Paxos Made Parallel

Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou and Li Zhuang
Microsoft Research Asia

Abstract

Standard state-machine replication involves consensus on a sequence of totally ordered requests through, for example, the Paxos protocol. Serialized request processing seriously limits our ability to leverage prevalent multi-core servers. This tension between concurrency and consistency is *not* inherent because the total-ordering of requests is merely a simplifying convenience that is unnecessary for consistency. Replicated state machines can be made *parallel* by consensus on partial-order traces, rather than on totally ordered requests, that capture causal orders in one replica execution and to be replayed in an order-preserving manner on others. The result is a new multi-core friendly replicated state-machine framework that achieves strong consistency while preserving parallelism in multi-threaded applications. On 12-core machines with hyper-threading, evaluations on typical applications show that we can scale with the number of cores, achieving up to a 16 times throughput increase over standard replicated state machines.

1 Introduction

Server applications that power on-line services typically run on a cluster of *multi-core commodity* servers. These applications must leverage an underlying multi-core architecture for highly concurrent request-processing, but must also often resort to replication to guard against the failure of commodity components. To guarantee consistency, replicas in the standard replicated state-machine approach [26, 36] start from the same initial state and process the same set of requests in the same *total order* deterministically. They use a consensus protocol to agree on each request in a total order that implies sequential request processing, leading to performance loss on a multi-core machine as many requests are in practice non-conflicting and can be processed concurrently.

We present Tribble, a new replication framework that maintains replica consistency while preserving concurrency in request processing, thereby refuting the misconception that serialized execution is necessary for state-machine replication. Tribble processes multiple requests concurrently on replicas; application writers provide processing logic that uses a set of *synchronization primitives* (e.g., locks and semaphores) for coordinating concurrent access to shared data. Tribble achieves replica consistency with the following key techniques. First, Tribble has a *primary* replica process requests concurrently, with others as *secondary* replicas that concurrently *replay* the primary execution. Second, Tribble uses a consensus protocol, such as Paxos, to maintain consistency even when replicas fail and the primary changes. Rather than agreeing on a sequence of total-ordering requests, Tribble replicas agree on a partially ordered *trace* that captures causal dependencies among synchronization events that occur in request processing. Replay of the trace on a secondary preserves these causal dependencies and the replica reaches the same consistent state as the primary, while still allowing for concurrency during replay.

Tribble is the first practical replication framework that approaches the ideal of achieving both high reliability through Paxos and high performance through concurrent multi-threaded execution. We design Tribble to be the new "replicated state-machine" approach that is multi-core friendly. To evaluate whether Tribble preserves concurrency in request processing under various circumstances and to understand its overhead, we have further developed a set of micro-benchmarks and identified several representative applications, including a global lock service, a thumbnail service, a key/value store, a simple file system, and Google's LevelDB. Our experiments on 12-core servers with hyper-threading have shown that applications achieve as high as 16 times the throughput on Tribble when compared to standard replicated state-machine.

The rest of the paper is organized as follows. Section 2 presents an architectural overview of Tribble. Section 3 details how Tribble uses replica consensus to agree on a trace. Section 4 presents how Tribble captures causal orders in traces during execution and replays execution while respecting the causal orders. We discuss state divergence and Tribble application scope in Section 5. Section 6 presents experimental results on both our micro-benchmarks and representative applications. We survey related work in Section 7 and conclude in Section 8.

2 Overview

We consider on-line service applications that serve incoming client requests with request *handler*s and a thread pool with a fixed number of threads to process requests. In contrast to standard state-machine replication, where totally ordered requests execute sequentially, request handlers in Tribble are executed concurrently using a set of standard synchronization primitives to coordinate their access to shared data. Each handler executes *deterministically*, where Tribble requires that the ordering of synchronization events be the only source of nondeterminism. Tribble degrades into state-machine replication if each request handler uses the same lock to protect the entire execution.

2.1 Causal-Order Traces

Figure 1 shows how request handlers use synchronization primitives. Two threads are working on two different requests, where lock *L* is used to coordinate the access to shared data. `Lock` and `Unlock` calls introduce causal dependencies between the two threads, which are shown as edges. Because each replica has the same request-handler code, an execution is uniquely determined by the set of incoming requests with their assignments to threads, as well as the synchronization events and their causal orders, which collectively constitute a *trace*. In a trace, a *synchronization event* is identified by its thread id and a local clock that increases for each local event; a causal order between two synchronization events is recorded as a directed *causal edge*, that is identified by a pair of event identifiers. As shown in Figure 1, a causal edge exists from the `Unlock` event $(t_1, 3)$ to the `Lock` event $(t_2, 2)$, where the `Unlock` event must precede the `Lock` event.

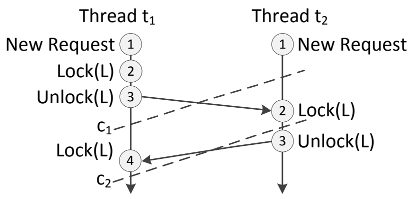


Figure 1: Request handlers & synchronization primitives.

The trace is *growing* as Tribble continuously handles incoming requests. We pick an event in a thread as a *cut point* for that thread. The collection of cut points, one for each thread, defines a *cut* on a trace. A cut includes all events up to the cut points in the threads, as well as the causal orders among them. A trace tr_p is considered a *prefix* of another trace tr if tr_p is a cut on tr. A cut is *consistent* if, for any causal edge from event e_1 to e_2 in the trace, e_2 being in the cut implies that e_1 is also included in the cut. An execution reaches only consistent cuts. Figure 1 shows two cuts c_1 and c_2, where c_1 is consistent, but c_2 is inconsistent because event $(t_1, 4)$ is in the cut, but $(t_2, 3)$ is not.

2.2 Execution, Consensus, and Replay

Figure 2 compares Tribble's processing steps with those in state-machine replication. Both use a consensus protocol such as Paxos as a basic building block. In its simplest form, the consensus module allows clients to *propose* values in consensus instances and notifies clients when a value is *committed* in an instance. The consensus protocol ensures that a value committed is the value that was proposed in that instance and that no other values are committed in the same consensus instance, despite possible failure of a minority of replicas.

With state-machine replication, replicas send client requests r_i as proposals to the consensus module (step ①), reach consensus on a sequence of requests (step ②), and process those requests in this order (step ③).

A Tribble primary executes first by processing requests concurrently and records causal orders in a trace tr_i (step ①). A primary periodically proposes the current cut as the value for the next consensus instance (step ②). Even in the presence of multiple primary replicas, replicas can reach consensus to commit on a sequence of traces (step ③). Once committed, a trace can be replayed on secondary replicas (step ④).

