All about Eve: Execute-Verify Replication for Multi-Core Servers

Manos Kapritsos*, Yang Wang*, Vivien Quema†, Allen Clement‡, Lorenzo Alvisi*, Mike Dahlin*
*The University of Texas at Austin †Grenoble INP ‡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 *execute* groups of requests concurrently and then *verify* that they can reach agreement on a state and output produced by a correct replica; if they can not, they roll back and execute the requests sequentially. Eve minimizes divergence using application-specific criteria to organize requests into groups of requests that are unlikely to interfere. Our evaluation suggests that Eve's unique ability to combine execution independence with nondetermistic interleaving of requests enables high-performance replication for multi-core servers while tolerating a wide range of faults, including elusive concurrency bugs.

1 Introduction

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

State machine replication (SMR) is a powerful fault tolerance technique [26, 38]. Historically, the essential idea is for replicas to deterministically process the same sequence of requests so that correct replicas 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 states and outputs of correct servers may diverge even if no faults occur. As a result, most SMR systems require servers to process requests sequentially: a replica finishes executing one request before beginning to execute the next [7, 27, 31, 39, 44, 50].

At first glance, recent efforts to enforce deterministic parallel execution seem to offer a promising approach to overcoming this impasse. Unfortunately, as we detail in the next section, these efforts fall short, not just because of the practical limitations of current implementations (e.g. high overhead [2, 3, 5]) but more fundamentally because, to achieve better performance, many modern replication algorithms do not actually execute operations in the same order at every replica (and sometimes do not even execute the same set of operations) [7, 11, 21, 47].

To avoid these issues, Eve's replication architecture eliminates the requirement that replicas execute requests in the same order. Instead, Eve partitions requests in batches and, after taking lightweight measures to make conflicts within a batch unlikely, it allows different replicas to execute requests within each batch in parallel, speculating that the result of these parallel executions (i.e. the system's important state and output at each replica) will match across enough replicas.

To execute requests in parallel without violating the safety requirements of replica coordination, Eve turns on its head the established architecture of state machine replication. Traditionally, deterministic replicas first *agree* on the order in which requests are to be executed and then *execute* them [7, 26, 27, 31, 38, 50]; in Eve, replicas first speculatively *execute* requests concurrently, and then *verify* that they have agreed on the state and the output produced by a correct replica. If too many replicas diverge so that a correct state/output cannot be identified, 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 the nondeterminism introduced by allowing parallel execution, replicas seldom diverge, and that, if they do, divergence is efficiently detected and reconciled. Eve minimizes divergence through a *mixer* stage that applies application-specific criteria to produce groups of requests that are unlikely to interfere, and it makes repair efficient through incremental state transfer and fine-grained rollbacks. Note that if the underlying program is correct under unreplicated parallel execution, then delaying agreement until after execution and, when necessary, falling back to sequential re-execution guarantees that replication remains safe and live even if the mixer allows interfering requests in the same group.

Eve's execute-verify architecture is general and ap-

plies to both crash tolerant and Byzantine tolerant systems. In particular, when Eve is configured to tolerate crash faults, it also provides significant protection against concurrency bugs, thus addressing 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 attempting to run only unlikely-to-interfere requests in parallel [25, 35]. Second, its *execute-verify* architecture allows Eve to detect and recover when concurrency causes executions to diverge, regardless of whether the divergence results from a concurrency bug or from distinct correct replicas making different legal choices.

In essence, Eve refines the assumptions that underlie the traditional implementation of state machine replication. In the agree-execute architecture, the safety requirement that correct replicas agree *on the same state and output* is reduced to the problem of guaranteeing that deterministic replicas process identical sequences of commands (i.e. agree *on the same inputs*). Eve continues to require replicas to be deterministic, but it no longer insists on them executing identical sequences of requests: instead of relying on agreement on inputs, Eve reverts to the weaker original safety requirement that replicas agree on state and output.

The practical consequence of this refinement is that in Eve correct replicas enjoy two properties that prior replica coordination protocols have treated as fundamentally at odds with each other: *nondeterministic interleaving of requests* and *execution independence*. Indeed, it is precisely through the combination of these two properties that Eve improves the state of the art for replicating multi-core servers:

1. *Nondeterministic interleaving of requests lets Eve provide high-performance replication for multi-core servers.* Eve gains performance by avoiding the overhead of enforcing determinism. For example, in our experiments with the TPC-W benchmark, Eve achieves a 6.5*x* speedup over sequential execution that approaches the 7.5*x* speedup of the original unreplicated server. For the same benchmark, Eve achieves a 4.7*x* speedup over the Remus primary-backup system [13] by exploiting its unique ability to allow independent replicas to interleave requests nondeterministically.

2. *Independence lets Eve mask a wide range of faults.* Without independently executing replicas, it is in general impossible to tolerate arbitrary faults. Independence makes Eve's architecture fully general, as our prototype supports tunable fault tolerance [9], retaining traditional state machine replication's ability to be configured to tolerate crash, omission, or Byzantine faults. Notably, we find that execution independence pays dividends even when Eve is configured to tolerate only crash or omission failures by offering the opportunity to 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 occassionally allow conflicting requests to be parallelized, Eve detected and corrected the effects of this bug 82% of the times it manifested, because Eve's independent execution allowed the bug to manifest (or not) in different ways on different replicas.

The rest of the paper proceeds as follows. In Section 2 we explain why deterministic multithreaded execution does not solve the problem of replicating multithreaded services. Section 3 describes the system model and Section 4 gives an overview of the protocol. In Section 5 we discuss the execution stage in more detail and in Section 6 we present the agreement protocols used by the verification stage for two interesting configurations and discuss Eve's ability to mask concurrency bugs. Section 7 presents an experimental evaluation of Eve, and Section 8 presents related work. Section 9 concludes the paper.

2 Why not deterministic execution?

Deterministic execution of multithreaded programs [2, 3, 5, 30] guarantees that, given the same input, all correct replicas of a multithreaded application will produce identical internal application states and outputs. Although at first glance this approach appears a perfect match for the challenge of multithreaded SMR on multi-core servers, there are two issues that lead us to look beyond it. The first issue [4] is straightforward: current techniques for deterministic multithreading either require hardware support [14, 15, 20] or are too slow (1.2x-10x overhead) [2, 3, 5] for production environments. The second issue originates from the semantic gap that exists between modern SMR protocols and the techniques used to achieve deterministic multithreading.

