Rex: Replication at the Speed of Multi-core

Zhenyu Guo[*] Chuntao Hong[*] Mao Yang[*] Dong Zhou[#] Lidong Zhou[*] Li Zhuang[*]

[*]Microsoft Research [#]CMU

[*]{zhenyug, chhong, maoyang, lidongz, lizhuang}@microsoft.com [#]dongz@cs.cmu.edu

Abstract

Standard state-machine replication involves consensus on a sequence of totally ordered requests through, for example, the Paxos protocol. Such a sequential execution model is becoming outdated on prevalent multi-core servers. Highly concurrent executions on multi-core architectures introduce non-determinism related to thread scheduling and lock contentions, and fundamentally break the assumption in state-machine replication. 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. Concurrent executions of the application can be decoupled with a sequence of consensus decisions through consensus on partial-order traces, rather than on totally ordered requests, that capture the non-deterministic decisions in one replica execution and to be replayed with the same decisions on others. The result is a new multi-core friendly replicated state-machine framework that achieves strong consistency while preserving parallelism in multi-thread 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 16 times the throughput of standard replicated state machines.

Categories and Subject Descriptors D.1.3 [*SOFTWARE*]: Concurrent Programming

General Terms Multi-core, Performance, Replication

Keywords Replicated State Machine, Multi-core, Replication

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 [28, 38] start from the same initial state and process the same set of requests in the same *total order* deterministically. The concurrent execution model of multi-thread server applications no longer matches this deterministic sequential execution model assumed by state-machine replication. The fundamental challenge in replicating highly concurrent multi-thread applications lies in the tension between determinism and parallelism. Deterministic parallelism [3, 5, 7, 15, 16, 21, 34] has been a promising research direction, but so far, without architectural changes to provide hardware support, determinism is achieved only at the cost of degraded expressiveness and/or performance. Eve [22] achieves *mostly* deterministic parallelism by using a mixer that groups requests into *mostly* non-conflicting batches, so that concurrent executions of a batch on different replicas are likely to have deterministic and consistent effects. Eve has built-in mechanisms to detect divergence, trigger rollback, and resort to serial re-execution for recovery.

We instead take a different approach to the problem: rather than forcing all replicas to execute in the same deterministic fashion, we achieve replica consistency with an *execute-agree-follow* model in our new replication framework called Rex. In this *execute-agree-follow* model, a *primary* replica *executes* freely, processing requests concurrently, while recording the non-deterministic decisions in a partially ordered *trace*. All replicas then run a consensus protocol, such as Paxos [29], to *agree* on the traces, despite replica failures and primary changes. Finally, *secondary* replicas *follow* the agreed-upon trace by making the same non-deterministic choices in a concurrent replay to reach the same consistent state as the primary.

We have implemented Rex. To evaluate whether Rex 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 show that applications on Rex

EuroSys 2014, April 13 - 16 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2704-6/14/04$15.00. . . $15.00.

http://dx.doi.org/10.1145/2592798.2592800

achieve as much as 16 times the throughput of standard state-machine replication.

The rest of the paper is organized as follows. Section 2 presents an overview of Rex's execute-agree-follow model. Section 3 details how Rex uses replica consensus to agree on growing traces. Section 4 presents how Rex captures causal order in traces during execution and allows secondary replicas to follow by replaying execution while respecting the causal order. Since Eve is so related to our work, we devote Section 5 to the discussion of the difference between Eve and Rex. 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

Rex assumes an application model that is close to how online service applications are commonly written. In this model, an application serves incoming client requests with request *handler*s. In contrast to standard state-machine replication, where totally ordered requests are processed sequentially, request handlers in Rex are executed concurrently in a thread pool using a set of standard synchronization primitives to coordinate their access to shared data. Each handler executes *deterministically*, where Rex requires that the ordering of synchronization events be the only source of nondeterminism. Rex degrades into state-machine replication if each request handler uses the same lock to protect the entire execution.

2.1 Rex vs. State-Machine Replication

Figure 1 compares Rex'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 a 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.

State-machine replication uses a *consensus-execution* model, where replicas first reach consensus on a sequence of requests r_i (① *consensus stage*), and then execute those requests in this order (② *execution stage*).

Rex instead uses an *execute-agree-follow* model. A Rex primary executes first by processing requests concurrently and records fine-grained causal dependencies among requests in a trace tr_i (① *execute stage*). A primary periodically proposes the up-to-date trace as the value for the next consensus instance. Even in the presence of multiple primary replicas, replicas can reach consensus on a sequence of traces (② *agree stage*). Once committed, secondary replicas faithfully follow the traces to replay the execution as on the primary (③ *follow stage*).

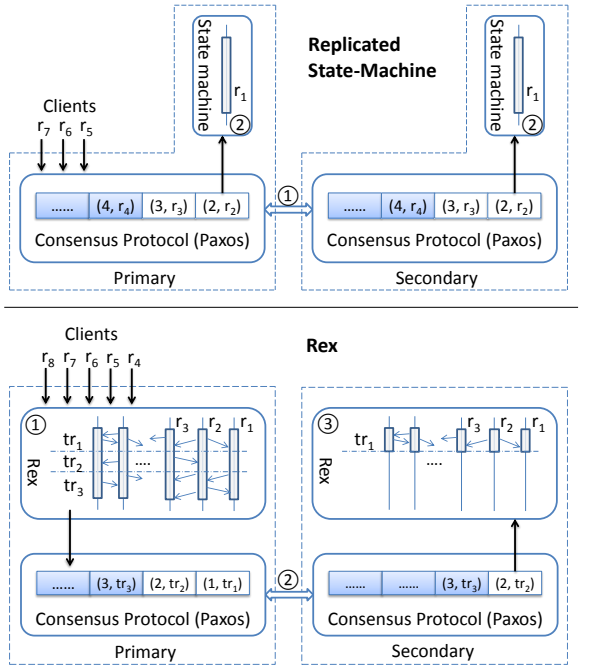


Figure 1. State-machine replication vs. Rex.