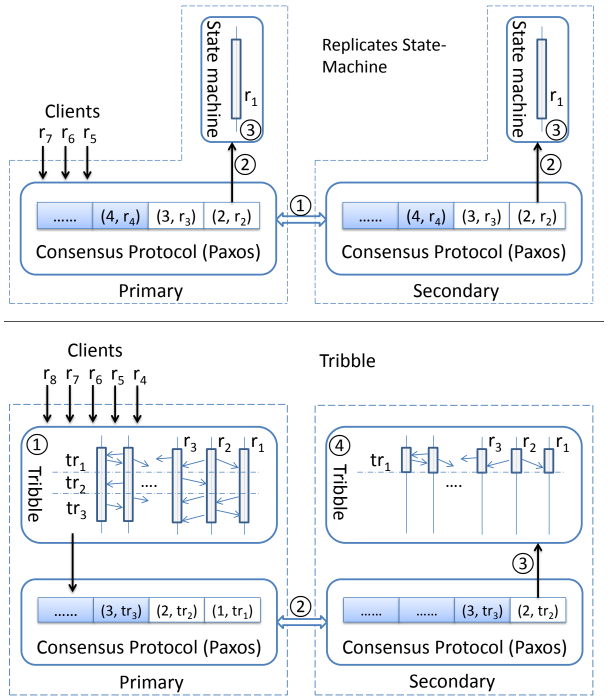


Figure 2: State-machine replication vs. Tribble.

Tribble executes on the primary before a consensus is reached, whereas consensus precedes execution in state-machine replication. This has a subtle implication on defining when the processing of a request is completed and when the service can respond to clients. In state-machine replication, request processing starts after reaching a consensus on that request and thus the primary can respond to clients right after execution. In Tribble, a primary cannot respond to clients right after finishing processing a request, but must wait until a trace containing the processing of that request and all its depending events has been committed in a consensus instance (step ③). However, the primary does not have to wait for the completion of the replay on a secondary.

Tribble reaches consensus on a sequence of traces, while replicated state machines agree on a sequence of requests. While requests in different consensus instances are independent, traces in different consensus instances are not: replicas must reach consensus on a sequence of growing traces that satisfy the *prefix condition*, where a trace committed in instance i is a prefix of the trace committed in instance i+1. The prefix condition must also hold during a primary change: a new primary must first learn what has been committed in previous consensus instances and replay to the trace of the last completed consensus instance, before it can continue execution to create longer traces as proposals to subsequent consensus instances.

2.3 Correctness and Concurrency

Correctness of Tribble can be defined as equivalent to a valid single-machine execution, and follows from the following three properties: (i) **Consensus:** all replicas reach a consensus on a sequence of traces, (ii) **Determinism:** a replica execution that conforms to the same trace reaches the same consistent state, and (iii) **Prefix:** a trace committed in an earlier consensus instance is a prefix of any trace committed in a later one. The first two properties ensure consistency across replicas, while the third property ensures that the sequence of traces constitute cuts on the same valid single machine execution. How Tribble satisfies the consensus and prefix properties is the subject of Section 3, while Section 4 presents how Tribble achieves the determinism property.

Compared to state-machine replication, where request processing is serialized, Tribble preserves concurrency in request processing: a primary processes requests concurrently using Tribble synchronization primitives, while recording causal orders that matter to the execution; a secondary replays the execution by respecting recorded causal orders and preserves the inherent parallelism of an application. Tribble introduces overhead in both execution of a primary for recording causal orders and execution of secondary replicas for respecting those orders. Our evaluations in Section 6 show that the overhead is manageable and higher concurrency leads to significant performance improvement on multi-core machines.

3 Consensus on a Growing Trace

Tribble uses Paxos to ensure the consensus and prefix properties in Section 2.3. As in state-machine replication, Tribble manages a sequence of consensus instances with the Paxos protocol [28] so replicas can agree on a trace. Across instances, Tribble must also ensure the prefix property. In this section, we describe how Tribble reaches consensus on a growing trace.

3.1 Consensus in Tribble

Tribble uses the classic two-phase Paxos protocol that relies on a failure detection and leader election module to detect replica failures and instruct a replica to become a leader (e.g., after suspecting that the current leader has failed). To become a new leader, a replica must execute the first phase of the Paxos protocol. When multiple replicas compete to become the new leader, their *ballot numbers* decide the winner. In this phase, the new leader must learn any proposal that could have been committed and proposes the same values to ensure consistency. A leader carries out the second phase to get a proposal committed in a consensus instance and does so in a round-trip when a majority of the replicas cooperate.

Beyond standard consensus, Tribble makes two noteworthy design decisions. First, Tribble has at most one *active* consensus instance at any time. A primary proposes to an instance only after it learns that any earlier instance has a proposal committed. This decision greatly simplifies the design of Tribble in the following three ways. First, Tribble does not have to manage multiple active instances or deal with “holes” where a later instance reaches an agreement before an earlier instance does. Second, the decision makes it easy to guarantee the prefix condition: during primary changes, the new primary simply learns the trace committed in the last instance, replays that trace, and uses that trace as the starting point for further execution. Finally, the design enables a simple optimization where a proposal to a new instance can contain not the full trace, but only the additional information on top of the committed trace in the previous instance. There is no risk of mis-interpretation because the base trace has already been committed.

This simplification does *not* come at the expense of performance. Normally, when a primary is ready to propose to instance $i + 1$, the proposal for instance i has already been committed. Even when a primary wants to propose to instance $i + 1$ before a consensus is reached in all previous instances, the primary can simply piggyback all the not-yet-committed proposals for previous instances in the proposal to the new instance: a secondary accepts the proposal for instance $i + 1$ only if it accepted the proposal for the previous instances.

Second, Tribble co-locates the primary with the leader in Paxos. This is a natural choice because the primary plays a similar role in Tribble as the leader in Paxos: they are both ideally unique and stable. Also primary or leader changes both affect performance adversely. To facilitate such co-location, Tribble's Paxos implementation exposes leader changes in the interface, in addition to the standard Paxos interface, as follows. `Propose(i, p)` is used to propose `p` to instance `i`; `OnCommitted(i, p)` allows Tribble to provide a callback to be invoked when proposal `p` is committed in instance `i`; `OnBecomeLeader()` is the callback to be invoked when the local replica becomes the leader in Paxos; `OnNewLeader(r)` is the callback to be invoked when another replica `r` becomes the new leader.