Seeking opportunities for higher throughput, SMR protocols have in recent years looked for ways to exploit the semantics of the requests processed by the replicas to achieve replica coordination without forcing all replicas to process identical sequences of inputs. For example, many modern SMR systems no longer insist that read requests be performed in the same order at all replicas, since read requests do not modify the state of the replicated application. This *read-only optimization* [7, 9, 24]

is often combined with a second optimization that allows read requests to be executed only at a *preferred quorum* of replicas, rather than at *all* replicas [21]. Several SMR systems [11, 47] use the preferred quorum optimization during failure-free executions also for requests that change the application's state, asking other replicas to execute these requests only if a preferred replica fails.

Unfortunately, deterministic multithreading techniques know nothing of the semantics of the operations they perform. Their ability to guarantee replica coordination of multithreaded servers is based purely on syntactic mechanisms that critically rely on the assumption that all replicas receive identical sequences of inputs: only then can deterministic multithreading ensure that the replicas' states and outputs will be the same. Read-only optimizations and preferred quorum operations violate that assumption, leading correct replicas to diverge. For instance, read-only requests advance a replica's instruction counter and may cause the replica to acquire additional read locks: it is easy to build executions where such low-level differences may eventually cause the application state of correct replicas to diverge [22]. Paradoxically, the troubles of deterministic replication stem from sticking to the letter of the state machine approach [26, 38], at the same time that modern SMR protocols have relaxed its requirements while staying true to its spirit.

3 System model

The novel architecture for state machine replication that we propose is fully general: Eve can be applied to coordinate the execution of multithreaded replicas in both synchronous and asynchronous systems and can be configured to tolerate failures of any severity, from crashes to Byzantine faults.

In this paper, 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 [6]—relies on timing assumptions for both safety and liveness.

Eve can be configured to produce systems that are *live*, i.e. provide a response to client requests, despite a total of up to u failures, whether of omission or commission, and to ensure that all responses accepted by correct clients are *correct* despite up to r commission failures and any number of omission failures [9]. Commission failures include all failures that are not omission failures. The union of omission and commission failures are Byzantine failures. However, we assume that failures do not break cryptographic primitives; i.e., a faulty node can never produce a correct node's MAC. We denote a message X sent by Y that includes an authenticator (a vector of MACs, one per receiving replica) as $\langle X \rangle_{\vec{\mu}_Y}$.

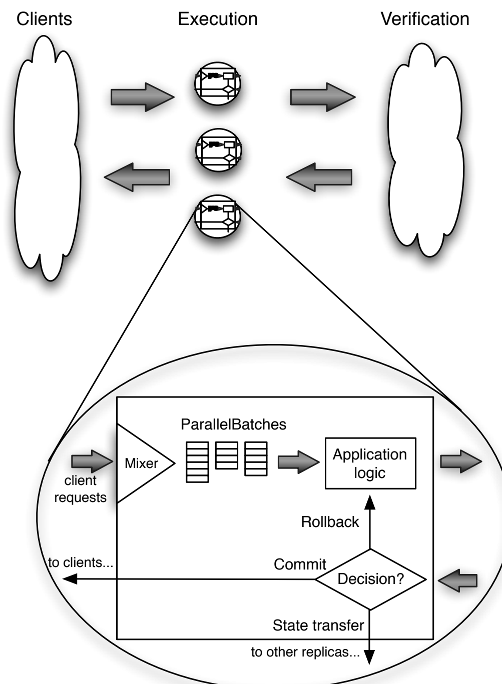

Figure 1: Overview of Eve.

4 Protocol overview

Figure 1 shows an overview of Eve, whose "execute-then-verify" design departs from the "agree-then-execute" approach of traditional SMR [7, 27, 50].

4.1 Execution stage

Eve divides requests in batches, and lets replicas execute requests within a batch in parallel, without requiring them to agree on the order of request execution within a batch. However, Eve takes steps to make it likely that replicas will produce identical final states and outputs for each batch.

Batching Clients send their requests to the current primary execution replica. The primary groups requests into batches, assigns each batch a sequence number, and sends them to all execution replicas. Multiple such batches can be in flight at the same time, but they are processed in order. Along with the requests, the primary sends any data needed to consistently process any nondeterministic requests in the batch (e.g. a seed for `random()` calls or a timestamp for `gettimeofday()` calls [7, 9]). The primary however makes no effort to

eliminate the nondeterminism that may arise when multithreaded replicas independently execute their batches.

Mixing Each replica runs the same deterministic *mixer* to partition each batch received from the primary into the same ordered sequence of *parallelBatches*—groups of requests that the mixer believes can be executed in parallel with little likelihood that different interleavings will produce diverging results at distinct replicas. For example, if *conflicting* requests ρ_1 and ρ_2 both modify object A, the mixer will place them in different parallelBatches. Section 5.1 describes the mixer in more detail.

Executing (in parallel) Each replica executes the parallelBatches in the order specified by the deterministic mixer. After executing all parallelBatches in the ith batch, a replica computes a hash of its application state and of the outputs generated in response to requests in that batch. This hash, along with the sequenceNumber i and the hash for batch $i-1$,[1] constitute a *token* that is sent to the verification stage in order to discern whether the replicas have diverged. Section 5.2 describes how we efficiently and deterministically compute the hash of the final state and outputs.

4.2 Verification stage

Eve's execution stage strives to make divergence unlikely, but offers no guarantees: for instance, despite its best effort, the mixer may inadvertently include conflicting requests in the same parallelBatch and cause distinct correct replicas to produce different final states and outputs. It is up to the verification stage to ensure that such divergences cannot affect safety, but only performance: at the end of the verification stage, all correct replicas that have executed the ith batch of requests are guaranteed to have reached the same final state and produced the same outputs.