Rex 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 Rex, 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 (② consensus). However, the primary does not have to wait for the completion of the replay on the secondaries.

Execute stage. Rex executes client requests on the primary before a consensus is reached. During execution, request handlers on the primary record causal dependencies among a set of synchronization events. Figure 2 shows one example. 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 order, 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 2, a causal edge exists from the `Unlock` event $(t_1, 3)$ to the `Lock` event

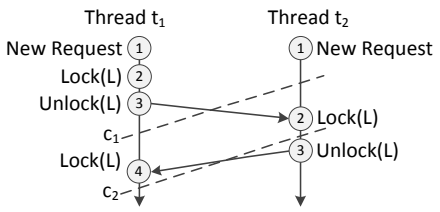


Figure 2. Request handlers & synchronization primitives.

$(t_2, 2)$, where the `Unlock` event must precede the `Lock` event.

The trace is *growing* as a Rex primary continuously executes 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 order 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 2 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.

Agree stage. Rex replicas reach consensus on a sequence of traces using the multi-instance Paxos protocol. While requests in different consensus instances are independent in replicated state-machine, traces in different consensus instances of Rex 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.

Follow stage. In the follow stage, a secondary replica in Rex follows the execution based on the traces that the replicas have reached consensus on. When processing requests on a secondary with respect to a trace, a synchronization event e is triggered only after the execution of all events causally ordered before e in the trace. If e in thread t has to wait for the execution of an event e' in thread t', Rex pauses thread t just before e and registers with t' to have it signal t after it executes e'. This way, the executions of corresponding threads at each secondary follow the same causal order. Because of 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 get 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 in the follow stage. Unnecessary causal dependencies are removed at the execute stage for better performance.

2.2 Correctness and Concurrency

The correctness of Rex 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 Rex satisfies the consensus and prefix properties is the subject of Section 3, while Section 4 presents how Rex achieves the determinism property.

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

3. Agreeing on Traces

Rex uses Paxos to ensure the consensus and prefix properties introduced in Section 2.2. As in state-machine replication, Rex manages a sequence of consensus instances with the Paxos protocol [30] so replicas can agree on a trace. Across instances, Rex must also ensure the prefix property. In this section, we describe how Rex reaches consensus on a growing trace.

3.1 Consensus in Rex

Rex uses the multi-instance Paxos protocol that relies on a failure detection and leader election module to detect replica failures and elect a new 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 Paxos, Rex makes two noteworthy design decisions. First, Rex has at most one *active* consensus

instance at any time. A primary proposes to an instance only after it learns that every earlier instance has a proposal committed. This decision greatly simplifies the design of Rex in the following three ways. First, Rex 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 from 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, the leader of Rex's Paxos is co-located on the primary. The 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 Rex 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 Rex primary is co-located with the leader except during transition, 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.

3.2 Reconfiguration

In Rex, leader changes trigger primary changes. A new leader in Paxos becomes the new primary; the old primary downgrades 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 primary. The consensus through Paxos ensures correctness by choosing only one trace, 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. For performance reasons, we let the threads log events and causal edges asynchronously. As a result, 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 primary. In that case, the promotion is forever blocked. Instead of making each cut consistent, Rex defines the last consistent cut contained in a trace as the meaning of the proposal. If the primary changes, then the residual of the trace after that consistent cut is ignored.

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. Since the primary executes *speculatively* in Rex, it must roll back its execution to the point of the last committed trace. One way to roll back is through checkpointing, as described next.

3.3 Checkpointing

Rex 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 developer to write application-specific checkpointing logic, Rex 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, thanks 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 Rex does not have sufficient information for a replica to continue processing an incomplete request when re-starting from that checkpoint. When Rex 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., 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. Execute and Follow

Rex leverages record (on the primary in the execute stage) and replay (on a secondary in follow stage) to achieve the determinism property as required in Section 2.2. The unique setting in Rex imposes three requirements that make previous approaches of record and replay insufficient.

First, Rex 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 previous record and replay systems, such as R2 [20] and Respec [32]. For example, a record and replay tool often records the return value of an `fopen` call during recording and simply returns the value without executing during replay. As 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.

A second unique requirement in Rex is *hybrid execution*, where a resource is concurrently manipulated by API invocations in replay mode and native mode. A secondary replica may serve read requests in a separate thread pool running in native mode without involving them in replication. This is generally not a problem as read requests only *read* states without making changes. However, a `Lock` invocation inside a read request handler may change the lock state from unlocked to locked, which would interfere with the lock invocation in write request handlers. This effect is called *lock state pollution*.

```
1  class RexLock {
2    void Lock() {
3      if (env::local_thread_mode == RECORD) {
4        AcquireLock(real_lock);
5        RecordCausalEdge();
6      } else if (env::local_thread_mode == REPLAY) {
7        WaitCausalEdgesIfNecessary();
8        AcquireLock(real_lock);
9      }
10   }
11   void Unlock() {
12     if (env::local_thread_mode == RECORD)
13       RecordCausalEdge();
14     ReleaseLock(real_lock);
15   }
16 }
```

Figure 3. Wrapper for `Lock` and `Unlock`.

Third, Rex demands *online replay* [32]: replay performance should be comparable to record performance so that a secondary can catch up to ensure stable system throughput. This is in contrast to offline replay for scenarios such as debugging. In particular, Rex must enable concurrency during replay.