Normally, a single primary is co-located with the leader, processes client requests, and periodically creates traces as proposals for consensus. Those traces satisfy the prefix condition naturally as they are cuts of the same execution. A secondary does not have to finish replaying before responding to the primary; a secondary replays only to catch up with the primary in order to speed up primary changes or to serve non-updating queries.

In Tribble, leader changes trigger primary changes. A new leader in Paxos will become the new primary; the old primary will downgrade itself to a secondary when a new leader emerges. In the (rare) cases where there are multiple leaders, multiple replicas might assume the role of the primary. The consensus through Paxos ensures correctness by choosing only one trace in such cases, although executions that are not selected are wasted and require rollbacks.

Promotion to primary. Our Paxos implementation signals `OnBecomeLeader()` when the local replica completes phase 1 of the Paxos protocol (across all instances) without encountering a higher ballot number. In that phase, the new leader must have learned all instances that might have a proposal committed and will re-execute phase 2 to notify all replicas about those proposals committed. The replica learns the trace committed in the last instance, replays that trace, and then switches from a secondary to a primary. Once the replica becomes the primary, it starts executing from the state corresponding to the last committed trace to create new proposals, thereby ensuring the prefix condition.

Concurrency and inconsistent cut. A new primary must replay the last committed trace to the end, which is feasible as long as the trace forms a consistent cut. Even though execution of a primary results in a consistent cut at any time, concurrent thread executions might log synchronization events and their causal edges in an order that is different from the execution order, thereby leading to the possibility of having an inconsistent cut as the trace for consensus. If an event e gets logged before another event e' that is causally ordered before e. A secondary would not be able to replay e fully because it would be blocked waiting for e'. This is particularly problematic when a secondary is promoted to a primary. In that case, the promotion is forever blocked. Instead of making each cut consistent, Tribble defines the last consistent cut contained in a trace as the meaning of the proposal. The residual of the trace after that consistent cut is ignored in the case of primary changes. Tribble attaches a ballot number in each proposal to indicate a primary change because a new leader/primary will use a higher ballot number.

Primary demotion. Our Paxos implementation signals `OnNewLeader(r)` whenever it learns a higher ballot number from some replica `r`. If the local replica is the primary, but is no longer a leader, the replica must downgrade itself to a secondary. As in Tribble where a primary executes *speculatively*, a downgrading replica must also roll back its execution to the point of the last committed trace. One way to roll back is through checkpointing, as described next.

3.2 Checkpointing and Garbage Collection

Tribble supports checkpointing (i) to allow a replica to recover from failures, (ii) to implement rollback on a downgrading replica, and (iii) to facilitate garbage collection. Although it is sometimes possible for an application writer to write application-specific checkpointing logic, Tribble resorts to a general checkpointing framework to alleviate this burden.

Having the primary checkpoint periodically during its execution turns out to be undesirable for several reasons. First, the primary's current state is speculative and might have to be rolled back; an extra mechanism is needed to check whether a checkpoint eventually corresponds to some committed state. Second, the primary is on the critical path of request processing, being responsible for taking requests, processing them, and creating traces for consensus. Any disruption to the primary leads directly to service unavailability. In contrast, due to redundancies needed for fault tolerance, a secondary can take the responsibility of creating checkpoints without significant disruptions, by coordinating with the primary.

Checkpointing cannot be done on a state where a request has not been processed completely because Tribble does not have sufficient information for a replica to continue processing an incomplete request when re-starting from that checkpoint. When Tribble decides to create a checkpoint, the primary sets the *checkpoint* flag, so that all threads will pause before taking on any new request. Threads working on background tasks (e.g., for compaction in LevelDB) must also pause (and resume) at a clean starting point. Instead of taking the checkpoint directly when all threads are paused, the primary marks this particular cut (as a list of the local virtual clock values for each thread) and passes the checkpoint request with the cut information in the proposal for consensus. A secondary receiving such a request waits until the replay hits the cut points in the checkpoint request and creates a snapshot through a checkpointing callback to the application. Some policy is put in place to decide how often to checkpoint and which secondary should create a snapshot. Once created, a secondary continues its replay and copies the checkpoint in the background to other replicas. When a checkpoint is available on a replica, any committed trace before the cut points of that checkpoint is no longer needed and can be garbage collected.

4 Capturing and Executing Traces

In addition to consensus, Tribble leverages record (on a primary) and replay (on a secondary) to achieve the determinism property in Section 2.3. The unique setting in Tribble imposes two requirements that make previous approaches of record and replay insufficient. First, Tribble demands the ability of *mode change* from replay to live execution, when a secondary is promoted as the primary. As a result, resources cannot be faked during replay, as is often done in traditional record and replay systems, such as R2 [19] and Respec [30]. For example, a record and replay tool often records the return value of a `fopen` call during recording and simply returns the value without executing during replay. Because the `file` resource is faked, the system *cannot* switch from replay to live execution after replaying `fopen`. The subsequent calls (e.g., `fread`, `fwrite`, and `fclose`) on this resource would fail without actually executing `fopen` first.

Second, Tribble demands *online replay* [30]: both record (on a primary) and replay (on a secondary) need to be carried out *efficiently* as they both affect the performance of a live system. This is in contrast to *offline replay* for scenarios such as debugging. In Tribble, replay performance should be comparable to record performance so that a secondary can catch up to ensure stable system throughput. In particular, Tribble must enable concurrency during replay.

None of the previous record-and-replay systems offer a satisfactory solution for Tribble to support both mode change and online replay, although many of the concepts and approaches are useful to Tribble. To support mode change, replay must maintain system resources faithfully by re-executing operations on resources. To ensure that these executions produce the same effect, Tribble must ensure the appropriate execution order of these operations. Tribble therefore provides wrappers for synchronization operations in order to record their order of execution, similar to RecPlay [34] and JaRec [17]. This way, the difference between record and replay lies purely in the working mode of the wrappers, making it easy to switch from replay to live execution while being transparent to applications. By wrapping basic synchronization primitives, Tribble introduces enough nondeterminism in a programming-friendly way to allow sufficient concurrency. To support online replay, we intentionally avoid providing programming abstractions that are not record-replay friendly, such as OS upcalls, and decide against developing a complete deterministic record and replay tool for arbitrary full-fledged concurrent programs due to the inherent complexity and performance overhead.

```
class TribbleLock;
class TribbleReadWriteLock;
class TribbleCond;
class TribbleRequest {
   virtual int Execute(Tribble* rsm);
   virtual ostream& Marshall(ostream& os);
   virtual istream& UnMarshall(istream& is);
   . . . }
class Tribble {
   virtual bool Start(int numThread);
   virtual int WriteCheckPoint(ostream& os);
   virtual int ReadCheckPoint(istream& is);
   . . .
   int AddTimer(Callback cb, int interval);
}
```

Figure 3: Programming API in Tribble.

