fig. 2. Halfmoon overview.

function context. However, the assignment of event timestamps is inherently non-deterministic, which must be logged to rule out the uncertainty.

At first glance, this seems to be contradictory to our design goal. To address this problem, Halfmoon leverages the fact that many serverless applications function well under less stringent guarantees than real-time consistency, i.e., linearizability. For example, Cloudburst [103] provides *repeatable reads* or *causal consistency* for SSFs; AFT [102] provides *read atomicity*. Therefore, the read/write timestamps do not have to reflect the real time, which is non-deterministic and unrecoverable without logging. Instead, we can deterministically generate *logical* timestamps for log-free operations based on the SSF context, similar to the way we infer the input parameters for reads or writes. In other words, it is possible to infer all events from a skeleton of the event stream. Only the skeleton needs to be persisted, thereby saving the logging overhead for the rest.

3 Halfmoon Overview

Figure 2 shows the overall architecture of Halfmoon. SSFs interact with the external state through Halfmoon's client library. The library exposes similar APIs as existing solutions [53, 123], including data operations such as read/write, and control flow operations such as invoke to enable stateful workflows. The APIs have the same signature as their raw counterparts, but automatically perform logging behind the scene to ensure idempotence. To achieve exactly-once semantics, Halfmoon relies on the serverless runtime to detect and re-execute crashed SSFs. This feature is widely supported among existing platforms [1, 8, 100, 123].

Halfmoon applies log-based fault tolerance with novel log-optimal protocols. The Halfmoon-read protocol is log-free on reads, and the Halfmoon-write protocol is log-free on writes. In line with the state-of-the-art solution Boki [53], Halfmoon decouples the log from the external state into a separate logging layer. The logging layer implements the shared log abstraction [21, 32, 53], which enforces a global total order of log records and serves as the event stream of Halfmoon. We note that Halfmoon is not tied to Boki's logging layer (Section 8). Our prototype uses Boki because it specifically optimizes logging for stateful serverless computing.

Figure 3 lists Halfmoon's log APIs. `logAppend` appends to the log and assigns a globally monotonically increasing sequence number (seqnum) to the log record. Each log record has a number of `tags` as specified in `logAppend`. The main log is logically divided into sub-streams where log records have a common tag, and a record may appear in several sub-streams. Because the order of records is determined by their seqnums, the order within each sub-stream is consistent with that of the main log. Sub-streams reduce log replay overhead by enabling selective reads,

```
# Return the sequence number of the log record
def logAppend(tags, record) -> seqnum
# Read the previous or next log record
# whose seqnum<=`max_seqnum` or >=`min_seqnum`
def logReadPrev(tag, max_seqnum) -> LogRecord
def logReadNext(tag, min_seqnum) -> LogRecord
# Delete log records up to `seqnum`
def logTrim(tag, seqnum)
# Conditional log append (Section 5.1)
def logCondAppend(tags, record, condTag, condPos)
    -> (seqnum, error)
```

Fig. 3. The log APIs in Halfmoon.

a common approach in shared log systems [22, 53, 111]. **logReadPrev** (**logReadNext**) seeks backward (forward) on a sub-stream specified by the **tag** parameter. **logTrim** garbage collects a sub-stream. Besides the APIs of prior work [53], we introduce **logCondAppend** to resolve conflicts between concurrent SSF instances (Section 5). Because all log APIs target specific sub-streams, we abbreviate sub-streams as streams for the rest of this article.

4 Halfmoon Design

This section presents the design of the Halfmoon-read and the Halfmoon-write protocol. We start by clarifying the concepts and assumptions.

Race conditions. Let *instances* denote executions of an SSF corresponding to a *single* function invocation. There can be multiple instances because an instance might crash and the serverless runtime would launch a new one to serve the SSF invocation. Without loss of generality, instances should be assigned a common identifier (tag) so that they may refer to the same log stream containing the SSF's execution history. We use *instanceID* to denote this common identifier. The lifetime of instances may be *disjoint*, where a second instance starts after the first one has crashed; or *overlapping*, where the first one is assumed to have crashed by the runtime, but is still actually alive. This could happen when the first instance fails to make progress in time, due to network errors or unexpected slow execution. We use *peers* specifically to denote such *concurrent* instances for an SSF invocation.

Consequently, there are two race conditions against the exactly-once semantics. First, an instance may race with another disjoint instance, at the risk of repeating a completed step. Second, an instance may also race with its peers, both attempting to execute the same step. For the sake of presentation, this section focuses on addressing the first race condition. We extend the protocols to handle the second one in Section 5. For now, we assume that the logging layer resolves conflicts between peers such that **logAppend** is idempotent; the formal proof is deferred to Section 5.

Time-related concepts. A call to the **logAppend** API returns a monotonically increasing *seqnum* from the logging layer (Figure 3). Within each SSF, we use a variable *cursorTS* to record the function-local seqnum that advances after each logging operation. As per Section 2, *timestamp* is a general concept that parameterizes the position of the associated event in the event stream. Note that depending on the logging protocol, timestamps can be based on seqnums (i.e., logical) or the real time.

Transactions. In line with previous works, we assume that SSFs are non-transactional by default [53]; to execute several steps atomically, SSFs should explicitly use transactions. Halfmoon can reuse existing transactional APIs, and focuses on optimizing the logging overhead for normal operations.

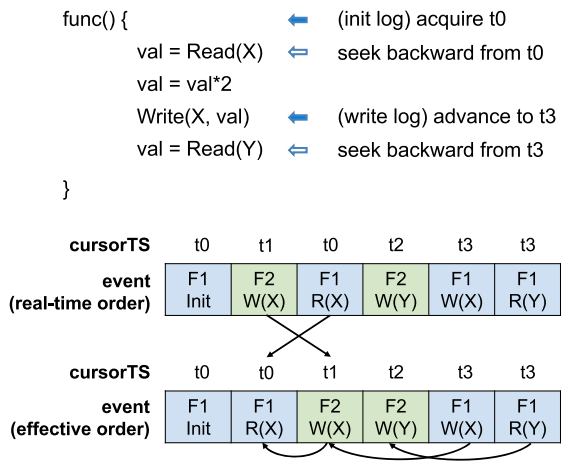


Fig. 4. Example of the Halfmoon-read protocol. F1 runs the pseudocode. F2 is another SSF. The top and bottom timeline show the real-time order and the effective order of events, respectively. The arrows below the bottom timeline show the dependencies between operations on object X and Y.

4.1 Halfmoon-Read: The Log-Free Read Protocol

Overview. Under Halfmoon-Read, all reads are log-free and only writes perform logging. The persistent part of the event stream consists only of writes. Writes achieve idempotence by checking the write logs in advance. For reads, we take two steps to ensure idempotence. First, we assign timestamps to reads in a deterministic manner. Second, we use the timestamps to map reads to existing writes in the write log. In other words, reads are inserted into the event stream given their timestamps, and are idempotent because their positions are deterministic.

In step one, we must infer the read timestamps from the SSF context. Our solution is to use cursorTS, i.e., the seqnum of the latest logged operation in the SSF; then the fault tolerance of the logging layer guarantees that read timestamps are deterministic.

In step two, we must ensure that writes are traceable. Our solution is to use multi-versioning to manage the external state. Each write creates a new version of the object and registers the version number in the logging layer; a read locates a specific object version by querying the write log (`logReadPrev`). Note that the version numbers are unordered by themselves; the write log defines the order. Therefore the external storage only needs to support plain key-value APIs; the version numbers serve as pointers to the actual object. Because the position of a write in the event stream is determined by the seqnum of the associated write log record, the write timestamp is set to that seqnum accordingly.

Example. Figure 4 shows an example of the Halfmoon-read protocol. F1's initial cursorTS is t_0. When reading object X, it does not see F2's write at t_1 because the read uses an earlier timestamp. However, when reading object Y, it sees F2's write at t_2 because it has advanced its cursorTS to t_3 after its previous write. The effective order of events follows the order of the seqnum-based logical timestamps. We show in Section 4.4 that Halfmoon-read provides ***sequential consistency (SC)*** [66].

4.1.1 *The Halfmoon-Read Primitives.*

Init. Figure 5 shows the pseudocode of the Halfmoon-read protocol. At the start of execution, the SSF appends an *init* log record, and uses the seqnum of this record as the initial cursorTS (line 7).

```
1  def Init(env, input):
2      # retrieve all log records of the SSF
3      env.stepLogs = getStepLogs(env.ID)
4      if env.stepLogs[0] is not None:
5          env.cursorTS = env.stepLogs[0]["seqnum"]
6      else:
7          env.cursorTS = logAppend([env.ID], LogRecord{
8              "step": 0, "op": "init",
9              "data": input,
10         })
11     env.step = 0
12
13 def Write(env, key, value):
14     # check if write can be skipped
15     env.step += 1
16     if env.stepLogs[env.step] is not None:
17         env.cursorTS = env.stepLogs[env.step]["seqnum"]
18         return
19     # deterministically generate version number
20     vNum = getVersionNumber(env)
21     DBWrite(key, value, version=vNum)
22     env.cursorTS = logAppend([env.ID, key], LogRecord{
23         "step": env.step, "op": "write",
24         "version": vNum,
25     })
26
27 def Read(env, key):
28     writeLog = logReadPrev(key, env.cursorTS)
29     return DBRead(key, version=writeLog["version"])
30
31 def Invoke(env, funcName, input):
32     # check if invoke can be skipped
33     env.step += 1
34     if env.stepLogs[env.step] is not None:
35         env.cursorTS = env.stepLogs[env.step]["seqnum"]
36         return env.stepLogs[env.step]["result"]
37     # deterministically generate ID
38     ID = getUUID(env)
39     # ID is passed into callee's env
40     result = InvokeFunc(ID, funcName, input)
41     env.cursorTS = logAppend([env.ID], LogRecord{
42         "step": env.step, "op": "invoke",
43         "result": result,
44     })
```

Fig. 5. Pseudocode of the Halfmoon-read protocol.

The cursorTS is recovered from the log record, if present (line 5). The SSF also retrieve all records from the log stream tagged by its instanceID (line 3), which contains the execution history of the SSF. We refer to this per-SSF log stream as the step log. The current records in the step log are stored in a local array **`env.stepLogs`**. Later the SSF may check this array for existing log records and skip finished operations.

Write first obtains a version number (line 20), performs multi-version **`DBWrite`**, and finishes by logging the version number (line 22). Specifically, Halfmoon-read first checks if the write log record

exists. If so, the write has been applied. Otherwise, it means that either the write has not yet created a new version of the object, or it has created the new version but crashed before logging.

To be idempotent, the write needs to use a deterministic version number. For example, it can generate the version number by simply concatenating the unique and deterministic InstanceID (`env.ID`) and the current step number (to distinguish different operations in the same SSF). Alternatively, if the version number is to be randomly generated, the SSF should log and check the version number *before* `DBWrite` to transform it into a deterministic operation. Our current prototype adopts the latter approach such that Halfmoon-read logs before and after `DBWrite`. This is because our primary baseline, Boki [53], also logs twice for each write. Our prototype aligns the logging overhead of writes such that our performance gains come solely from eliminating the logging of reads. Alternatively, if the external state exposes its internal logging and ordering of events, then `DBWrite` and logging can be merged into a single step, where it atomically creates a new object version and receives a timestamp.

Read first queries the per-object write log tagged by `key`. It passes the cursorTS into `logReadPrev` to retrieve a particular write log record. The "version" attribute in the record points to the actual object the read should see (line 28). To facilitate this process, we pass two tags into `logAppend` when logging the write, namely the instanceID of the SSF (`env.ID`) and the `key` of the target object, such that the record is visible in two log streams, namely the SSF's step log and the object's write log. Therefore, a write log record serves a dual purpose. First, it checkpoints the progress of the initiating SSF, allowing the write to be skipped during re-execution by checking the step log. Second, it functions as the commit point of the write where it becomes visible to other SSFs in the write log. Because the logging layer assigns monotonically increasing seqnums, a read has full visibility of all writes with smaller logical timestamps, thereby allowing the read to seek backward in the event stream deterministically.

The second purpose requires that the logging be performed after `DBWrite`, as opposed to **write-ahead logging** (**WAL**). The reason is that the logging layer is decoupled from the external state. If the version number is made visible in advance, then reads could obtain version numbers with no matching objects in the external state. In contrast, Halfmoon-read ensures that the exposed object versions are always available.

In spite of being log-free, `Read` still needs to pay one round of `logReadPrev`. This overhead is implementation specific. Because Boki caches log records on function nodes, `logReadPrev` takes 0.12 ms at the median and 0.72 ms at the 99%-tile [53], which is negligible compared to `DBRead` (Table 1). Therefore our prototype of Halfmoon-read provides near-zero overhead for reads. Note that the critical data of a write log record consists only of its seqnum and version number, which can be covered in a few dozen of bytes. Therefore the cache size is not a concern here. Alternatively, if the logging layer is merged with the external state, a read can directly issue a query with a filter on object versions, instead of retrieving the version and the object separately.

Invoke first generates the callee's instanceID (line 38), calls the function, and finishes by logging the result (line 41). In case the log record already exists, the actual invocation can be skipped. The callee's instanceID must be deterministic to ensure idempotence. Similar to the version numbers in `Write`, the SSF can deterministically generate the instanceID from its context, or randomly generate it and perform additional logging and checking to turn it into a deterministic operation. In line with Boki, our prototype adopts the latter approach. Note that by logging the result, the SSF ensures that the cursorTS is monotonic even across invocations. Because each individual SSF is idempotent, by induction the entire workflow is also idempotent.

Remark. Halfmoon-read is primarily designed for key-value stores with read/write interfaces. To perform table-level queries, e.g., scan, join, and aggregation, one should first use `logReadPrev`

to get a list of version numbers for all objects in the table. This list captures a snapshot of the table at a given timestamp. It is necessary because the ordering of individual writes is defined by the write log, and the version numbers are not ordered by themselves. As an optimization, it is possible to cache the database index in the logging layer to reduce the size of the returned list. Alternatively, the table should be read-only to bypass any logging or version lookup. Our prototype implementation of Halfmoon-read does not support queries over *mutable* tables.

As Halfmoon-read requires multi-versioning, an important concern is the garbage collection and the storage overhead. We address this problem in Sections 4.6 and 4.7, respectively.

4.1.2 Idempotence of Halfmoon-Read. We now prove that Halfmoon-read provides idempotence for SSFs regardless of failure and re-execution. We start with the following definition and assumption.

Definition 4.1 (Context). For a particular function instance, let *context* be the set of local variables containing: (1) function input; (2) key-value pairs retrieved so far from `Read` operation; (3) cursorTS.