No previous record-and-replay systems offer a satisfactory solution for Rex to support all these requirements, although many of the concepts and approaches are useful to Rex. To support mode change, a replaying replica must maintain system resources faithfully by re-executing operations on resources. To allow hybrid execution, we must carefully avoid lock state pollution from reading threads. To achieve online replay, trade-offs and optimizations are adopted. This section first provides an overview of record and replay and then focuses on the solutions to the challenges.

4.1 Wrapping Synchronization Primitives

Rex must ensure the same execution ordering of synchronization operations from all threads, such that executions in follow stage would produce the same effect as in execute stage. Rex therefore provides wrappers for those synchronization primitives to capture and enforce their order of execution, similar to RecPlay [37] and JaRec [19]. A list of synchronization primitives is shown in Section 6.

Figure 3 is the pseudo code of the `Lock` and `Unlock` wrappers. During recording, the wrapper invokes `RecordCausalEdge` (lines 5,13) to capture a causal edge between two consecutive successful `ReleaseLock` and `AcquireLock` calls of `real_lock`; a causal edge remembers the source and destination events, including the thread identifiers and the logical clock of the events in the thread. Before replaying the destination event of a causal edge, `WaitCausalEdgesIfNecessary` (line 7) pauses the current thread until the source event on that edge happens. The `WaitCausalEdgesIfNecessary` call before `ReleaseLock` (line 14) is not neccessary as its source event `AcquireLock` must have already happened. Wrappers must also ensure atomicity of an event and its causal

edge logging, which usually requires an additional lock. In this case, atomicity is already ensured by the mutex lock of `real_lock`.

The difference between record and replay lies purely in the working mode of the wrappers, making it easy to switch from replay to record while being transparent to applications. By wrapping synchronization primitives, Rex introduces enough non-determinism in a programming-friendly way to allow sufficient concurrency. Rex intentionally avoids providing programming abstractions that are not record-replay friendly, such as OS upcalls, and decides against developing a complete deterministic record and replay tool for arbitrary full-fledged concurrent programs because of the inherent complexity and performance overhead, as well as the unique requirements in Rex.

4.2 Dealing with Challenges

Order replay for mode change. To faithfully maintain resources during mode change, Rex uses *order replay* [37]. When invoking a sequence of function calls, order replay requires that all resource states and return values of function calls be the same as long as the invoking order is the same. All locks in Rex conforms to this requirement, as well as operations over other resources such as disk files because: (i) the sequence of operations remains the same since concurrent operations in multiple threads are protected with locks and lock invoking order is enforced by wrappers; (ii) operations are assumed to be synchronous only.

Avoid lock state pollution from hybrid execution. The key to enable hybrid execution is to circumvent the effect of lock state pollution from reading threads. Fortunately, lock state changes from read requests are transient and always restored by a later `Unlock` invocation in the same read request handler. This transient state change is safe for lock functions that have no return values, except the completion time is delayed until the read thread releases the lock. For lock functions that have return values such as `TryLock`, the state change may result in different return values. Rex invokes the lock function repeatedly during replay inside the wrapper until it gets the same return value as in record.

Tradeoff between record overhead and replay parallelism. With only `Lock` and `Unlock`, causal order captured during record 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), and as a result sacrifices certain parallelism during replay. The left figure in Figure 4 shows an example with three threads. If Rex imposed a total order on all those events, it would have to record edges (A, B, C, D). As a result, t_3@1 (read in t_3 at clock 1) needs to wait for t_2@1, and t_2@2 needs to wait for t_3@1 during replay; both are not true causal edges; therefore the wait leads to unnecessarily reduced parallelism. The right figure shows the ground-truth causal edges, in which all the `TryLock(F)`s have causal order with `Lock` and `Unlock` on t_1 (edges A, Y, W, Z, X, D). The execution would be equivalent by preserving all these edges, with higher parallelism during replay as t_3 and t_2 will not wait for each other. To capture the ground-truth causal edges is however not free. It requires recording more causal edges and maintaining more information in the shared resource. (In this case, six edges will be recorded instead of four, and as a result, since vertexes in the graph now have multiple incoming edges, it is now necessary for the lock to keep the list of "incoming events", instead of remembering only the last incoming event.) Rex carefully balances the tradeoff between the record overhead and the replay parallelism so that the performance between record and replay is comparable.

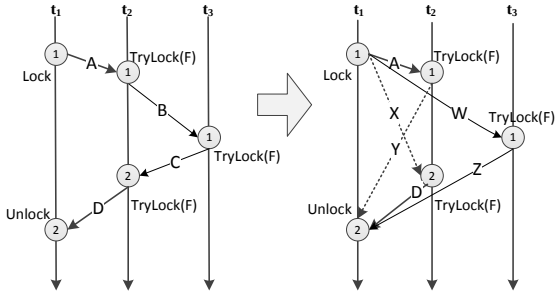


Figure 4. Total order vs. partial order. `TryLock(F)` indicates that the invocation fails to gain the lock.

Remove unnecessary causal edges. Rex removes unnecessary causal edges to reduce both the trace size and the replay cost of examining whether a causal edge is satisfied or not. Because Rex uses the same number of threads to serve write requests and timer events on all replicas, causal ordering within each thread is preserved and therefore can be safely removed. In addition, Rex removes unnecessary causal edges if the causal order indicated by an edge can be implicitly derived from other edges. In the example of Figure 4, causal edge X is removed because it follows from causal edge A and the intra-thread causal order in thread t_2. The same applies to causal edge Y.

5. Discussion

The execute-agree-follow model in Rex and the execute-verify model in Eve [22] represent two different approaches to replication on multi-core servers, especially in the assumptions to achieve consistency among replicas and in the mechanisms for performance.

