Communicating Replicated State Machines

David Wetterau

April 2015

Abstract

State Machine Replication (SMR) is a method to provide a highly available distributed service that traditionally relies on deterministic sequential execution to guarantee that replicas all agree on the same final state. In addition to relying on sequential execution, many SMR protocols adhere to the client-server model that no longer fits many applications today. This thesis presents a novel approach to provide the replicated state machine abstraction in an environment with communicating services that leverages both the multicore nature of today's machines and pipelining techniques to improve performance and resource utilization.

Contents

Contents **1**

1 Introduction **3**

2 Background **5**
- 2.1 State Machine Replication 5
- 2.2 Eve: Execute-Verify Replication 6
  - 2.2.1 The Mixer 8
  - 2.2.2 The Execution Stage 8
  - 2.2.3 The Verification Stage 9
  - 2.2.4 Performance Results 10

3 Adam: State Machine Replication with Communicating Services **12**
- 3.1 System Model 13
- 3.2 Initial Approaches 14
  - 3.2.1 Naive Sequential Implementation 14
  - 3.2.2 Eve Modification 15
  - 3.2.3 Single-Threaded Pipelining 16
- 3.3 Initial Results 17

4 Multi-Threaded Pipelining in Adam **18**
- 4.1 Design Overview 18
- 4.2 Theoretical Benefits 19

5 Adam Implementation **21**
- 5.1 Initial Code Structure 21
  - 5.1.1 Existing Multi-Threaded Adam Implementation 21
  - 5.1.2 Sequential Pipeline Manager 22
- 5.2 Core Components 23
  - 5.2.1 The Parallel Pipeline Manager 23
  - 5.2.2 The Parallel Group Manager and Execution Thread Modifications 24
  - 5.2.3 Verification Stage 25

5.3 Rollback and Correctness Guarantees 26

6 Evaluation and Results **28**

7 Future Work **32**

8 Related Work **34**
8.1 Replicated Remote Procedure Calls 34
8.2 Passive Replication 34

9 Conclusion **36**

Bibliography **38**

Chapter 1

Introduction

Computing systems today are rarely self-contained units. The desire for more levels of abstraction and cleaner separation of responsibilities has led to systems being modularized into services that must communicate and work together to accomplish tasks. Not only is this separation observed within standalone systems, but also in systems that interact with external services such as cloud based storage or computing providers [1, 5, 8]. In this thesis we will present a way to build such systems in a highly available and fault tolerant manner by building upon the existing techniques of State Machine Replication (SMR) [2, 6, 15–18] and pipelining.

Traditional SMR protocols suffer from fundamental performance deficiencies in today's computing environment. Many SMR protocols rely on the sequential execution of commands to guarantee that replicas operate in a deterministic manner [2, 3, 6, 16]. Not only does sequential execution do a poor job of leveraging abundantly available multicore machines, it also requires that a system that communicates with other services remain idle while waiting for the response of a request issued to another service.

In this thesis, we present a technique to efficiently perform SMR in a system where replicated services must communicate while processing client requests. We first leverage speculative execution [14, 15] to allow our system to be multi-threaded and therefore better utilize the multicore nature of machines today. We then show how to pipeline a speculative execution system to avoid unnecessary blocking when waiting for the responses of requests sent to other services. This work builds on contributions originally made by the Laboratory for Advanced Systems Research at the University of Texas at Austin [13, 14] and was a joint effort between David

Wetterau and Jim Given. We highly recommend reading both this thesis and Given's thesis [11] on this work to get a complete understanding of this project.

The rest of this thesis is structured as follows. Chapter 2 provides background on the replicated state machine abstraction and the Eve system, which pioneered the use of multi-threaded speculative execution in an SMR system. Chapter 3 introduces Adam, a system to provide high availability in an environment with services that communicate not only with clients, but also with other services. Chapter 4 describes at a high level the design and benefits of multi-threaded pipelining in Adam. Chapter 5 explains our implementation of Adam in detail along with the correctness guarantees Adam provides. Chapter 6 presents our evaluation and the performance of Adam. Chapter 7 discusses some of the future directions we believe would be fruitful for this system. Chapter 8 discusses work related to Adam and the conclusion of this thesis is presented in Chapter 9.

Chapter 2

Background

2.1 State Machine Replication

One way to allow a distributed system to be highly available in the presence of failures is to use State Machine Replication (SMR). SMR is a way to guarantee that multiple replicas, either virtual or physical, produce the same outputs when given the same commands. In this section we explain the abstraction of a replicated state machine along with how SMR has been traditionally implemented.