4.1 Application Model and Assumptions

Tribble supports a constrained application model, with its programming API shown in Figure 3. A programmer builds an application by inheriting the `Tribble` and `TribbleRequest` classes. The processing logic for a request is encoded in an `Execute` function as a request handler. The `Tribble` class implements initialization and also the functions used for checkpointing, as described in Section 3.2. During initialization, the application can add background tasks such as garbage collection using the `AddTimer` method, which implicitly creates a background thread. Multiple instances of request handlers and background tasks might be executed concurrently, using built-in Tribble synchronization primitives (e.g., `TribbleLock`) to coordinate.

Just as all request processing must be determinis-

```
int last_thread_id;
int last_local_clock;
MarkCausalEdge() {
   CausalEdge ce;
   ce.SrcThreadId = last_thread_id;
   ce.SrcThreadClock = last_local_clock;
   ce.DstThreadId = my_thread_id;
   ce.DstThreadClock = ++my_local_clock;
   Runtime::MarkCausalEdge(ce);
   last_thread_id = my_thread_id;
   last_thread_clock = my_local_clock; }
Lock() {
   AcquireLock(real_lock);
   MarkCausalEdge(); }
Unlock() {
   MarkCausalEdge();
   ReleaseLock(real_lock); }
```

Figure 4: Wrappers for `Lock` and `Unlock`.

tic in the standard state-machine replication, we assume that there is no non-determinism other than the order of the Tribble synchronization operations. This assumption makes mode switch easy. We discuss the cases where such an assumption might be violated, with potential countermeasures, in Section 5.

4.2 Capturing Causal Orders

In Tribble, a primary captures causal orders among synchronization events during its execution, while a secondary replays the execution by preserving those causal orders. To capture causal orders, Tribble implements a set of wrappers for commonly used synchronization primitives. Figure 4 shows the pseudo code of the wrappers for `Lock` and `Unlock`. The wrapper maintains a real mutex lock `real_lock` and the identifier for the last `Lock`/`Unlock` event. For each `lock` event, Tribble creates a causal edge from the recorded last `Unlock` event. The wrappers must ensure atomicity of an event and its causal edge logging for faithful recording. Intuitively, an additional lock can be used to combine the event invocation and logging for atomicity. In this case, the mutex lock already ensures the atomicity when applications use `Lock` and `Unlock` properly.

The example in Figure 4 is deceivingly simple. We use `TryLock` to illustrate the subtle complications that could arise in implementing those wrappers. A `TryLock` call does not block even if it fails to obtain the lock. We use `TryLock(F)` to refer to a `TryLock` event that returns failure and `TryLock(T)` to a `TryLock` event that returns success. The subtle complication is related to two characteristics of `TryLock`: the non-blocking nature that affects thread safety and the partial-order nature of causal orders for better replay concurrency. The same type of techniques can be applied to other synchronization primitives, such as Readers/Writer locks and semaphores, as well as covering the case where an application inappropriately invokes `Unlock` multiple times.

```
RealLock data_lock;
Lock() {
   AcquireLock(real_lock);
   AcquireLock(data_lock);
   ...
   ReleaseLock(data_lock); }
TryLock() {
   AcquireLock(data_lock);
   bool ret = TryAcquireLock(real_lock);
   ...
   MarkCausalEdge();
   ReleaseLock(data_lock);
   ... }
```

Figure 5: Wrappers for `Lock` and `TryLock`.

Thread safety through detection and retry. With `TryLock`, Tribble can no longer rely on the `real_lock` for thread safety because it is possible to have concurrent `TryLock` calls, even along with a concurrent `Lock`. Tribble therefore has to introduce a second mutex lock `data_lock` to protect the context in each lock wrapper, as shown in Figure 5. Now a `Lock` operation needs to acquire both `real_lock` and `data_lock` atomically. This cannot be done because acquiring a lock is a blocking call. While `TryLock` can acquire `data_lock` first, `Lock` must acquire the `real_lock` first because acquiring `data_lock` first and holding it when it acquires the `real_lock` would cause deadlocks: no one can acquire the `data_lock` when this thread blocks on the `real_lock`. However, the implementation in Figure 5 is not yet thread safe. Figure 6 shows a case where thread safety is violated. We consider a thread interleaving, where thread t_1 calls `Lock`. Thread t_2 executes `TryLock` completely after t_1 acquires the `real_lock`, but before it acquires the `data_lock`. t_2 will get the context before the completion of `Lock` because t_1 has not updated the context. This leads to the false causal orders (in dotted arrows) shown on the right of Figure 6, where the `TryLock` event is falsely considered to be causally ordered after the `Unlock` event. In reality, it should be causally ordered after the `Lock` event in order to return false correctly. The root cause of the problem is the loss of atomicity between acquiring the `real_lock` and updating the context accordingly.

Rather than *preventing* such bad cases from happening, Tribble uses a *detection and retry* approach to address this problem, as shown in Figure 7. We add a flag `locked` to indicate that the lock is being held. The code

```
t1                     t2
AcquireLock(
    real_lock);
                       AcquireLock(data_lock);
                       ...
                       MarkCausalEdge();
                       // outdated data !!!
                       ...
AcquireLock(           ReleaseLock(data_lock);
    data_lock);
…
ReleaseLock(
    data_lock);
```

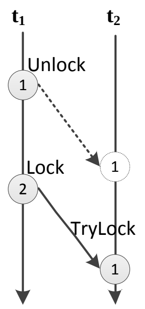


Figure 6: Thread safety violation in `TryLock`.

```
bool locked;
Lock() {
   AcquireLock(real_lock);
   AcquireLock(data_lock);
   . . .
   locked = true;
   ReleaseLock(data_lock); }
Unlock() {
   AcquireLock(data_lock);
   . . .
   locked = false;
   ReleaseLock(data_lock);
   ReleaseLock(real_lock); }
TryLock() {
   bool ret = TryAcquireLock(real_lock);
   AcquireLock(data_lock);
   while (ret == false && !locked) {
      ReleaseLock(data_lock);
      ret = TryAcquireLock(real_lock);
      AcquireLock(data_lock);
   }
   if (ret) locked = true;
   . . .
   ReleaseLock(data_lock); }
```

Figure 7: Thread safety through detection and retry.

of `TryLock` checks the consistency between the return value of the call and the flag. Any inconsistency indicates a race condition, causing the `TryLock` to re-execute.

