Eve: Execute-Verify Replication for Multi-Core Servers

Manos Kapritsos*, Yang Wang*, Vivien Quema†, Allen Clement‡, Lorenzo Alvisi*, Mike Dahlin*
*University of Texas at Austin †INRIA ‡MPI-SWS

Abstract: This paper presents Eve, a new Execute-Verify architecture that allows state machine replication to scale to multi-core servers. Eve departs from the traditional agree-execute architecture of state machine replication: replicas first concurrently and nondeterministically execute requests; then they verify, agree, and converge on the state and the outputs produced by a correct replica. Eve minimizes divergence through a mixer stage that applies application-specific rules to organize requests into batches of requests that are unlikely to interfere. Our evaluation suggests that Eve's unique ability to combine execution independence with nondeterminism enables high-performance replication for multi-core servers while offering tolerance to a wide range of faults, including elusive concurrency bugs.

1 Introduction

This paper presents Eve, a new Execute-Verify replication architecture that allows state machine replication to scale to multi-core servers.

State machine replication is a powerful fault-tolerance technique [10, 24, 39]. Historically, the essential idea is for replicas to deterministically process the same sequence of requests so that they traverse the same sequence of internal states and produce the same sequence of outputs.

Multi-core servers pose a challenge to this approach. To take advantage of parallel hardware, modern servers execute multiple requests in parallel. However, if different servers interleave requests' instructions in different ways, the servers' states and outputs may diverge. As a result, most state machine replication systems implement sequential execution where a replica finishes executing one request before beginning to execute the next [10, 25, 27, 40, 45, 51].

At first glance, a promising idea is to leverage recent efforts to enforce deterministic parallel execution. Unfortunately, existing techniques may require rewriting applications around new synchronization abstractions [7, 36] or may incur high overheads [14, 15, 47].

The Eve replication architecture allows parallel request execution with low overhead. Eve does this by eliminating the requirement that execution be deterministic. Instead, Eve allows execution to be internally nondeterministic, but it speculates that the results of parallel execution (including the system's important state [37] and outputs) will match across replicas.

To execute nondeterministically without violating the safety requirement of replica coordination, Eve turns on its head the tenet of traditional state machine replication: traditionally, deterministic replicas first *agree* on the order in which requests are to be executed and then *execute* them [10, 25, 27, 39, 51]. In Eve, replicas first speculatively *execute* requests concurrently and nondeterministically; they then *verify* and agree on the state and the outputs produced by a correct replica. If replicas diverge, Eve guarantees safety and liveness by rolling back and sequentially and deterministically re-executing the requests.

Critical to Eve's performance are mechanisms that ensure that despite nondeterminism, replicas seldom diverge, and that, if they do, divergence is efficiently detected and reconciled through state-transfer and fine-grained rollbacks. Eve minimizes divergence through a *mixer* stage that applies application-specific rules to organize requests into batches of requests that are unlikely to interfere. Note that if the underlying program is correct under unreplicated parallel execution, then delaying agreement until after execution and falling back on sequential re-execution guarantees that replication remains safe and live even if the mixer allows interfering requests in the same batch.

Eve's execute-verify architecture also allows us to use replication to protect against nondeterministic concurrency bugs ("Heisenbugs" [19].) Eve thus explores a region of the design space that falls short of Byzantine fault tolerance but that strengthens guarantees compared to standard crash-tolerance. Eve's robustness stems from two sources. First, Eve's *mixer* reduces the likelihood of triggering latent concurrency bugs by running only unlikely-to-interfere requests in parallel [23, 35]. Second, Eve's *execute-verify* architecture allows it to detect and recover from cases where concurrency causes executions to diverge regardless of whether the divergence was the result of different legal choices by different replicas or was caused by a Heisenbug.

This paper refines the fundamental assumptions of state machine replication. For decades, the traditional view has been that if all replicas execute independently, then producing the same sequences of outputs to a set of clients requires them to be deterministic [24, 39]. Eve

leverages the observation that although deterministic execution of a sequence of inputs is sufficient to produce identical outputs across replicas, it is not a fundamental requirement to maintaining execution independence. The practical consequence of refining our assumptions is that Eve can use its execute-verify architecture to bring together *nondeterminism* and *independence* to improve the replication of multi-core servers:

1. *Nondeterminism → Eve provides high-performance replication of multi-core servers.* Eve gains performance by avoiding the overhead of enforcing determinism. For example, in our experiments with the TPC-W benchmark, with 16 cores Eve achieves a 7.2*x* speedup for synchronous, crash-tolerant server-pair replication, while the original unreplicated server achieves a 8.8*x* speedup. Additionally, in our experiments Eve has competitive overheads and better scalability than the Remus primary-backup system [13], and Eve outperforms DPG [6], a replication system that handles multi-core servers by ensuring determinism across executions.

2. *Independence → Eve masks a wide range of faults.* Eve's architecture is general, and our prototype supports tunable fault tolerance [11], retaining state machine replication's ability to be configured to tolerate crash, omission, or Byzantine faults. Notably, even when an Eve system is configured to tolerate crash or omission failures, it can also mask some concurrency failures. Although we do not claim that our experimental results are general, we find them promising: for the TPC-W benchmark running on the H2 database, executing requests in parallel on an unreplicated server triggered a previously undiagnosed concurrency bug in H2 73 times in a span of 750K requests. Under Eve, our mixer *eliminated* all manifestations of this bug. Furthermore, when we altered our mixer to allow batches of conflicting requests, Eve detected and corrected this bug 82% of the times it manifested.

The rest of the paper proceeds as follows

2 Related Work

The challenge in implementing state machine replication on multi-core servers is its reliance on replica determinism: even when replicas are allowed to maintain some nondeterministic internal state, the abstract state they expose to the outside world must be deterministic [37].

Sources of nondeterminism can in principle be encapsulated as clients to the state machine [10, 39]. This works well when the opportunities for nondeterminism are relatively few: for instance, in UpRight [11] each execution replica receives with each batch of requests forwarded by the Order stage both a seed for the replica's pseudo-number generator and a time-of-day value to be used, if necessary, while processing the requests. The increasing ubiquity of multi-core processors, however, takes the challenge of nondeterminism to a new level: when nondeterminism is at the scale of individual memory accesses, it becomes infeasible to model it as a client to the state machine.

A recent paper suggests that it may be viable to enforce deterministic concurrency control in transactional systems [42], but the general case remains hard and, despite their apparent promise, recent efforts to enforce deterministic execution of multithreaded programs running on multi-core are unable to come to the rescue.

The Determinator operating system [2] can enforce determinism efficiently on multithreaded and multiprocess applications, but only if they are written using new deterministic synchronization abstractions [16, 30]. When it comes to legacy code, the best available option is to remove nondeterminism by leveraging the recently proposed Deterministic Process Group (DPG) abstraction [5, 6]. Unfortunately, the overhead imposed by DPG is high: for a non-replicated server running Apache, using DPG results in a four-fold reduction on the throughput achievable on the same hardware running nondeterministically [6]; when the server is replicated, throughput is further reduced by another factor of 6.

One alternative is to use a replication technique other than state machine replication. *Semi-active replication* [33] weakens state machine replication with respect to both determinism and execution independence: *one* replica, the primary, executes nondeterministically and logs all the nondeterministic actions it performs. All other replicas then execute by deterministically reproducing the primary's choices.