5.1 Consistency

For consistency, the execute-verify model in Eve assumes that the application state that "matters" can be marked, checkpointed, compared, rolled back, and transferred (and used in a different replica). Replica consistency is then guaranteed directly in the verification phase where the application states on replicas are compared to detect divergence. This is particularly attractive because (i) verifica-

tion offers high assurance of consistency, against different possible sources of divergence, even including Byzantine faults and concurrency bugs, and (ii) replicas are allowed to execute independently. In contrast, the execute-agree-follow model in Rex assumes that replicas making the same non-deterministic decisions are consistent and that all non-deterministic decisions can be captured. In this model, replicas no longer execute independently. Whereas correctness for the execute-verify model hinges on accurately marking the states, correctness for the execute-agree-follow model hinges on capturing all sources of non-determinism completely.

Our experiences with the six applications described in Section 6 have indicated that finding locks and non-deterministic functions is usually easy to do, because most applications, especially those developed with cross-platform capabilities in mind, usually wrap different implementations of these functions into uniform interfaces. By replacing these interfaces with Rex's synchronization primitives, we can easily capture all the locks and non-deterministic functions. Data races, however, can be harder to find. Fortunately, more and more people have realized the hazard of data races and have begun to take precautions when using data races as synchronization. In the six applications we have tested, we have only fixed two benign data races: one is a double-check lock and the other is a NULL-pointer check, both of which already regarded as bad practices [42].

Through our own exercises of applying the execute-verify model to the applications we have examined in Section 6, we have discovered some subtleties related to this approach. State marking essentially divides the program state into two parts; we refer to the unmarked states as the *context*. Because rollback and state transfer apply only to the marked states, it is important to ensure that the marked state, when rolled back or transferred, is consistent with the current context, possibly on a different replica. Normally, a checkpoint for an application should contain all the marked state, but this turns out to be insufficient in this case because loading a checkpoint to reconstruct the program state often resets the context appropriately. Rollback and state transfer in Eve require that the marked state be consistent with the given context after rollback or state transfer; e.g., through regenerating soft states in the context if needed.

Another subtlety we have encountered is related to background tasks; one such example is the compaction tasks in LevelDB [14]. Eve uses the end of processing a request batch as the point to check state consistency, assuming that the incoming requests are the only triggers to state changes. The assumption no longer holds with background tasks. Synchronizing replicas on background tasks to define consistency-checking points breaks execution independence and might not always be feasible. Writing a wrapper to mask the effect of background tasks could always work in theory, but is non-trivial to do. In general, we have found that handling on-disk state is tricky in the execute-verify model.

```
1 class Singleton {
2   static Singleton *ptr_;
3   static Lock lock_;
4   Singleton* GetInstance() {
5     NATIVE_EXEC    // introduced by Rex
6     if (!ptr_) {
7       AcquireLock(lock_);
8       if (!ptr_) ptr_ = new Singleton();
9       ReleaseLock(lock_);
10    }
11    return ptr_;
12  }
13 }
```

Figure 5. A double-lock implementation of `Singleton` initialization.

Verification in Rex. The idea of verification is powerful and can benefit Rex as well. In particular, we implemented opportunistic validity checking in Rex to detect data races. We have implemented both result checking and resource-version checking. Result checking is implemented by logging the return value on the primary and checking the result on secondary replicas. Result checking can be late in reporting state divergences that have occurred earlier but only reflected in the return value much later. To detect data races earlier, one could include more states in the trace along with return values, which would increase the overhead significantly. Instead, we also implemented resource-version checking as follows. We add a version number to each resource (e.g., a lock or a semaphore). Each time the resource is used, its version number increases. The primary records version numbers along with the causal events. While replaying, a secondary checks the version number of a resource against the one logged in traces. Thus, Rex detects whether or not a resource is used in the same order in replicas. (Note that we are simplifying here by assuming total order of events here, actual implementation is more complex because of partial order optimization as discussed in Section 4.2.) Version checking is useful as it detects state divergences early and helps programmers identify the root cause of data races. Those validity checks are particularly useful when testing Rex-enabled applications.

Manual exclusion in Rex. Just as Eve can choose to ignore certain insignificant state differences (e.g., IP addresses) on different replicas, Rex allows developers to exclude certain portions of the executions out of the agree-follow scope by using a special macro `NATIVE_EXEC` in order to allow benign data races explicitly. In the scope of `NATIVE_EXEC`, Rex stops recording/replaying the synchronization primitives so that different replicas can execute the code with different threads, just like in a non-Rex environment. Taking as an example the `Singleton` code we found in LevelDB (Figure 5), the initialization of a `Singleton` object may execute in any thread (which invokes several lock function calls). However, the resulting state is the same no matter which thread initializes this object. We use `NATIVE_EXEC`

to allow different threads initializing the object on different replicas.

5.2 Performance

For performance, there are three aspects to compare: (i) the level of parallelism on multi-core servers in normal cases, (ii) the overhead introduced, and (iii) the probability of exceptions and the performance under exceptions.

To enable high concurrency on multi-core servers, Eve introduces a mixer, which packages requests into (largely) non-conflicting batches that can be executed concurrently. This is a general and powerful mechanism that does not depend on application semantics. The level of parallelism is however bounded by the sizes of such batches in the workload. There is also the traditional trade-off between latency and throughput on batching: waiting for a larger batch increases the level of parallelism, but at the price of introducing longer latencies.