At the core of SMR is a deterministic state machine. A state machine is a model of computation that consists of state variables, which store the state of the machine, and commands that transform this state. In a SMR implementation, requests are issued by clients and contain the name of the state machine to operate on, the command to perform, and any other information needed to perform the command. When a replica in a SMR system receives a request, it performs the command by transitioning between states and finally producing output. For a write request from a client, the state machine might end in a state different from where it began (this can be thought of as the value of some variable being updated). In the case of a read request, the state machine returns to the same state it started in (no state was changed as a result of the client request) [18].

The safety requirement of SMR is that the observable state and outputs of are the same across all correct replicas and the liveness requirement is that all client requests eventually commit. The correctness goal of SMR is to provide a system that retains both safety and liveness in the presence of replica failures. To provide correctness, the requirement of SMR is that the observable state (including but not

limited to the output) of each state machine must *always* be the same across all correct replicas. As long as all correct replicas have the same observable state and produce the same output, voting can be used to mask the outputs of faulty replicas. In order for an SMR system to tolerate more faults, one simply increases the number of replicas in the system according to the failure model being considered and the SMR protocol being used.

To provide this correctness requirement, SMR has traditionally been implemented by only allowing sequential execution of commands [16–18]. If commands are first ordered the same way on each replica, and are then executed one at a time in a deterministic way, then all correct replicas will arrive at the same state. This is easy to see because each correct state machine simply follows the exact same state transitions in the same order. This design is referred to as the agree-execute approach, and has been applied in a wide range of systems including the original Paxos protocol [16].

Sequential execution was an acceptable option when SMR was first introduced since single core performance was rapidly increasing and multicore machines were not widespread. Now that the world has instead shifted to increasing the amount of parallelism to improve performance, we find that sequential execution is a serious limitation. If we tried to naively relax this sequential execution restriction and run commands using multiple threads, there would be no guarantee that the correct replicas would arrive in the same final state since the commands could be interleaved non-deterministically across the threads. In the next section, we show one way that we can still provide the replicated state machine abstraction without sequential execution by using speculative execution instead.

2.2 Eve: Execute-Verify Replication

The agree-execute approach with sequential execution guarantees convergence because sequential execution is deterministic. One observation that can be made about this approach is that it has effectively reduced the actual safety requirement of SMR (that the observable state of each state machine is the same across all correct replicas) to the problem of guaranteeing that all correct replicas agree on an order of commands to execute. Eve is a system that was developed at the University of Texas at Austin that uses the actual safety requirement of SMR to allow for multi-threaded execution [13, 14].

Eve uses what is referred to as speculative execution [14, 15] in a way that provides the safety guarantee of SMR while not requiring that requests are executed sequentially. The Eve protocol begins by clients sending requests to the current primary execution replica. This primary replica then gives a set, or batch, of client requests to each replica, however the replicas are not required to execute these requests sequentially in the order that they appear in the set. The primary replica also gives each replica a seed to a random number generator and other environment variables (such as the current time) that could potentially be other sources of non-determinism during execution. Replicas then fully leverage the potential of their underlying multicore machines to execute all of the requests in parallel. At the end of the execution, replicas compare states with each other in a verification stage and if enough correct replicas arrived at the same state, the results of the client requests are released. This step can be thought of as resolving the speculative execution. If not enough replicas agree on the final state, it must be due to a non-deterministic interleaving of the requests, so each replica simply re-executes all of the requests in the group in a deterministic order sequentially to guarantee both safety and liveness. This process of speculatively executing requests in parallel and then verifying that all correct replicas arrived at the same ending state is referred to as the execute-verify approach. Figure 2.1 depicts an overview of the Eve protocol which uses this approach.

In theory the execute-verify approach is straightforward, but in order for Eve to work well, it is important to make the case that requests have to be re-executed in a sequential manner rare. If this were not the case, Eve would be strictly worse than SMR implemented sequentially because of the need to execute requests multiple times to guarantee safety. To reduce the likelihood that request interleavings cause the final states of replicas to diverge, the authors introduce what they call the mixer which functions to separate requests that modify the same state variables in the system (discussed in Section 2.2.1). After Eve runs the mixer on the incoming requests, it proceeds to the execution stage described in Section 2.2.2. After executing a batch of requests, Eve moves on to the verification stage which guarantees that replicas compare their states efficiently and in a way that tolerates even Byzantine failures. This verification stage is described in Section 2.2.3 and the final results that the authors are able to produce with Eve are presented in Section 2.2.4.

2.2.1 The Mixer

Given a set of requests submitted to each replica in the Eve system by the current primary, each replica's mixer constructs a batch of requests to schedule for execution in the following way. The mixer examines each request to determine if it conflicts with any request already added to the next batch to execute. If the request does not conflict, it adds the request to the next batch to execute. If it does conflict, the mixer passes over the request. After a complete pass, the mixer releases the batch to be executed and after doing so, repeats this process on the requests that were left out of the batch on the previous pass.