In this context, one may hope to be able to leverage the large body of work on deterministic multiprocessor replay [1, 15, 26, 31, 32, 38, 47, 49, 50]: while these systems focus mostly on debugging and security, they work by recording all nondeterministic events, which can then be used for replay. The problem, once again, is overhead: simply recording every memory access is too expensive to be practical [15]; trading replay accuracy to reduce the recording overhead [1, 32] no longer ensures that all correct replicas will produce identical outputs; logging only synchronization operations [4, 28, 29, 38] is insufficient for application with data races, a commonplace occurrence in real applications. DoublePlay [47] succeeds in reducing the logging overhead but at the cost of more than doubling the resources consumed by the primary. The best match for multi-core replication is Respec [26],

which supports online record and replay. Respec and Eve share some high-level design decisions that reduce overhead: they both rely on speculation (speculative logging in Respec; speculative execution in Eve) and both require correct replicas to agree on their final state and outputs rather than on the detailed execution path that led them there. Speculating on execution not only allows Eve to tolerate commission failures, which Respec does not tackle, but involves minimal overhead; Respec instead must still pay the cost of logging nondeterministic events, even if speculation helps reduce their number. As a result, in Respec the execution time of running an Apache benchmark actually *increases* as more execution threads are added.

Vandiver et al. [46] describe a Byzantine-tolerant semi-active replication scheme for transaction processing database systems. Their system supports concurrent execution of queries but its scope is limited: it applies to the subset of transaction processing systems that use strict two-phase locking (2PL).

In Hypervisor-based fault-tolerance [8] the primary sends to the backup the result of all the nondeterministic events it executes. The system assumes that both primary and backup execute requests sequentially.

Remus [13], a high availability solution for virtual machines running Xen [3], takes a different approach: it circumvents the overhead and complexity of explicitly handling the nondeterminism in multi-core executions by completely renouncing independent execution in favor of a primary-backup architecture [9]. In Remus, the backup does not execute requests, but instead passively absorbs state updates from the primary: since execution occurs only at the primary, the costs and difficulty of coordinating memory accesses are sidestepped. These advantages however come at a significant price in terms of fault coverage: in Remus, correct replica coordination depends on strong synchrony assumptions that are hard to enforce in high-availability deployments, where it may be desirable to have primary and backup located in different racks or in different data centers—leaving Remus at the mercy of the correctness of a switch. Further, Remus can only tolerate omission failures—all commission failures, including common failures such as concurrency bugs, are beyond its reach. Like Remus, Eve neither tracks nor eliminates nondeterminism, but it manages to do so without forsaking fault coverage; further, despite its stronger guarantees, Eve's performance on a single CPU is comparable to Remus's, and it appears more scalable (see Section 6) because it can ensure that the states of replicas converge without requiring the transfer of all modified state.

One of the keys to Eve's ability to efficiently combine nondeterminism and independent execution is the use of the mixer, which allows replicas to execute requests concurrently with low chance of interference. Kotla et al. [23] use a similar mechanism to improve the throughput of BFT replication systems. However, since they still assume a traditional agree-execute architecture, the safety of their system depends on the assumption that the rules used by the mixer never mistakenly place interfering requests in the same batch: a single unanticipated conflict can lead to a safety violation.

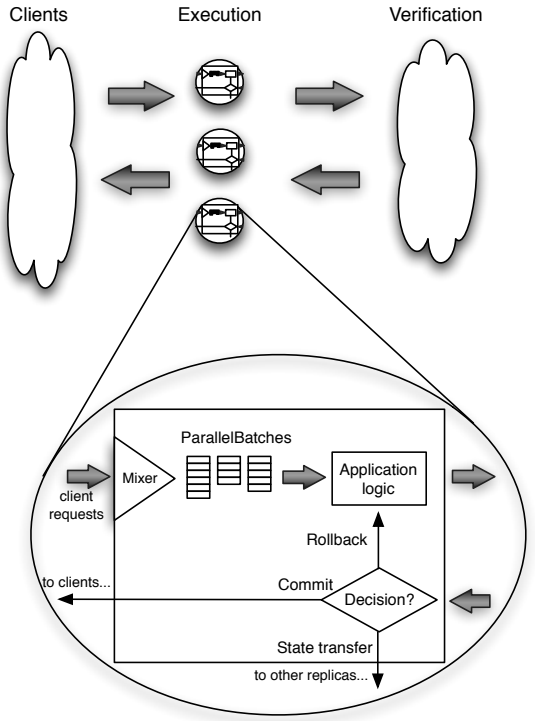

Figure 1: Overview of Eve.

Agreement can always be viewed as an ensemble of proposers, accepters, and learners [25], and protocol designers manipulate these stages to optimize for different goals [22, 44]. For example, both Eve and Zyzzyva [22] allow speculative execution that precedes completion of agreement. In these systems, nodes that can be considered learners (voters at clients in Zyzzyva and verifiers at servers in Eve) trigger additional agreement phases and roll-back/reexecution to repair unsuccessful speculation. However, where Zyzzyva speculates on the order of requests that will be agreed, Eve speculates not only on the request order and but also on how nondeterministic state machines will process requests.

3 Architecture overview

The goal of Eve is to replicate servers that run multiple requests in parallel. To achieve this, we have to rethink the architecture of replicated systems. Figure 1 shows an overview of Eve. This design is a departure from the traditional approach of "agree-execute." Instead, Eve ex-

ecutes requests without reaching agreement on the order [22]. After executing the requests, replicas try to agree whether they have reached the same state and produced the same results. If not, they take additional steps to repair the divergence.

Because our goal is to tolerate nondeterminism in the execution, correct replicas can execute requests differently and report different results. Although we were motivated to allow nondeterminism for performance and although Byzantine fault tolerance was not our primary goal, our techniques do draw from the BFT literature, and this convergence has the side effect of strengthening fault tolerance: Eve protects against concurrency bugs and can be configured to tolerate Byzantine faults.

In this section we give an overview of the design of Eve and in Sections 4-5 we describe the challenges and proposed solutions to efficiently implement this design.

3.1 Basic operation

In the common case, client requests are sent to the current *primary* execution replica, which groups them into batches and forwards them to all other execution replicas. Then each replica applies a deterministic, application-specific *mixer* on each batch. The mixing process divides a single *batch* into a sequence of *parallelBatches*, such that each parallelBatch contains requests that the mixer believes can be executed in parallel. Since the mixer is deterministic, it results in all replicas having the same sequence of identical parallelBatches.

Each replica executes the parallelBatches in the given order. Every parallelBatch is assigned a *sequenceNumber i*. After executing the i^{th} parallelBatch, a replica computes a hash of its application state and of the responses that correspond to requests in that parallelBatch. This hash, along with the hash for parallelBatch $i-1$ and the sequence number (i), constitute a *token* that is sent to the verification cluster in order to discern whether the replicas have diverged from each other.

Prior to verification, a number of things could go wrong. For example, different nodes may believe they are the primary and produce and execute different batches of requests or different execution nodes may produce different results when nondeterministically executing the same parallelBatch. Verification provides an end-to-end check in which nodes agree on the state of the system before they proceed.