Assumption 4.2 (Deterministic Execution). *Let $F1$, $F2$ be two instances of an SSF invocation. If $F1$ and $F2$ have identical contexts, then they will be performing the same operation (`Read` or `Write`) with the same arguments in the next step.*

Assumption 4.2 states that SSF execution is deterministic based on its context. In practice, nondeterminism (e.g., calling random number generators or reading the clock) can be made deterministic by logging the results and restoring them during re-execution. For the sake of clarity, we only consider read and write operations using the key-value API, which can be generalized to other mechanisms of accessing or modifying the external state.

We do not consider `Invoke` operations here because the case for nested or asynchronous SSFs can be reduced to a single, sequential SSF. Specifically, the SSF operation sequence (with loops unrolled) can be represented as a **directed acyclic graph** (**DAG**). An asynchronous invocation creates a new branch, running concurrently with the main branch; synchronous invocations stay in the main branch. To show that the entire DAG is idempotent, we only need to prove the case for every sequential path from the root. We now have the following theorem for Halfmoon-read.

Theorem 4.3 (Operation Sequences are Identical Across Instances). *Let $F1$ and $F2$ be two instances of an SSF invocation. Let $\{S_i^1\}_{1\le i\le n}$ and $\{S_i^2\}_{1\le i\le m}$ be the sequence of operations performed by $F1$ and $F2$, respectively. Then $S_i^1 = S_i^2$ for $1 \le i \le \min(n, m)$.*

Proof. We prove the theorem by induction on the length of the common prefix of operations sequences. For the base case, we show that $F1$ and $F2$ have identical contexts after `Init` (Lemma 4.4). Then given Assumption 4.2, the prefix is identical for length one. We then show that `Write` or `Read` in Halfmoon-read always renders the same context given the same arguments (Lemma 4.5 and Lemma 4.6), which extends the induction hypothesis to longer prefixes, completing the proof. □

Lemma 4.4 (Idempotence of Init). *If $F1$ and $F2$ are two instances of an SSF invocation, then $F1$ and $F2$ have identical contexts after `Init`.*

Proof. The fact that $F1$ and $F2$ correspond to the same invocation implies that their inputs are identical. Since the set of key-value pairs retrieved from `Read` is empty for both instances, it remains to show that the initial cursorTSs are identical. Given the definition of `Init` in Section 4.1.1, the SSF either recovers the cursorTS from an existing log record or appends a new one. Because we assume that the logging layer is idempotent (Section 4), in both cases, the cursorTS cannot diverge between $F1$ and $F2$. □

Lemma 4.5 (Idempotence of Write). *`Write` in Halfmoon-read deterministically creates a version of the object in the external storage and deterministically advances the cursorTS in the context.*

Proof. Based on Assumption 4.2, `Write` uses deterministic version number (Section 4.1.1) and value for the object, which implies that each version of the object is immutable once created. Because `logAppend` is idempotent, the seqnum of the write log record is deterministic. Therefore the cursorTS is also deterministic after the write operation. □

Lemma 4.6 (Idempotence of Read). *`Read` in Halfmoon-read deterministically adds a key-value pair to the SSF context.*

Proof. Given its definition in Section 4.1.1, `Read` seeks backward on the write log to find the latest write log record whose seqnum does not exceed the cusorTS. Because the logging layer is totally ordered and issues monotonically increasing seqnums globally, the retrieved write log record is deterministic. It remains to show that the associated version of the object in the external storage is deterministic. Based on Assumption 4.2, `Write` uses deterministic version number (Section 4.1.1) and value for the object, which implies that each version of the object is immutable once created, concluding the proof. □

In conclusion, Theorem 4.3 guarantees that every instance of an SSF invocation performs exactly the same sequence of operations. Combining it with Lemma 4.5 further ensures that the write operations in that sequence are idempotent in terms of modifying the external state. Therefore, Halfmoon-read provides idempotence for SSFs.

4.2 Halfmoon-Write: The Log-Free Write Protocol

Overview. The Halfmoon-read protocol is ideal for read-intensive workloads. We now present the Halfmoon-write protocol that supports log-free writes. Similar to Halfmoon-read, the idea is to assign deterministic timestamps to log-free operations, but with roles reversed. Under Halfmoon-write, all reads are logged, while writes use the cursorTS. Because reads directly log the real-time data they have seen from the external state, they are idempotent on their own. Moreover, there is no need to use multi-versioning to keep the history of writes. Instead, writes in Halfmoon-write perform *conditional* updates to the *single-version* external state. Specifically, a write deterministically generates a version number based on the cursorTS, and updates the object only if the stored version number is smaller. The monotonicity of version numbers and the determinism of the cursorTS ensures that a write is always applied at the same point in the event stream, thereby achieving idempotence.

For writes, version numbers serve as their logical timestamps. Read timestamps, in contrast, are based on real time because reads always target the latest data. Note that the read timestamp is *implicit* and unknown to Halfmoon; SSFs have no need to explicitly parameterize reads with timestamps given the already *materialized* read log records. We define the read timestamps only to help understand the ordering of events in Halfmoon-write.

Example. Figure 6 shows an example of the Halfmoon-write protocol. F1 initializes its cursorTS to t_0, and issues Write(X) using t_0 as the version number. At this point, F2 has already applied Write(X) with version number t_1. Consequently, F1 does *not* overwrite F2's Write(X) because its version number is smaller. The effect on the external state is equivalent to a virtual interleaving where F1's Write(X) does "happen" before F2's Write(X), which adheres to the definition of exactly-once semantics. In contrast, F1's later Write(Z) overwrites F2's Write(Z), in accordance with the real-time order. This is because F1 has advanced its cursorTS to t_2 after reading the latest value of Y.

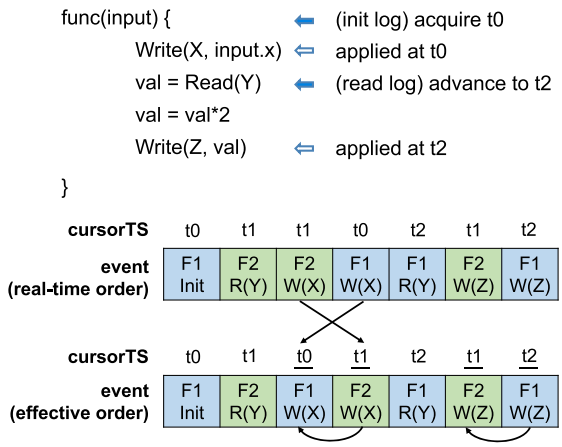


Fig. 6. Example of the Halfmoon-write protocol. F1 runs the pseudocode. F2 is another SSF. The top and bottom timeline show the real-time order and the effective order of events, respectively. The arrows below the bottom timeline show the dependencies between operations on object X and Z.

Note that there is a major difference between Halfmoon-read and Halfmoon-write in terms of the ordering of events. In Halfmoon-read, the effective order of events follows the order of logical timestamps. In Halfmoon-write, logical timestamps (version numbers) are only relevant for writes, while reads are based on real time.[1] Consequently, the effective order under Halfmoon-write combines the real-time and logical-timestamp order. We derive this order through the following steps. First, we order all events by real time. Second, for write events only, we reorder them according to their version numbers (Section 4.4).

For example, Figure 6 underlines the version numbers of writes. F1's Write(X) with version t_0 is ordered immediately before F2's Write(X) with t_1, but still *after* F2's Read(Y). Formally, we show in Section 4.4 that the ordering under Halfmoon-write enforces a sequential history for each SSF except that consecutive log-free writes to different objects may commute. For now, we give an intuitive interpretation of Halfmoon-write's reordering of writes. Because the cursorTS is refreshed after logging each read, a higher cursorTS implies that the SSF has seen "fresher" data, which in turn gives a higher priority to the SSF's writes. In Figure 6, F1's Write(X) is reordered because F2 has seen a fresher value of Y. F1's later Write(Z) is not reordered because F1 is at least as fresh as F2.

4.2.1 The Halfmoon-Write Primitives. Figure 7 shows the pseudocode of the Halfmoon-write protocol. It reuses the `Init` and `Invoke` functions from the Halfmoon-read protocol and differs only in `Read` and `Write`.

Write performs *conditional* update by comparing the version numbers. It is applied to the object only if the stored version number is smaller (line 4) [53]. The version number is structured as a tuple (line 3). The first field is the cursorTS; the second is a counter that records the number of *consecutive* writes. For simplicity, we omit the counter in Figure 6. The counter is incremented upon writes and reset upon reads. The purpose of the second field is to break ties between consecutive writes to the *same* object. A version number V1 is smaller than V2 if V1's cursorTS is smaller, or if they have equal cursorTS but V1's counter is smaller.

[1]Realtime-ness applies to failure-free reads. Since exactly-once semantics ensures that operations appear exactly once in the event stream, regardless of failure and re-execution, we consider only the failure-free case when discussing the ordering of events.

```
def Write(env, key, value):
    env.consecutiveW += 1
    vNum = (env.cursorTS, env.consecutiveW)
    DBWrite(key, cond="VERSION < {vNum}",
        update="VALUE={value}; VERSION={vNum}")

def Read(env, key):
    env.step += 1
    env.consecutiveW = 0
    if env.stepLogs[env.step] is not None:
        env.cursorTS = env.stepLogs[env.step]["seqnum"]
        return env.stepLogs[env.step]["data"]
    value = DBRead(key)
    env.cursorTS = logAppend([env.ID], LogRecord{
        "step": env.step, "op": "read",
        "data": value,
    })
    return value
```

Fig. 7. Pseudocode of the Halfmoon-write protocol.

Read first recovers the previous result from the step log if possible. Otherwise, it reads the current object and logs the result. The cursorTS is updated accordingly. CursorTS is only relevant to subsequent log-free writes. Reads always see the latest state regardless of the cursorTS. Note that there is no per-object read log in Halfmoon-write, as opposed to the write log in Halfmoon-read. This is because read log records are only checked by the initiating SSF, so there is no need to tag them with the object's key to make them publicly visible. Halfmoon-write only maintains the per-SSF step logs.

4.2.2 Idempotence of Halfmoon-Write. We now prove that Halfmoon-write provides idempotence for SSFs regardless of failure and re-execution. Similar to Halfmoon-read, our goal is to show that Theorem 4.3 also applies to Halfmoon-write. Using the same inductive proof, it remains to validate the idempotence of `Write` and `Read` for Halfmoon-write. The difference between Halfmoon-read and Halfmoon-write in terms of write is that the latter does not change the SSF context. Based on Definition 4.1 and Assumption 4.2, we have the following lemmas.

Lemma 4.7 (Idempotence of Write). *Let $\{W_i\}_{i\in\mathbb{N}}$ be the sequence of writes applied to an object in the external storage under Halfmoon-write. Let $src(W_i)$ be a function that maps a write W_i to a particular operation in an SSF instance, i.e., $src(W_i) = (instanceID, stepnumber)$. Then $src(W_i) = src(W_j)$ implies $i = j$.*

Proof. Note that W_i must have succeeded in the conditional update; otherwise, it would be ignored by the external state. We prove the lemma by contradiction. Suppose there exists $src(W_i) = src(W_j)$ and $i < j$. By Assumption 4.2, W_i and W_j must have identical version numbers (essentially the cursorTS). Let it be v. Because $i < j$, the stored version number in the external state should be at least as high as v by the time W_j is applied. This contradicts the conditional update, concluding the proof. □

Lemma 4.8 (Idempotence of Read). ***Read** in Halfmoon-write deterministically adds a key-value pair to the SSF context and deterministically advances the cursorTS in the SSF context.*

Proof. While `DBRead` of different instances may not be the same, the read log record in the logging layer serves as a single source of truth. Because appending to the logging layer is assumed

to be idempotent (Section 4), the read log record is guaranteed to be unique. Therefore the read result and the updated cursorTS must be deterministic. □

In conclusion, provided that the sequence of operations is identical across instances, Lemma 4.7 further ensures that the external effect of each write is idempotent. Consequently, SSFs enjoy idempotence using Halfmoon-write.

4.3 Log Optimality

Given the two logging protocols that enable log-free exactly-once reads or writes, it is worth exploring whether there is still room for further optimization. We now prove that our protocols are log-optimal, i.e., no other log-based fault-tolerant protocol can achieve lower worst-case logging overhead than our protocols. Note that we only consider protocols that provide fault tolerance (i.e., idempotence) for *all* workloads over general mutable shared states. This implies that idempotence cannot be conditioned upon application semantics. For example, if reads in an application always target a read-only object, then there may be a more efficient protocol than Halfmoon that always skips logging these reads; however, this protocol cannot provide idempotence for all scenarios. Nevertheless, Halfmoon can utilize program analysis and verification tools to better understand application semantics and further optimize the logging overhead (Section 8).

We start by formalizing the concepts and assumptions. For simplicity, we focus on accessing a single object in this section.

Definition 4.9 (Write). Let S be the set of valid states. A write operation is a function $w : S \to S$ that transforms the current state s to a new state $w(s)$.

Definition 4.10 (Read). A read is a function $r : S \to V$ that maps the current state s to a value $r(s) \in V$. A read is logged if there is a fault-tolerant record containing the read result.

Without loss of generality, writes transform the current state while reads interpret it. Note that a read may not be defined over some states. For example, in Halfmoon-read, a read with cursorTS t is only valid if the latest seqnum in the logging layer is larger than t. A correct protocol must ensure at any time that any read allowed by the protocol is defined over the state at that time.

Next, we clarify the concept of logged and log-free writes. A write is logged if it is associated with a *standalone* record in fault-tolerant storage. Moreover, the record is either created by the write itself, e.g., in WAL, or it should be publicly visible, e.g., in Halfmoon-read. If there is no such record, the write is defined to be log-free. Consequently, in case a read is logged, we assume that the log record is private to the SSF; otherwise, it would be equivalent to associating a publicly visible log record with the write that created the object, so this write is also considered logged. Formally, we have the following assumption for log-free writes.

Assumption 4.11 (Log-free Writes are Memoryless). *Let w be a log-free write over state s_1 with visible external effect. Then there exist $s_2 \in S$ and read r s.t. $w(s_1) = w(s_2)$ and $r(s_1) \neq r(s_2)$.*

Intuitively, a log-free write directly *overwrites* the object. It does not create a standalone record such that the old state is lost after the write. Consequently, there are multiple distinct old states that end with the same new state after the write. One cannot determine the actual old state from the new state alone. Note that the existence of r and s_2 entails that r is defined over both s_1 and s_2 and thus allowed by the protocol. Moreover, because $r(s_1) \neq r(s_2)$, at least one of them is not equal to $r(w(s_1))$, the read result over the current state. This implies that the write has visible external effects.