4.3 Processing Requests on a Secondary

A secondary in Tribble processes requests while respecting causal orders captured in a primary's execution. When processing requests on a secondary, a synchronization event e is triggered only after the execution of all events causally ordered before e. In the case where e in thread t has to wait for the execution of an event e' in thread t', Tribble pauses thread t just before e and registers with t' to have it signal t after it executes e'. This way, a secondary attempts to execute in the same causal orders on the same set of threads. Due to differences in thread interleaving, a replay on a secondary might introduce extra waiting. For example, in the execution on a primary, thread t_1 might get a lock before t_2; during replay, t_2 might be scheduled first and gets to the point of acquiring the lock before t_1 does. In this case, t_2 still has to wait for t_1 to respect the same ordering as in the execution of the primary. Causal dependencies captured in causal edges decide the level of concurrency during replay. To achieve better performance and concurrency, we discuss a couple of implementation tradeoffs and optimizations below in Section 4.4.

4.4 Tradeoffs and Optimizations

`TryLock`. With only `Lock` and `Unlock`, causal orders are simply the total order of all events on the same lock. Such total ordering turns out to be overly stringent for mutex locks that support `TryLock` (and similarly for readers/writer locks and semaphores). Figure 8 shows an example with three threads. If Tribble imposed a total order on all those events, it would have to record edges (A, B, C, D). For correctness only, the causal orders between the `TryLock(F)`s in t_2 and the `TryLock(F)` in t_3 are unnecessary, while all these `TryLock(F)`s have causal orders with `Lock` and `Unlock` on t_1 (edges A, D, W, X, Y, Z). For example, the execution would be equivalent even if the `TryLock(F)` in t_3 executes before the two `TryLock(F)`s in t_2, as long as it is after the `Lock` in t_1 and before the corresponding `Unlock`.

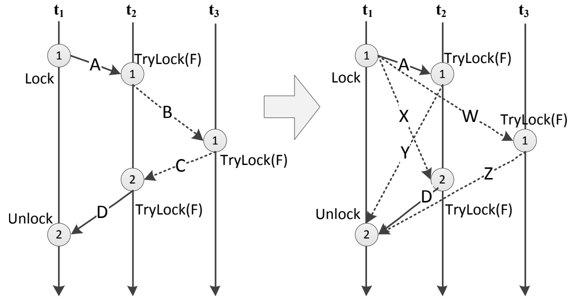


Figure 8: Total order vs. partial order.

To capture these causal orders precisely, Tribble's implementation tracks not only the last `Lock`, `TryLock(T)`, and `Unlock` events, but also all the `TryLock(F)` events before an `Unlock` event that releases the lock that cause those `TryLock` calls to fail. A `TryLock(F)` event is causally ordered after the preceding `Lock` or `TryLock(T)` event. An `Unlock` event is causally ordered after all preceding `TryLock(F)` events that fail due to the corresponding `Lock` or `TryLock(T)` event. Capturing the precise partial order helps improve replay concurrency, but often at the expense of increased complexity in capturing those causal orders while sometimes also increasing the total number of events.

Removing unnecessary causal edges. In the implementation, Tribble makes a simplifying assumption that all replicas initiates a thread pool of the same number of threads. With this assumption, causal orders within each thread are preserved. In the example of Figure 8, causal edge X is unnecessary because it follows from causal edge A and the intra-thread causal order in thread t_2. The same is true for causal edge Y. Tribble remove unnecessary causal edges if the causal order indicated by an edge has been implicitly decided from other edges.

5 The Trouble with Tribble

Tribble is no panacea and does have limitations, which we elaborate here.

Guarding against state divergence. Tribble request handlers must contain no sources of non-determinism beyond synchronization events. We discuss several cases where this requirement is violated, causing replica-state divergence.

An application could use non-deterministic APIs such as those related to random numbers and time. Because Tribble achieves replica consistency via replay on a secondary, it is straightforward to extend the current implementation to cover these simple sources of non-determinism. The results of those calls during the execution of the primary will be added to the trace, so that a secondary can simply use those values during replay to maintain consistency with the primary.

Developers could potentially write applications that contain races and other concurrency problems, causing replicas to diverge. Some races are bugs in applications; others might actually be *benign* in that they do not affect correctness. For example, an application that needs highly concurrent appends to a file might allow concurrent appends to be non-deterministic in terms of the order of those appends for better performance. A replay on a secondary might cause concurrent appends to result in different orders, causing state divergence. To make such an application deterministic, the application must serialize append calls, leading to performance loss. We have not seen such cases in the applications we have studied, but this example shows the limitation of Tribble in terms of exposing full concurrency.

A developer might also introduce data races that are benign to the application but hazardous to Tribble. Figure 9 shows the `GetInstance` member function of a Singleton class, which causes a secondary to diverge from the primary. On the primary, thread t_1 executes `GetInstance` before t_2 and creates a new singleton object. However, during replay, if t_1 lags behind and t_2 executes `GetInstance` before t_1, so that t_2 will enter the code to create a new singleton object. This violates Tribble's assumption that each thread executes the same sequence of synchronization events and so the replay fails.

```
Singleton* ptr;
Lock lock;
Singleton* GetInstance() {
   if (!ptr) {
      AcquireLock(lock);
      if (!ptr) ptr = new Singleton;
      ReleaseLock(lock);
   }
   return ptr;
}
```

Figure 9: Pseudo code for a singleton class.

The root cause of this problem is non-determinism that affects the control flow of an execution. This benign race has to be prevented; for example, by using a Readers-Writer lock to protect the pointer or to create all singletons in the main thread before creating worker threads. We plan to investigate the possibility of developing a tool to flag such problems automatically, possibly leveraging previous work on detecting concurrency bugs [35, 34, 20, 40]. The wrapping and logging for record and replay in Tribble, as well as its checkpointing facility, offer a powerful tool for debugging those problems. We have built a *diff* tool to compare the original execution on a primary and its replay on a secondary to detect divergence. We are also building an implementation model checker and have found that the simple interface helps greatly in building such tools. Another interesting approach is to tolerate such divergence through verification and re-execution, following the speculative Execute-Verify approach in, for example, Respec [30] and Eve [22].