Agreement The verification stage runs an agreement protocol to determine the final state and outputs of all correct replicas after each batch of requests. The input to the agreement protocol (see Section 6) are the tokens received from the execution replicas. The final decision is either *commit* (if enough tokens match) or *rollback* (if too many tokens differ). In particular, the protocol first verifies whether replicas have diverged at all: if all tokens agree, the replicas' common final state and outputs are committed. If there is divergence, the agreement protocol tallies the received tokens, trying to identify a final state and outputs pair reached by enough replicas to guarantee that the pair is the product of a correct replica. If one such pair is found, then Eve ensures that all correct replicas commit to that state and outputs; if not, then the agreement protocol decides to roll back.

Commit If the result of the verification stage is *commit*, the execution replicas mark the corresponding sequence number as committed and send the responses for that parallelBatch to the clients.

Rollback If the result of the verification stage is *rollback*, the execution replicas roll back their state to the latest committed sequence number and re-execute the batch sequentially to guarantee progress. A rollback may also cause a primary change, to deal with a faulty primary. To guarantee progress, the first batch created by the new primary, which typically includes some subset of the rolled back requests, is executed sequentially by all execution replicas.

A serendipitous consequence of its "execute-verify" architecture is that Eve can often mask replica divergences caused by *concurrency bugs*, i.e. deviations from an application's intended behavior triggered by particular thread interleavings [18]. Some concurrency bugs may manifest as commission failures; however, because such failures are typically triggered probabilistically and are not the result of the actions of a strategic adversary, they can be often masked by configurations of Eve designed to tolerate only omission failures. Of course, as every system that uses redundancy to tolerate failures, Eve is vulnerable to *correlated failures* and cannot mask concurrency failures if too many replicas fail in exactly the same way. This said, Eve's architecture should help, both because the mixer, by trying to avoid parallelizing requests that interfere, makes concurrency bugs less likely and because concurrency bugs may manifest differently (if at all) on different replicas.

5 Execution stage

In this section we describe the execution stage in more detail. In particular, we discuss the design of the mixer and the design and implementation of the state management framework that allows Eve to perform efficient state comparison, state transfer, and rollback.

5.1 Mixer design

Parallel execution will result in better performance only if divergence is rare. The mission of the mixer is to identify requests that may productively be executed in parallel and to do so with 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

[1] We include the hash for the previous batch to make sure that the system only accepts valid state transitions. Verification replicas will only accept a token as valid if they have already agreed that there is a committed hash for sequence number $i-1$ that matches the one in the ith token.

Transaction	Read and write keys
getBestSellers	**read**: item, author, order_line
getRelated	**read**: item
getMostRecentOrder	**read**: customer, cc_xacts, address, country, order_line
doCart	**read**: item **write**: shopping_cart_line, shopping_cart
doBuyConfirm	**read**: customer, address **write**: order_line, item, cc_xacts, shopping_cart_line

Figure 2: The keys used for the 5 most frequent transactions of the TPC-W workload.

of the false negative and false positive rates of the mixer. A good mixer is just a performance optimization (albeit an important one).

The mixer we use for our experiments parses each request, trying to predict which state it will access: depending on the application, this state can vary from a single file or application-level object to higher level objects such as database rows or tables. Two requests conflict when they access the same object in a read/write or write/write manner. To avoid putting together conflicting requests, the mixer starts with an empty parallelBatch and two (initially empty) hash tables, one for objects being read, the other for objects being written. The mixer then scans in turn each request, mapping the objects accessed in the request to a read or write key, as appropriate. Before adding a request to a parallelBatch, the mixer checks whether that request's keys have read/write or write/write conflicts with the keys already present in the two hash tables. If not, the mixer adds the request to the parallelBatch and adds its keys to the appropriate hash table; when a conflict occurs, the mixer tries to add the request to a different parallelBatch—or creates a new parallelBatch, if the request conflicts with all existing parallelBatches.

In our experiments with the H2 Database Engine and the TPC-W workload, we simply used the names of the tables accessed in read or write mode as read and write keys for each transaction[2] (see Table 2). Note that because the mixer can safely misclassify requests, we need not explicitly capture additional conflicts potentially generated through database triggers or view accesses that may be invisible to us: Eve's verification stage allows us to be safe without being perfect. Moreover, the mixer can be improved over time using feedback from the system (e.g. by logging parallelBatches that caused rollbacks).

Although implementing a perfect mixer might prove tricky for some cases, we expect that a good mixer can be written for many interesting applications and workloads with modest effort. Databases and key-value stores are examples of applications where requests typically identify the application-level objects that will be affected—tables and values respectively. Our experience so far is encouraging. Our TPC-W mixer took 10 student-hours to build, without any prior familiarity with the TPC-W code. As demonstrated in Section 7, this simple mixer achieves good parallelism (acceptably few false positives), and we do not observe any rollbacks (few or no false negatives).

[2]Since H2 does not support row-level locking, we did not implement conflict checks at a granularity finer than a table.

5.2 State management

Moving from an agree-execute to an execute-verify architecture puts pressure on the implementation of state checkpointing, comparison, rollback, and transfer. For example, replicas in Eve must compute a hash of the application state reached after executing every batch of requests; in contrast, traditional SMR protocols checkpoint and compare application states much less often (e.g. when garbage collecting the request log).

To achieve efficient state comparison and fine-grained checkpointing and rollback, Eve stores the state using a copy-on-write Merkle tree, whose root is a concise representation of the entire state. The implementation borrows two ideas from BASE [36]. First, it includes only the subset of state that determines the operation of the state machine, omitting other state (such as an IP address or a TCP connection) that can vary across different replicas but has no semantic effect on the state and output produced by the application. Second, it provides an abstraction wrapper on some objects to mask variations across different replicas.

Similar to BASE and other traditional SMR systems such as PBFT, Zyzzyva, and UpRight, where programmers are required to manually annotate which state is to be included in the state machine's checkpoint [7, 9, 24, 36], our current implementation of Eve manually annotates the application code to denote the objects that should be added to the Merkle tree and to mark them as dirty when they get modified.

Compared to BASE, however, Eve faces two novel challenges: maintaining a deterministic Merkle tree structure under parallel execution and parallel hash generation as well as issues related to our choice to implement Eve in Java.