Assumption 4.12 (Arbitrary Interleaving). *SSFs are not aware of operations from others ahead of time, such that there can be arbitrary interleaving of reads and writes.*

Based on the definitions and assumptions, we have the following lemma that captures the relationship between the logging of reads and writes.

Lemma 4.13. *If a fault-tolerant logging protocol allows an object to be updated with log-free writes (i.e., with visible external effect), then the protocol cannot be log-free on reads.*

Proof. We prove by contradiction. We construct a counterexample that violates the idempotence of reads. Suppose an SSF performs a log-free read r over external state s_1 and then crashes. During the crash failure, s_1 is modified by a log-free write w. When the SSF re-executes the read, it cannot always recover the previous result based on the modified state. This is because given Assumption 4.11, we can choose s_2 and r s.t. $w(s_1) = w(s_2)$ and $r(s_1) \neq r(s_2)$. Moreover, given Assumption 4.12, the chosen s_2 and r in the counter example is always possible to happen in the worst case. Consequently, it is impossible for the read to choose between $r(s_1)$ and $r(s_2)$ as the previous read result, violating idempotence. □

Theorem 4.14. *In the worst case, a fault-tolerant logging protocol either logs all reads or all writes with visible external effects.*

Proof. As per Lemma 4.13, if there are writes that are log-free and have visible external effects, then the protocol must log all reads so as to avoid the counterexample (which is otherwise possible based on Assumption 4.12). If there are no such writes, then by definition all writes with visible external effects are logged. □

Remark. We only consider writes with visible external effects because a protocol may skip logging some writes by making sure that none can read them. For example, a protocol may handle log-free writes like Halfmoon-write do, occasionally take a snapshot of the object, and serve all reads using the snapshots like Halfmoon-read do. This implies that only the writes captured by the snapshots are visible and logged; the rest are effectively *stateless* operations with no visible effects.

We also note that the concept of logging in Lemma 4.13 and Theorem 4.14 is a general abstraction for installing a standalone record in fault-tolerant storage, independent of implementation. The actual cost of logging, however, is implementation-specific. For example, the write logging in Halfmoon-read can be overlapped with execution, or merged with `DBWrite` if the external state exposes its internal logging and ordering of events. Nonetheless, all writes in Halfmoon-read are considered logged because they are all associated with persistent objects in the multi-version external storage at the least. In contrast, even if an SSF performs logging in `Init` under Halfmoon-read and then issues a log-free read, the preceding logging operation is not considered part of the read because the `Init` call is not even aware of such an operation. As we show in Section 4.4, the logging in `Init` serves to bring the cursorTS up-to-date, and is not necessary for idempotence. The initial cursorTS is only required to be deterministic, which can be inherited from the parent SSF, or if there is no such SSF, be arbitrarily out-of-date.

Halfmoon measures the logging overhead as the number of *abstract* logging operations. Given any particular implementation, a fault-tolerant logging protocol either logs more reads than Halfmoon-read or logs more writes than Halfmoon-write in the worst case. Therefore our protocols identify two minimums in the design space. The actual choice between them depends on the implementation and the workload. We present an analysis in Section 4.7 to quantify this decision.

4.4 Consistency

The primary goal of this article is to explore the minimal logging overhead required for idempotence. In doing so, Halfmoon relaxes the real-time guarantees of linearizability [48] such that events can be persisted without logging. Specifically, Halfmoon-read provides SC, which guarantees that events are totally ordered, and the order adheres to the program order of each process in

the system. The ordering under Halfmoon-write is semantically equivalent to SC as long as consecutive writes to different objects can commute. Formally, we have the following propositions. We include TLA+ verification in our technical report [6].

Proposition 4.15. *Halfmoon-read orders events according to their logical timestamps. This ordering provides SC.*

Proof. For simplicity, we consider only read and write events; nested SSFs can be flattened into a single SSF with auxiliary logging at the function boundaries. It follows that writes are already totally ordered by the logging layer according to the write timestamps, because the creation of a write log record serves as the commit point of the corresponding write where it becomes visible to other SSFs. Because log-free reads are mapped to writes by seeking backward on the write log, the effective interleaving of reads and writes is equivalent to the one produced by inserting the reads in the ordered sequence of writes based on read timestamps.

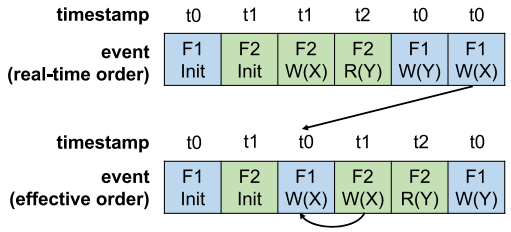
To see that the timestamp order provides SC, note that the cursorTS is monotonically increasing within each SSF. Suppose that there is a violation of program order, then there must be a timestamp inversion in the total order, contradicting the way it is constructed. □

Proposition 4.16. *Halfmoon-write orders events through the following steps. First, all events are ordered by real time. Second, write events are reordered according to their version numbers. Specifically, a write is not reordered if it succeeds in conditional update (Section 4.2); otherwise, it is placed immediately before the next successful write to the same object with a higher version. This total ordering enforces a sequential history for each SSF except that consecutive log-free writes to different objects, i.e., those between two logged events, may commute.*

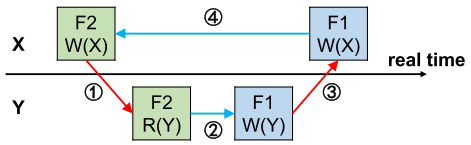
Proof. In line with the proof of Proposition 4.15, we only consider read and write operations. All reads and all successful writes are processed in real-time in the external state, which naturally adheres to the actual interleaving of reads and writes. Writes that get reordered do not change the external state and are not visible to any other SSFs. The placement of these writes in the event stream should maintain the monotonicity of version numbers for each object, as a result of conditional updates, and also respect the data dependencies of reads. Therefore we insert a reordered write immediately before the next successful write to the same object with a higher version.

Taking the sequential history of each SSF from the total ordering, we note that reads and successful writes adhere to the program order because they are ordered by real time. It remains to show that a reordered write w may never go backward across a preceding read r in the SSF. Suppose w is placed immediately before w', the next successful write. Then it suffices to show that w' happens later in real time than r. For w' to succeed in the conditional update, it must have acquired a higher cursorTS than that of w, which equals the one obtained after logging r. Because the logging layer issues monotonically increasing seqnums respecting the monotonicity of the real time, w' indeed happens later than r in real time. Note that by combining cursorTS with a local counter, version numbers are always monotonic within each SSF. Therefore consecutive log-free writes to the *same* object are never reordered. □

Example. Figure 8(a) presents an example to illustrate the total ordering under Halfmoon-write. For simplicity, we consider only the cursorTS instead of the entire tuple for version numbers. F2's Write(X) with version t_1 and F1's Write(Y) with version t_0 succeed in conditional update and are not reordered. F1's Write(X) with version t_0, however, is reordered before F2's Write(X). Consequently, the program order of F1 is changed such Write(X) happens before Write(Y). However, although F1's Write(X) is reordered, it will never go past the preceding logged operation (Init in this case).



(a) Example of the ordering of events under Halfmoon-write.



(b) Enforcing program order among consecutive writes creates a dependency cycle in the example. Red edges (①③) represent the program order; blue edges (②④) represent the data dependencies. Note that the direction of edges indicates *precedence*, unlike the other figures in this paper.

Fig. 8. Example of Halfmoon-write where consecutive log-free writes to different objects may commute.

Ordering of consecutive writes. The example shows that Halfmoon-write may only change the program order of consecutive log-free writes to different objects. Consequently, the ordering of Halfmoon-write is exactly the same as SC when SSFs do not perform consecutive writes to different objects, and is semantically equivalent to SC when such writes can commute. In case the ordering of consecutive writes must be preserved, one can perform extra logging between the writes such that every dependent pair cannot be reordered. The best practice under Halfmoon-write is to make dependencies explicit through invocation or triggers in the SSF workflow, and utilize the logging in the init step of each SSF to prevent reordering. We observe this design pattern in a variety of workloads [3, 4, 10, 11, 33, 53, 123]. We also provide an extension of Halfmoon-write in Section 6 that preserves the ordering of consecutive writes. The extended protocol is log-free on writes in the best case.

Remark. One might wonder why the two protocols are dual to each other in terms of design but differ in the ordering of events. The reason is that writes have external effects while reads do not. Therefore, there is more freedom in placing log-free reads in the event stream than placing log-free writes. For a log-free read under Halfmoon-read, no matter when in real time the read is actually performed, we can always safely insert it in the event stream based on its assigned logical timestamp. In contrast, under Halfmoon-write, the placement of a log-free write not only depends on its version number (logical timestamp), but also on the real-time state of the object. If the conditional update succeeds, then the write must be placed precisely at this point in real time, since it would be immediately visible to reads after that point. This additional constraint leads to the permutation of program order in Figure 8.

Moreover, by advancing the cursorTS in `Init`, our protocols enforce a *real-time* property at the *boundary* of SSFs: if an operation finishes at t in real time, then all SSFs that start after t are guaranteed to see the external effects of that operation. This property is well suited for a variety of serverless applications that use event triggers to invoke downstream tasks [5, 7, 9]. Optionally, an SSF can perform linearizable reads/writes by explicitly advancing the cursorTS beforehand. Similar to `Init`, the SSF appends a special *sync log* to acquire the up-to-date seqnum in the system.

Halfmoon offers the flexibility for users to enforce linearizability if necessary, or achieve minimal logging overhead when our consistency guarantees suffice.

4.5 The Hybrid Protocol

So far, each protocol applies to accessing the entire external state. However, it is possible to use *independent* protocols per object, resulting in a hybrid protocol. This is because Halfmoon-read and Halfmoon-write differ only in the handling of reads and writes; they can share the same cursorTS of the SSF.

Idempotence. We note that the hybrid protocol is also idempotent. Because the cursorTS is still monotonically increasing within each SSF, we can directly use the inductive proofs of Halfmoon-read (Section 4.1.2) and Halfmoon-write (Section 4.2.2) to derive the idempotence of the hybrid protocol.

Consistency. The consistency of the Hybrid protocol is the weaker of Halfmoon-read and Halfmoon-write. We have the following proposition.

Proposition 4.17. *The hybrid protocol orders events through the following steps. First, we order the following events by real time: (1) all writes of the Halfmoon-read; (2) all reads and writes from Halfmoon-write. Second, we reorder the log-free writes from Halfmoon-read as described in Proposition 4.16. Finally, we insert the log-free reads from Halfmoon-read according to their logical timestamp. Specifically, a read with timestamp t is inserted immediately after the logged event that receives seqnum t and advances the cursorTS to t. This total ordering enforces a sequential history for all logged events in each SSF; similar to Halfmoon-write, consecutive log-free operations may commute but never cross the surrounding logged events.*

Proof. For the first step, we note that the real time of a write in Halfmoon-write is determined by the point when the write log record is appended in the logging layer, i.e., its commit point. Because the logging layer issues globally monotonic seqnums, the ordering of writes in Halfmoon-write based on real time is the same as the original one based on logical timestamps (Proposition 4.15). Therefore we can safely align the ordering of Halfmoon-read and Halfmoon-write into a total ordering.

Second, we reorder the log-free writes from Halfmoon-read in the same way as Proposition 4.16. We now show that the reordering of log-free writes would never cross the surrounding logged events. We have already proved the case for logged reads in Halfmoon-write in the proof of Proposition 4.16. We now prove the case for writes in Halfmoon-read. We use lowercase w to denote log-free writes from Halfmoon-write, and uppercase W to denote logged writes from Halfmoon-read. Specifically, we wish to show that a reordered log-free write w may never go backward across a preceding logged write W in the SSF. Suppose w is placed immediately before w', the successful log-free write that is applied after w. Then it suffices to show that w' happens later in real time than W. For w' to succeed in the conditional update, it must have acquired a higher cursorTS than that of w, which equals the one obtained after logging W. Because the logging layer issues monotonically increasing seqnums respecting the monotonicity of the real time, w' indeed happens later than W in real time.

Finally, the log-free reads in Halfmoon-read cannot be associated with real time. Therefore, we insert them based on their logical timestamps, relative to logged events that are also assigned logical seqnums. Note that the logged events can be writes from Halfmoon-read or reads from Halfmoon-write. The invariant is that the placement of a log-free read never goes past the preceding logged event in the SSF that advances the cursorTS to that used by the read; nor does it go past the succeeding one that advances the cursorTS to a higher value. After this step, the ordering of all

reads and writes from Halfmoon-read is still monotonic concerning logical timestamps, identical to that in Proposition 4.15, which preserves the semantics of Halfmoon-read. □

4.6 Garbage Collection

In line with previous work [53, 123], Halfmoon uses a **garbage collector** (**GC**) function to remove the log records of finished SSFs. The GC is periodically invoked. For Halfmoon-write, the lifetime of a read log record is equal to that of the initiating SSF. For Halfmoon-read, the GC should delete both the write log records and the matching object versions in the external state. Because a write log record has a dual purpose (Section 4.1), its lifetime should be the maximum of the SSF's and the object version's lifetime. Finally, the object version should outlive all SSFs that might read it.

Consequently, to garbage collect an object version whose matching write log record has seqnum t, the GC must wait until the following conditions are met: (a) there exists another record in the object's write log with seqnum $t' > t$, and (b) all SSFs that *start before* t', i.e., with initial cursorTS less than t', finish. Condition (b) entails the completion of the initiating SSF, so (a) and (b) also apply to garbage collecting the write log. Both conditions can be checked during the GC scan [53, 123]. Specifically, the GC tracks the latest seqnum $\bar{t}$ that satisfies (b). Upon advancing $\bar{t}$, for each per-object write log, the GC marks the latest log record whose seqnum falls below the new $\bar{t}$. These records point to the earliest object versions that might still be observed by current or future SSFs. Therefore, for each per-object write log, it deletes all records preceding the marked records, as well as the corresponding object versions in the external state.

4.7 Choosing the Right Protocol

As described in Section 4.5, we can choose to use Halfmoon-read or Halfmoon-write on a per-object basis. Qualitatively, we should use the Halfmoon-read/write protocol for read/write-intensive objects, respectively. We now quantify the decision. For simplicity, we consider only read and write operations. Let P_r, P_w as the probability that an SSF reads and writes the object, respectively. Let λ be the average arrival rate of SSFs. Then we can express the read/write intensity by multiplying P_r or P_w with λ.