The authors found that mixers were both easy to implement for various real applications such as the TPC-W benchmarks, and greatly reduced the likelihood that replicas would diverge because of thread interleavings [14]. It is important to note that while the mixer does improve performance of the system greatly, it is not needed for either the safety or liveness conditions of SMR. Because Eve can re-execute requests sequentially as a fallback, there is no requirement that the requests in a batch do not conflict and therefore there is no required accuracy of the mixer in order for Eve to be correct.

2.2.2 The Execution Stage

The execution stage in Eve is where the system gains its massive performance improvements over a sequential agree-execute SMR system. The output of running the mixer on the set of client requests is a batch of requests that are unlikely to cause replicas to diverge. This could possibly happen (and is much more likely without a mixer) because of non-deterministic interleavings of the operations performed by the parallel threads. In the execution stage, these requests are distributed out across a number of execution threads to be run in parallel concurrently. When all of the execution threads in a replica finish performing the commands specified by the client requests in the batch, a hash of the final state of the state machine is computed efficiently using a deterministic Merkle tree implementation. This hash of the final state is then used in the verification stage to determine whether or not the parallel execution caused a divergence in the final states of the replicas.

2.2.3 The Verification Stage

In Eve's verification stage, replicas must compare their final states in an efficient manner that is also resistant to even Byzantine failures. This is accomplished by relying on cryptographic primitives such as Message Authentication Codes (MACs) and collision resistant hashes of the states [14]. As mentioned in the previous subsection, Eve replicas compute a hash of their state after processing a batch of requests by each using a deterministic Merkle tree for efficiency [14]. After computing the root hash of the tree, replicas sign and send to a separate set of verification machines a signed token that consists of the root hash they started with before processing the current batch, and the hash of their state after processing the batch in a parallel manner.

The verification machines then run an agreement protocol on the tokens (which are the hashes of each replica's state) submitted to them by all the replicas. This protocol is explained at length in Kapritsos' doctoral thesis and the original Eve protocol paper [13,14]. During this agreement protocol, the verifiers can decide that a view change is required which allows this system to remain live in the presence of a faulty primary execution replica. If all of the tokens match at the end of agreement, the verifiers instruct the primary to release the result of the computation to the clients. This is achieved by responding to the replicas with a "commit" message. If instead some tokens differ but there are enough submissions of the same token to guarantee that a correct replica submitted it, the verification machines respond with a commit for that token but require the replicas that did not reach that state to receive a state transfer from the replicas that did. This state transfer is performed in an incremental manner, allowing Eve to save considerable bandwidth for this operation.

In the worst case, the verifiers do not identify a token that was reached by enough replicas. In this case, the verifiers require all replicas to re-execute the requests in the batch in a sequential manner. When this case arises, a view change is also performed and the primary replica is rotated to ensure progress. After the replicas have re-executed the requests in the same order, there is no need to enter the verification stage again because the execution was performed deterministically. This deterministic execution guarantees that all replicas arrive at the same ending state.

Regardless of if the verification machines agreed on a token or not, the correct

execution replicas always know after just one round of communication with the verification machines whether they can go on and execute the next batch of requests, or should re-execute the current batch before going on. Even in the case where the execution replicas diverge (due to non-deterministic interleavings or faulty replicas), a second round of communication with the verification machines is not needed because the sequential deterministic re-execution of requests guarantees convergence of all correct replicas. This verification stage maintains the liveness property because each batch of requests only requires at most one round of communication with the verification machines.

2.2.4 Performance Results

The experimental evaluation of Eve shows promising results for this approach to a multi-threaded replicated state machine. In particular, the designers of Eve demonstrate with microbenchmarks that Eve is capable of a 12.5x speedup over sequential execution using 16 core machines with 10ms requests. These numbers do however fall slightly with lightweight requests. They find that the improvement decreased to 10x for 1ms requests and 3.3x for 0.1ms requests mainly because of the inability to saturate the system with client requests and the increased overhead of the checkpointing system relative to the time spent performing computation for each request.

The authors also compare Eve's performance to an existing attempt at performing multi-threaded SMR called Remus [7]. In Remus, the primary replica executes client requests with multiple threads and then transfers its entire state to the other replicas. This approach, referred to as passive replication, both requires considerably more bandwidth than Eve (because of the transfer of full states) and does not protect against commission failures, while Eve does. Eve is shown in the authors' tests to outperform Remus throughput wise by a factor of 4.7x while using orders of magnitude less network bandwidth.