5.2.1 Deterministic Merkle trees

To generate the same checksum, different replicas must put the same objects at the same location in their Merkle tree. In single-threaded execution, determinism comes 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 address the problem. The first option is to make memory allocation synchronized and deterministic. This approach not only negates efforts toward concurrent memory allocation [17, 40],

but is unnecessary, since the allocation order usually does not fundamentally affect replica equivalence. The second option is to generate an ID based on object content and to use it to determine an object's location in the tree; this approach does not work, however, since many objects have the same content, especially at creation time.

Our solution is to postpone adding newly created objects to the Merkle tree until the end of the batch, when they can be added deterministically. Eve scans existing modified objects, and if one contains a reference to an object not yet in the tree, Eve adds that object into the tree's next empty slot and iteratively repeats the process for all newly added objects.

Object 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 in which references are defined in a class. However some classes, like *Hashtable*, do not store their references in a deterministic order; we discuss how to address these classes in Section 5.2.2.

We do not parallelize the process of scanning for new objects, since it has low overhead. We do parallelize hash generation, however: we split the Merkle tree into subtrees and compute their hashes in parallel before combining them to obtain the hash of the Merkle tree's root.

5.2.2 Java Language & Runtime

The choice of implementing our prototype in Java provides us with several desirable features, including an easy way to differentiate references from other data that simplifies the implementation of deterministic scanning; at the same time, it also raises some challenges.

First, objects to which the Merkle tree holds a reference to are not eligible for Java's automatic garbage collection (GC). Our solution is to periodically perform a Merkle-tree-level scan, using a mark-and-sweep algorithm similar to Java's GC, to find unused objects and remove them from the tree. This ensures that those objects can be correctly garbage collected by Java's GC. For the applications we have considered, this scan can be performed less frequently than Java's GC, since objects in the tree tend to be "important" and have a long lifetime. In our experience this scan is not a major source of overhead.

Second, several standard set-like data structures in Java, including instances of the widely-used Hashtable and HashSet classes, are not oblivious to the order in which they are populated. For example, the serialized state of a Java Hashtable object is sensitive to the order in which keys are added and removed. So, while two set-like data structures at different replicas may contain the same elements, they may generate different checksums when added to a Merkle tree: while semantically equivalent, the states of these replicas would instead be seen as having diverged, triggering unnecessary rollbacks.

Our solution is to create wrappers [36] that abstract away semantically irrelevant differences between instances of set-like classes kept at different replicas. The wrappers generate, for each set-like data structure, a deterministic list of all the elements it contains, and, if necessary, a corresponding iterator. If the elements' type is one for which Java already provides a comparator (e.g. Integer, Long, String, etc.), this is easy to do. Otherwise, the elements are sorted using an ordered pair *(requestId, count)* that Eve assigns to each element before adding it to the data structure. Here, *requestId* is the unique identifier of the request responsible for adding the element, and *count* is the number of elements added so far to the data structure by request *requestId*. In practice, we only found the need to generate two wrappers, one for each of the two interfaces (Set and Map) commonly used by Java's set-like data structures.

6 Verification stage

The goal of the verification stage is to determine whether enough execution replicas agree on their state and responses after executing a batch of requests. Given that the tokens produced by the execution replicas reflect their current state as well as the state transition they underwent, all the verification stage has to decide is whether enough of these tokens match.

To come to that decision, the verification replicas use an agreement protocol [7, 27] whose details depend largely on the system model. As an optimization, read-only requests are first executed at multiple replicas without involving the verification stage. If enough replies match, the client accepts the returned value; otherwise, the read-only request is reissued and processed as a regular request. We present the protocol for two extreme cases: an asynchronous Byzantine fault tolerant system, and a synchronous primary-backup system. We then discuss how the verification stage can offer some defense against concurrency bugs and how it can be tuned to maximize the number of tolerated concurrency bugs.

6.1 Asynchronous BFT

In this section we describe the verification protocol for an asynchronous Byzantine fault tolerant system with $n_E = u + max(u,r) + 1$ execution replicas and $n_V = 2u + r + 1$ verification replicas [8, 9], which allows the system to remain live despite u failures (whether of omission or commission), and safe despite r commission failures and any number of omission failures. Readers familiar with PBFT [7] will find many similarities between these two protocols; this is not surprising, since both protocols attempt to perform agreement among $2u + r + 1$ replicas ($3f + 1$ in PBFT terminology). The main differences between these protocols stem from two factors. First, in

PBFT the replicas try to agree on the output of a single node—the primary. In Eve the object of agreement is the behavior of a collection of replicas—the execution replicas. Therefore, in Eve verification replicas use a quorum of tokens from the execution replicas as their "proposed" value. Second, in PBFT the replicas try to agree on the inputs to the state machine (the incoming requests and their order). Instead, in Eve replicas try to agree on the outputs of the state machine (the application state and the responses to the clients). As such, in the view change protocol (which space considerations compel us to discuss in full detail elsewhere [22]) the existence of a certificate for a given sequence number is enough to commit that sequence number to the next view—a prefix of committed sequence numbers is no longer required.

When an execution replica executes a batch of requests (i.e. a sequence of parallelBatches), it sends a $\langle \text{VERIFY}, \upsilon, n, T, e \rangle_{\vec{\mu}_e}$ message to all verification replicas, where υ is the current view number, n is the batch sequence number, T is the computed token for that batch, and e is the sending execution replica. Recall that T contains the hash of both batch n and of batch $n-1$: a verification replica accepts a VERIFY message for batch n only if it has previously committed a hash for batch $n-1$ that matches the one stored in T.

When a verification replica receives $max(u,r)+1$ VERIFY messages with matching tokens, it marks this sequence number as *preprepared* and sends a $\langle \text{PREPARE}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ message to all other verification replicas. Similarly when a verification replica receives $n_V - u$ matching PREPARE messages, it marks this sequence number as *prepared* and sends a $\langle \text{COMMIT}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ to all other verification replicas. Finally, when a verification replica receives $n_V - u$ matching COMMIT messages, it marks this sequence number as *committed* and sends a $\langle \text{VERIFY-RESPONSE}, \upsilon, n, T, v \rangle_{\vec{\mu}_v}$ message to all execution replicas. Note that the view number υ is the same as that of the VERIFY message; this indicates that agreement was reached and no view change was necessary.