In contrast, Rex treats faithfulness as an important design goal by supporting an application model that uses common synchronization primitives, which is close to how on-line multi-thread applications are written. Rex therefore supports parallelism at the same fine granularity as in the original application, does not rely on any knowledge about the conflicts between requests, and does not interfere with the grouping of requests for processing. Our experiences with real-world multi-thread applications have confirmed the significance of faithfulness to performance, especially related to fine-granularity parallelism. Rex preserves lock granularity by providing programmers with commonly used Lock/Unlock APIs. In modern multi-thread server applications, programmers usually spend a lot of time trying to minimize lock contentions in order to achieve best scalability. Instead of simply locking the whole data structure with one big lock, they use multiple locks to protect different data. In the meantime, they try to reduce the time a lock is held as much as possible. They may even redesign data structures so as to reduce lock contention. For example, KyotoCabinet [27] uses one lock to protect the meta-data, and 1024 other locks to protect different key ranges. Releasing locks before performing time-consuming operations is another widely used technique. Most of the benchmarks we have used take advantage of this technique. On the data structure side, LevelDB [14] uses reference-counting pointers to avoid holding a lock to prevent the pointer from being deleted. It is through these clever designs that modern multi-thread server applications fully leverage the power of multi-core servers. In Rex, we honor these designs by adapting the Lock/Unlock API and preserving the lock granularity, thereby faithfully preserving the scalability of the original application. Locks in the primary behave exactly like traditional locks. In this way, optimization techniques that reduce lock contention can also be used in Rex based applications. Whenever desirable, the idea of a mixer can also be introduced into Rex-enabled applications because the mechanism is largely orthogonal to those in Rex.

In the normal case, the overhead introduced by Eve involves running the mixer, tracking state changes in a Merkle tree, and detecting divergence after each batch. Rex instead incurs overhead in capturing non-deterministic decisions (e.g., those involving synchronization primitives) and in following those decisions on secondary replicas. Such overhead has been shown to be reasonable in our evaluations (Section 6). Eve has to roll back for a sequential re-execution whenever state divergence is detected. Eve optimistically assumes that divergence from non-determinism does not happen often. Eve's optimism comes from its use of a mixer that repackages requests into mostly non-conflicting batches. The quality of its mixer is critical to its performance. The mixer must have the knowledge on when the processing of two requests introduces conflicts. False positives and false negatives from the mixer both have negative impact on the performance of Eve. In contrast, for Rex, a rollback is needed only when a primary is falsely suspected and replaced by a new one: in this case, the uncommitted portion of the traces on that server has to be rolled back. Compared to Eve, which has to support fine-granularity partial rollbacks, full-machine rollback is sufficient for Rex: the weaker requirement makes checkpointing and rollback in Rex simpler.

6. Experience and Evaluation

In this section, we discuss the details of the API and share our experiences with Rex. We further use a combination of real-world applications and micro-benchmarks to evaluate the following aspects of Rex:

- How well does Rex scale with the number of cores?
- What is the impact to performance if lock granularity is not preserved?
- How do queries (read requests) perform under different semantics?
- How well does Rex cope with checkpointing, primary changes, and replica recovery?

6.1 Building Applications with Rex

We have implemented Rex 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. The API of Rex is shown in Figure 6.

A programmer builds an application by inheriting the `RexRSM` and `RexRequest` classes. The processing logic for a request is encoded in an `Execute` function as a request handler. The `RexRSM` class implements initialization and the functions used for checkpointing. During initialization, the application can add background tasks such

```
class RexLock;
class RexReadWriteLock;
class RexCond;
class RexRequest {
    virtual int Execute(RexRSM * rsm);
    virtual ostream& Marshall(ostream& os);
    virtual istream& UnMarshall(istream& is);
    ...
};
class RexRSM {
    virtual bool Start(int numThread);
    virtual int WriteCheckPoint(ostream& os);
    virtual int ReadCheckPoint(istream& is);
    ...
    int AddTimer(Callback cb, int interval);
};
```

Figure 6. Programming API in Rex.

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 Rex synchronization primitives to coordinate. Built-in synchronization primitives in Rex include `RexLock`, `RexReadWriteLock`, and `RexCond`.

Adapting a service application to Rex requires two steps: wrapping the service interface API with `RexRSM` and `RexRequest`, and replacing the synchronization primitives with those in Rex. Both are straightforward and we have written only 300-500 lines of code for each of the applications. Table 1 shows the type of locks used in some of the applications we built. The effort lies mostly in writing the functions for checkpointing, which may not have counterparts in the original applications (as they may not have been designed to support checkpointing).

Debugging data races. Rex provides a set of mechanisms to detect data races and help programmers fix data races (see Section 5). We have found those mechanisms effective when building applications.

We have identified a benign data race related to singleton initialization as shown in Figure 5. If a different thread creates the singleton when executing on a secondary replica, compared to what happened on the primary, the secondary can no longer "follow" the trace created by the primary. After enabling resource version checking in Rex, the secondary throws an exception when a "wrong" thread tries to acquire the lock and initialize the singleton. Along with the exception, Rex reports the name and the version of the lock, the expected thread, and the actual thread who is accessing the lock. As a result, we quickly narrow down the cause of the problem and fix it with the `NATIVE_EXEC` macro.

We found another benign data race in LevelDB [14], where an in-memory table is written to disk regularly. A timer routine checks the pointer to the table periodically and triggers an I/O function to dump the table if the pointer is not NULL. Although the pointer is guarded by a lock, the timer does not acquire the lock when checking whether the lock is NULL. Instead, it acquires in the I/O function to ensure the table is not modified during the disk write. As a result, the timer would behave differently on replicas if the pointer checking returns different results. Though the root cause of the problem is the unguarded read, the problem is exposed only when the I/O function tries to acquire the lock. By extracting and visualizing the causal edges from the transmitted trace from the primary, and comparing against the current in-memory state, we find that the I/O function was unexpectedly triggered on the secondary because of this unguarded read. Because the pointer is checked infrequently, we fix the problem by acquiring the lock before reading the pointer.

Application	Synchronization Primitives
Thumbnail Server	Lock
File System	Lock
Lock Server	ReadWriteLock
LevelDB	Lock, Cond
Memcached	Lock, Cond
Kyoto Cabinet	Lock, Cond, ReadWriteLock

Table 1. Synchronization primitives used.