Storage overhead. Let t be the average function lifetime (including re-execution in case of failures). The lifetime analysis in Section 4.6 assumes that GC is performed as soon as possible. To account for the periodicity of GC, we define T_{gc} as the average delay between the completion of an SSF and the next GC scan.

We start by deriving the storage overhead for Halfmoon-write, which consists of the read log records and a single version of the object. Let N_r be the average number of read log records across time. By Little's Law [76], N_r equals the effective arrival rate of reads, which is $P_r\lambda$, times the average lifetime of read log records. We therefore have $N_r = P_r\lambda(t + T_{gc})$, and the time-averaged storage overhead S_{read} is

$$S_{read} = S_{val} + N_r(S_{meta} + S_{val}) \quad (1)$$

$$= S_{val} + P_r\lambda(t + T_{gc})(S_{meta} + S_{val}), \quad (2)$$

where S_{meta} and S_{val} denote the metadata size of a read log record and the object size, respectively. The full size of a read log record is $S_{val} + S_{meta}$.

For Halfmoon-read, the storage overhead consists of the write log and several versions of the object. Let N_w be the average number of write log records across time, which is also the average number of object versions. Let T_w be the average time gap between two consecutive writes to the object. According to Section 4.6, the average lifetime of a write log record and the corresponding object version is $T_w + t$, where T_w enforces condition (a), and t enforces (b). Considering T_{gc}, we

have $N_w = P_w\lambda(T_w + t + T_{gc})$. Assuming a Poisson arrival of SSFs [69], we have $T_w = 1/(P_w\lambda)$. The average storage overhead S_{write} is

$$S_{write} = N_w(2S_{meta} + S_{val}) \tag{3}$$

$$= (1 + P_w\lambda(t + T_{gc}))(2S_{meta} + S_{val}). \tag{4}$$

Equation (3) assumes that the size of a write log is equal to S_{meta}. Note that there is a coefficient of two because our prototype of Halfmoon-read also logs *before* each write to align the write logging overhead with Boki (Section 4.1). We further assume that S_{meta} is negligible compared to S_{val}. Dividing both S_{read} and S_{write} with S_{val}, we derive the boundary condition as $P_r = P_w$. A higher read intensity means that Halfmoon-read has lower storage overhead, and vice versa.

Runtime overhead. Let C_w be the extra cost of a write in Halfmoon-read compared to that of Halfmoon-write. Similarly, let C_r be the extra cost of a read in Halfmoon-write over Halfmoon-read. The extra cost takes all relevant factors into account, including logging and multi-versioning. In a given time period T, the expected numbers of reads and writes to the object are $P_r\lambda T$ and $P_w\lambda T$, respectively. Then the expected extra costs of the two protocols are $P_w\lambda TC_w$ and $P_r\lambda TC_r$ in total, respectively. For our system prototype, we have $C_w \approx 2C_r$, where the coefficient of two is due to the same reason as that of Equation (3) (aligning our write logging overhead with Boki). The boundary condition is then $P_r = 2P_w$. A higher read intensity means that Halfmoon-read has lower runtime overhead, and vice versa.

Remark. Note that the runtime analysis assumes that all SSFs have equal importance. To differentiate between SSFs, we can analyze the read and write activity of each SSF respectively, finally taking a weighted sum. We can also combine the runtime overhead with the storage overhead by taking another weighted sum, e.g., by their monetary cost, to facilitate the final decision.

4.8 Switching between Protocols

The intensity of read and write, namely P_r and P_w, may change over time for a particular object. Therefore, we design a switching procedure between the two protocols. Note that switching is a *global* decision over individual objects; the read or write intensity considered in this section is aggregated over all SSFs during a certain period.

We first provide an overview of the switching (Section 4.8.1). We then present a detailed description of the transitional read and write operations during the switching, utilizing a novel locking primitive to reconcile the single- and multi-version schemas under the two protocols (Section 4.8.2). Finally, we introduce a mechanism to optionally bump the external state up-to-date after the switching (Section 4.8.4).

4.8.1 Overview. There are three major requirements. First, the switching must not violate Theorem 4.14. Second, the switching should not be a stop-the-world operation; SSFs should be able to run concurrently. Third, the switching must be fault-tolerant, i.e., SSFs must consistently use the same protocol for each step during re-execution.

To satisfy the above requirements, we maintain a *transition log* to record the switching history for each object. It is necessary because there can be an arbitrary delay between SSF failure and re-execution, possibly spanning several switching events. The runtime starts the switching by appending a "BEGIN" record to the transition log. It also scans the init log records (Section 4.1) to find all running SSFs that start before the switching. When all of these SSFs finish, the runtime completes the switching by appending an "END" record.

We now describe the switching from the perspective of an SSF. The first time an SSF reads or writes an object, it queries the transition log of the object to determine which protocol

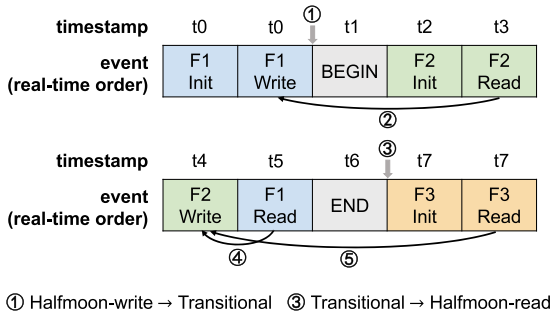


Fig. 9. Example of the switching procedure.

to use, and consistently uses the same protocol for that object in subsequent operations. The `SelectProtocol` in Figure 10 shows the pseudocode (line 1–6). Specifically, it calls `logReadPrev` to retrieve a transition log record, using the *initial* cursorTS acquired in `Init`. This ensures that the switching is fault-tolerant, since both the cursorTS and the transition log are persistent. If the log record is "END", the SSF should use the target protocol specified in the record as normal. However, if the record is "BEGIN", the SSF should use a special *transitional* protocol that logs *all* reads and writes. It cannot use the new protocol immediately; otherwise, SSFs using the old protocol will run concurrently with those using the new, violating Theorem 4.14. Once all SSFs using the old protocol finish, the runtime can safely switch to the new protocol.

Concerning the external state, we need to support both single- and multi-versioning as required by Halfmoon-write and Halfmoon-read. Our solution is to treat single-versioning as a special case of multi-versioning. The former corresponds to a special LATEST version (managed by Halfmoon-write) among other versions (managed by Halfmoon-read). As per Section 4.1, multi-versioning can be implemented over plain key-value APIs where each version is represented by a separate key. Therefore the two versioning schemas can be seamlessly integrated. There is no intersection between them except during the switching.

Example. Figure 9 shows an example where the runtime switches from Halfmoon-write to Halfmoon-read. F1 (using Halfmoon-write) and F3 (using Halfmoon-read) perform single- and multi-version reads and writes, respectively. The handling of F2 is more complicated. F2's write should be visible to both F1 and F3 (④ and ⑤), so it has to modify the LATEST version as well as create a standalone version. F2's read should target both the LATEST version using Halfmoon-write (②) as well as some other version using Halfmoon-read (not shown in the figure), because its lifetime may overlap with both F1 and F3. F2 should compare the freshness of the data returned by the two protocols, namely the "version" attribute of the LATEST object version, and the seqnum of the write log record matching the other version. The fresher one is chosen as the read result and logged for idempotence.

4.8.2 Transitional Reads and Writes with Visibility Lock. We now present the details of the transitional reads and writes during the switching. Figure 10 shows the pseudocode. We redesign the `Read` and `Write` operation to handle the transition between Halfmoon-read and Halfmoon-write. We introduce a novel locking primitive, the *visibility lock*, to atomically commit single- and multi-version writes.

Write with visibility lock. Transitional writes during the switching (`TransWrite`) should modify the LATEST version (for Halfmoon-write) as well as create a standalone version (for

```
def SelectProtocol(env, key):
    transLog = logReadPrev(key+"TRANSITION", getInitTS(env))
    if type(transLog) is BEGIN:
        env.protocol[key] = TRANSITIONAL
    else: # is END
        env.protocol[key] = transLog["protocol"]

def visLock(env, key, id):
    DBWrite(key, update="insert {id} -> LOCK")

def visUnlock(env, key, id):
    DBWrite(key, update="delete {id} -> LOCK")

def TransWrite(env, key, value):
    env.step += 1
    stepID = (env.ID, env.step)
    visLock(env, key, stepID)
    vNum = getVersionNumber(env)
    # standalone version
    # omit checking if the log record already exists
    DBWrite(key+str(vNum), value)
    env.cursorTS = logCondAppend([env.ID, key],
        LogRecord{
            "step": env.step, "op": "write",
            "version": vNum,
        }, condTag=[env.ID], condPos=[env.step])
    # the LATEST version
    vNum = env.cursorTS
    DBWrite(key+"LATEST", cond="VERSION < {vNum}",
        update="VALUE={value}; VERSION={vNum}")
    visUnlock(env, key, stepID)

def TransRead(env, key):
    env.step += 1
    # standalone version
    writeLog = logReadPrev(key, env.cursorTS)
    vNum = writeLog["version"]
    standalone = DBRead(key+str(vNum))
    # the LATEST version
    latest = DBRead(key+"LATEST")
    # get the fresher data from the two versions
    value = getHigherVersionOf(standalone, latest)
    # omit checking if the log record already exists
    env.cursorTS = logCondAppend([env.ID], LogRecord{
        "step": env.step, "op": "read",
        "data": value,
    }, condTag=[env.ID], condPos=[env.step])
    return value
```

Fig. 10. Pseudocode of the switching procedure.

Halfmoon-read). The problem is that the two versioning schemas have different commit points for writes. The LATEST version, being a normal object in the external state, is committed once the conditional write succeeds. The standalone versions, being unordered by themselves, are committed once the corresponding write log records are appended to the logging layer. The

transitional write should ensure that the two commits are atomic; otherwise, a single write may appear at two different points in the event stream, violating exactly-once semantics.

Our solution is to introduce a new locking primitive for the LATEST version during the switching. The *visibility lock* is implemented as a hash map associated with the LATEST object version. It is designed to block reads when the object is being updated by transitional writes. `TransWrite` acquires the lock by inserting its identifier, i.e., the (env.ID, step number) tuple, into the hash map (line 17 in Figure 10). Then it sequentially performs a multi-version write (line 19–25) and a single-version write (line 27–29). The seqnum of the write log record from the multi-version write is assigned as the version number for the single-version write (line 27). Finally, it releases the lock by deleting its identifier from the hash map. Because the identifier of the write operation is unique, acquiring and releasing the lock is inherently idempotent.

Note that the commit point of the entire write coincides with that of the constituent multi-version write. The visibility lock "hides" the LATEST version from reads so as not to expose a partially committed state where the single-version part has yet to be applied. Note that we perform the multi-version write first because it allows us to parameterize the commit point using the seqnum. Should we perform single-version write first, the commit point would be based on the real time, making it impossible to commit on the multi-version side, which is ordered by logical timestamps. While the visibility of the LATEST version is delayed by the lock, the multi-version write becomes immediately visible once committed, because its commit point extends to the entire transitional write. Because the write operations on the LATEST version, either from transitional writes during the switching or from log-free writes under Halfmoon-write, are not dependent on each other, Halfmoon-write can safely ignore the visibility lock during its log-free writes.

Read with visibility lock. For an SSF that starts during a switching procedure, its lifetime may overlap with SSFs both before and after the switching. Therefore, a transitional read `TransRead` should target both the LATEST version as well as some multi-version object. Because we perform the multi-version write prior to the single-version write in `TransWrite`, any uncommitted writes to LATEST version, if issued by a transitional write, are guaranteed to be present as a standalone object version. Therefore, the transitional reads have no need to block on the visibility lock. Instead, it first reads from some standalone version and then from the LATEST version (line 34–39), and compares the freshness of the data, i.e., the seqnum of the write log record associated with the standalone version and the version number of the LATEST version (line 41). The seqnum and the version number are comparable because the latter is based on cursorTS, which is in turn based on seqnums. The fresher data is chosen as the read result and logged for idempotency. Similarly, reads under Halfmoon-read can also bypass the visibility lock.

However, because reads under Halfmoon-write only target the LATEST version and may miss uncommitted writes, they must wait until the visibility lock is free, i.e., the hash map becomes empty. In practice, Halfmoon-write does not actually block its reads. Instead, a read is considered valid only if the hash map in the retrieved object is empty; a non-empty hash map indicates that the read should be retried later. To avoid starving these reads, we can borrow ideas from the well-studied reader-writer problem [88] to introduce a reader lock beside the visibility lock.

4.8.3 Idempotence and Consistency.

Idempotence. As described above, a `TransWrite` involves the following sequential steps: (1) acquiring the visibility lock, (2) performing a multi-version write as in Halfmoon-read, (3) performing a single-version write as in Halfmoon-write, and (4) releasing the visibility lock. Each step is idempotent and deterministically updates the SSF context. A `TransRead` is idempotent because it logs the read result. Therefore, the switching procedure retains idempotence.

Consistency. With the help of the visibility lock, the `TransWrite` is atomic despite performing a multi-version write and a single-version write. Therefore we can safely insert `TransWrite` and `TransRead` into the existing total order of events (Section 4.5). We have the following proposition.

Proposition 4.18. *Introducing the switching procedure retains the same level of consistency in Proposition 4.5.*

Proof. Based on the way we construct the total ordering in Proposition 4.5, we insert the `TransWrite`s between step one and step two. Specifically, because the commit point of a `TransWrite` is the same as that of the multi-version write it performs, we insert them in the same way as we insert the writes of Halfmoon-read, i.e., based on the real time of the commit point.

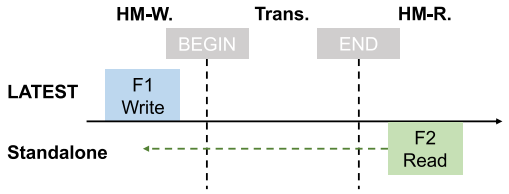
We insert the `TransRead`s as a final step after the original step three. At this point all writes are already inserted. The position of a `TransRead` depends on the data it observes. If the data is written by a successful log-free write from Halfmoon-write, the `TransRead` is inserted based on its real time, the same way we insert the reads from Halfmoon-write. Otherwise, it is inserted based on its logical timestamp (Proposition 4.17), the same way we insert the reads from Halfmoon-reads. Note that despite using a mixture of real time and logical timestamps for insertion, consecutive `TransRead` in an SSF never gets reordered. This is because the logging of the read result always advances the cursorTS for subsequent usage of logical timestamps, in accordance with the real time.