If agreement can not be reached, either because of diverging replicas, asynchrony, or because of a Byzantine execution primary, the verification replicas initiate a view change.[3] During the view change, the verification replicas identify the highest sequence number (and corresponding token) that has been prepared by at least $n_V - u$ replicas and start the new view with that token. They send a $\langle \text{VERIFY-RESPONSE}, \upsilon+1, n, T, v, f \rangle_{\vec{\mu}_v}$ message to all execution replicas, where f is a flag that indicates that the next batch should be executed sequentially to ensure progress. Note that in this case the view number has increased; this indicates that agreement was not reached and a rollback to sequence number n is required.

[3]The view change is triggered when the commit throughput is lower than expected, similar to [10].

Commit, State transfer and Rollback Upon receipt of $r+1$ matching VERIFY-RESPONSE messages, an execution replica e distinguishes three cases:

Commit If the view number has not increased and the agreed-upon token matches the one e previously sent, then e marks that sequence number as stable, garbage-collects any portions of the state that have now become obsolete, and releases the responses computed from the requests in this batch to the corresponding clients.

State transfer If the view number has not increased, but the token does not match the one e previously sent, it means that this replica has diverged from the agreed-upon state. To repair this divergence, it issues a state transfer request to other replicas. This transfer is incremental: rather than transferring the entire state, Eve transfers only the part that has changed since the last stable sequence number. Incremental transfer, which uses the Merkle tree to identify what state needs to be transferred, allows Eve to rapidly bring slow and diverging replicas up-to-date.

Rollback If the view number has increased, this means that agreement could not be reached. Replica e discards any unexecuted requests and rolls back its state to the sequence number indicated by the token T, while verifying that its new state matches the token (else it initiates a state transfer). The increased view number also implicitly rotates the execution primary. The replicas start receiving batches from the new primary and, since the flag f was set, execute the first batch sequentially to ensure progress.

6.2 Synchronous primary-backup

A system configured for synchronous primary-backup has only two replicas that are responsible for both execution and verification. The primary receives client requests and groups them into batches. When a batch $\mathcal{B}$ is formed, it sends a $\langle \text{EXECUTE-BATCH}, n, \mathcal{B}, ND \rangle$ message to the backup, where n is the batch sequence number and ND is the data needed for consistent execution of nondeterministic calls such as `random()` and `gettimeofday()`. Both replicas apply the mixer to the batch, execute the resulting parallelBatches, and compute the state token, as described in Section 4. The backup sends its token to the primary, which compares it to its own token. If the tokens match, the primary marks this sequence number as stable and releases the responses to the clients. If the tokens differ, the primary rolls back its state to the latest stable sequence number and noti-

fies the backup to do the same. To ensure progress, they execute the next batch sequentially.

If the primary crashes, the backup is eventually notified and assumes the role of the primary. As long as the old primary is unavailable, the new primary will keep executing requests on its own. After a period of unavailability, a replica uses incremental state transfer to bring its state up-to-date before processing any new requests.

6.3 Tolerating concurrency bugs

A happy consequence of the execute-verify architecture is that even when configured with the minimum number of replicas required to tolerate u omission faults, Eve provides some protection against concurrency bugs.

Concurrency bugs can lead to both omission faults (e.g., a replica could get stuck) and commission faults (e.g., a replica could produce an incorrect output or transition to an incorrect state). However, faults due to concurrency bugs have an important property that in general cannot be assumed for Byzantine faults: they are easy to repair. If Eve detects a concurrency fault, it can repair the fault via rollback and sequential re-execution.

Asynchronous case When configured with $r = 0$, Eve provides the following guarantee:

Theorem 1. *When configured with $n_{exec} = 2u+1$ and $r = 0$, asynchronous Eve is safe, live, and correct despite up to u concurrency or omission faults.*

Note that safety and liveness refer to the requirements of state machine replication—that the committed state and outputs at correct replicas match and that requests eventually commit. Correctness refers to the state machine itself; a committed state is correct if it is a state that can be reached by the state machine in a fault-free run.

Proof sketch: The system is always safe and correct because the verifier requires $u+1$ matching execution tokens to commit a batch. If there are at most u concurrency faults and no other commission faults, then every committed batch has at least one execution token produced by a correct replica.

The system is live because if a batch fails to gather $u+1$ matching tokens, the verifier forces the execution replicas to roll back and sequentially re-execute. During sequential execution deterministic correct replicas do not diverge; so, re-execution suffers at most u omission faults and produces at least $u+1$ matching execution tokens, allowing the batch to commit. □

When more than u correlated concurrency faults produce exactly the same state and output, Eve still provides the safety and liveness properties of state machine replication, but can no longer guarantee correctness.

Synchronous case When configured with just $u+1$ execution replicas, Eve can continue to operate with 1 replica if u replicas fail by omission. In such configurations, Eve does not have spare redundancy and can not mask concurrency faults at the one remaining replica.

Extra protection during good intervals During *good intervals* when there are no replica faults or timeouts other than those caused by concurrency bugs, Eve uses spare redundancy to boost its best-effort protection against concurrency bugs to $n_E - 1$ execution replicas in both the synchronous and asynchronous cases.

For example, in the synchronous primary-backup case, when both execution replicas are alive, the primary receives both execution responses, and if they do not match, it orders a rollback and sequential re-execution. Thus, during a good interval this configuration masks one-replica concurrency failures. We expect this to be the common case.

In both the synchronous and asynchronous case Eve, when configured for $r = 0$, enters *extra protection mode* (EPM) after k consecutive batches for which all n_E execution replicas provided matching, timely responses. While Eve is in 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_E responses. If the verifiers receive all n_E matching responses, they commit the response. Otherwise, they order a rollback and sequential re-execution. Then, if they receive n_E matching responses within a short timeout, they commit the response and remain in EPM. Conversely, if sequential re-execution does not produce n_E matching and timely responses, they suspect a non-concurrency failure and exit EPM to ensure liveness by allowing the system to make progress with fewer matching responses.