If the verification cluster agrees that all (or sufficiently many) replicas converged on the same state, then the execution replicas can *commit* the results of that parallelBatch, which includes sending the corresponding responses back to the clients. Any replica that does not have the agreed upon state can request a state transfer from the other replicas.

If too few execution nodes converge, the verification cluster notifies the execution replicas. This notification includes the sequenceNumber and state hash of a stable, agreed upon parallelBatch to which they should roll back as well as the identity of the primary. Two things are done to ensure progress. First, to guard against concurrency conflicts missed by the mixer, execution nodes revert to sequential execution as in standard deterministic state machine replication [10, 11] until they succeed in making progress.

Second, to guard against failures of the primary, the verifiers identify a new primary if progress is too slow [10, 12]. When this happens, execution nodes discard unexecuted parallelBatches and begin processing batches from the new primary using the stable checkpoint. Clients eventually timeout, learn the new primary, and resend their requests there.

The internals of the verification stage closely resemble agreement in previous replication systems [10, 11, 25, 40, 48], and we omit the details. It should be noted that in previous designs where agreement precedes execution, separating agreement and execution raises an issue: the system must ensure agreement on each execution checkpoint, but there are not enough execution nodes to run any agreement protocol. In such systems, the solution is to add an additional "upstream" flow where hashes of execution checkpoints are fed back through the agreement nodes [11, 48]. Eve avoids this complexity: agreement always *follows* execution and naturally covers all earlier phases (e.g., request ordering, responses, and execution checkpoints.)

4 Design Issues

4.1 Configurations

We primarily target asynchronous environments where the network can arbitrarily delay, reorder, or lose messages without imperiling safety. For liveness, we require the existence of synchronous intervals during which the network is well behaved and messages sent between two correct nodes are received and processed with bounded delay. Because synchronous primary-backup with reliable links is a practically interesting configuration [13], we also evaluate Eve in a server-pair configuration that—like primary-backup [9]—relies on timing assumptions for both safety and liveness.

Our primary focus is on tolerating omission failures. However, our design is general. We target systems that are *live*, i.e. provide a response to client requests, despite up to u arbitrary failures. We ensure that all responses accepted by correct clients are *correct* despite up to r commission failures and any number of omission failures [11]. Commission failures include all failures that are not omission failures, and the union of omission and commission failures are Byzantine failures. Note, how-

	General	Omission-only (r=0)	BFT (u=r=f)
N_{exec}	$u+max(u,r)+1$	$2u+1$	2f+1
N_{verify}	$2u+r+1$	$2u+1$	3f+1

Figure 2: Configurations for asynchronous replication

ever, that we assume that failures do not break cryptographic primitives; i.e., a faulty node can never produce a correct node's signature or MAC.

Figure 2 summarizes the minimum number of execution and verification replicas required for various configurations in an asynchronous environment. The omission-only column shows the standard number of replicas needed to remain safe and live despite u omission failures, assuming no commission failures. Note that when Eve is configured this way, it also tolerates concurrency failures (Section 4.2.) For completeness, we include the BFT column, which shows the number of replicas needed to remain safe and live despite f arbitrary failures.

4.1.1 Synchronous process pair

Synchronous 2-node replication is the minimal system able to tolerate 1 crash failure. It requires a reliable synchronous network for safety and liveness [9]. To guarantee liveness, if one node crashes, the other becomes solely responsible for processing requests and clients must accept that node's responses. For safety, we must ensure that there is only one primary at a time.

This safety property is commonly enforced by engineering a monitor process on each node, along with conservative timeouts and a simple, highly reliable network (e.g., a dedicated direct link). Rather than build our own monitor, we use ZooKeeper [21] to ensure that across all clients and servers, at most one server is considered to be the primary (assuming ZooKeeper's synchrony assumptions hold).

To run Eve, the primary selects a set of requests to be in batch i and sends the batch to the secondary. The primary and the secondary then apply the mixer to the requests and execute the resulting parallelBatches. Following execution of a parallelBatch, the secondary computes a hash of the state that was updated during that parallelBatch and sends this hash to the primary. The primary detects divergence by checking the hash of the secondary against its own state update. If these two match, the primary responds to the client. Otherwise, the primary rolls back its state and reexecutes sequentially and orders the secondary to do the same. If the secondary fails to provide a hash within a timeout, because of a crash or a concurrency bug, then the primary times out and issues a rollback, as above. If a replica has crashed, this failure is eventually detected by ZooKeeper and the remaining node assumes the responsibility of executing the requests by itself. When the crashed node recovers, the primary brings it up to date by initiating a state transfer and starts operating again in the primary-secondary mode.

4.2 Tolerating Heisenbugs

Eve's mixer and nondeterministic, replicated parallel execution provide opportunities to mask some concurrency failures. The basic ideas are simple.

For the mixer, our intuition is that systems that avoid executing requests that appear likely to access the same data will also tend to avoid triggering Heisenbugs. The effectiveness of the mixer will depend on the chosen application-specific rules and on details of the application and workload.

For detection and correction via nondeterministic, replicated parallel execution, we speculate that at least some Heisenbugs depend on the interleavings of requests and threads and that reexecuting requests in a different way may sometimes avoid them. This is not a new idea [18, 19].

Concurrency failures. We define a concurrency failure as follows:

> *concurrency failure*—a failure that can manifest during parallel execution but not during sequential execution.

Concurrency failures are a subset of Byzantine failures. Concurrency failures include both omission failures (e.g., a node could get stuck) and commission failures (e.g., a node could produce an incorrect output or transition to an incorrect state.) However, concurrency failures have an important property that cannot be assumed to hold in general for Byzantine failures: concurrency failures are easy to repair. If a system detects a concurrency failure, it can repair it via rollback and sequential reexecution.

Not all divergences among replicas are concurrency failures; systems can have legal nondeterminism. A divergence is a concurrency failure only if the resulting output or state is "illegal." Of course, Eve doesn't know if a divergence is simple nondeterminism that would be "legal" on a single node or a concurrency failure that would be "illegal" on a single node. This informality is OK. Eve resolves both types of divergence in the same way, so we do not need to draw a more precise line between these two cases.

Conversely, not all concurrency failures result in divergence. As in any system that uses redundancy to mask failures, Eve is vulnerable to *correlated failures* and cannot mask concurrency failures if all nodes fail in exactly the same way.

This said, Eve's departure from traditional state machine replication architectures helps here. Eve's mixing stage identifies requests that should not interfere, and different Eve replicas may schedule the parallel execution

	Asynchronous	Synchronous
Execution replicas	n = 2u+1	n = u+1
Concurrency failures tolerated		
Always (worst case)	u	0
During good intervals	n-1	n-1

Figure 3: Number of concurrency failures tolerated by Eve when configured to tolerate a desired number of omission failures u.

of these requests differently to introduce diversity among replicas and reduce the likelihood that a concurrency bug manifests in exactly the same way at all nodes. Below, we bound the number of concurrency failures that Eve is guaranteed to tolerate, and in Section 6.4 we evaluate Eve's effectiveness at masking concurrency failures in one case study.

Also note that not all Heisenbugs are concurrency failures [19]. Although this section is called *Tolerating Heisenbugs*, we only claim that Eve can tolerate *some* manifestations of Heisenbugs, not all of them.