In this total ordering, the placement of `TransRead` or `TransWrite` are all reduced to the placement of reads and writes from Halfmoon-read or Halfmoon-write, without introducing new dependencies. Therefore, the consistency of the hybrid protocol is retained. □

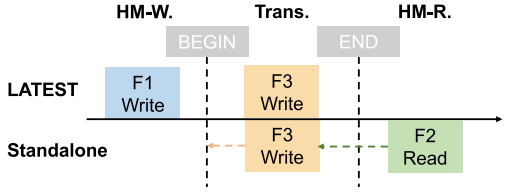
4.8.4 Bringing the External State Up-To-Date After Switching. The transitional reads and writes bridge the two Halfmoon protocols by targeting both versioning schemas. Nonetheless, the transitional SSFs that use such operations during the switching are only responsible for objects in their read and write sets. Suppose we are switching from Halfmoon-write to Halfmoon-read. Let there be an object that has been updated by Halfmoon-write before the switching ends (i.e., by SSFs that start before and overlap with the switching), but has not been touched by any transitional SSF (i.e., those that start after the switching). Then the latest state of that object is present in the LATEST version but not as any standalone version, thereby being invisible to Halfmoon-read after the switching (Figure 11(a)).

While this phenomenon is benign to the consistency of Halfmoon, which does not always enforce real-time reads and writes, it would be beneficial to bring the external state up-to-date after the switching. Following the previous example, for a log-free read after the switching, let *wlog* be the write log record it retrieves in `logReadPrev` (Figure 5) during the read. Then the read result is considered up-to-date only if the seqnum of *wlog* is higher than that of the latest BEGIN record in the transition log, i.e., the one that initiates the transition from Halfmoon-write to Halfmoon-read (Figure 11(b)). This essentially requires that the read should target some write either from transitional SSFs (during the switching), or from SSFs using Halfmoon-read (after the switching). If there is no such write, then it is the responsibility of the read to create a standalone version that reflects the possibly fresher state in the LATEST version.

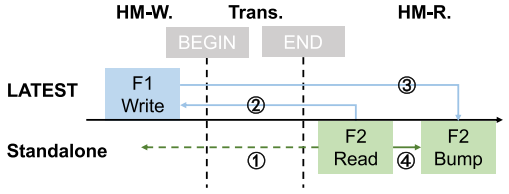
To this end, we extend the `Read` operation of Halfmoon-read and Halfmoon-write to `BumpRead`. Here we focus on `BumpReadHMR` as we are using Halfmoon-read after the switching (Figure 11(c)). The pseudocode of `BumpReadHMR` is shown in Figure 12. Specifically, it first performs a normal log-free read as in Halfmoon-read (① in Figure 11(c), line 4–7 in Figure 12), and checks the seqnum of the write log record if it satisfied the "freshness" criterion (line 9 in Figure 12). If not, it proceeds to read from the LATEST version as Halfmoon-write would do, without the logging step (② in Figure 11(c), line 14 in Figure 12). Then it performs a complete multi-version



(a) The external state could be stale after the switching.



(b) A read is up-to-date only if it maps to a write that took place no earlier than the beginning of the switching.



(c) A `BumpRead` operation (`BumpReadHMR` here) eagerly syncs between the LATEST version and standalone versions to bring the external state up-to-date after the switching.

Fig. 11. Example of switching from Halfmoon-write to Halfmoon-read.

write as in Halfmoon-read to create a standalone version and the corresponding write log record (③ in Figure 11(c), line 17–27 in Figure 12). The exception is that appending to the write log is conditional—the standalone version must map to the first log record in the per-object write log after the switching starts (line 21 in Figure 12). Otherwise, there could be a conflicting multi-version write that takes a higher priority over the bumped version. In the presence of a conflict, the `logCondAppend` should return the seqnum of the conflicting write log record along with an error message. `BumpReadHMR` would then return the up-to-date object version based on the conflicting record (④ in Figure 11(c), line 31 in Figure 12).

In case we are switching from Halfmoon-read to Halfmoon-write, we use `BumpReadHMW` to bring the external state (the LATEST version) up-to-date (Figure 13). The logic of `BumpReadHMW` is simpler. It first fetches the LATEST version as in Halfmoon-read (line 6), and checks if it satisfied the "freshness" criterion (line 8). If not, it reads from a standalone version using a log-free read as in Halfmoon-read (line 11–14). Then it performs a conditional update on the LATEST version to apply the fresh data. By requiring that the version number be monotonically increasing on the LATEST object version, the conflict between the bumped version and concurrent writes, if any, is automatically resolved.

5 Implementation

We implement the Halfmoon prototype on top of Boki. We modify 2300 lines of C++ in the logging layer. Halfmoon's client library consists of 1700 lines of Go.

```
def BumpReadHMR(env, key):
    # get the last BEGIN record in the transition log
    transLogBegin = lastBeginFromTransLog(env)
    # normal log-free read
    writeLog = logReadPrev(key, env.cursorTS)
    vNum = writeLog["version"]
    standalone = DBRead(key+str(vNum))
    # up-to-date
    if writeLog["seqnum"] >= transLogBegin["seqnum"]:
        return standalone
    # possibly stale
    env.step += 1
    # at this point there is no SSF that uses HM-W to update the LATEST version,
     so ignore visibility lock
    latest = DBRead(key+"LATEST")
    # get the fresher data from the two versions
    value = getHigherVersionOf(standalone, latest)
    # create up-to-date standalone version
    vNum = getVersionNumber(env)
    DBWrite(key+str(vNum), value)
    # conditional append to the write log; omit checking if the log record
     already exists
    wLogPos = getWriteLogLengthAt(
        tag=key, at=transLogBegin["seqnum"])
    env.cursorTS, err = logCondAppend([env.ID, key],
        LogRecord{
            "step": env.step, "op": "write",
            "version": vNum,
        },
        condTag=[key, env.ID],
        condPos=[wLogPos, env.step])
    if err is None:
        return value
    else:
        return getObjFromConflict(err["seqnum"])
```

Fig. 12. Pseudocode of **BumpHMR**.

5.1 Resolving Conflicts Among Peer Instances

We discuss in Section 4 that there are two race conditions against the exactly-once semantics. We handle the first one in Section 4.1–4.2. To address the second one (the race between peer instances), We introduce **logCondAppend** (Figure 3). Compared with **logAppend**, it takes two additional parameters. **condTag** is a set of log tags that each maps to a specific log stream. For the sake of resolving conflicts, **condTag** is set to the caller SSF's instanceID (**env.ID**); the matching log stream contains the log records created by the SSF. **condPos** is the current step number of the caller. **logCondAppend** first tries to append to the log normally as **logAppend** does. It then checks the offset of the log record in the caller's log stream specified by **condTag**. The conditional append succeeds if the offset is equal to **condPos**, i.e., the step number is as expected and the log record appears in the right position in the SSF's execution history. Otherwise, it undoes the log append and returns the seqnum of the log record at the expected offset, along with an error message. In case there are multiple **condTag** and **condPos** pairs, the log append is successful only if the record's offset meets expectation on all log streams.

```
def BumpReadHMW(env, key):
    env.step += 1
    # get the last BEGIN record in the transition log
    transLogBegin = lastBeginFromTransLog(env)
    # omit checking visibility lock
    latest = DBRead(key+"LATEST")
    # up-to-date
    if latest["VERSION"] >= transLogBegin["seqnum"]:
        value = latest
    else:
        # get standalone version
        writeLog = logReadPrev(key, env.cursorTS)
        vNum = writeLog["version"]
        standalone = DBRead(key+str(vNum))
        # get the fresher data from the two versions
        value = getHigherVersionOf(standalone, latest)
        # update the LATEST version
        DBWrite(key, cond="VERSION < {vNum}",
            update="VALUE={value}; VERSION={vNum}")
    # append to the read log; omit checking if the read log record already exists
    env.cursorTS = logCondAppend([env.ID], LogRecord{
            "step": env.step, "op": "read",
            "data": value,
        }, condTag=[env.ID], condPos=[env.step])
    return value
```

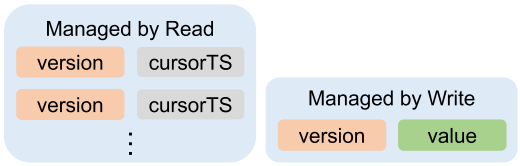
Fig. 13. Pseudocode of **BumpHMW**.

logCondAppend is similar to the compare-and-swap operation in concurrent programming. It resolves conflicts *in place* to ensure that only a single instance succeeds in logging a certain operation. For symmetric protocols, logging and conflict resolution can be performed *separately*. For example, Boki immediately reads the caller's stream after appending to the log, and only honors the first log record of the current operation. This approach is practical because the sole purpose of logging in symmetric protocols is checkpointing SSF progress. Therefore the caller's log stream is visible to the peer instances but not to other SSFs. Conflict resolution, though as a separate step, naturally serves as a synchronization point among peers, i.e., it ensures that all instances have identical states after this step. However, Halfmoon's use of the write log (Figure 5) presents a new synchronization problem. Because the write log records also appear in the object's log stream specified by the **key** tag, they are visible to other SSFs as well. Suppose we resolve conflicts separately, then a concurrent log-free read might see an inconsistent state of the object's stream. In contrast, **logCondAppend** greatly simplifies the reasoning about concurrent instances.

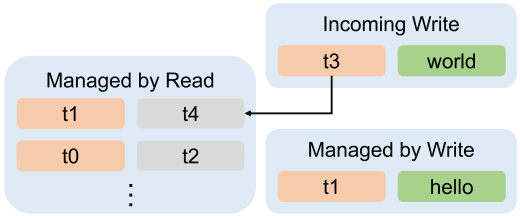
We extend the implementation in Section 4 to handle the race condition among peers by simply replacing all **logAppend** with **logCondAppend**. If the conditional append fails, we let developers decide how to handle the error. For example, the SSF instance can use the returned seqnum to read the log records at the expected offset, and proceed with an identical state as its peers. Alternatively, it can quit the race by exiting.

6 Extensions of Halfmoon

This section presents two extensions of Halfmoon. The first extension enforces strict ordering of consecutive writes, which provides SC. The second extension exports the current protocols to a partially ordered shared log, which generalizes Halfmoon to geo-distributed settings.



(a) Object layout.



(b) A timestamp inversion where the write's version is lower than the pre-read cursorTS of a previous read.

Fig. 14. Example of the extended Halfmoon-write protocol.

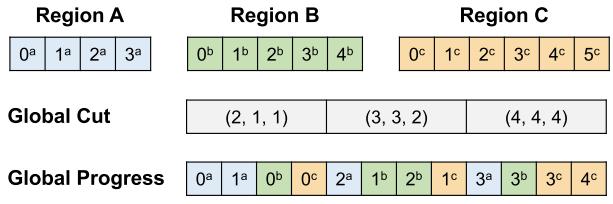
6.1 Strict Ordering of Consecutive Writes

We design an extension of Halfmoon-write that eliminates possible reordering of consecutive writes (Section 4.4), thereby providing SC. The key idea is to utilize the pre-read cursorTS, i.e., the SSF's cursorTS before logging the read result, to detect and break possible dependency cycles in Figure 8(b). By definition, the pre-read cursorTS represents the highest possible version number this SSF might have used in its writes. This allows SSFs to check during writes if there exists a *timestamp inversion* edge, e.g., edge ② in Figure 8(b), where a write has data dependency on a read with a higher cursorTS. If so, the write should perform extra logging to advance the cursorTS for subsequent writes, thereby inverting edge ④ in the dependency cycle.

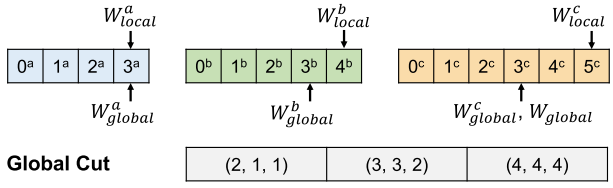
The extended protocol is log-free on writes in the best case. By best case, we mean that whether the protocol requires extra write logging depends only on the actual interleaving of operations, regardless of the dependencies in the SSF code. In the best case, e.g., under a write-intensive workload, there is no timestamp inversion and the protocol can be log-free on writes.

To check for timestamp inversion, we let reads maintain a monotonically increasing sequence of (*version*, *pre-read cursorTS*) pairs for each object. Figure 14(a) shows the layout of an object under the extended protocol. The version field is equal to the version of the object the read has seen. A read creates such a pair only if its cursorTS is higher than (*i*) the current object version, which indicates a potential timestamp inversion, and (*ii*) the largest pre-read cursorTS in the sequence of pairs, so as to ensure monotonicity. If several consecutive pairs have the same object version, they are merged by overwriting the tail of the sequence. Updating the sequence should be atomic with reading the object, which can be implemented using existing techniques such as DAAL [100, 123].

Upon visiting the object, a write atomically updates the object and searches for a pair in the sequence that has the largest object version below the version of the write. If the cursorTS of this pair is higher than the write version, then there is a timestamp inversion (Figure 14(b)). The subsequent extra logging step need not be atomic with the above step; the atomicity of reading the object and updating the sequence of pairs ensures idempotence in the detection of timestamp inversion. We provide a TLA+ verification of idempotence and SC of the extended protocol in our technical report [6].



(a) Synchronizing regions with global cuts.



(b) Reconciliating global and local progress.

Fig. 15. Example of the extended Halfmoon-read protocol over a partially ordered log across three regions.

6.2 Halfmoon Over Partially Ordered Shared Log

The current design of Halfmoon depends on a totally ordered shared log, which can be costly to implement in geo-distributed settings [81, 105]. We present an extension of Halfmoon based on a partially ordered shared log, where users are clustered into different regions and each region has a locally ordered log. Because the logs across regions are only partially ordered, the extension provides the weaker but more scalable causal consistency [80] than the SC of original Halfmoon. Causal consistency only requires that causally dependent operations are delivered in the same order across regions, which allows us more flexibility in the design of the extension.

Halfmoon-read. Because a log-free read in Halfmoon-read (Section 4.1.1) seeks backward from its read timestamp on the write log and returns the latest preceding write. For such a read to be idempotent, we must maintain the invariant that once a read obtains t as its read timestamp, no subsequent writes can be ordered before t in the shared log. In the presence of a total order, this invariant is easily satisfied. With a partially ordered log, however, there is no restriction against the ordering of concurrent writes across regions. Fortunately, by targeting causal consistency, we can instead restrict the scope of reads. That is, we let a read seek backward on the locally ordered write log within its region. Consequently, reads and writes from different regions are causally independent and can be arbitrarily ordered, eliminating the need for cross-region synchronization on the critical path. Another benefit of this design is that SSFs can still use the seqnum of its local log as cursorTS and read timestamps.