7 Evaluation

Our evaluation tries to answer the following questions:

- What is the throughput gain that Eve provides compared to a traditional sequential execution approach?
- How does Eve's performance compare to an unreplicated multithreaded execution and alternative replication approaches?
- How is Eve's performance affected by the mixer and by other workload characteristics?
- How well does Eve mask concurrency bugs?

We address these questions by using a key-value store application and the H2 Database Engine. We implemented a simple 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, in terms of both false positives and false negatives. For the H2

Database Engine we use an open-source implementation of the TPC-W benchmark [42, 43]. For brevity, we will present the results of the browsing workload, which has more opportunities for concurrency.

Our current prototype omits some of the features described above. Specifically, although we implement the extra protection mode optimization from Section 6.3 for synchronous primary-backup 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. Finally, our current prototype only supports in-memory application state.

We run our microbenchmarks on an Emulab testbed with 14x 4-core Intel Xeon @2.4 GHz, 4x 8-core Intel Xeon @2.66 GHz, and 2x 8-core hyper-threaded Intel Xeon @1.6 GHz, connected with a 1 Gb Ethernet. We were able to get limited access to 3x 16-core AMD Opteron @3.0 GHz and 2x 8-core Intel Xeon L5420 @2.5 Ghz. We use the AMD machines as execution replicas to run the TPC-W benchmark on the H2 Database Engine for both the synchronous primary-backup and the asynchronous BFT configuration (Figure 3). For the asynchronous BFT configuration we use 3 execution and 4 verifier nodes, which are sufficient to tolerate 1 Byzantine fault ($u = 1$, $r = 1$). The L5420 machines are running Xen and we use them to perform our comparison with Remus (Figure 10 and Figure 11).

7.1 H2 Database with TPC-W

Figures 3 demonstrates the performance of Eve for the H2 Database Engine [19] with the TPC-W browsing workload [42, 43]. We report the throughput of Eve using an asynchronous BFT configuration (*Eve-BFT*) and a synchronous active primary-backup configuration (*Eve-PrimaryBackup*). We compare against the throughput achieved by an unreplicated server that uses sequential execution regardless of the number of available hardware threads (*sequential*). Note that this represents an upper bound of the performance achievable by previous replication systems that use sequential execution [7, 9, 27, 31]. We also compare against the performance of an unreplicated server that uses parallel execution.

With 16 execution threads, Eve achieves a speedup of 6.5x compared to sequential execution. That approaches the 7.5x speedup achieved by an unreplicated H2 Database server using 16 threads.

In both configurations and across all runs and for all data points, Eve never needs to roll back. This suggests 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.

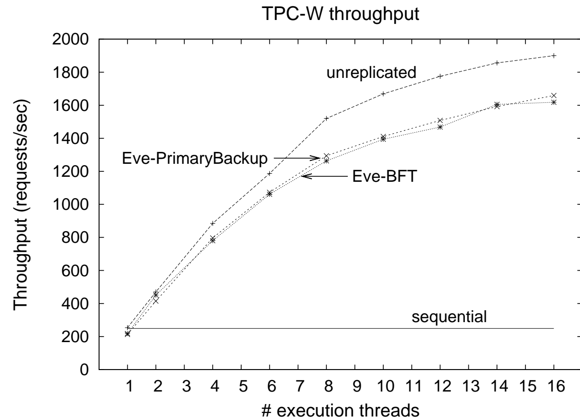

Figure 3: The throughput of Eve running the TPC-W browsing workload on the H2 Database Engine.

7.2 Microbenchmarks

In this section, we use a simple key-value store application to measure how various parameters affect Eve's performance. Due to lack of space, we only show the graphs for the primary-backup configuration; the results for asynchronous replication are similar. Except when noted, the default workload consumes 1 ms of execution time per request, each request updates one application object, and the application object size is 1 KB.

Figure 4 shows the impact of varying the CPU demand of each request. We observe that heavier workloads (10 ms of execution time per request) scale well, up to 12.5x on 16 threads compared to sequential execution. As the workload gets lighter, the overhead of Eve becomes more pronounced. Speedups fall to 10x for 1 ms/request and to 3.3x for 0.1 ms/request. The 3.3x scaling is partially an artifact of our inability to fully load the server with lightweight requests. In our workload generator, clients have 1 outstanding request at a time, thus requiring a high number of clients to saturate the servers; this causes our servers to run out of sockets before they are fully loaded. We measure our server CPU utilization during this experiment to be about 30%.

In Figure 4 we plot throughput speedup, so that trends are apparent. For reference, the absolute peak throughputs in requests per second are 25.2K, 10.0K, 1242 for the 0.1 ms, 1 ms, 10 ms lines, respectively.

The next experiment explores the impact of the application object size on the system throughput. We run the experiment using object sizes of 10 B, 1 KB, and 10 KB. Figure 5 shows the results. While the achieved throughput scales well for object sizes of 10 B and 1 KB, its scalability decreases for larger objects (10 KB). This is an artifact of the hashing library we use, as it first copies the object before computing its hash: for large objects, this memory copy limits the achievable throughput. Note that in this figure we plot throughput speedup rather than

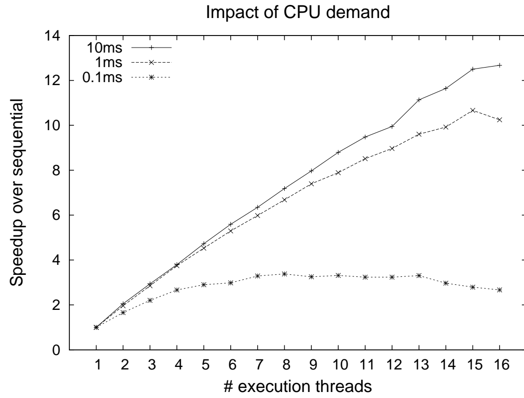

Figure 4: The impact of CPU demand per request on Eve's throughput speedup.