Database replication. Even though the Tribble's idea of introducing higher concurrency by preserving only partial orders applies to other domains such as database replication, Tribble's approach is not necessarily always ideal. Database engines use locks extensively, especially those low-overhead interlocks. Handling those locks appropriately as Tribble does currently is likely to introduce significant overhead. Fundamentally, for database systems, defining partial orders at the coarse granularity of transactions is a better design choice, as transactions are natural atomic units in such systems, compared to defining partial orders on synchronization operations.

6 Evaluation

We have implemented Tribble with about 30,000 lines of C++ code, in which 17,500 lines are for implementing Paxos and common libraries for RPC, logging, and so on. The rest is almost equally divided into the implementation of the wrappers for synchronization primitives, of the runtime support for replay, and test cases. We use a combination of real applications and micro-benchmarks to evaluate the following aspects of Tribble: (i) How well does Tribble scale with the number of cores, especially compared to the replicated state-machine approach? (ii) How much overhead does Tribble introduce compared to native execution without record and replay? (iii) What factors are significant to Tribble's relative performance with respect to its replicated state-machine and native counterparts? (iv) How do queries perform under different semantics? And (v) How well does Tribble cope with checkpointing, primary changes, and replica recovery?

6.1 Experimental Setup

All experiments are conducted on 12-core machines with hyper-threading, 72 GB memory, 3 SCSI disks with RAID5 supports, and interconnected via 40 Gbits switch. We run applications on a group of three replicas to tolerate one fail-stop failure, with enough clients submitting requests so that the machines are fully loaded. Requests are batched to reduce the communication cost between clients and primary.

6.2 Real Applications Performance

We have built or ported a set of real applications on top of Tribble and found it easy to use its simple interface. We first adapt each application to scale well on a single multi-core machine and then port it to Tribble. The main effort during porting has been to replace synchronization primitives used in applications with those supported by Tribble. Mutex locks with try-locks, semaphores, and readers-writer locks are mostly sufficient for those applications. We did have to replace some more efficient inter-locks with less efficient mutex locks, and find additional sources of non-determinism.

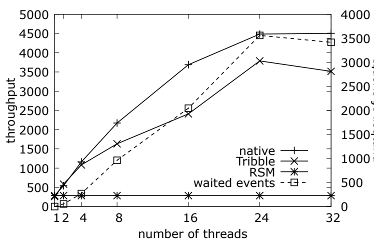
Thumbnail server is an existing application that manages picture thumbnails. It contains an in-memory hash table to store metadata and an in-memory cache to store thumbnails, using a set of keys to protect these data structures. In each request, it computes the thumbnail of a picture and obtains locks to update data structures related to the thumbnail. **Lock server** is a distributed lock service similar to Chubby [10] that was previously built on top of a replicated state-machine library. According to the report on Chubby [10], we create a workload with 90% of requests for renewing leases of locked files and the rest as "create" or "update" operations on locked files. File sizes vary from 100 bytes to 5k bytes. In the **file system workload** experiment, we measure performance of synchronized random read/write on 64 files, each with a size of 128 MB. Each request is 16 KB with 2:8 read/write split. **LevelDB** [18] is a fast key-value store library that provides an ordered mapping from string keys to string values. Data is stored sorted in skip lists with $O(\log n)$ lookup time. The database is divided into 256 slices with one lock for each slice. **Kyoto Cabinet**'s HashDB [25] is a lightweight hash database library whose database is divided into 1024 slices with each slice protected by a readers-writer lock. **Memcached** [1] is another in-memory key-value store for object caching. We build replicated storage services on top of Level DB, Kyoto Cabinet and Memcached by wrapping the libraries they provide, and then replacing the synchronization primitives by their Tribble counterparts. The benchmark used is a dataset with 1 million entries where each operation has a 16-byte key and a 100-byte value, which is commonly used in key-value stores.

We have measured each application with different workload configurations, but we report only the one described above due to space limit. Performance measured under other configurations yields the same conclusions.

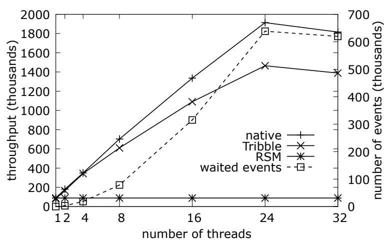
Each application runs in three modes: a *native* mode where the application runs on a single server without replication, a *rsm* mode where the application runs on replicated state-machine, and a *Tribble* mode where the application is replicated with Tribble. In the Tribble mode, for fairness, we apply flow control on the primary to wait for secondary replicas. The throughput is therefore the lower of the throughput on the primary and on a secondary. It turns out that execution with recording on the primary is not the bottleneck, incurring only within 5% overhead compared to the native mode. The end-to-end throughput in the Tribble mode essentially is bounded by the throughput for replay. We vary the number of threads and record the throughput for each application in each mode. The results are shown in Figure 10.

All applications except *Memcached* scale well as we increase the number of working threads. Memcached contains three frequently used global locks (slabs lock, cache lock, and status lock), as well as *interlocked* (increment and decrement) instructions which are replaced in Tribble with locks. It has heavy lock contentions and therefore does not scale well even in the native mode. Figure 10 (f) shows that Tribble introduces an overhead of 70%. Tribble clearly does not work well under heavy lock contentions.

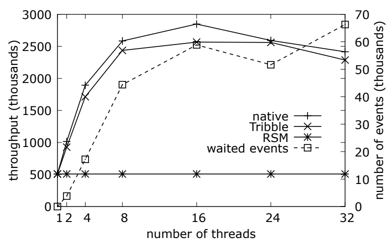
The scalability of Tribble is highly related to the scalability of an application itself in native mode. The thumbnail server is compute intensive and shows perfect performance scalability until the number of threads exceeds CPU cores. The lock server scales well till the number of CPU cores is reached. Both LevelDB and Kyoto Cabinet scale to about 8 cores. LevelDB is slightly



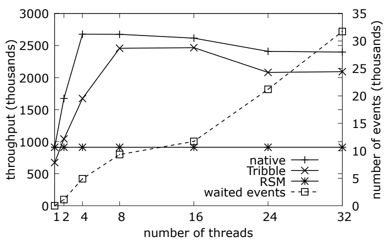
(a) Thumbnail server



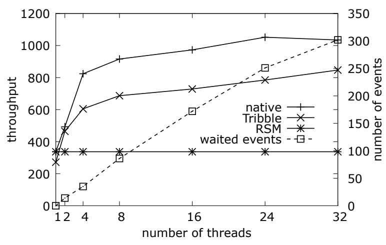
(b) Lock Server



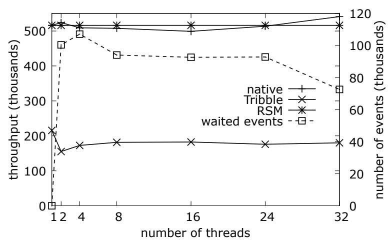
(c) LevelDB



(d) Kyoto Cabinet



(e) File System



(f) Memcached

Figure 10: Throughput of real applications

better due to its use of light-weight mutex locks instead of the readers/writer locks of Kyoto Cabinet. In the file system experiment, batched requests allow the underlying disk driver to optimize disk access, thus concurrent execution increases the throughput.