6.2 Experimental Setup

Our experiments are conducted on 12-core machines with hyper-threading, 72 GB memory, 3 SCSI disks with RAID5 support, and interconnected via 40 Gbps network. 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 the clients and the primary.

6.3 Real-world Application Performance

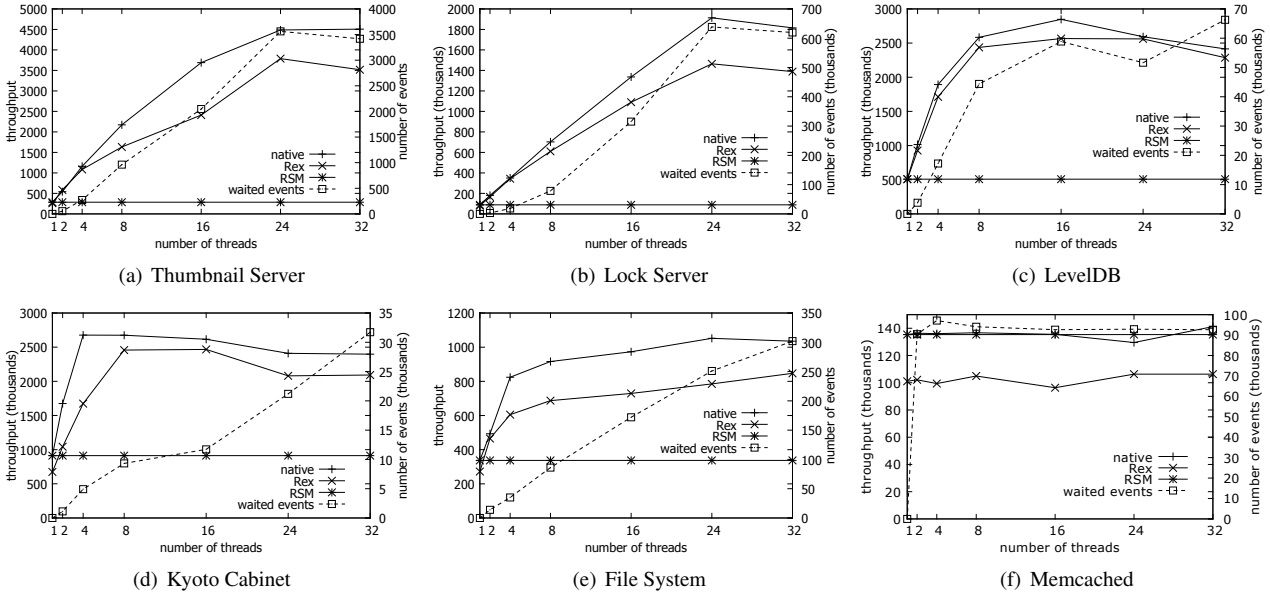
We have built or ported a set of real-world applications on top of Rex. For each application, we first optimize them to achieve best scalability and then port them to Rex.

Thumbnail server is an existing application that manages picture thumbnails. It maintains an in-memory hash table to store meta-data and an in-memory cache to store thumbnails, as well as a set of locks 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]. It was previously built on top of a replicated state-machine library. Similar to the report on Chubby [10], we create a workload with 90% of the requests as "leases renew" 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** experiment, we measure the performance of synchronized random read/write on 64 files of size 128MB. Each request either reads or writes 16 KB of data. The read/write ratio is 1:4.

LevelDB [14] is a fast key-value store library that provides an ordered mapping from string keys to string values.



(a) Thumbnail Server (b) Lock Server (c) LevelDB

(d) Kyoto Cabinet (e) File System (f) Memcached

Figure 7. Throughput of real-world applications.

The database is divided into 256 slices with one lock for each slice.

Kyoto Cabinet's HashDB [27] is a lightweight hash database library whose key space is divided into 1024 slices with each slice protected by a readers-writer lock.

Memcached [18] 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 Rex counterparts. The benchmark used is a dataset with 1 million entries where each operation has a 16-byte key and a 100-byte value as commonly used in key-value stores.

We have measured each application with different workload configurations, but we report only the ones described previously because of 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, an *RSM* mode where the application runs on replicated state-machine, and a *Rex* mode where the application is replicated with Rex. In Rex mode, for fairness, we apply flow control on the primary so it waits for secondary replicas. The throughput is therefore the lower of the throughput on the primary and on the secondaries. It turns out that execution and recording on the primary is not the bottleneck, incurring only within 5% overhead compared to the native mode. The end-to-end throughput in Rex 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 7.

All applications except *Memcached* scale well as we increase the number of worker threads. Memcached contains three frequently used global locks (slabs lock, cache lock, and status lock). The application does not scale well even in native mode, because the regions guarded by the locks are large, therefore introducing heavy lock contention. Rex clearly does not work well in this case.

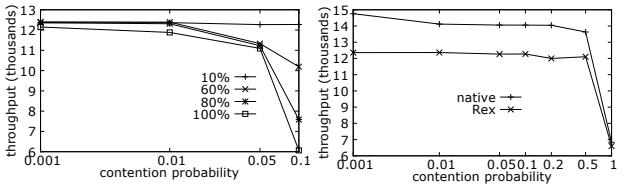
The scalability of Rex is highly related to the scalability of an application itself in native mode. The thumbnail server is computation intensive and shows perfect scalability until the number of threads exceeds the number of CPU cores. The lock server scales well until the number of CPU cores is reached. Both LevelDB and Kyoto Cabinet scale to about 8 cores. LevelDB is slightly better thanks to its use of lightweight 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 accesses. Therefore, concurrent execution increases the throughput.