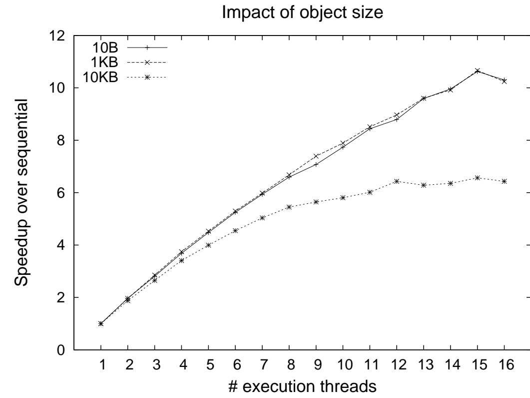

Figure 5: The impact of application object size on Eve's throughput speedup.

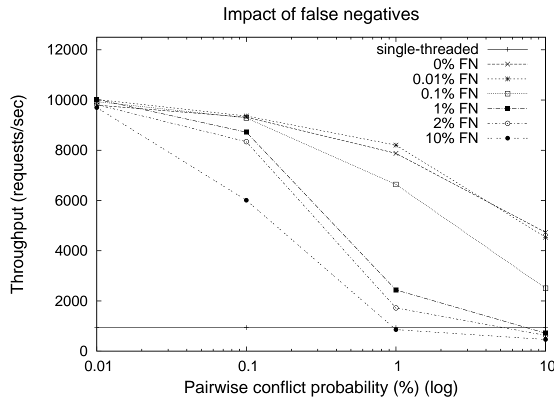

Figure 6: The impact of conflict probability and false negative rate on Eve's throughput.

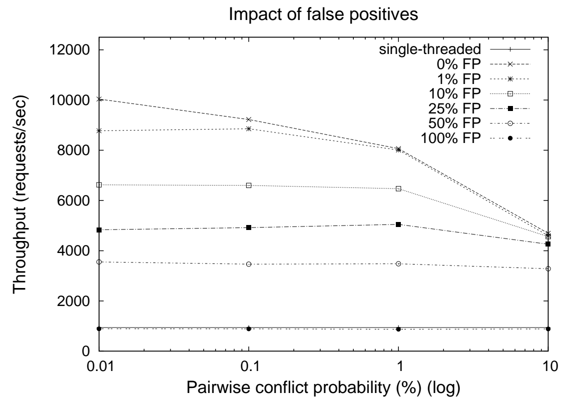

Figure 7: The impact of conflict probability and false positive rate on Eve's throughput.

absolute throughput to better indicate the trends across workloads. For reference, the absolute peak throughput values in requests per second are 10.0K, 10.0K, 5.6K for the 10 B, 1 KB, 10 KB lines, respectively.

Next, we evaluate Eve's sensitivity to inaccurate mixers. Specifically, we 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 probability: the probability that two requests have a conflict. In practice, we achieve this by having each request modify one 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 6 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 be 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 contain 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 7 shows the effect of false positives on throughput. As expected, increased false positive ratios can lead

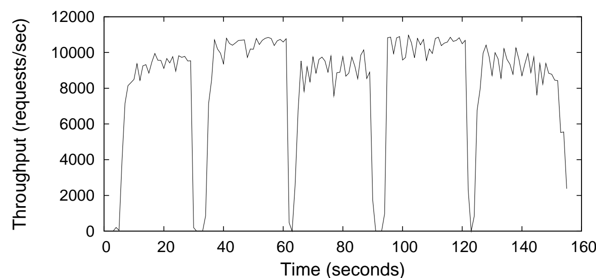

Figure 8: Throughput during node crash and recovery for an Eve primary-backup configuration.

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 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 [25]. 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.

7.3 Failure and recovery

In Figure 8, we demonstrate Eve's ability to mask and recover from failures. In the primary-backup configuration 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 observe that after the first failure the throughput drops to zero until the backup realizes that the primary is dead after a timeout of 4 seconds.[4] The backup then assumes the role of the primary and starts processing requests. The throughput during this period is higher because the new primary knows that the other node is crashed and does not send any messages to it. At $t = 60$, the first node recovers, and the throughput drops to zero for about one second while the newly recovered node catches up. 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.

7.4 Concurrency faults

To evaluate Eve's ability to mask concurrency faults, we use a primary-backup configuration 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 explore how well Eve's second line of defense—parallel execution—works in masking this bug.

Figure 9 shows 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.

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 original 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.

7.5 Remus

Remus [13] is a primary-backup system that uses Virtual Machines (VMs) to send modified state 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. Second, Remus operates on the VM level, which forces it to send entire pages, rather than just the modified objects. Also, because Remus is using passive replication, it tolerates a narrower range of faults than Eve. Our experiments show that, despite Eve's stronger guarantees, it outperforms Remus by a factor of $4.7x$, while using two orders of magnitude less network bandwidth.

Figure 10 shows the throughput achieved by Remus and Eve on the browsing workload of the TPC-W bench-

[4] One could use a fast failure detector [29] to achieve sub-second detection.

	Group all	1% FN	0.5% FN	0.1% FN	Original Mixer
Times bug manifested	73	51	29	4	0
Fixed with rollback	60	38	18	3	0
All identical (not masked)	13	13	11	1	0
Throughput	1104	1233	1240	1299	1322

Figure 9: Effectiveness of Eve in masking concurrency bugs when various mixers are used.

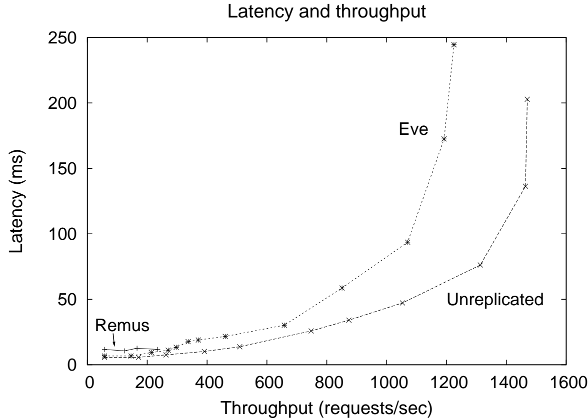

Figure 10: The latency and throughput of Remus and Eve running the H2 Database Engine on Xen. Both systems use a 2-node configuration. The workload is the browsing workload of the TPC-W benchmark.