We see up to a 25% overhead compared to the native version, but the increased concurrency more than compensate for this overhead. To understand the main factor of the overhead, we also count the number of all causal orders, actually recorded causal edges, and causal events that a secondary waits on during replay. The waited events demonstrate the number of synchronization events that causes a thread to wait for another thread at secondary replicas. It strongly corresponds to the performance gap between native and Tribble - the more waited events there is, the wider the gap is. We also see between 58% to 99% in causal-edge reduction with the optimization to remove unnecessary causal edges described in Section 4.4. Overall we see up to 3 to 16 times the throughput compared to that of serialized execution on replicated state-machine.

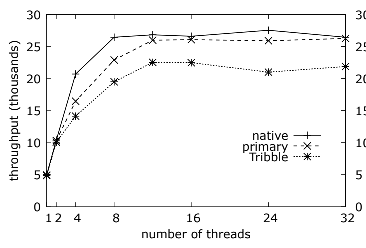
The log shipped from the primary to the secondary replicas contains client requests, as well as the synchronization events recorded by Tribble. Each synchronization event adds around 16 bytes in the trace. Synchronization events add 0 to 70% overhead to log sizes; the exact number varies with applications and with the number of threads used. This overhead is never the bottleneck of the whole system in our experiments.

6.3 Micro Benchmark

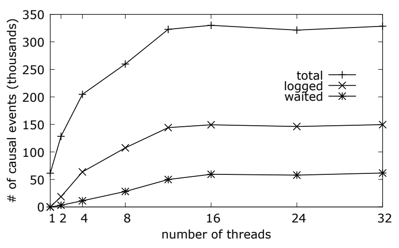
We study the impact of lock contention on Tribble in a micro benchmark experiment. Each request does a total of 200,000 multiplications, with 30% of them done when holding a lock. Each request randomly picks a lock from a pool of l locks. By changing the parameter l, we control the probability p of lock contention, where $p = 1/l$. The experiment uses three modes: in addition to native and Tribble modes, we measure a primary-only mode to understand recording overhead. For each mode we also count the number of all causal dependencies, logged causal edges, and waited events during replay.

We first measure the scalability (in terms of throughput) of Tribble with $p = 0.1$. In Figure 11 (a,b), our optimization manages to reduce the number of logged events to 15-45% of total events. Only 15-40% of logged events introduce waiting at secondary replicas. Recording overhead is about 5% while replay waiting overhead is about 20%, which is consistent with real application experiments. We have also found that (i) the throughput gap between the native mode and the primary-only mode increases as the number of logged events increases, and (ii) the gap between the native mode and Tribble mode increases as the number of waited events increases.

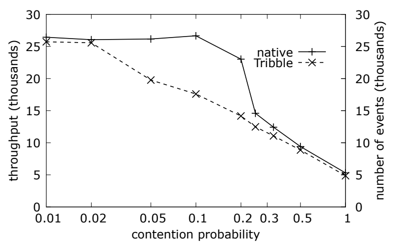
We further vary p to see how the gap between Tribble and the native mode changes. The gap widens first and then narrows. The gap is small when the probability of contention is below 0.02, indicating that Tribble introduces little overhead. The native mode continues to achieve higher throughput until the contention probabil-



(a) Throughput ($p = 0.1$)



(b) # of causal events ($p = 0.1$)

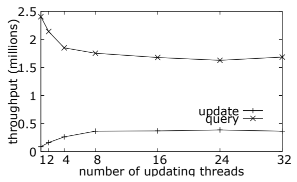


(c) Throughput vs. contention probability

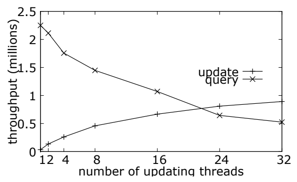
Figure 11: Micro benchmark results.

ity reaches 0.1. The number of causal edges increases as lock contention becomes more severe, leading to higher overhead in Tribble mode and a widening gap. However, when the contention probability is higher than 0.1, the throughput in the native mode drops quickly due to serious lock contention, causing the gap to narrow again and eventually disappearing when $p = 1$.

6.4 Query Performance and its Impact



(a) Query secondary



(b) Query primary

Figure 12: Different query semantics

Tribble supports querying against latest committed state by considering a query request as an update request, while it can also execute a query directly on a replica without going through the replication protocol. The semantics offered to query requests differ: a request on a secondary is executed on a committed but possibly outdated state, whereas a request on a primary might be executed speculatively on a yet-to-be committed state. This is because a Tribble primary executes before consensus, whereas a secondary executes after consensus. Because requests are processed concurrently, query requests must use synchronization primitives to access shared data. Tribble maintains a separate thread pool to handle non-updating queries that acquire only reader locks and ignores any causal orders related to those events.

We use the lock server application to understand the impact of these query semantics; normal request cases are omitted here as it is already included in previous experiments. We use 24 threads for processing query requests (to keep all cores busy), while varying the number of threads for processing update requests from 1 to 32. Interestingly, query-primary and query-secondary exhibit different behavior, as shown in Figure 12. In both cases, the update throughput increases as the number of cores for update requests increases. However, the query throughput manages to stay mostly flat in Figure 10 (a) (for query-secondary), while noticeably decreasing in Figure 10 (b) (for query-primary) as the update throughput goes up more significantly. Because we use readers-write locks, waiting for other events during replay on a secondary gives query processing more opportunity to grab its lock. Overall, Tribble can support a high query throughput when not under a heavy update request load.