We see up to 25% overhead compared to the native version, but the increased concurrency more than compensates for this overhead. To understand the main source of the overhead, we also count the number of all causal events, the number of actually recorded causal edges, and the number of causal events that a secondary waits on during replay. The waited events legend shown in Figure 7 presents the number of synchronization events per second that cause threads to wait for others at secondary replicas. It strongly corresponds to the performance gap between native and Rex: the higher the value is, the wider the gap. We also see 58% to 99% reduction of causal edges with the optimization to remove unnecessary causal edges described in Section 4.2. Overal-

l, we see 3 to 16 times throughput on Rex compared to the serialized execution in traditional replicated state-machine.

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

6.4 Lock Contention and Lock Granularity



(a) Lock granularity (b) Lock contention

Figure 8. Impact of different lock granularities and lock contention probabilities.

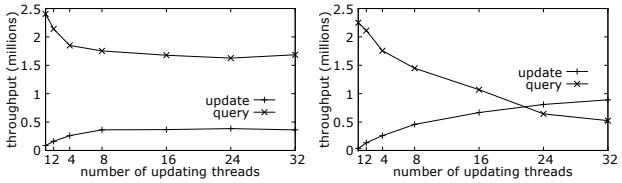
In order to understand the impact to the performance if lock granularity is not preserved, e.g., with the request handler granularity, we compose a micro-benchmark to study the performance with different lock granularities and lock contention probabilities.

In the benchmark, each request performs computation for approximately 10 milliseconds, with part of them done within a lock. By controlling the percentage of computation done in the lock, we can simulate applications with different lock granularities. The lock is randomly picked from a pool of l locks. By changing the parameter l, we control the probability p of lock contention, where $p = 1/l$. These experiments are conducted on 16-core machines.

Figure 8(a) presents the throughput of Rex for four different lock granularities, as we vary the lock contention probability. We show the performance of four settings, in which 10%, 60%, 80% and 100% of the computation is done in locks. The X-axis indicates the conflict probability of the locks. We can see that different lock granularities do not have much impact on throughput when conflict probability is smaller than 0.05, because there are still enough independent requests to keep all the 16 cores busy. However, as the conflict probability grows to 0.1, the throughput drops by almost a half in the case where 100% computation is done in locks, while there is almost no performance degradation at all in the case of 10%. This experiment demonstrates the importance to minimize lock granularity in server applications. Rex honors these optimization efforts and preserves as much parallelism as the application can offer. Relying on mixers to batch requests into independent groups, and treating each request as a unit of parallelism as Eve [22] does, is exactly the case where 100% of the computation is done within locks, given there is neither false positive nor false negative in the mixers.

Figure 8(b) further presents the performance degradation as we vary the lock contention probability p from 0.001 to 1, with 10% of computation done in locks. The throughput with Rex is compared against that from the native run of the benchmark. We can see that the gap between native and Rex is consistently around 10-20% when the lock contention probability is below 0.5. When the probability is greater than 0.5, the throughput drops quickly for both the native run and Rex's run. In all cases, Rex preserves the parallelism from the native run.

6.5 Query Performance and its Impact



(a) Query on secondary (b) Query on primary

Figure 9. Performance with different query semantics.

In Rex, read-only requests (queries) can be treated as update requests and go through the same replication protocol. In this case, the read requests will have the same semantics as in a non-replicated system. The read requests in the experiments shown in Figure 7 are executed in this way.

In addition, Rex supports executing read requests directly on primary or secondary replicas without going through the replication protocol, as discussed in Section 4. The semantics offered to query requests in this case differ depending on where the requests are executed: a read 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 Rex primary executes before consensus, whereas a secondary executes after consensus.

In this experiment, we use the lock server application to analyze the performance implications of these two different query semantics. 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 9. In both cases, the update throughput increases as the number of threads for update requests increases. However, the query throughput manages to stay mostly flat in Figure 7 (a) (for query-secondary), while noticeably decreasing in Figure 7 (b) (for query-primary) as the update throughput goes up more significantly. As threads on a secondary sometimes wait on synchronization events, there is a higher chance that a reader thread can grab that lock, compared with that on the primary, resulting in higher query throughput.

6.6 Checkpointing and Primary Changes

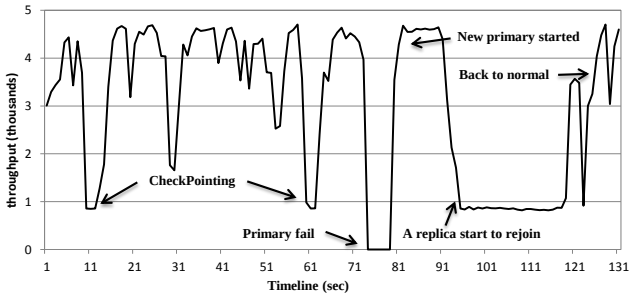


Figure 10. Failover of thumbnail server.

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 as the benchmark. We first create two checkpoints at an interval of 50 seconds and then kill the primary replica at the 71st second, restarting it 20 seconds later. Figure 10 shows the throughput fluctuation.

We measure the result of a stress test in which all CPUs are saturated. Because 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 affecting the overall performance as long as it is not promoted to the primary. We can see that during checkpointing the throughput drops for about 2 seconds and then recovers. At the point when 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 because of our aggressive flow control. If the system were not fully loaded, other replicas could 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

Rex uses Paxos [29, 30] as its underlying consensus protocol, which has become a standard practice [11, 24] in distributed systems. However, the Rex approach can also be applied to other replication protocols, such as primary/backup replication [1] and its variations (e.g., chain replication [39]).

Lamport points out in the *Generalized Paxos* protocol [31] 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. Similar to Generalized Paxos, *Generic Broadcast* [36] orders the delivery of messages only if needed, based on the semantics of messages provided by programmers.