Finally note that we restrict our attention to concurrency failures by the state machine being replicated, and we assume that the Eve infrastructure is not subject to concurrency failures (though it is subject to other failures.) For example, we assume that if the only failure in a system is a concurrency failure, Eve can always roll the state machine back and reexecute requests.

How many concurrency failures can Eve tolerate? As noted in Section 4.1, Eve can be configured to tolerate Byzantine failures. Since the Byzantine model covers any failure, clearly Eve can be configured to tolerate some number of concurrency failures.

However, we expect it to be more common to configure systems for omission fault tolerance than for Byzantine fault tolerance, and Eve can provide protection in these cases as Figure 3 indicates. This figure and the remainder of this subsection assume that the system is not configured with extra redundancy to protect safety against commission failures (i.e., r = 0.)

In the asynchronous case, the verifier must always wait for at least u+1 responses from the 2u+1 execution nodes.[1] If any of these u+1 responses differ, Eve rolls back and sequentially reexecutes the parallelBatch. Thus, Eve is guaranteed to detect and correct u concurrency failures. Of course, even if all u+1 responding execution nodes suffer concurrency failures, Eve could get lucky and have them fail in different ways, triggering a rollback, but Eve can only *guarantee* that it masks concurrency failures when at least one of the $u+1$ responses is correct.

If, instead, a synchronous model is assumed, fewer execution replicas are needed because it is possible to identify failed nodes by their silence. This reduces replication costs, but it also reduces the "spare" redundancy that we exploited in the asynchronous case. In particular, under a synchronous model, the system can sometimes operate with only a single active execution node, and thus Eve cannot always promise to correct even a single concurrency failure. Nonetheless, Eve still provides substantial protection to synchronous configurations.

Extra protection during good intervals. During "good intervals" when all nodes and the network are working well, there is spare redundancy that Eve uses to boost its "best effort" protection to tolerate concurrency bugs at n-1 execution replicas in both the synchronous and asynchronous cases.

For example, in the synchronous process-pair case, when both execution nodes are alive, the primary receives both execution responses, and if they don't match, it orders a rollback and sequential replay. Thus, the synchronous process-pair configuration masks one-node concurrency failures when both nodes are functioning. We expect this to be the dominant common case.

More generally, during a *good interval* Eve may transition to *extra protection mode* that masks n-1 concurrency failures for the remainder of the good interval.

In a *good interval*, no faults or timeouts occur except those caused by a concurrency failures. Thus, during the interval all nodes are either correct or suffer concurrency failures, and correct nodes and the network are timely.

When Eve is in *extra protection mode* (EPM), after the verifiers receive the minimum number of execution responses necessary for progress, they continue to wait for up to a short timeout to receive all n responses. If the verifiers receive all n matching responses, they commit the response. Otherwise, they order a roll-back and sequential replay. Then, if they receive n matching responses within a short timeout, they commit the response and remain in EPM. Conversely, if sequential reexecution does not produce n matching and timely responses, they suspect a non-concurrency failure and exit EPM.

Notice that as long as Eve remains in EPM, safety is assured even if $n-1$ execution nodes suffer a concurrency failure (or any other failure) for a parallelBatch. Also notice that EPM does not affect liveness, since repeated rollbacks cause Eve to exit EPM. Finally notice that during a good interval, once Eve enters EPM, it stays there until the good interval ends (e.g., due to a non-concurrency node failure or network timeout.)

The difference between the synchronous and asynchronous case is how nodes make a decision to enter EPM. In the synchronous case, the decision is easy because the current primary always knows which replicas are alive, and it always receives responses from all live

[1] It has to wait for u+1 because u nodes could be slow but correct and then u of the nodes that did respond could crash, leaving just 1 node able to continue with the current state.

replicas. In the asynchronous case, Eve enters EPM after executing k consecutive batches for which all n execution replicas provided matching, timely responses.

4.3 Mixer design

Nondeterministic parallel execution will only be productive if divergence is rare. Eve therefore uses a mixer to identify requests that may usefully be executed in parallel. Such a mixer must have low false negative and false positive rates. False negatives will cause conflicting requests to be executed in parallel, creating the potential for divergence and rollback. False positives will cause requests that could have been successfully executed in parallel to be serialized, reducing the parallelism of the execution. Note however that Eve remains safe and live independent of the false negative and false positive rate of the mixer. A good mixer is just a performance optimization (although an important one).

The mixer is by nature application-specific. The basic idea is to parse each request and predict what parts of the state it will access. Depending on the application this can vary from single application objects and file system files to high-level abstractions such as database rows or tables. After getting a prediction for each request the mixer simply avoids putting two conflicting requests in the same parallelBatch. Two requests conflict when they access the same object in a read/write or write/write manner.

We expect that a good mixer can be written for most application and workloads. Although implementing a perfect mixer might prove tricky for some cases, a mixer that is allowed to misclassify requests occasionally is much easier to achieve. Moreover, we can use feedback from the system to improve the mixer over time (e.g. by logging parallelBatches that caused rollbacks). In our experience with the H2 Database Engine and the TPC-W workload, we observe that the mixer can be inferred from the tables that each transaction accesses. As demonstrated in section 6, we achieve good parallelism (acceptably few false positives) and do not observe any rollbacks (few or no false negatives). Kotla et al. [23] also report that they were able to implement conflict avoidance filters with good parallelism.

5 Implementing state management

Moving from an agree-execute architecture to an execute-verify one puts pressure on the implementation of state checkpointing, comparison, rollback, and transfer. For example, in Eve, after executing a parallelBatch we have to compute a hash of a new checkpoint before we respond to the clients. In contrast, traditional approaches only need to agree on the order of requests before responding to the clients. In these systems checkpoints are created and compared less often (e.g. when garbage collecting the request log).

At a high level our approach is straightforward—we store the state using a copy-on-write Merkle tree to provide both efficient state comparison and fine grained checkpointing and rollback.

Our implementation uses two ideas from BASE [37]. First, it includes only the subset of state that determines the operation of the state machine. We omit temporary state, like an IP address or a TCP connection, which can vary across different nodes but has no semantic effect on abstract state or outputs. Second, it provides an abstraction wrapper on some objects to mask variations across different replicas.

Compared to BASE, three sets of issues arise: maintaining a deterministic Merkle tree structure under parallel execution, parallel hash generation, and Java language and runtime issues.

5.1 Deterministic Merkle trees

To generate the same checksum for equivalent states, different replicas must put equivalent objects at the same location in the Merkle tree. In single-threaded execution, determinism can be achieved easily by adding an object to the tree when it is created. Determinism is more challenging in multithreaded execution when objects can be created concurrently.

There are two intuitive ways to solve the problem. The first option is to make memory allocation synchronized and deterministic, but this negates efforts toward concurrent memory allocation [17, 41]. Furthermore, it is unnecessary, since the allocation order usually does not fundamentally affect replica equivalence. Second, an ID could be generated based on object content, and this ID might be used to determine an object's location in the tree; this approach does not work, since many objects have the same content, especially at creation time.

Our solution is to defer adding newly created objects to the Merkle tree until the end of the batch. Eve scans existing modified objects, and if one contains a reference to an object not in the tree, it adds that object into the next empty slot in the tree. After that Eve iteratively repeats the process for all newly added objects.