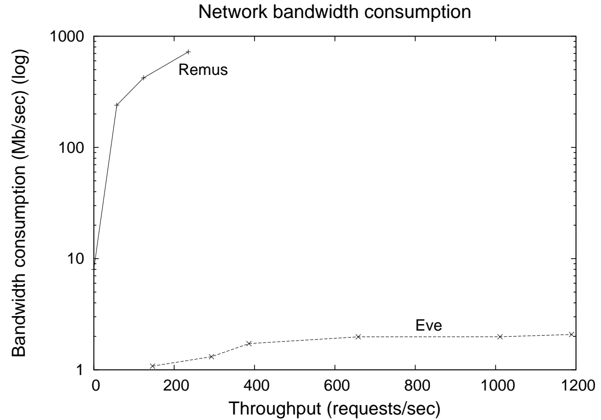

Figure 11: The bandwidth consumption of Remus and Eve for the experiment shown in Figure 10.

mark. We also show the latency and throughput of the unreplicated system for the same workload. Both systems run the H2 Database Engine on Xen and using a 2-node (primary-backup) configuration. Remus achieves a maximum throughput of 235 requests per second, while Eve peaks at 1225 requests per second. Remus crashes for loads higher than 235 requests per second, as its bandwidth requirements approach the capacity of the network, as Figure 11 shows. In contrast with Remus, Eve executes requests independently at each replica and does not need to propagate state modifications over the network. The practical consequence is that Eve uses significantly less bandwidth, achieves higher throughput, and provides stronger guarantees compared to a passive replication approach like Remus.

7.6 Latency and batching

Figure 10 provides some insight in Eve's tradeoff between latency and throughput. When Eve is not saturated, its latency is only marginally higher than that of an unreplicated server. As the load increases, Eve's latency increases somewhat, until it finally spikes up at the saturation point, at a throughput of 1225 requests per second; the unreplicated server's latency spikes up at around 1470 requests per second. To keep its latency low while maintaining a high peak throughput, Eve uses a dynamic batching scheme: the batch size decreases when the demand is low (providing good latency), and increases when the system starts becoming saturated, in order to leverage as much parallelism as possible.

8 Related Work

Vandiver et al. [45] describe a Byzantine-tolerant semi-active replication scheme for transaction processing 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). A recent paper suggests that it may be viable to enforce deterministic concurrency control in transactional systems [41], but the general case remains hard. Kim et al. [23] recently proposed applying this idea to a transactional operating system. This approach assumes that all application state is manageable by the kernel and does not handle in-memory application state.

One alternative is to use a replication technique other than state machine replication. *Semi-active replication* [34] 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, 12, 16, 28, 32, 33, 37, 46, 48, 49]. Unfortunately, relaxing the requirement of independent execution makes these systems vulnerable to commission failures. Also, similar to determin-

istic multithreaded execution approaches, record and replay approaches assume that the same input is given to all replicas. As discussed in Section 2 this assumption is violated in modern replication systems.

The Remus primary-backup system [13] takes a different approach: 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 parallel execution are sidestepped. These advantages however come at a significant price in terms of fault coverage: 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 outperforms Remus by a factor of $4.7x$ and uses two orders of magnitude less network bandwidth (see Section 7.5) 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 combine independent execution with nondeterministic interleaving of requests is the use of the mixer, which allows replicas to execute requests concurrently with low chance of interference. Kotla et al. [25] 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 criteria used by the mixer never mistakenly parallelize conflicting requests: a single unanticipated conflict can lead to a safety violation.

Both Eve and Zyzzyva [24] allow speculative execution that precedes completion of agreement, but the assumptions on which Eve and Zyzzyva rest are fundamentally different. Zyzzyva depends on correct nodes being deterministic, so that agreement on inputs is enough to guarantee agreement on outputs: hence, a replica need only send (a hash of) the sequence of requests it has executed to convey its state to a client. In contrast, in Eve there is no guarantee that correct replicas, even if they have executed the same batch of requests, will be in the same state, as the mixer may have incorrectly placed conflicting requests in the same parallelBatch.

We did contemplate an Eve implementation in which verification is not performed within the logical boundaries of the replicated service but, as in Zyzzyva, it is moved to the clients to reduce overhead. For example, a server's reply to a client's request could contain not just the response, but also the root of the Merkle tree that encodes the server's state. However, since agreement is not a bottleneck for the applications we consider, we ultimately chose to heed the lessons of Aardvark [10] and steer away from the corner cases that such an implementation would have introduced.

9 Conclusion

Eve is a new execute-verify architecture that allows state machine replication to scale to multi-core servers. By revisiting the role of determinism in replica coordination, Eve enables new SMR protocols that for the first time allow replicas to interleave requests nondeterministically and execute independently. This unprecedented combination is critical to both Eve's scalability and to its generality, as Eve can be configured to tolerate both omission and commission failures in both synchronous and asynchronous settings. As an added bonus, Eve's unconventional architecture can be easily tuned to provide low-cost, best-effort protection against concurrency bugs.

Acknowledgements

We thank our shepherd Robert Morris, and the OSDI reviewers for their insightful comments. This work was supported by NSF grants NSF-CiC-FRCC-1048269 and CNS-0720649.

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] 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.

[4] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

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

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

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

[8] A. Clement. *UpRight Fault Tolerance*. PhD thesis, The University of Texas at Austin, Dec. 2010.

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

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

[11] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[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] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In *ASPLOS*, 2011.

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

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

[18] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

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

[20] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In *HPCA*, 2011.

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

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers (extended version). Technical Report TR-12-23, Department of Computer Science, The University of Texas at Austin, September 2012.

[23] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. Improving server applications with system transactions. In *EuroSys*, 2012.

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

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

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

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

[28] 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.

[29] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *SOSP*, 2011.

[30] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

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

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

[33] 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.

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

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

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

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

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

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

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

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

[42] TPC-W. Open-source TPC-W implementation. http://pharm.ece.wisc.edu/tpcw.shtml.

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

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

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

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

[47] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. In *Eurosys*, 2011.

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

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

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