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 for concurrent requests. Remus [13] achieves high availability by frequently propagating the checkpoints of a virtual machine to another. Eve [22] is closely related to Rex; detailed comparisons with Eve appear in Section 5. CBASE [25] leverages a user-defined parallelizer module, similar to the mixer in Eve, to identify the dependencies between requests. It then executes the independent requests in parallel on different replicas. Unlike Eve, it assumes a perfect parallelizer module. The LSA algorithm proposed by Basile et al. [4] ensures replica consistency by 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. HP's NonStop Advanced Architecture [8] captures synchronization events and replicates them to processes on the same machine. Hybrid Transactional Replication (HTR) [23] replicates transactions between machines using a hybrid mode of transactional replication and state machine replication. It improves replication performance by switching between the two modes for different workloads.

To capture and preserve partial order among requests, Rex leverages previous work of faithful record and replay of multi-thread programs. An incomplete sample of such work includes RecPlay [37], JaRec [19], ReVirt [17], R2 [20], PRES [35], ODR [2], SCRIBE [26], Respec [32] and its follow-on work [40, 41], and many others [12, 33]. However, mode change, hybrid execution, and online replay are the unique requirements that drive the design of Rex. Most of the previous work (e.g., Revirt, PRES, and ODR) target offline debugging and forensics, hence do not take these requirements into consideration. For example, PRES reduces recording overhead by making the replay take the extra overhead of searching for the identical executions, which is a reasonable tradeoff for offline debugging, but undesirable for the scenario of Rex. Although Respec is also designed for online replay, its implementation only allows replicas on the same machine because of its use of multi-thread fork, while Rex's replay happens on different secondary servers.

Deterministic parallel execution is another promising direction and can be done with new OS abstractions (e.g., Determinator [3] and dOs [7]), by runtime libraries (e.g., Kendo [34]) with compiler support (e.g., CoreDet [5]), or with hardware support (e.g., DMP [15], Calvin [21], and R-CDC [16]). With deterministic execution, traditional state-machine replication can be applied directly to multi-core environments. 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 [6].

8. Concluding Remarks

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

Acknowledgement

We are particularly grateful to our shepherd Lorenzo Alvisi for his valuable feedbacks and to the reviewers for their insightful comments. We would also like to thank Sean McDirmid for his suggestions that helped improve the paper.

References

[1] 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. IEEE, 1976.

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

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX symposium on operating systems design and implementation*, OSDI'10, pages 1–16. USENIX, 2010.

[4] 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.

[5] 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 15th international conference on architectural support for programming languages and operating systems*, ASPLOS '10, pages 53–64. ACM, 2010.

[6] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: how well does it actually pound nails? In *Proceedings of the 2nd workshop on determinism and correctness in parallel programming*, WODET '11, pages 448–465. ACM, 2011.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOs. In *Proceedings of the 9th USENIX symposium on operating systems design and implementation*, OSDI'10, pages 1–16. USENIX, 2010.

[8] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. In *Proceedings of the 35th international conference on dependable systems and networks*, DSN '05, pages 12–21. IEEE, 2005.

[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 symposium on networked systems design and implementation*, NSDI'11, pages 11–11. USENIX, 2011.

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

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th annual ACM symposium on principles of distributed computing*, PODC '07, pages 398–407. ACM, 2007.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM symposium on operating systems principles*, SOSP '11, pages 337–351. ACM, 2011.

[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. USENIX, 2008.

[14] J. Dean and S. Ghemawat. LevelDB: A fast and lightweight key/value database library by Google., 2011. `http://code.google.com/p/leveldb`.

[15] 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. ACM, 2009.

[16] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In *Proceedings of the 16th international conference on architectural support for programming languages and operating systems*, ASPLOS '11, pages 67–78. ACM, 2011.

[17] 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 USENIX symposium on operating systems design and implementation*, OSDI '02, pages 211–224. ACM, 2002.

[18] B. Fitzpatrick. memcached - a distributed memory object caching system, 2011. `http://memcached.org/`.

[19] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software: practice and experience*, 34:523–547, 2004.

[20] 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 symposium on operating systems design and implementation*, OSDI'08, pages 193–208. USENIX, 2008.

[21] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: deterministic or not? Free will to choose. In *Proceedings of the 2011 IEEE 17th international symposium on high performance computer architecture*, HPCA '11, pages 333–334. IEEE, 2011.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX symposium on operating systems design and implementation*, OSDI'12, pages 237–250. USENIX, 2012.

[23] T. Kobus, M. Kokociński, and P. T. Wojciechowski. Hybrid replication: state-machine-based and deferred-update replication schemes combined. In *Proceedings of the 33rd international conference on distributed computing systems*, ICDCS '13, pages 286–296. IEEE, 2013.

[24] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: state machine replication based on the Paxos protocol. Technical report, EPFL, 2011.

[25] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 34th international conference on dependable systems and networks*, DSN '04, pages 575–. IEEE, 2004.

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

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

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

[29] L. Lamport. The part-time parliament. *ACM transaction on computer systems*, 16(2):133–169, 1998.

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

[31] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft, 2005.

[32] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor 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, 2010.

[33] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM symposium on operating systems principles*, SOSP '11, pages 327–336. ACM, 2011.

[34] 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. ACM, 2009.

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

[36] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th international symposium on distributed computing*, DISC '99, pages 94–106. Springer Verlag, 1999.

[37] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM transaction on computer systems*, 17(2):133–152, 1999.

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

[39] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX symposium on operating systems design and implementation*, OSDI'04, pages 7–7. USENIX, 2004.

[40] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM symposium on operating systems principles*, SOSP '11, pages 369–384. ACM, 2011.

[41] 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, 2011.

[42] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad-hoc synchronization considered harmful. In *Proceedings of the 9th USENIX conference on operating systems design and implementation*, OSDI'10, pages 1–8. USENIX, 2010.