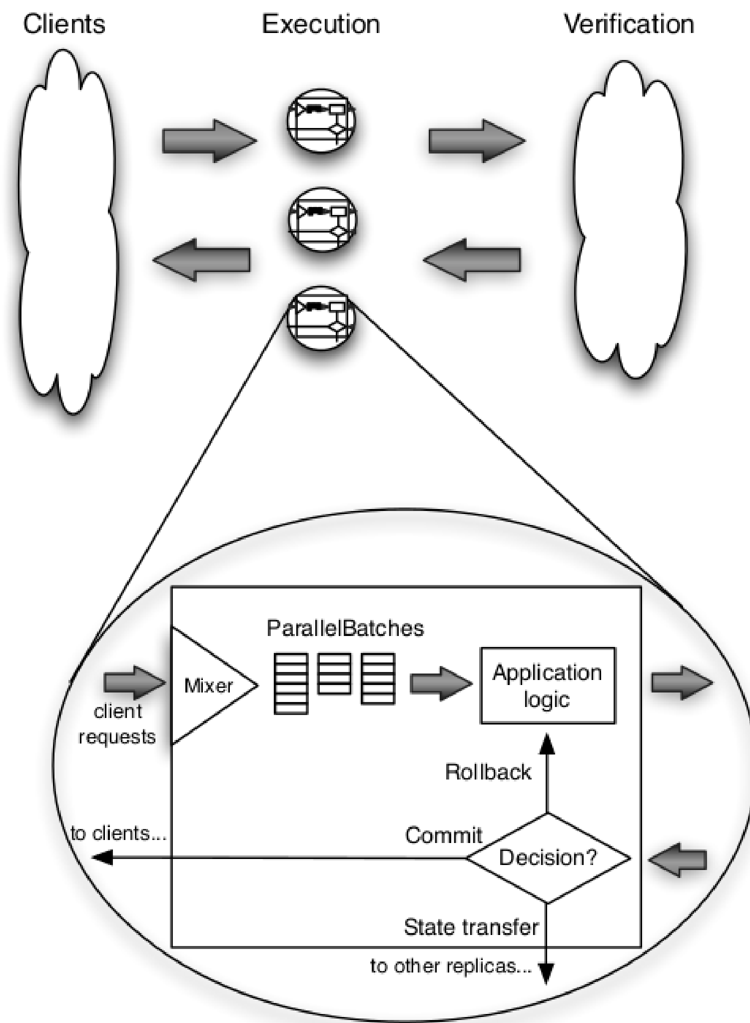

Figure 2.1: An overview of the Eve protocol as it appears in the original paper [14].

Chapter 3

Adam: State Machine Replication with Communicating Services

The Eve protocol can be used successfully in a parallel environment to achieve significant performance improvements in terms of request throughput. Systems today however rarely fit the simple client-server model that Eve adheres to. Instead, most services need to interact with other services while processing a request from a client [1, 5, 8].

For example, if a client visits a cloud file storage website [9], the server handling the client's request needs to make what we will refer to as a *nested request* to a separate service. The function of this nested request might be to retrieve the names of the client's files which must be done before sending the contents of the webpage back to the client. Figure 3.1 shows the anatomy of a client request that we consider for the for the rest of this paper. In the diagram, the local computation can be thought of as the time that the webserver spends generating the HTML to return and the remote computation can be thought of as the remote service loading the names of the user's files from a disk.

Eve's approach to multi-threaded SMR successfully leverages the multicore nature of today's machines, but it is not immediately clear how to extend the Eve protocol to allow replicated services to communicate. In this chapter, we explore the Adam protocol and the previous work that was performed to convert the Eve protocol to work in this new environment: one where servers not only expose their

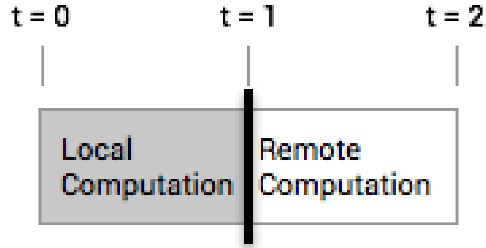

Figure 3.1: The anatomy of a client request that must perform a nested request. The dark line is used to indicate where we ensure all correct replicas have the same state. We say that a request requires two time slices to complete for simplicity: the first for the local computation and an equal amount of time for the remote computation.

states to clients, but also to other services through nested requests.

3.1 System Model

We assume a system model similar to that of Eve. Adam requires periods of synchrony in an asynchronous environment to guarantee liveness although it always guarantees safety. In these periods of synchrony we assume that messages sent between two correct processes are delivered within a bounded time. Outside of these periods, Adam tolerates arbitrary network delays and lost messages without violating safety.

The Adam prototypes described in this paper can be independently configured with their own fault tolerance parameters. Just like the UpRight system [3], Adam is guaranteed to respond to clients in the presence of at most u failures (commission or omission) and guarantees that any response sent to a correct client is correct in the presence of up to r commission failures and any number of omission failures. We assume that no failures in the system can break cryptographic primitives which could for example allow failed nodes to forge a correct node's MAC. This assumption was also made in Eve [13, 14].