The scanning is deterministic for two reasons: First, existing objects are already put at deterministic locations in the tree. Second, for a single object, Eve can iterate all its references in a deterministic order. Usually, we can use the order that references are defined in a class. There are some problems with special classes, like Hashtable, and we will discuss them in Section 5.3.2.

Note that scanning new objects is single-threaded. We have not attempted to parallelize it, because it is much less expensive than hash generation, which we do parallelize.

5.2 Generating Merkle hashes in parallel

For scalability, the Merkle tree must generate its checksum in parallel. Eve's approach is as follows: First, when an object is updated, the path from the object to the root of the tree is marked as dirty. At the end of a batch, Eve divides the whole tree into a number of subtrees and assigns each subtree with dirty nodes to a thread. Then each thread calculates its subtree's new hash, using the dirty bit to avoid unchanged branches. Finally, after all worker threads finish, the main thread collects all hashes from different subtrees and generates a global hash.

5.3 Java Language & Runtime

Our prototype is implemented in Java. Java provides us with important features that help us implement the previous deterministic scanning algorithm, but it also raises several challenges: First, some of its basic data structures, like Hashtables, can be equivalent but have different internal states. Putting them directly into the Merkle tree will of course yield spurious diverged checksums. Our solution is to provide wrappers for certain classes to abstract away differences across machines [37]. Second, as long as the Merkle tree holds a reference to an object, this object is not eligible for Java's automatic garbage collection (GC); our solution is to periodically perform a Merkle-tree-level garbage collection.

5.3.1 Java Benefits

Our state management system is based on objects and a typed programming language, Java, instead of on memory bytes. This gives us several benefits. First, it's easy to differentiate references from other data, which simplifies scanning for new objects to add to the tree. When generating a checksum for an object, it also allows us to replace a nondeterministic memory address by its deterministic location in the tree, which eliminates the problem of nondeterministic memory allocation. Second, Java provides protection against certain kinds of bugs, such as buffer overflow. This makes Eve's runtime system more likely to survive concurrency bugs.

5.3.2 Data structure abstraction

This problem comes from standard set-like data structures in Java. Those data structures, including HashTable, HashSet, etc. are widely used in real applications. According to the definition of Set, two sets are equivalent if all their elements are equivalent, independent of their order. However, if we check the implementation of these data structures, few of them are order oblivious under concurrent executions. For example, a Hashtable implemented using buckets is sensitive to adding order if added objects fall into the same bucket. For this kind of data structures, if we use the naive approach, two equivalent Sets will be identified as diverged. Rollback and safe reexecution could fix this, but since these data structures are used so widely and frequently, we do not want them to cause frequent rollbacks.

Eve creates a wrapper for Set-like data structures: when a checksum is needed, the wrapper creates a deterministic list of all its elements, completely ignoring the internal data structure. When an iteration is needed, the wrapper also provides that deterministic iterator. The wrapper sorts the Set elements to achieve deterministic order. For some data structures, a specialized wrapper could achieve better performance. For example, for a bucket-based hashtable, we could sort inside each bucket instead of sorting all elements. Note that it's possible that elements are not sortable. This case also requires a specialized wrapper. Eve uses the request ID plus the number of objects added by the request to order such objects. Our experience with applications has been that most Set-like data structures use sortable values as keys, like Integer, Long, String, etc. Even more complex data types usually have a primitive identifier, which can be used to sort. Thanks to the well-defined Java interface, we only need to create two general purpose wrappers: one for Set and one for Map. In Java, almost all Set-like data structures implement one of these two interfaces.

5.3.3 Automatic garbage collection

One of the most important features of the Java Virtual Machine (JVM) is garbage collection (GC), in which the JVM can deallocate an object if it is not reachable from any root object. However, GC cannot collect an object if the Merkle tree always holds a reference to that object. Requiring the programmer to remove the object from the tree manually is a solution, but this eliminates the benefits of automatic GC. Our solution is to periodically perform a Merkle-tree-level GC. We use a mark and sweep algorithm, similar to JVM's GC: start scanning from some root objects and scan their references recursively. If some objects are not touched during this scan, they can be removed from the tree. From our experience with the applications, objects in the tree typically have a long lifetime, since they are "important" objects, so the Merkle-tree-level GC can be performed less frequently than JVM's GC.

6 Evaluation

In our evaluation, we try to answer the following questions:

- What is the throughput gain that Eve provides compared to a traditional sequential execution approach?
- How do Eve's overheads compare to those of unreplicated multithreaded execution and to alternative replication approaches?
- How is Eve's performance affected by the mixer and by other workload characteristics?

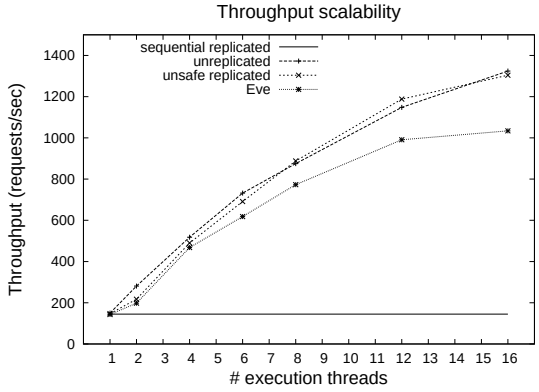

Figure 4: This graph shows the throughput of a) a sequential unreplicated server b) a multithreaded unreplicated server c) an unsafe replicated server (requests executed in parallel, but no verification performed) and d) an Eve replicated multithreaded server. The replicated experiments use a server-pair configuration on 16-core machines. The servers run the TPC-W browsing workload on the H2 Database Engine.

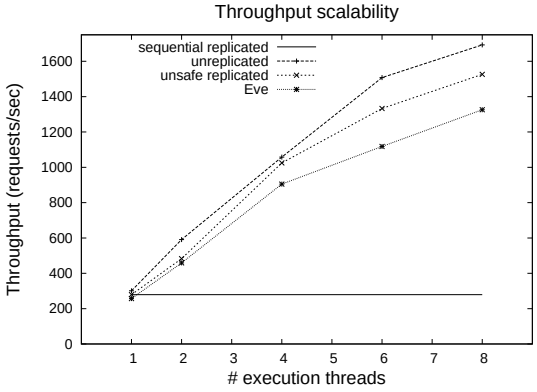

Figure 5: This graph shows the throughput of a) a sequential unreplicated server b) a multithreaded unreplicated server c) an unsafe replicated server (requests executed in parallel, but no verification performed) and d) an Eve replicated multithreaded server. The replicated experiments use an asynchronous configuration (3-way replication) on 8-core machines. The servers run the TPC-W browsing workload on the H2 Database Engine.

- How well can Eve mask concurrency bugs?

We address these questions by using a simple key-value store application, the H2 Database Engine, and the Apache Web Server. We use the key-value store application to perform microbenchmark measurements of Eve's sensitivity to various parameters. Specifically, we vary the amount of execution time required per request, the size of the application objects and the accuracy of our mixer, both in terms of false positives and false negatives. For the H2 Database Engine we use 3 TPC-W workloads. For brevity, we will present the results of the browsing workload, which has more opportunities for concurrency.