However, being causally consistent allows regions to diverge, even permanently. Therefore it is desirable to add a convergence property to causal consistency [80], i.e., being both causally and eventually consistent. Our solution is to periodically synchronize the logs across regions, but do so asynchronously without pausing reads or writes. Figure 15(a) shows the synchronization process. Similar to Boki's metalog [53], a global sequencer collects the progress of per-region logs and periodically publishes a cut. The increment between the two cuts indicates the global progress. To allow regions to converge, we should still order the log records within the global progress. However, the order can be arbitrary because records from different regions are causally independent. For example, we can simply order the records by the region ID, which allows each region to independently derive the same order.

Because the global cut is issued asynchronously, it could lag behind the local log. Each region then must reconcile the global progress with the local one. Let there be a read with timestamp t in region i. Let W^i_{local} denote the local log record it seeks backward to. Regarding the latest global cut received by region i as of timestamp t, let W^i_{global} denote the latest write log record from region i in that cut, and let W_{global} be the latest write log record from all regions after ordering the global progress. Then there are two possible situations. If W^i_{global} does not lag behind W^i_{local}, then the read should seek to W_{global}. Otherwise, the read should seek to W^i_{local}. Figure 15(b) shows an example. Region A should use W_{global}, which is W^C_{global}, while Region B and C use W^B_{local} and W^C_{local}, respectively. In simple terms, local writes generally take precedence over writes from other regions, in favor of causal consistency; but for "quiet" regions that are not ahead of the global progress, we respect the global progress in favor of convergence.

Note that Halfmoon-read relies on the shared log instead of the external store for the ordering of events. Moreover, because it uses multi-versioning, the external store is only required to be eventually consistent. If an object version is not found at the moment, Halfmoon-read should retry the read later.

Halfmoon-write. The extension of Halfmoon-write is more straightforward. Unlike Halfmoon-read, Halfmoon-write is mainly dependent on the external store for consistency. Consequently, the external storage should be causally consistent, which is known to be practical and scalable for geo-distributed settings [80]. Because external storage handles the ordering of writes from different regions, it remains for Halfmoon-write to enforce idempotence per region. Therefore the read log can be purely local. The only requirement is that the version field of an object should be a vector, where each element corresponds to a region [19, 68]. The conditional update in the write operation only needs to check the respective region's version.

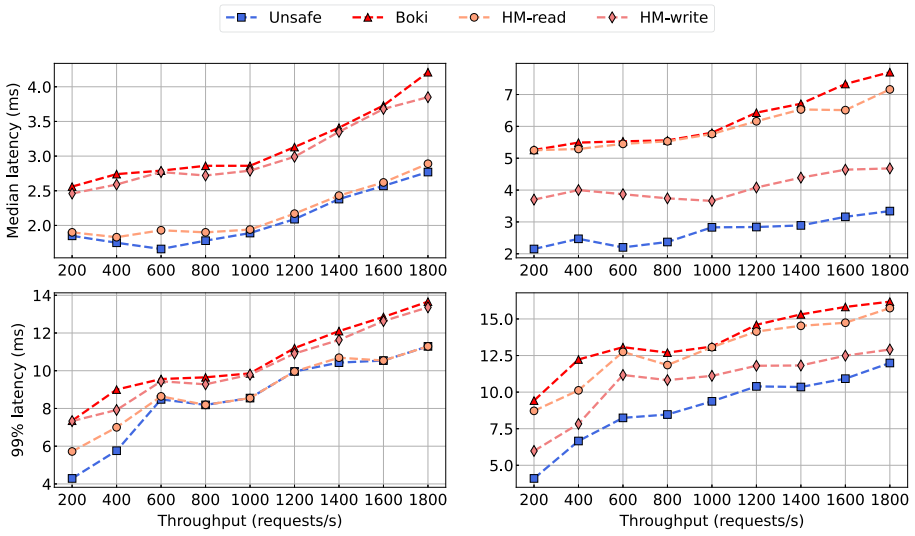
7 Evaluation

This section compares Halfmoon's performance with the state-of-the-art solution Boki [53], using microbenchmarks (Section 7.1) and realistic applications (Section 7.2). We also include an unsafe baseline with no logging. It does not offer exactly-once semantics and serves as the lower bound of Halfmoon. We evaluate Halfmoon's storage and runtime overhead under different read/write intensity in Section 7.3 and explore the switching delay between Halfmoon's protocols in Section 7.4.

Experimental setup. We conduct all our experiments on AWS EC2 c5d.2xlarge instances using the configuration reported in Boki [53]. Each instance has 8 vCPUs, 16GiB of DRAM, and 200GiB NVMe SSD. The runtime infrastructure consists of eight function nodes and one function gateway; the logging layer consists of three storage nodes and one sequencer node. Both Boki and Halfmoon use Amazon DynamoDB [37] as the external storage.

7.1 Microbenchmarks

We measure the median and 99%-tile tail latency of reads and writes over a period of 10 minutes under various request rates. In line with previous works [53, 123], we use a synthetic SSF that issues one read and write per request. We populate the external state with 10K objects, each consisting of 8B key and 256B value. Figure 16 shows the results. Compared to Boki, Halfmoon-read offers ~30% lower latency on reads and achieves similar performance on writes (as per Section 4.1, we deliberately align Halfmoon-read's write logging overhead with Boki). It offers exactly-once reads with only ~15% overhead over unsafe raw reads, which is 4–5× lower than Boki. Halfmoon-write also achieves ~30% lower latency than Boki on writes, and has similar performance on reads.



(a) Read. (b) Write.

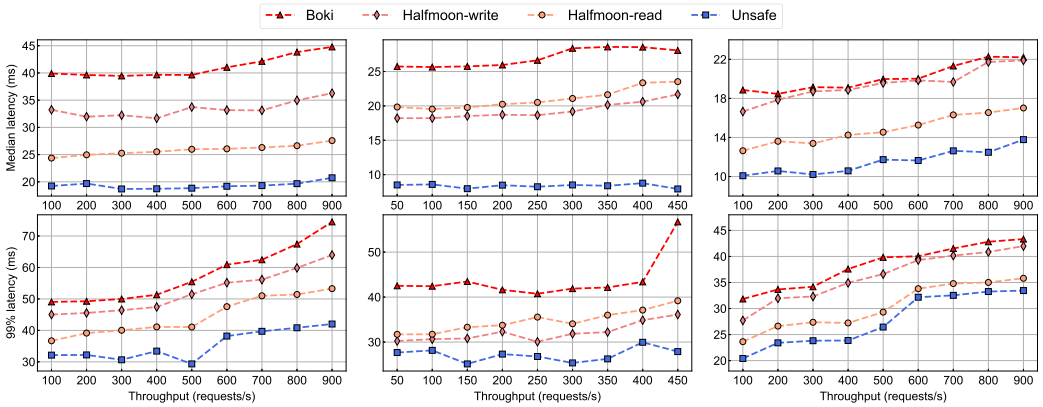
Fig. 16. Latency of read and write under different request rates. Halfmoon-read/write is abbreviated as HM-read/write.

The latency of log-free writes is still higher than raw writes, because Halfmoon-write performs *conditional* updates (Section 4.2), which is more expensive than directly updating the object. However, the overhead is still 2–4× lower than Boki, of which writes are also conditional and require logging. The curves of median and tail latency show similar dynamics as the baselines, indicating that Halfmoon's protocols do not introduce performance bottlenecks in the logging layer or the external storage.

7.2 End-to-End Application Workloads

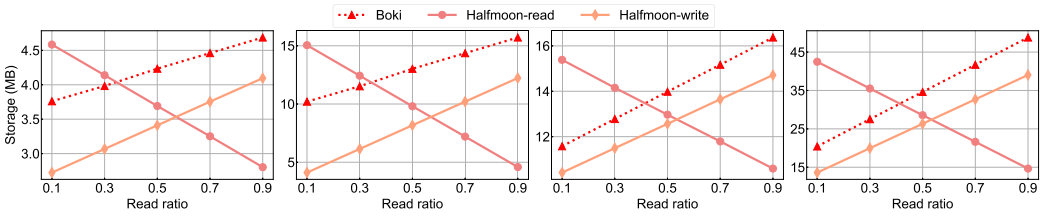
We evaluate Halfmoon over three realistic application workloads. The first two workloads, travel reservation and movie review, are adapted from DeathStarBench [3, 43] and are commonly used in Boki and Beldi. Travel reservation runs a workflow of 10 SSFs. Users search for nearby hotels based on distance and ratings, and then make reservations. This workload is read-intensive. Movie review runs a workflow of 13 SSFs. It is slightly skewed towards writes as posting user reviews makes up its core functionality. The third workload is Retwis, a simplified X (formerly known as Twitter) clone [13]. Retwis consists of several X functions (e.g., post tweet, get timeline) that perform PUTs and GETs on a key-value store. This workload is also read-intensive. All three workloads store application data in DynamoDB. We evaluate both Halfmoon-read and Halfmoon-write to demonstrate the benefits of using the appropriate protocol, as well as the worst-case performance penalties when using the wrong protocol.

Figure 17 shows the results. Using the appropriate protocol, Halfmoon offers 20%–40% lower median latency for the three workloads, and achieves 1.5–4.0× lower overhead above the unsafe baseline. Logging is typically not the bottleneck of Boki, so it saturates at approximately the same load as Halfmoon. Halfmoon-read outperforms Halfmoon-write in read-intensive workloads (Figure 17(a) and (c)), and otherwise in write-intensive workload (Figure 17(b)). Halfmoon outperforms Boki even under the wrong protocol, because Boki either logs more reads than Halfmoon-read or logs more writes than Halfmoon-write.



(a) Travel reservation. (b) Movie Review. (c) Retwis.

Fig. 17. End-to-end performance of Boki and Halfmoon under three application workloads.



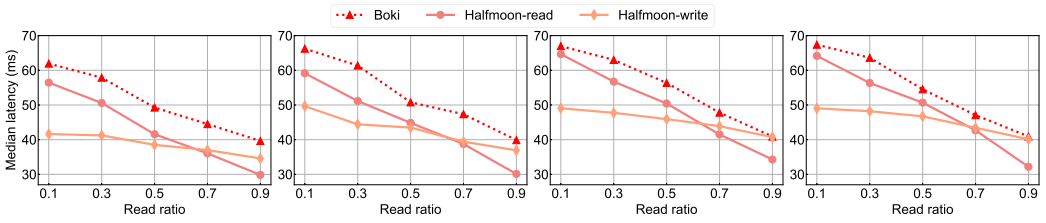
(a) size=256B, GC=10s. (b) size=256B, GC=60s. (c) size=1KB, GC=10s. (d) size=1KB, GC=60s.

Fig. 18. Storage overhead of Boki and Halfmoon under different object size and GC interval.

7.3 System Overhead

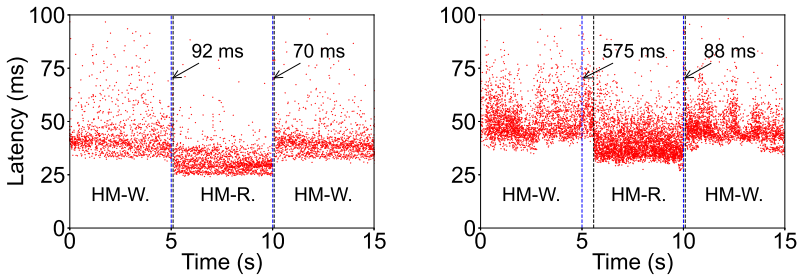
We use a synthetic SSF to validate the overhead analysis in Section 4.7. The SSF issues 10 operations to the database. We vary the read and write intensity of the workload by changing the ratio of reads among the operations. We populate 10K objects in the database. Each operation in the SSF targets a random object, such that the read ratio is representative of the read intensity over each object. We compare the two protocols to determine the boundary condition when they have equal overhead. For reference, we also measure the overhead of Boki.

Storage overhead. We measure the time-averaged storage usage over a period of 10 minutes. The overall usage consists of log and database storage. Halfmoon-write stores a single version of each object, while Halfmoon-read stores multiple versions. We vary the size of each object and the GC interval. Figure 18 shows the results. Section 4.7 predicts that the boundary condition is when the read and write intensity are equal, i.e., the read ratio is 0.5. The theoretical boundary is an asymptotic result when the log storage is negligible compared with database storage. The actual boundary is slightly higher, because Halfmoon-read logs twice for each write (Section 4.1), while Halfmoon-write logs once for each read. As the object size increases, the boundary moves closer to 0.5 as the database storage becomes the decisive part. As per our analysis in Section 4.7, the boundary condition is not affected by the choice of GC interval in Figure 18. Halfmoon-read has a higher storage usage than Boki under low read ratio, i.e., high write intensity, because the overhead of multi-versioning outweighs that of the read log records, which are rather scarce. Compared with Boki, Halfmoon requires 1.2–3.4× less storage on average.



(a) 100 requests/s. (b) 200 requests/s. (c) 300 requests/s. (d) 400 requests/s.

Fig. 19. Runtime overhead of Boki and Halfmoon under different request rates.



(a) 300 requests/s. (b) 600 requests/s.

Fig. 20. The switching delay between Halfmoon's protocols (abbreviated as HM-R/W). The red dot shows SSF latency over time. The blue vertical line marks the beginning of switching. The black line marks its end.

Runtime overhead. We measure the median latency under different request rates. The object size is set to 256B and the GC interval to 10s. Figure 19 shows the results. We predict in Section 4.7 that the boundary condition is when the read intensity is twice the write intensity, i.e., the read ratio is 2/3. The actual boundary is slightly higher, because C_w is more than twice C_r in practice (Figure 16). Figure 19 also confirms that the request rate has little impact on the boundary condition. Both of our protocols have lower latency than Boki, with an improvement factor of 1.2–1.5×. We empirically validate that Halfmoon's runtime performance is insensitive to the GC interval. This is because the overhead of querying the log or database index typically scales logarithmically with the number of log records or object versions.

7.4 Switching Delay

We use the same SSF as Section 7.3 to evaluate the switching delay. The workload has two phases. The first phase runs Halfmoon-write with a read ratio of 0.2. The second runs Halfmoon-read with a read ratio of 0.8. We change the phase every five seconds. Figure 20 shows the dynamics of SSF latency over time. Under a moderate load of 300 requests/s, the switching takes less than 100 ms. Under 600 requests/s (the workload saturates at about 800 requests/s), it takes longer to switch from Halfmoon-write to Halfmoon-read than the other way around. This is because SSFs in the first phase have a longer completion time due to the higher write intensity, and Halfmoon needs to wait until all SSFs using the old protocol finish before switching to the new protocol (Section 4.8).