3.2 Initial Approaches

Before describing Adam, it is important to understand why current SMR protocols that use speculative execution such as Eve do not work in this environment with communicating services. Consider a service A that is using Eve and processing client requests. While processing some client requests, A must make a call to a separate service B. Now suppose that because of the speculative nature of Eve, too many replicas diverge while processing this client request so service A rolls back the execution of the client requests. When service A re-executes the requests sequentially, there is no guarantee that it makes exactly the same nested requests. Non-determinism caused by thread interleaving could result in a nested request made as a result of the same client request that is different from the nested request that is made when processing the same client request sequentially. If the nested requests performed by A when re-executing the client requests differ from the first time, the state of service B is now inconsistent because B has executed requests that A will never re-execute.

This well-known issue is called the *output commit* problem [10] and a very simple solution to the problem is to always take a consistent snapshot before producing output or exposing internal state to an outside observer. The first attempt to convert Eve to work in this communicating services setting takes exactly this approach, where consistent snapshots are performed by performing agreement and verification on replica states before producing output, and is described in Section 3.2.2. A separate idea to achieve higher performance in this setting is to simply run in a deterministic and sequential manner, but to leverage deterministic pipelining for increased performance. With this idea, Adam would not be speculative and no rollback would ever be needed to maintain correctness. This second approach is described in Section 3.2.3.

3.2.1 Naive Sequential Implementation

The simplest way to imagine implementing SMR with communicating replicated services is to use the typical agree-execute approach. This approach works, although we see in the next few sections that we can do much better. Figure 3.2 shows what such an implementation looks like when executing 12 client requests. Since an agree-execute implementation does not use speculative execution, no checkpoints are needed in this approach; however while remote computation is happening, no

local computation can be performed. As a result, the execution of 12 client requests that incur an equal amount of work on the local and remote services (a simplification we will use for ease of explanation) takes 24 time slices.

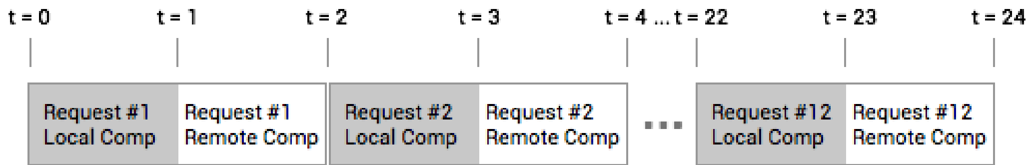

Figure 3.2: An example execution of 12 client requests with a naive sequential implementation. No checkpoints are needed for this approach.

3.2.2 Eve Modification

In an effort to leverage the performance gains that speculative execution approaches such as Eve promise, the first attempt to construct Adam focuses on solving the output commit problem that Eve faces in this environment. To do this, the initial Adam contributors at the University of Texas at Austin modified the existing Eve implementation to take checkpoints before making nested requests.

For the purposes of illustration consider an Eve-like system with multiple execution threads that are processing client requests in a speculative manner. When the execution threads arrive at the part of processing the client request that requires a nested request to another service, they inform the system of their progress and wait. We refer to a thread reaching a point where it must checkpoint as "hitting the wall" throughout the remainder of this paper. After the last thread reaches such a point, the system computes a hash of its state and verifies that all replicas are consistent in exactly the same way that Eve does after processing an entire batch (described in Section 2.2.3). If all replicas are in the same state after processing these prefixes of the client requests, the threads can safely make their nested requests and continue executing until the next time that they make a nested request, or until the end of their last request.

Figure 3.3 shows what an execution of 12 client requests would look like with this approach using just two execution threads. Notice that checkpoints are taken right before the execution threads begin to make their nested requests to the remote service. Also notice that an additional checkpoint is needed at the end to make sure

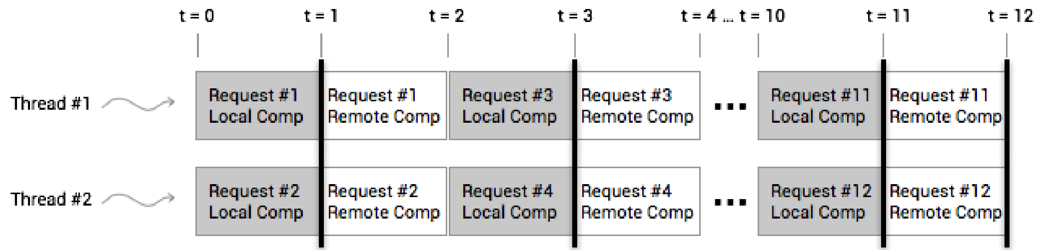

Figure 3.3: An example execution of 12 client requests with the modified Eve parallel approach. For 12 requests on two execution threads, 12 time slices are used and seven total checkpoints (three of which are not pictured) are performed.

all replicas have the same state at the end of the batch. This approach (with two execution threads) finishes the batch of 12 requests in just 12 time slices. Theoretically this would mean a 2x increase in performance when compared to the 24 time slices that the naive approach requires.