Note that our prototype omits some of the features described above. Specifically, although we implement the extra protection mode (EPM) optimization from Section 4.2 for synchronous server-pair replication, we do not implement it for our asynchronous configurations. Also, our current implementation does not handle applications that include objects for which Java's *finalize* method modifies state that needs to be consistent across replicas.

We run our experiments on an Emulab testbed with 24x 4-core Intel Xeon @2.4GHz, 3x 8-core Intel Xeon @2.66GHz, and 2x 16-core Intel Xeon @1.6GHz, connected with a 1Gb Ethernet. The 2 16-core machines are only enough to support our server-pair experiments. Therefore, our asynchronous replication experiments run on 8-core machines. For the asynchronous configuration we use 3 execution and 4 verifier nodes, which is sufficient to tolerate 1 arbitrary fault ($u = 1$, $r = 1$).

6.1 H2 Database with TPC-W

Figures 4 demonstrates the performance of Eve for the H2 Database Engine [20] with the TPC-W workload [43], for a synchronous server-pair configuration with 2 execution nodes. We observe that with 16 execution threads, Eve achieves a speedup of 7.2x, compared to the sequential execution that previous state machine replication systems have used. That compares favorably with the 8.8x speedup that is achieved by an unreplicated server using 16 cores.

For reference, we also measure the performance of an *unsafe* multithreaded replicated system that executes requests in parallel but that omits Eve's correctness checks. The performance of this system is close to that of the unreplicated server. Note, however, that this arrangement is not only unsafe in theory but also in practice. We find that the value produced for a given request frequently differs between the primary and secondary. To complete the benchmark, we only send the primary's answers to the clients.

Figures 5 demonstrates the performance for a similar experiment, only this time in an asynchonous replication setting with 3 execution servers. We observe a similar scalability pattern for this figure. Note that in this experiment we use 8-core machines and that each core of our 8-core machines is faster than the cores of our 16-core machines, so the y-axis is shifted. At 8 cores, Eve's speedup compared to the single-threaded unreplicated server is 4.4x, while the multithreaded unreplicated server has a speedup of 5.6x.

In both configurations and across all runs and for all

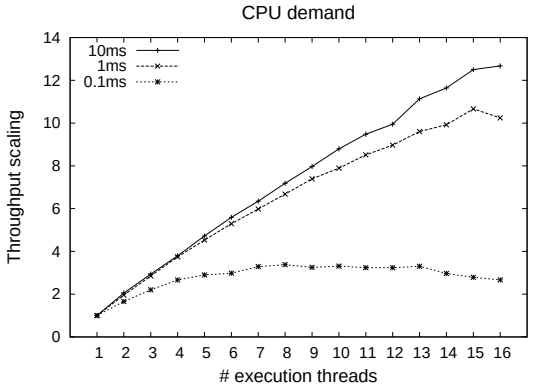

Figure 6: This graph shows the impact of CPU demand per request on Eve's throughput scalability. The workloads are 10ms, 1ms, and 0.1ms of CPU time per request. The servers run our key-value store application with a server-pair configuration on 16-core machines. The application object size is 1KB.

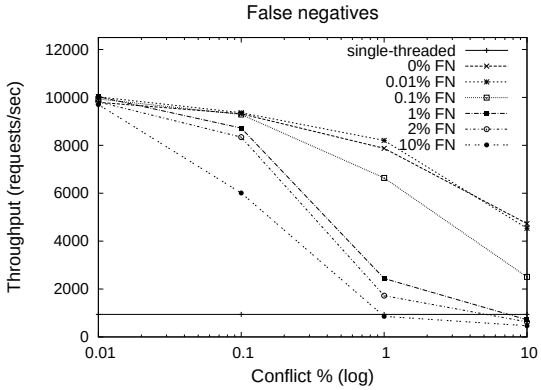

Figure 8: This graph shows the impact of conflict chance and false negative rate on Eve's throughput. We vary the conflict chance from 0.01% to 10% and the false negative rate from 0% to 10%. The servers run our key-value store application with a server-pair configuration on 16-core machines. Application object size is 1KB and the CPU time per request is 1ms.

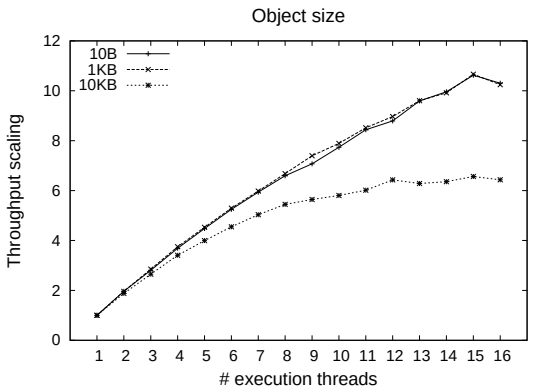

Figure 7: This graph shows the impact of application object size on Eve's throughput scalability. Object sizes are 10bytes, 1KB, and 10KB. The servers run our key-value store application with a server-pair configuration on 16-core machines. The CPU time per request is 1ms.

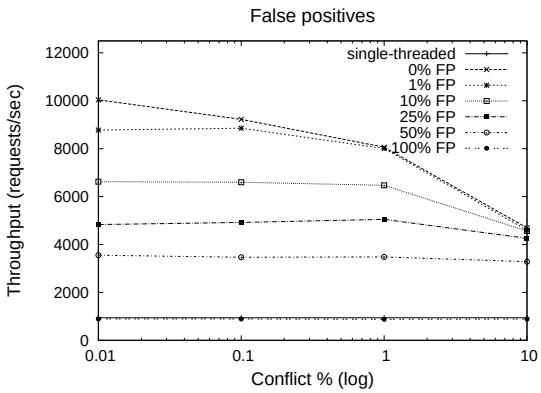

Figure 9: This graph shows the impact of conflict chance and false positive rate on Eve's throughput. We vary the conflict chance from 0.01% to 10% and the false positive rate from 0% to 100%. The servers run our key-value store application with a server-pair configuration on 16-core machines. Application object size is 1KB and CPU time per request is 1ms.

data points, Eve never needs to roll back. This indicates that our simple mixer never parallelized requests it should have serialized. At the same time, the good speedup indicates that it was adequately aggressive in identifying opportunities for parallelism.

6.2 Microbenchmarks

In this section, we report on how various parameters affect Eve's performance. Due to lack of space, we only show the graphs for the server-pair configuration; the results for asynchronous replication are similar. Unless varied by a particular experiment, the default workload incurs 1ms of execution time per request and the application object size is 1KB.

Figure 6 shows the impact of varying the CPU demand of the workload. We observe that heavier workloads (10ms of execution time per request) scale well, up to a 12.5x on 16 cores compared to sequential execution. As the workload gets lighter, the effect of our overhead becomes more pronounced. Speedups fall to 10x for 1ms/request and to 3.3x for 0.1ms/request. The 3.3x scaling is partially an artifact of our inability to fully load the server when requests are light. In our workload generator, clients have 1 outstanding request at a time, thus

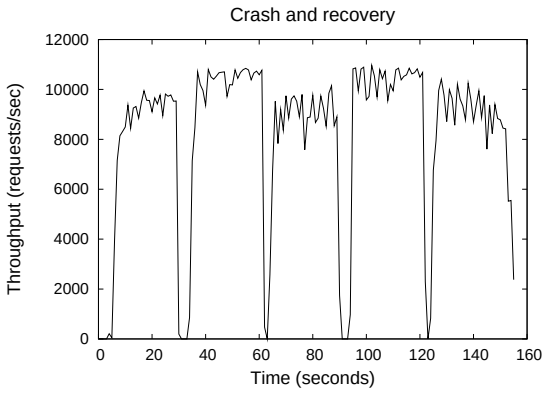

Figure 10: This graph shows the impact of node crash and recovery on throughput for an Eve server-pair configuration on 16-core machines.

requiring a high number of clients to saturate the server; this causes our server to run out of sockets. We measure our server CPU utilization during this experiment to be about 30%.