6.5 Checkpointing and Primary Changes

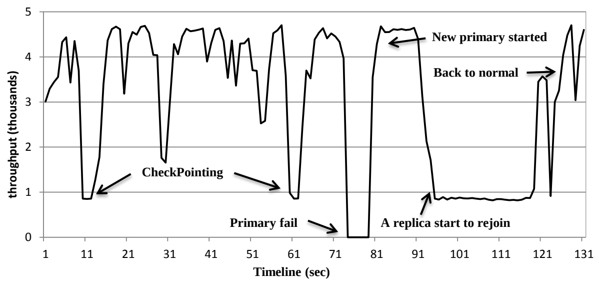


Figure 13: Failover of thumbnail server

We have so far focused on normal-case performance. In this experiment, we aim at understanding the impact of potential disruptions, such as checkpointing and primary changes. We take the thumbnail server that creates two checkpoints at an interval of 50 seconds and then kill the primary replica at the 71st second, restarting it 20 seconds later. Figure 13 shows the throughput fluctuation.

We measure the result of a stress test where all CPUs are saturated. As a result of our aggressive flow control, any abnormal operation can lead to significant performance variation. This does not have to be the case in practice: a secondary can try to catch up over time without impacting the overall performance as long as it does not get promoted to the primary. During checkpointing, the throughput drops for about 2 seconds and then recovers. At the point where the primary fails, the throughput drops to zero and recovers after five seconds when the new primary takes over. The old primary replica is then back as a new secondary and starts learning the committed traces that it has missed, causing the throughput to drop for about 30 seconds due to our aggressive flow control under stress test. If the system were not fully loaded, other replicas can proceed without waiting for the newly joining replica, thus avoiding such performance impact. However, in a stress test setting, a new replica may never catch up if others do not wait. After all replicas are back, the throughput is back to normal.

7 Related Work

Tribble uses Paxos [27, 28] as its underlying consensus protocol. It has become a standard practice [11, 23] to build replicated state-machine [26, 36] using Paxos. The Tribble approach can be applied to other replication protocols, such as primary/backup replication [2] and its variations (e.g., chain replication [37]).

Gaios [9] shows how to build a high performance data store using the Paxos-based replicated state machine approach. Gaios' disk-efficient request processing satisfies both the in-order requirement for consistency as well as the disk's inherent need of concurrent request. Remus [13] achieves high availability by frequently propagating the checkpoints of a virtual machine to another.

Lamport points out in a *generalized Paxos* protocol [29] that defining an execution of a set of requests does not require totally ordering them. It suffices to determine the order in which every pair of conflicting requests are executed. The proposal does not address any practical issues of checking whether requests are conflicting, but simply assumes that such information is available.

Eve [22] uses execute-verify, rather than agree-execute in state-machine replication or execute-replay in Tribble, and resorts to re-execution in cases of detected divergence, which is complementary with Tribble. The LSA algorithm in Basile et al. [5] ensures replica consistency through enforcing the order of synchronization operations on replicas, but it does not consider the complications related to leader changes, as well as the resulting mode changes.

To capture and preserve partial orders among requests, Tribble leverages previous work of faithful record and replay of multi-threaded programs. An incomplete sample of such work includes RecPlay [34], JaRec [17], ReVirt [16], R2 [19], PRES [33], ODR [3], SCRIBE [24], Respec [30] and its follow-on work [39, 38], and many others [31, 12]. Mode change and online replay are the two requirements that drive the design of Tribble. Most of the previous work on record and replay (e.g., Revirt, PRES and ODR) target offline debugging and forensics, where replay performance is not important. For example, PRES reduces record-time overhead by making the replay take the extra overhead of searching for the identical execution, a reasonable tradeoff for offline debugging, but undesirable for the scenarios of Tribble. Although Respec is also designed for online replay, its implementation only allows replicas on the same machine due to its use of multi-threaded fork, while Tribble's replay happens on different secondary servers. Tribble further avoids using barriers, which introduces non-negligible overhead.

Tribble ensures the atomicity of API invocations and recording logic. The atomicity can be achieved in many ways. For example, SCRIBE ensures such atomicity by associating each shared resource with a wait queue and a unique sequence number counter, serializing thread access to it. During replay, the same serialized order is enforced when threads access the shared resource, which guarantees the same execution order but downgrades the parallelism of the system as some API calls that should be able to run concurrently are now serialized.

Deterministic parallel execution is another promising direction and can be done with new OS abstractions (e.g., Determinator [4] and dOs [8]), by runtime libraries (e.g., Kendo [32]) with compiler support (e.g., Coredet [6]), or with hardware support (e.g., DMP [14], Calvin [21], and RCDC [15]). With deterministic execution, traditional state-machine replication can be applied directly. However, without architectural changes to provide hardware support, determinism is achieved at the cost of degraded expressiveness and/or performance. For instance, Determinator allows only race-free synchronization primitives natively such as fork, join as well as barrier, and supports others using emulation; Kendo supports deterministic lock/unlock using deterministic logical time, which may sacrifice performance. Overall, the overhead of such solutions (CoreDet, dOS, and Determinator) is not yet low enough for production environments [7].

8 Concluding Remarks

The prevalence of the multi-core architecture has created a serious performance gap between a native multi-threaded application and its replicated state-machine counterpart. Tribble closes this gap using a carefully designed execute-replay approach. By defining a set of simple user-friendly APIs and by building on a well-known consensus protocol, we hope that Tribble would contribute to a new replication foundation that is appropriate for the multi-core era, replacing the classic state-machine replication approach.

References

[1] memcached - a distributed memory object caching system. `http://memcached.org/`.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[5] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):448–465, 2006.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 53–64, New York, NY, USA, 2010. ACM.

[7] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? WODET '11, 2011.

[8] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOs. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[9] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[15] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 67–78, New York, NY, USA, 2011. ACM.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 211–224, New York, NY, USA, 2002. ACM.

[17] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere. JaRec: a portable record/replay environment for multi-threaded java applications. volume 34, pages 523–547, 2004.

[18] Google. LevelDB: A fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb`.

[19] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[20] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 231–240, New York, NY, USA, 2008. ACM.

[21] D. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *HPCA'11*, pages 333–334, 2011.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX conference on Operating systems design and implementation (to appear)*, OSDI'12.

[23] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, July 2011. 38pp.

[24] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM.

[25] F. Labs. Kyoto Cabinet: a straightforward implementation of dbm. `http://www.fallabs.com/kyotocabinet/`.

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

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

[28] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

[29] L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft, Mar. 2005.

[30] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocesor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 77–90. ACM, March 2010.

[31] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[32] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 97–108, New York, NY, USA, 2009. ACM.

[33] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[34] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. volume 17, pages 133–152, New York, NY, USA, May 1999. ACM.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[36] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[37] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[38] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 369–384, New York, NY, USA, 2011. ACM.

[39] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 15–26. ACM, March 2011.

[40] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 251–264, New York, NY, USA, 2011. ACM.