3.2.3 Single-Threaded Pipelining

Kapritsos et al. also design and implement a prototype system that leverages single-threaded deterministic pipelining to try and solve this issue. The reason that pipelining is useful is because when a single thread is processing client requests and makes a request to a separate service, it must wait synchronously for the response. If this communication with the remote service takes as long as the computation that the first service must perform, the system is idle approximately half the time. Therefore it is clear that if it is possible to perform more work while waiting for a nested request's response, the performance of the system increases.

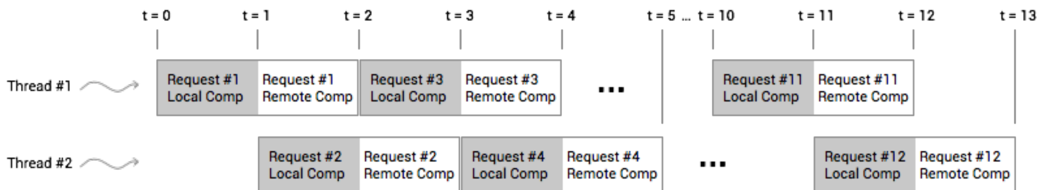

Figure 3.4: An example execution of 12 client requests with the single-threaded pipelining approach. For 12 requests on two execution threads, 13 time slices are used and no checkpoints are performed because all execution is deterministic.

To build a system that performs pipelining, Kapritsos et al. use multiple threads for ease of implementation but only run them one at a time and schedule them in a deterministic way. Deterministic scheduling and only having a single execution thread running at any point in time guarantees that all requests are executed deterministically, and therefore that all replicas in the system converge without the need for checkpointing or potentially rolling back.

In Figure 3.4 the execution of 12 client requests in a single-threaded pipelined system is depicted. Notice that the grey rectangles indicate local computation, or that the replica machine is actually performing work locally. Only one thread is ever performing local computation at a time (guaranteeing determinism), but because of pipelining, the total number of time slices needed is cut almost in half over the naive approach.

3.3 Initial Results

The initial findings in terms of the performance gain that applying speculative execution or pipelining techniques have over a naive sequential implementation in this environment with communicating replicated services are quite promising. In Kapritsos' doctoral defense, he reports that the throughput of this speculatively executing Adam system steadily increases with the number of cores according to the microbenchmarks that he performs [13]. Even though the throughput increases, it does not seem to be scaling particularly well with the number of cores. Servers with 16 cores running 16 execution threads in this implementation of Adam only have about 4 times the throughput of the same servers running with a single thread each.

The single-threaded pipelining technique described in this chapter also shows great potential. For very short client requests (0.1ms each), the service manages to achieve even more than the expected double throughput gain, probably because of the communication itself and the batching process at the remote service creating what acted in practice like an additional pipeline stage. For requests that take longer to process (1-10ms each) however, the system successfully achieves an approximately double throughput increase because of the two stage nature of the pipeline: the first being the local computation and the second being the computation on the remote service.

Chapter 4

Multi-Threaded Pipelining in Adam

4.1 Design Overview

The core of the work that we perform is to adapt the initial work on the Adam project to support both multi-threading and pipelining simultaneously. As discussed in Chapter 3, initial work on the Adam protocol yields some promising results both in the multi-threaded non-pipelined approach and the single-threaded pipelined approach.

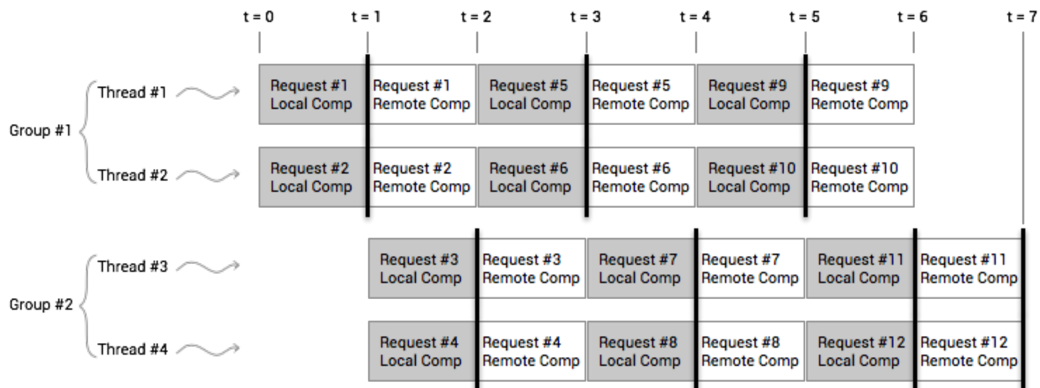

Figure 4.1: An example execution of 12 client requests in the multi-threaded pipelining approach we implement. Two groups of two execution threads each process 12 client requests in just seven time slices.