In the graph, we plot throughput scalability, so that trends across workloads are apparent. The absolute peak throughputs in requests per second are 25.2K, 10.0K, 1242 for the 0.1ms, 1ms, 10ms lines, respectively.

The next experiment explores the impact of the application object size on the system throughput. We expect larger objects to decrease the performance, since calculating the hashes of these objects will become the bottleneck. We run the experiment using object sizes of 10 bytes, 1 KB, and 10 KB, respectively, with 1ms of processing time per request in all cases. Figure 7 shows the results. In this graph, we plot scalability rather than throughput to better indicate the trends across workloads. The absolute peak throughput values in requests per second are 10.0K, 10.0K, 5.6K for the 10byte, 1KB, 10KB lines, respectively.

Next, we evaluate the sensitivity of Eve to inaccuracy in the mixer. Specifically, we want to explore the limits of tolerance to false negatives (misclassifying conflicting requests as non-conflicting) and false positives (misclassifying non-conflicting requests as conflicting). The effect of these parameters is measured as a function of the pairwise conflict chance; the chance that 2 requests have a conflict. In practice, we achieve this by having each request modify 1 object and then varying the number of application objects. For example, to produce a 1% conflict chance, we create 100 objects. Similarly, a 1% false negative rate means that each pair of conflicting requests has a 1% chance of being classified as non-conflicting.

Figure 8 shows the effect of false negatives on throughput. First notice that, even for 0% false negatives, the throughput drops as the pairwise conflict chance increases due to the decrease of available parallelism. For example, if a batch has 100 requests and each request has a 10% chance of conflicting with each other request, then a perfect mixer is likely to divide the batch into about 10 parallelBatches, each with about 10 requests.

When we add false negatives, we add rollbacks, and the number of rollbacks increases with both the underlying conflict rate and the false negative rate. Notice that the impact builds more quickly than one might expect because there is essentially a birthday "paradox"—if we have a 1% conflict rate and a 1% false negative rate, then the probability that any pair of conflicting requests is misclassified is 1 in 10000. But in a batch of 100 requests, each of these requests has about a 1% chance of being party to a conflict, which means there is about a 39% chance that a batch of 100 requests contains an undetected conflict. Furthermore, with a 1% conflict rate, the batch will be divided into only a few parallelBatches, so there is a good chance that conflicting requests will land in the same parallelBatch. In fact, in this case we measure 1 rollback per 7 parallelBatches executed. Despite this high conflict rate and this high number of rollbacks, Eve achieves a speedup of 2.6x compared to sequential execution.

Figure 9 shows the effect of false positives on throughput. As expected, increased false positive ratios can lead to lower throughput, but the effect is not as significant as for false negatives. The reason is simple: false positives reduce the opportunities for parallel execution, but beyond that they don't incur any additional overhead.

From these experiments, we conclude that Eve does require a good mixer to achieve good performance. This requirement does not particularly worry us. We found it easy to build a mixer that (to the best of our knowledge) detects all conflicts and still allows for a good amount of parallelism. Others have had similar experience [23]. Although creating perfect mixers may be difficult in some cases, we speculate that it will often be feasible to construct mixers with the low false negative rates and modest false positive rates needed by Eve.

6.3 Failure and recovery

In Figure 10, we demonstrate Eve's ability to mask and recover from failures. In the server-pair configurations we run an experiment where we kill the primary node n_1 at $t = 30$ seconds and recover it at $t = 60$ seconds (by which time the secondary n_2 has become the new primary). We then kill the new secondary (n_1) at $t = 90$ seconds and recover it at $t = 120$ seconds. We present the throughput over time. We observe that the throughput drops to zero after the first failure until the backup realizes that the primary is dead. It then assumes the role of the primary and starts processing requests. Note that the throughput during this period is higher because the

	Group all	1% FN	0.5% FN	0.1% FN	Default Mixer
Times bug manifested	73	51	29	4	0
Fixed with rollback	60	38	18	3	0
All identical	13	13	11	1	0
Throughput	1104	1233	1240	1299	1322

Figure 11: Effectiveness of Eve in masking concurrency bugs when various mixers are used.

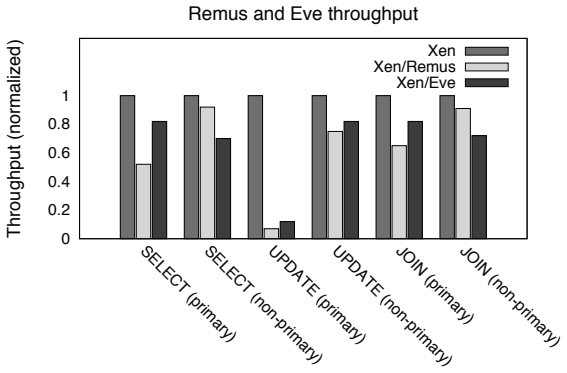

Figure 12: This graph shows the throughput of Remus and Eve normalized against that of an unreplicated H2 Database Engine, running on Xen with 1 virtual CPU. The various workloads are: SELECT, UPDATE and JOIN on primary and non-primary key. Both systems use a 2-node configuration.

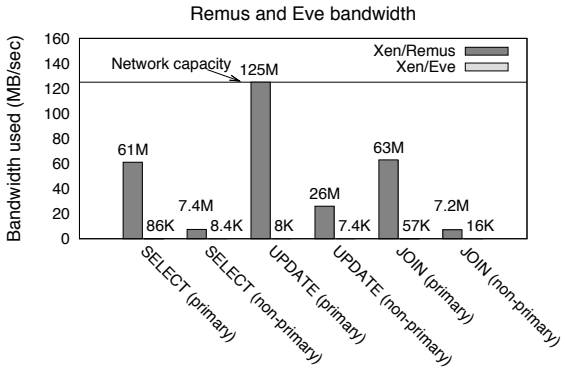

Figure 13: This graph shows the bandwidth utilization of Remus and Eve running the H2 Database Engine on Xen with 1 virtual CPU. The various workloads are: SELECT, UPDATE and JOIN on primary and non-primary key. Both systems use a 2-node configuration.

new primary knows that the other node is crashed and does not have to send any messages to it. At $t = 60$, the first node recovers. Again, the throughput drops to zero for a short span while the primary catches up the newly recovered node. Then the throughput returns to its original value. The process repeats when n_1 crashes again at $t = 90$ seconds and recovers at $t = 120$ seconds. When the secondary fails, the primary has to wait until it learns that the secondary is dead, before it resumes operation in single-node mode.