7.5 Recovery Cost

Halfmoon's asymmetric protocols are designed for the failure-free common case. During re-execution, Halfmoon always replays log-free operations. In contrast, symmetric logging protocols

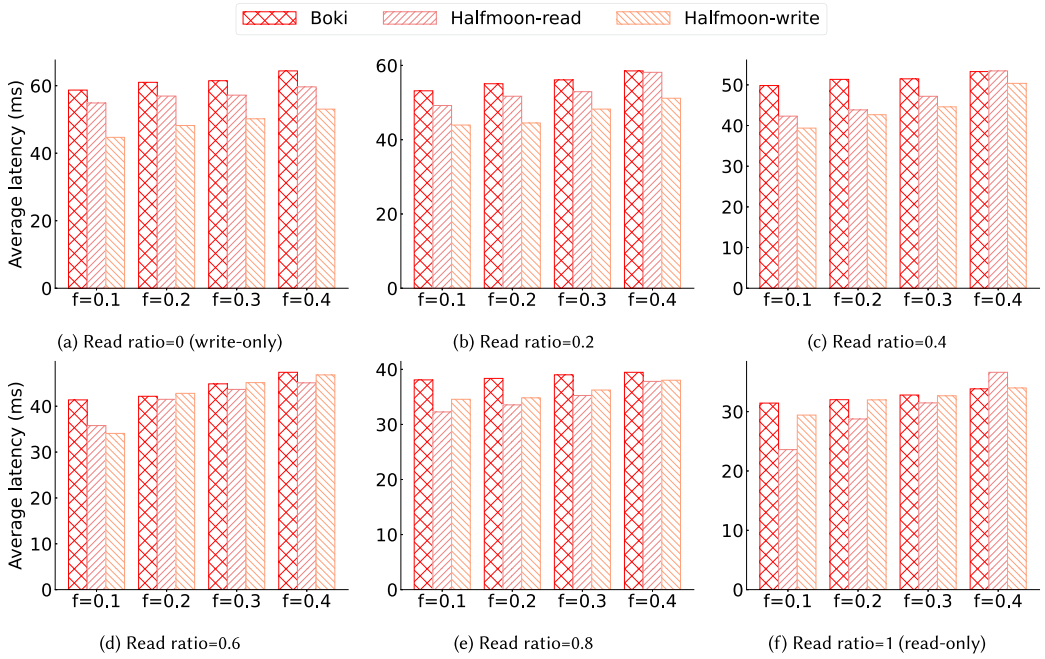


Fig. 21. The average end-to-end latency of Halfmoon and Boki under different failure rates.

such as Boki may skip logged operations. The speedup is implementation-specific. For Boki, the overhead of reading a log record is negligible compared to database operations. Therefore Boki always checks the logs before performing the actual read or write. In the extreme case, its recovery cost consists only of log reads.

We now incorporate the recovery cost into the end-to-end latency. We model the execution of an SSF as a Bernoulli Process. Let f be the probability of SSF failure. In each round, the SSF finishes with probability $1-f$ and crashes with probability f. Then the expected rounds of execution n before the SSF returns successfully is $1/(1-f)$. As an extreme case, we assume that the symmetric protocol has *zero* recovery cost, and the asymmetric protocol needs to replay *every* operation such that there is no speed up for recovery. Consequently, the total expected SSF latency under the symmetric protocols is simply the runtime cost of a single failure-free round. For our asymmetric protocols, the total expected latency consists of a single failure-free round and $n-1$ recovery rounds (with the same runtime cost as the failure-free round). Suppose that our protocols have x percent less runtime cost than the symmetric protocols in the failure-free case. Substituting x into the total expected latency, we see that the boundary condition is when f equals x, and a lower f implies that our asymmetric protocols have lower latency. According to Figure 16, the boundary condition between Boki and Halfmoon is $f \approx 30\%$, which far exceeds the actual failure rate of real applications.

We use the synthetic SSF workload in Section 7.3 to evaluate the end-to-end latency in the presence of failures. We run the workload at 100 requests/s and vary the read ratio of the workload from 0 to 1. Specifically, when the read ratio is 1, i.e., the workload is read-only, Halfmoon-read needs to replay every operation whereas Boki has approximately zero recovery cost. The same holds for Halfmoon-write when the read ratio is 0, i.e., the workload is write-only. Figure 21 shows the average latency of Halfmoon and Boki under different failure rates. Halfmoon has

comparable performance to Boki up to 40% failure rate. As for the extreme cases, Halfmoon outperforms Boki when f is less than 30% and 40% under the read-only and write-only workloads, respectively.

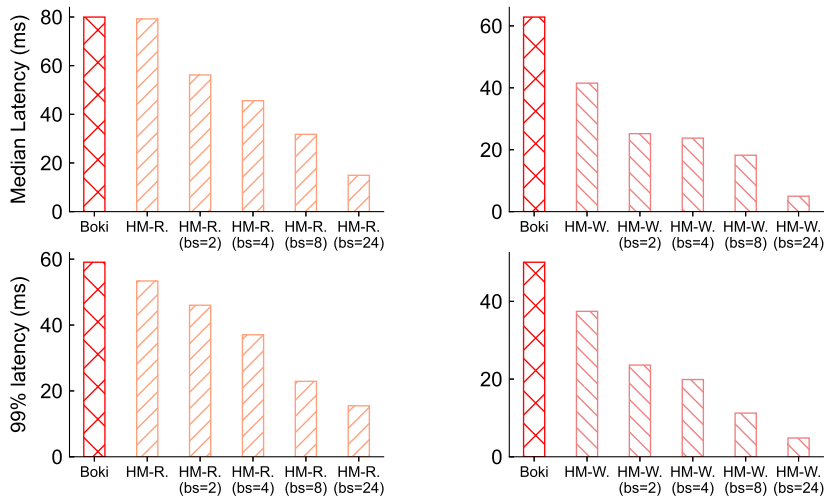
7.6 Overlapping Logging with Execution

For the sake of clarity, the logging operations in Halfmoon's protocol are issued synchronously with reads or writes. In practice, SSFs do not need to wait for the completion of logging unless the current operation is dependent on the seqnum returned from a previous logging operation, i.e., logging can be overlapped with execution (Section 4.3). In Halfmoon-read, such dependency only involves write-read pairs. Because consecutive writes install standalone versions and the logging layer guarantees that log records are added following program order, SSFs can proceed without blocking on the logging of writes. The log-free reads, however, require the cursorTS to be at least as high as the seqnum of any preceding writes, such that they may fetch the up-to-date object version. Consequently, it suffices to ensure, as a pre-condition of issuing log-free reads, that all previous logging operations have finished, and the cursorTS is set to the highest seqnum of those. Similarly, in Halfmoon-write, only log-free writes are dependent on previous logging operations. Reads can proceed without waiting for logging operations to complete.

Overlapping logging with execution can further reduce the logging overhead. We also evaluate the end-to-end performance of Halfmoon under different degrees of overlapping. The implementation we have described earlier yields the highest degree of overlapping, i.e., only waiting for completion before issuing a log-free operation. We explore other degrees by issuing and waiting for a batch of logging operations. We use the workload in Section 7.3 where the SSF issues a series of reads and then writes. The total number of reads and writes is set to 30. To demonstrate the performance bonus of overlapping logging with execution, we explore a write-mostly and a read-mostly setting, using a read ratio of 0.2 (6 reads, 24 writes) and 0.8 (24 reads, 6 writes), respectively. As an adversary, we use the Halfmoon-read protocol for the write-mostly workload and Halfmoon-write for the read-mostly workload. We vary the bs by which we wait for the completion of logging from two to eight, and the highest degree of overlapping is achieved at a bs of 24. Figure 22 shows the logging overhead, i.e., the end-to-end latency minus that of the unsafe baseline, in line with the overhead measurement in Section 7.3. In spite of using the unsuitable protocol, overlapping can effectively reduce the logging overhead. Both protocols can match the performance of the unsafe baseline under a large bs.

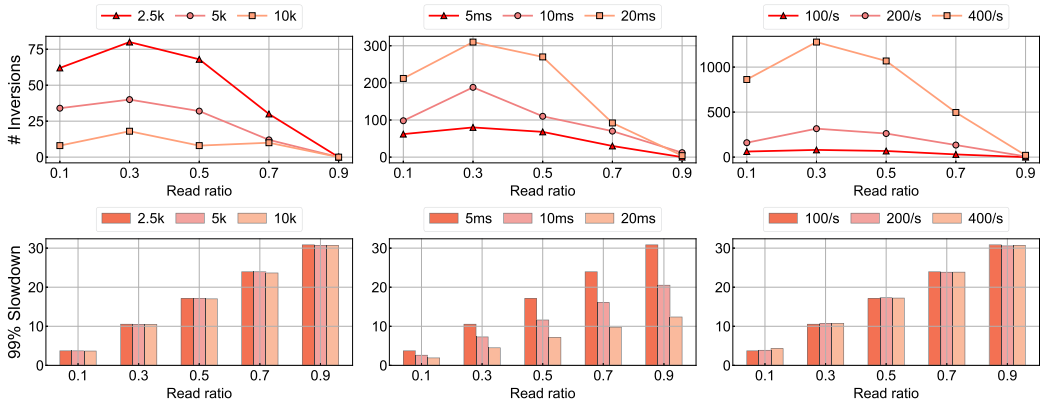
7.7 Overhead of the Halfmoon-Write Extension

We evaluate the overhead of the extended Halfmoon-write protocol in Section 6.1 compared to the vanilla Halfmoon-write. We also measure the number of timestamp inversions (Section 6.1) to understand the cause of overhead and the necessity of the extension. We use the workload in Section 7.3, where an SSF issues 10 read or write operations to the database. We change the ratio of reads among the operations to vary the read or write intensity. We explore three other dimensions of configurations that may affect the performance. First, we vary the number of objects in the database. A smaller number of objects increases the probability that an object is targeted by concurrent SSFs, resulting in more timestamp inversions. Second, we vary the time interval between reads and writes in an SSF. As demonstrated in the example in Section 8, the longer an SSF holds a stale cursorTS, the more likely it is to encounter a timestamp inversion when issuing a log-free write. Third, we vary the request rate, which also affects the probability of concurrent accesses and timestamp inversions. As the base case, we populate 2.5K objects, set the time interval to 5 ms, and issue requests at 100/s. We vary the number of objects up to 10K, the time interval up to 20 ms, and the request rate up to 400/s. The experiments run for 5 minutes.



(a) Halfmoon-read under write-mostly workload (read ratio=0.2). (b) Halfmoon-write under read-mostly workload (read ratio=0.8).

Fig. 22. Logging overhead when overlapping logging with execution. We vary the **batch size** (**bs**) by which we check the completion of logging. Logging overhead is measured as the end-to-end latency minus that of the unsafe baseline.



(a) Vary number of objects. (b) Vary time interval between operations. (c) Vary request rate.

Fig. 23. Overhead of the extended Halfmoon-write protocol. The top row shows the average number of timestamp inversions per minute. The bottom row shows the relative slowdown of the extended protocol over vanilla Halfmoon-write.

The top row of Figure 23 shows the average number of timestamp inversions per minute. As expected, the number of inversions is negatively correlated with the number of objects, and positively with the time interval and the request rate. As for the read ratio, the number of inversions has a turning point. This is because it involves both a read and a write to the same object to trigger an inversion; the probability is higher when the read and write intensity are balanced. Under all configurations, the SSF encounters at most *one* inversion per invocation. This suggests that timestamp inversions are rare in practice.

The bottom row of Figure 23 shows the relative slowdown of the extended protocol over the vanilla Halfmoon-write. The extended `Read` adds 1.8–5.7 ms overhead. The extended `Write` adds 0.5–1.6 ms overhead alone, and an additional 1.1–2.0 ms logging overhead if it encounters a timestamp inversion. Because `Read` is more expensive, the overall slowdown is positively correlated with the read ratio under all configurations. The number of objects and the request rate have little impact on the overhead, while a larger interval between operations amortizes the relative slowdown despite an increased number of timestamp inversions, since the extra logging overhead is not the dominant factor (at most one per SSF invocation). In the write-intensive workload that Halfmoon-write is intended for, the extended protocol adds 3–10% to the 99%-tile latency.

8 Discussion

Ordering and timestamps. SMR [15, 30, 38, 60, 63, 94] and its equivalents [20, 23, 25, 28, 44, 55, 82, 110] are common ways of building fault-tolerant distributed services [16, 56]. SMR ensures that each process replicating the service executes commands following a global total order. Timestamp-based ordering is widely adopted in SMR. Global ordering can be established using synchronized clocks [29, 61] or timestamp agreement mechanisms [32, 38, 53]. Halfmoon's event stream is built on top of SMR. It utilizes the event timestamps to deterministically parameterize log-free reads and writes. Halfmoon also requires that the event stream be traceable, which is naturally satisfied by existing SMR protocols that maintain a command log [20, 67, 93].

Security and privacy. The write log in Halfmoon-read serves a dual purpose (Section 4.7) and is visible to other SSFs. To avoid leaking private data, we can perform access control or encryption in the logging layer or the client library [126]. Only the shared fields should be exposed to other SSFs.

Program analysis and verification. Halfmoon takes no application-specific hints from SSF code. That is, it assumes the most pessimistic setup where all external operations are non-idempotent, which must be logged or deterministically parameterized. This is not necessarily true in practice. For example, if an object is read-only, then all reads to that object are inherently idempotent. Similarly, consider pushing to a message queue. If the receiver SSF can handle duplicate messages, then the push operation is also idempotent on its own. To avoid unnecessary logging, we can use existing tools [41, 42, 46, 50, 64, 71, 73, 84, 85, 90, 131] to analyze the SSF code beforehand to rule out such operations [71, 77, 83, 117, 118, 122]. Halfmoon is log-optimal in providing idempotent access to general mutable shared states.

Database logs. Logging is widely adopted in transactional databases [14, 92]. For example, WAL is a common technique to ensure durability and atomicity [89, 100]. The WAL defines the order of state transitions in the database, which allows for multi-versioning, recovery, and rollback. The write logging in Halfmoon-read works in the same way to support multi-version deterministic log-free reads. In principle, Halfmoon can reuse the WAL in the database for idempotence. However, the WAL is usually internal to the database and not available to the application layer. This is why our underlying system, Boki, decouples the logs to a separate logging layer where it can explicitly order events with log seqnums. Alternatively, the database can expose its internal logs to inform the application whether it has already performed a certain operation, which could further reduce the overhead of providing exactly-once semantics.