At a very high level, our approach is to adapt the existing single-threaded pipelining code to have it schedule groups of threads to run in parallel instead of a single thread. Chapter 5 explains at length the steps we took to convert the existing code we started with, as well as the new code we wrote to make this idea a reality.

Figure 4.1 shows what an execution of 12 client requests looks like in our multi-threaded and pipelined system with two groups of two execution threads each. Notice that only one group is performing local computation at a time (the dark rectangles) but multiple threads within a group are running simultaneously. Also notice that the full execution of all 12 client requests requires only seven of the same time slices used in the other diagrams. This abstraction also does not affect the total number of checkpoints that must be performed from the original multi-threaded approach.

4.2 Theoretical Benefits

By combining both pipelining and multi-threading, we expect our speedups over a naive replicated state machine in this system model to both scale linearly with the number of execution threads per group as well as double because of the two stage pipeline. In total this means that with the 16 core machines we have available to test with, we hope to see approximately 32 times the throughput of a naive sequential implementation in our microbenchmarks.

Figure 4.1 is an example of the improvement that we expect by using two groups with just two execution threads each on a batch of 12 requests. Notice that since only two execution threads are performing local computation at any point in time, our system only utilizes two cores on the execution replica's machine. We would expect a 2x throughput increase from using two threads concurrently as well as approximately a 2x throughput increase from the two stage pipeline for about a 4x throughput increase over the naive sequential approach.

Visually, this is exemplified by the number of time slices in each diagram of this paper. The naive sequential implementation (Figure 3.2) required 24 time slices for 12 client requests while the execution in our system required only seven.

$$24/7 \approx 3.43$$

This number actually converges to the desired 4x as the number of requests run in each batch approaches infinity. A formal proof of this fact is available in Jim Given's thesis [11] but conceptually it can be seen that the larger the number of requests that each thread must execute, the less significant the "unmatched" slices at the beginning and end of the batch become.

Chapter 5

Adam Implementation

5.1 Initial Code Structure

When we began our contributions to the Adam project, prototype implementations existed for both approaches described in Chapter 3. Before explaining our work to create a multi-threaded pipelined SMR system that uses speculative execution, we examine in detail the existing building blocks that we use to construct our final implementation.

5.1.1 Existing Multi-Threaded Adam Implementation

The existing multi-threaded Adam implementation we started with is a modification to the existing Eve [14] code that allows the prototype to perform nested requests. Some of the core changes involve the use of Apache's Commons Javaflow library [12] to easily allow execution threads to rollback their state to a checkpoint, even if they are currently in the middle of executing a client request. The other modifications to Eve involve adding support for taking multiple checkpoints per batch, one after every time the execution threads "hit a wall", meaning that all currently active execution threads are either finished with the entire batch of requests or are ready to make a nested request.

Execution threads are controlled in this implementation by the Batch Manager (BM) which handles the logic of making sure all threads have "hit a wall" and also initiates the checkpoint operation for each execution thread. This BM also ensures that all replicas are at the same state at each wall, which involves sending off a hash of the replica's state and initiating a rollback if needed. The verification process is

largely not finished in the BM as the existing multi-threaded attempt is intended to simply demonstrate that speculative execution can still provide performance gains even with communicating replicated services.

5.1.2 Sequential Pipeline Manager

The Sequential Pipeline Manager (SPM) was first implemented by Kapritsos to demonstrate that the single-threaded pipelining approach had promise in this system model. The SPM in particular is the part of the system that controls which execution thread is currently running and handles scheduling the next thread when the active thread desires to yield. This means that the SPM is also responsible for enforcing the condition that only one execution is being performed at any point in time and therefore that all execution is deterministic. This code is very straightforward and serves as the inspiration for the more complex pipeline manager that we build to schedule groups of threads.

The SPM begins the execution of a batch of requests by deterministically assigning requests to each thread in a round-robin fashion. Even though the pipeline executes sequentially, the SPM uses many lightweight threads to simplify the scheduling process. After assigning work to threads, the SPM notifies the first thread to begin. That thread then starts processing a client request until it is ready to make a nested request. The active thread at this point calls a yield function in the SPM to notify it to schedule the next thread. After notifiying, the active thread immediately makes a blocking call to the remote service. When the SPM receives this yield notification, it notifies the thread with the next higher id (modulo the number of threads) to begin execution. Eventually, the last thread with work to do yields the pipeline, at which point the SPM schedules the first thread again.