6.4 Concurrency failures

In this section, we evaluate Eve's ability to mask concurrency failures. We configure the system as server-pair with 16 execution threads and run the TPC-W browsing workload on the H2 Database Engine with various mixers. H2 has a previously undiagnosed concurrency bug in which a row counter is not incremented properly when multiple requests access the same table in *read_uncommitted* mode. Our standard mixer completely masks this bug because it does not let requests that modify the same table execute in parallel. By introducing less accurate mixers we can explore how well Eve's second line of defense—parallel nondeterministic execution—works in masking this bug.

We report the number of times that the bug manifested in one or both replicas. When the bug manifests only in one replica, Eve detects that the replicas have diverged and repairs the damage by rolling back and reexecuting sequentially. If the bug happens to manifest in both replicas in the same way, Eve will not detect it. Table 11 summarizes our results.

The first column shows the results when there is a trivial aggressive mixer that places all requests of batch i in the same parallelBatch. In this case, all requests that arrive together in a batch are allowed to execute in parallel. Naturally, this case has the highest number of bug manifestations. We observe that even when the mixer does no filtering at all, Eve masks 82% of the instances where the bug manifests. In the remaining 18% of the cases, the bug manifested in the same way in both replicas and was not corrected by Eve. In columns 2 through 4, we introduce mixers with high rates of false negatives. This results in fewer manifestations of the bug, with Eve still masking the majority of those manifestations. In the fifth column, we show results for our default mixer, which (to the best of our knowledge) does not introduce false negatives. In this case, the bug does not manifest at all.

Although we do not claim that these results are general, we find them promising.

6.5 Other approaches

Remus [13] is a passive replication system that uses Virtual Machines (VMs) to frequently send state updates from the primary to the backup. An advantage of this approach is that it is simple and requires no modifications to the application. A drawback of this approach is that it aggressively utilizes network resources to keep the backup consistent with the primary. The issue is aggravated by two properties of Remus. First, Remus does not make fine-grain distinctions between state that is required for the state machine and temporary state, thus sending more pages than those required for replication. Second, Remus operates on the VM level, which forces it to send entire pages, rather than just the modified objects. Also, because of Remus's passive replication, primary-backup approach, it tolerates a narrower range of faults than Eve.

The Remus paper [13] reports numbers only for 1 CPU, and in all our experiments it crashed with more CPUs than that. Remus developers report that Remus can run with 2 CPUs [34]. Also, in our experiments Remus would cause the network of the guest OS to become unresponsive, so we disabled the network and had the server read the workload from a local trace file, which greatly reduces the overhead of Remus. Also, Remus consistently crashes when we attempt to run the TPC-W workload. We therefore report on microbenchmark workloads with specific SQL queries taken from TPC-W. For fair comparison, we run Eve itself on Xen, to incorporate any overheads Xen itself may incur.

In Figure 12 we plot the throughput of Remus and Eve normalized against the throughput achieved by an unreplicated H2 Database Engine, running on Xen with 1 virtual CPU. Figure 13 shows the corresponding bandwidth utilizations. In this figure, the reason Eve results do not register next to Remus's bandwidth utilization is that Eve uses about 3 orders of magnitude less bandwidth.

With 1 CPU, Eve's performance is competitive with Remus's, and both impose moderate overheads compared to an unreplicated Xen server. We again note that supplying requests from a local trace rather than network clients, clearly gives Remus an advantage in this experiment.

Although we could not perform a direct comparison with multiple CPUs, Remus's heavy use of the network to transfer modified pages from the primary to the replica appears to limit its scalability. As Figure 13 indicates, Eve uses drastically less network bandwidth, because it transfers hashes of its state, rather than the state itself. Note that hash computation is not only faster than network transfer, but also parallelizable across CPUs.

The next experiment compares Eve's throughput to that reported by DPG [6]. We run the Apache Web Server, using a workload of 10KB static pages. We measure the throughput of the original Apache server and that of a server-pair Eve configuration. Figure 14 shows the results of our experiment, as well as the results reported for DPG on comparable hardware [6]. We observe that the original Apache achieves the same throughput in both cases. DPG incurs a 4x slowdown in order to make a single Apache server deterministic, with a throughput of 2.4K requests per second; when the server is replicated, throughput is further reduced by another factor of 6 to 372 requests per second. In contrast, Eve achieves a throughput of 7.4K requests per second for the replicated Apache server.

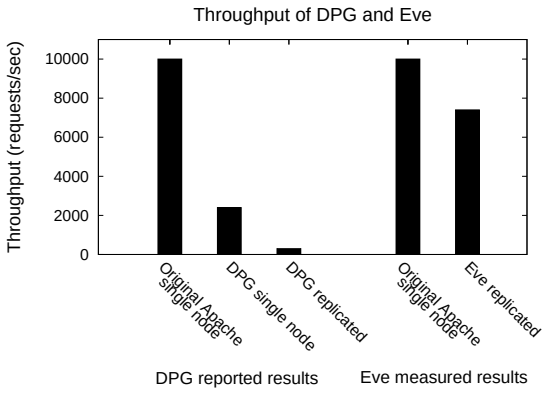

Figure 14: This graph shows the report throughput of DPG (single node and replicated) and Eve (replicated) compared to an original Apache Web Server with a workload of 10KB static pages.

7 Conclusion

Eve is a new Execute-Verify architecture that allows state machine replication to scale to multi-core servers. Eve accomplishes this by revisiting fundamental assumptions. Where the traditional view has been that state machines' independent executions must be deterministic, Eve demonstrates that nondeterministic execution can be used by employing an execute-verify architecture along with a mixer that reduces misspeculation. The practical consequence of refining our assumptions is that Eve uses its execute-verify architecture to bring together *nondeterminism* and *independence* to improve the *performance* and *fault tolerance* of replicated multi-core servers.

References

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[4] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE TPDS*, 2006.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 2010.

[6] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 1996.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *CDCCA*, 1992.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[11] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, 2009.

[12] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[15] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay for multiprocessor virtual machines. In *VEE*, 2008.

[16] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: compiling SHIM to Pthreads. In *DATE*, 2008.

[17] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD, 2006.

[18] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Eurosys*, 2011.

[19] J. Gray. Why do computers stop and what can be done about it. Technical Report 85.7, Tandem Computers, June 1985.

[20] H2. The H2 home page. http://www.h2database.com.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[23] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, 2004.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[25] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[26] 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 *ASPLOS*, 2010.

[27] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *OSDI*, 2008.

[28] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant Java virtual machine. In *DSN*, 2003.

[29] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[30] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[31] J. T. Pablo Montesinos, Luis Ceze. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[32] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessor. In *SOSP*, 2009.

[33] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4. *ACM OSR*, 1991.

[34] S. Rajagopalan. Personal communication.

[35] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[36] J. Robert L. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[37] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *SOSP*, Oct. 2001.

[38] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM TCS*, 1999.

[39] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 1990.

[40] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *NSDI*, 2009.

[41] Sun Microsystems, Inc. Memory management in the Java HotSpot virtual machine, 2006.

[42] A. Thomson and D. J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[43] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[44] R. van Renesse. Refining the way to consensus. In *PODC*, 2009.

[45] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 1996.

[46] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.

[47] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS*, page 15, 2011.

[48] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. In *Eurosys '11*, 2011.

[49] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[50] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS*, 2007.

[51] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, Oct. 2003.