Transactions. In line with previous works, we assume that SSFs are non-transactional by default [53]; to execute several steps atomically, SSFs should explicitly use transactions. Nevertheless, to enforce exactly-once semantics, it is indeed a viable approach to model the entire SSF as a transaction, which reduces the problem to atomicity [102]. However, it also implies that all

external effects must be invisible to other SSFs until the transaction is committed, otherwise risking the *read uncommitted* anomaly [107]. Because most serverless functions do not require all of their external effects to be transactional [13, 53, 103, 123], the delayed visibility could potentially change the application semantics, especially for function triggers. For example, suppose function A writes to an object and continues its execution. If function B is to be triggered upon changes to that object, then B's execution would be delayed until A is committed. While it is possible to insert early commit points in the function, it also involves manual effort. In contrast, we believe that providing idempotence at the operation granularity is a more flexible approach.

9 Related Work

Stateful serverless computing on top of the stateless FaaS paradigm has been identified as a challenging task by many prior works [40, 47, 62, 103, 124]. A line of research optimizes the transfer of intermediate state across functions in analytic or stream processing workflows [62, 65, 74]. This article targets a different scenario of sharing *mutable* state across SSFs. In terms of accessing the shared state, some works [53, 102, 103, 115, 123] adopt a data-oriented approach using table or key-value APIs, while others [2, 24, 26, 27, 31] adopt an object-oriented approach by encapsulating the state and access methods. Another line of research focuses on the formal semantics [27, 52, 59] and verification of SSFs [17, 33]. Halfmoon studies the orthogonal problem of the minimally necessary logging overhead for idempotence. It is an interesting direction to integrate Halfmoon's idea with these works to automatically generate log-optimal SSF implementations.

Log-based replay is a well studied approach for fault tolerance [34, 49, 96, 100, 109] and troubleshooting [41, 58, 86, 97, 108, 120, 121, 128, 129, 132]. The idea has been broadly embraced by serverless computing [53, 59, 123]. Olive [100] proposes to use write-ahead redo logging to achieve exactly-once semantics. Beldi [123] extends Olive to the serverless environment and supports transactional workflows. Boki [53] implements Beldi's techniques using the shared log [20, 32, 111]. In terms of reducing logging overhead, DDOS [51] enforces determinism in distributed systems and only requires logging message arrival times. Halfmoon has similar inspirations in stabilizing timestamps, and moves further to eliminate the logging overhead for either reads or writes.

Multi-versioning has been widely adopted in databases for concurrency control (MVCC) [91, 112, 116], where reads target old versions and writes install new versions. In the context of serverless computing, AFT [102] uses multi-versioning to provide read atomic isolation. The Halfmoon-read protocol applies the idea of existing works in a novel way to enable log-free exactly-once reads.

10 Conclusion

Halfmoon introduces novel optimizations to the log-based fault tolerance of SSFs. We design two logging protocols that enforce exactly-once semantics while providing log-free reads and writes, respectively. Instead of symmetrically logging every single read and write, our key insight is that it suffices to log *either* reads *or* writes, i.e., asymmetrically. We theoretically prove that our protocols are log-optimal. Therefore, Halfmoon pushes the overhead of log-based fault tolerance to its lower bound. We provide a criterion for choosing the right protocol for a given workload, and a pauseless switching procedure to switch protocols for dynamic workloads. Experiments show that Halfmoon achieves 20%–40% lower latency and 1.5–4.0× lower logging overhead than the state-of-the-art solution.

Acknowledgements

We thank the anonymous reviewers from both SOSP and TOCS and our SOSP shepherd, Ryan (Peng) Huang, for their insightful feedback.

References

[1] 2023. AWS Step Functions. Retrieved April 17, 2023 from https://aws.amazon.com/step-functions/

[2] 2023. Azure Durable Entities. Retrieved April 17, 2023 from https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities

[3] 2023. DeathStarBench. Retrieved April 17, 2023 from https://github.com/delimitrou/DeathStarBench/

[4] 2023. Functionbench. Retrieved April 17, 2023 from https://github.com/kmu-bigdata/serverless-faas-workbench

[5] 2023. Google Cloud Functions Triggers. Retrieved April 17, 2023 from https://cloud.google.com/functions/docs/calling

[6] 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing (Extended Version). Retrieved September 11, 2023 from https://tomquartz.github.io/files/SOSP23_Halfmoon_extended.pdf

[7] 2023. Logging in Azure Durable Functions. Retrieved April 17, 2023 from https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations

[8] 2023. Retrying Event-Driven Functions in Google Cloud. Retrieved April 17, 2023 from https://cloud.google.com/functions/docs/bestpractices/retries

[9] 2023. Sample Projects for AWS Step Functions. Retrieved April 17, 2023 from https://docs.aws.amazon.com/step-functions/latest/dg/create-sample-projects.html

[10] 2023. Serverless Examples. Retrieved April 17, 2023 from https://github.com/serverless/examples

[11] 2023. Serverlessbench. Retrieved April 17, 2023 from https://serverlessbench.systems/en-us/

[12] 2023. Statelessness of Google Cloud Functions. Retrieved April 17, 2023 from https://cloud.google.com/functions/docs/concepts/execution-environment

[13] 2023. Tutorial: Design and Implementation of a Simple Twitter Clone Using PHP and the Redis Key-Value Store. Retrieved September 11, 2023 from https://redis.io/topics/twitter-clone

[14] 2024. PostgreSQL. Retrieved October 20, 2024 from http://www.postgresql.org/

[15] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *USENIX OSDI*.

[16] Remzi Can Aksoy and Manos Kapritsos. 2019. Aegean: Replication beyond the client-server model. In *ACM SOSP*.

[17] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. 2021. Cloud-scale runtime verification of serverless applications. In *ACM SoCC*.

[18] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2023. Groundhog: Efficient request isolation in FaaS. In *EuroSys*.

[19] Özalp Babaoğlu and Keith Marzullo. 1993. Consistent global states of distributed systems: fundamental concepts and mechanisms. In *Distributed Systems* (2nd Ed.). ACM Press/Addison-Wesley Publishing Co., USA, 55–96.

[20] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. 2020. Virtual consensus in Delos. In *USENIX OSDI*.

[21] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. In *USENIX NSDI*.

[22] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed data structures over a shared log. In *ACM SOSP*.

[23] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, et al. 2021. Log-structured protocols in Delos. In *ACM SOSP*.

[24] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM TOSEM* 31, 3 (2022), 1–38.

[25] Ken Birman and Thomas Joseph. 1987. Exploiting virtual synchrony in distributed systems. In *ACM SOSP*.

[26] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1591–1604.

[27] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: Semantics for stateful serverless. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[28] Binbin Chen, Haifeng Yu, Yuda Zhao, and Phillip B. Gibbons. 2014. The cost of fault tolerance in multi-party communication complexity. *Journal of the ACM* 61, 3 (2014), 1–64.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2012. Spanner: Google's globally-distributed database. In *USENIX OSDI*.

[30] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos made transparent. In *ACM SOSP*.

[31] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *DEBS*.

[32] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *USENIX NSDI*.
[33] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. 2023. Automated verification of idempotence for stateful serverless applications. In *USENIX OSDI*.
[34] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-grained re-execution for efficient batched commit of distributed transactions. *vldb* 16, 8 (2023), 1930–1943.
[35] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *ACM ASPLOS*.
[36] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ACM ASPLOS*.
[37] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service. In *USENIX ATC*.
[38] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *EuroSys*.
[39] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. 2022. DGSF: Disaggregated GPUs for serverless functions. In *IPDPS*.
[40] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX ATC*.
[41] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *ACM SOSP*.
[42] Xinwei Fu, Dongyoon Lee, and Changwoo Min. 2022. {DURINN}: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *USENIX OSDI*.
[43] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ACM ASPLOS*.
[44] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Exploiting nil-externality for fast replicated storage. In *ACM SOSP*.
[45] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *ACM ASPLOS*.
[46] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM* 60, 7 (2017), 83–92.
[47] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).
[48] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.
[49] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2019. TxFS: Leveraging file-system crash consistency to provide ACID transactions. *ACM Transactions on Storage* 15, 2 (2019), 1–20.
[50] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *USENIX OSDI*.
[51] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. 2013. DDOS: Taming nondeterminism in distributed systems. *ACM SIGPLAN Notices* 48, 4 (2013), 499–508.
[52] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *popl* 3, OOPSLA (2019), 1–26.
[53] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful serverless computing with shared logs. In *ACM SOSP*.
[54] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ACM ASPLOS*.
[55] Ricardo Jiménez-Peris, Gustavo Alonso, and Bettina Kemme. 2003. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)* 28, 3 (2003), 257–294.
[56] Xin Jin, Zhen Zhang, Yunshan Jia, Yun Ma, and Xuanzhe Liu. 2024. SDCC: Software-defined collective communication for distributed training. *Science China Information Sciences* 67, 9 (2024), 192104.

[57] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for serverless functions. In *ACM SoCC*.
[58] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *ACM SOSP*.
[59] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing microservice applications on serverless, correctly. *popl* 7, POPL (2023), 367–395.
[60] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve: Execute-verify replication for multi-core servers. In *USENIX OSDI*.
[61] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *ACM ASPLOS*.
[62] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *USENIX OSDI*.
[63] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *EuroSys*.
[64] Eric Koskinen and Junfeng Yang. 2016. Reducing crash recoverability to reachability. In *ACM POPL*.
[65] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating function-as-a-service workflows. In *USENIX ATC*.
[66] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* (1979).
[67] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C–28, 9 (1979), 690–691.
[68] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565.
[69] Günter Last and Mathew Penrose. 2017. *Lectures on the Poisson Process*. Vol. 7. Cambridge University Press.
[70] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *vldb15* 13 (2022), 4023–4037.
[71] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. Dfix: Automatically fixing timing bugs in distributed systems. In *PLDI*.
[72] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just say no to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*.
[73] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. 2023. Performance bug analysis and detection for distributed storage and computing systems. *ACM Transactions on Storage* 19, 3 (2023), 1–33.
[74] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. Faasflow: Enable efficient workflow execution for function-as-a-service. In *ACM ASPLOS*.
[75] Barbara Liskov, Liuba Shrira, and John Wroclawski. 1991. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems* 9, 2 (1991), 125–142.
[76] John D. C. Little. 2011. Little's Law as viewed on its 50th anniversary. *Operations Research* 59, 3 (2011), 536–549.
[77] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 677–691.
[78] Xuanzhe Liu, Gang Huang, and Hong Mei. 2009. Discovering homogeneous web service community in the user-centric web environment. *IEEE Transactions on Services Computing* 2, 2 (2009), 167–181.
[79] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. 2023. LegoDroid: Flexible Android app decomposition and instant installation. *Science China Information Sciences* 66, 4 (2023), 142103.
[80] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM SOSP*.
[81] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A partially ordered shared log. In *USENIX OSDI*.
[82] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A partially ordered shared log. In *USENIX OSDI*.
[83] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *EuroSys*.

[84] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. 2022. Sift: Using refinement-guided automation to verify complex distributed systems. In *USENIX ATC*.
[85] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *ACM SOSP*.
[86] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *ACM SOSP*.
[87] Kostas Meladakis, Chrysostomos Zeginis, Kostas Magoutis, and Dimitris Plexousakis. 2022. Transferring transactional business processes to FaaS. In *WoSC*.
[88] John M. Mellor-Crummey and Michael L. Scott. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices* 26, 7 (1991), 106–113.
[89] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.
[90] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM SOSP*.
[91] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *ACM SIGMOD*.
[92] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *USENIX ATC, FREENIX Track*. 183–191.
[93] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *USENIX ATC*.
[94] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying state-machine replication through randomization. In *ACM SOSP*.
[95] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *ACM SOSP*.
[96] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2017. Scalable replay-based replication for fast databases. *vldb10* 13 (2017), 2025–2036.
[97] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the {OmniTable} way. In *USENIX OSDI*.
[98] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A transparent auto-scaling cache for serverless applications. In *ACM SoCC*.
[99] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM* 64, 5 (2021), 76–84.
[100] Srinath T. V. Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *USENIX OSDI*.
[101] Simon Shillaker and Peter Pietzuch. 2020. FAASM: Lightweight isolation for efficient stateful serverless computing. In *USENIX ATC*.
[102] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A fault-tolerance shim for serverless computing. In *EuroSys*.
[103] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *vldb* 13, 12 (2020), 2438–2452.
[104] Yang Tang and Junfeng Yang. 2020. Lambdata: Optimizing serverless computing by making data intents explicit. In *IEEE CLOUD*.
[105] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. 2020. {Near-Optimal} latency versus cost tradeoffs in {Geo-Distributed} storage. In *USENIX NSDI*.
[106] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ACM ASPLOS*.
[107] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2023. When is it safe to run a transactional workload under read committed? *ACM SIGMOD Record* 52, 1 (2023), 36–43.
[108] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–24.
[109] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: Fault tolerance off the critical path. In *ACM SOSP*.
[110] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the parallels between Paxos and Raft, and how to port optimizations. In *ACM PODC*.

[111] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. 2017. vCorfu: A cloud-scale object store on a shared log. In *USENIX NSDI*.

[112] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *USENIX NSDI*.

[113] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing.

[114] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic Review. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–61.

[115] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional causal consistency for serverless computing. In *ACM SIGMOD*.

[116] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *vldb* 10, 7 (2017), 781–792.

[117] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In *USENIX OSDI*.

[118] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-driven automated invariant learning for distributed protocols. In *USENIX OSDI*.

[119] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *ACM SoCC* .

[120] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ACM ASPLOS*.

[121] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *USENIX OSDI*.

[122] Xinhao Yuan and Junfeng Yang. 2020. Effective concurrency testing for distributed systems. In *ACM ASPLOS*.

[123] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *USENIX OSDI*.

[124] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *ACM SoCC*.

[125] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A programming framework for serverless computing. In *ACM SoCC*.

[126] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2022. Blockaid: Data access policy enforcement for web applications. In *USENIX OSDI*.

[127] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *ACM SOSP*.

[128] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Non-intrusive failure reproduction for distributed systems using the partial trace principle. In *ACM SOSP*.

[129] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *ACM SOSP*.

[130] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. BeeHive: Sub-second elasticity for web services with Semi-FaaS execution. In *ACM ASPLOS*.

[131] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *ACM SOSP*.

[132] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *PLDI*.

Received 19 May 2024; revised 29 October 2024; accepted 17 January 2025