The first thread then either continues waiting for the nested request to complete, or continues executing the client request until it hits another wall or finishes the request. After completing a client request, the active thread either starts executing its next assigned request or notifies the SPM if it has executed all of its assigned requests. Once all threads have responded in this way, the SPM finishes the batch and releases the results to the clients.

5.2 Core Components

To implement Adam with the multi-threaded pipelined approach, we first combine the two existing approaches into one system before implementing the parts of the functionality that were not yet present. We begin by adapting the Sequential Pipeline Manager (SPM) to schedule groups of threads instead of just individual threads, and then modify the Batch Manager (BM) to control a group of execution threads that can be scheduled onto and off of the pipeline. After making these modifications, we finish implementing parts of Adam that have not been implemented before for the initial proof of concepts such as the verification stage and make appropriate modifications to other existing parts of the code base to support both multi-threading and pipelining simultaneously.

5.2.1 The Parallel Pipeline Manager

By taking the structure of the existing SPM written by Kapritsos (and described in Section 5.1.2), we build what we call the Parallel Pipeline Manager (PPM) to schedule groups of threads in the context of the pipeline. The changes that we make to the SPM to control groups of threads mostly center around initialization and the detection of the correct time to swap groups onto or off of the execution pipeline.

Within the SPM, a simple mechanism is used to schedule client requests to execution threads that run one at a time in a pipelined fashion. Essentially, the SPM assigns the requests to threads by giving each thread one request before giving the first thread a second request and so on. In the PPM, we instead must schedule requests within a batch to both groups of threads and to threads within those groups. This is performed after the mixer has been run on the batch, so we are able to assume that requests are mostly non-conflicting. We experiment with a couple of ways of assigning the requests to the threads but eventually discover that our system works best when we assign every thread in the first group a request before beginning to assign requests to threads in the second group. This process is continued with the remaining groups before assigning a second request to each thread within the first group. We believe that this is the best way to assign the requests because assigning them in this way makes sure that all cores are utilized as best as possible, leveraging the multicore nature of the machine before taking advantage of the pipelining by using multiple groups. We argue this claim more formally in Jim Given's thesis [11].

The SPM also has a method that execution threads call when the active execu-

tion thread is about to make its nested request and therefore is ready to yield the pipeline to the next thread. In the PPM, this procedure is slightly more complicated because the PPM must wait for all threads to be in a state that they can yield the pipeline in. Each execution thread makes the same sort of yield request, but the result of the function call is simply a flipped bit in a bitarray that stores which threads are ready to yield. When the last thread in the currently active group (the group that the PPM is allowing to execute) calls this yield function, the last thread notifies the PPM's control thread. This control thread then realizes that it is time to schedule a new group of threads to run and updates the active group accordingly.

This bitarray also must be updated when threads no longer want to yield and are ready to execute again (e.g. when the nested request returns for a thread). We decide in our implementation to only allow the PPM's control thread to mark when threads no longer want to yield because of the following scenario.

Imagine that in a group of two execution threads, the first thread reaches the point where it is ready to yield and makes its nested request. Pretend that the second thread requires more local computation time and therefore is slightly behind the first thread in making its nested request. The second thread then does not reach the point to yield before the nested request that the first thread made returns. If we allow the first thread to mark that it no longer wants to yield the pipeline (since its nested request had completed), the PPM will not schedule the next group of threads to execute because not all of the threads in the currently active group want to yield anymore. This is an issue because the second request's remote computation could take a long time, in which case no thread is currently executing on the local machine and the entire local service is therefore idle. A simple workaround we found for this issue is to only allow the PPM to mark the active group's threads as not wanting to yield after they have been swapped off the pipeline at least one time.

5.2.2 The Parallel Group Manager and Execution Thread Modifications

The second part of our system that enables groups of threads to run in a speculative manner is the Parallel Group Manager (PGM). This part of the system controls the execution threads within a single group. It is at this layer of abstraction in our system that we handle taking checkpoints, possibly rolling back, and informing the PPM when all of the threads in the group are finished executing their assigned requests within the batch. The Batch Manager (described in Section 5.1.1) served as

our starting point for this part of our implementation, as it was already designed to control multiple execution threads in an environment with communicating services. Some of the major modifications we make to the existing BM code involve making sure that the PGM was able to be scheduled onto and off of the pipeline controlled by the PPM, and that verification was performed in a way to guarantee safety and liveness.

In order to make sure that PGM does not continue executing after being swapped off of the pipeline, we require that it communicate with the PPM to determine the number of the currently active group. If the active group number corresponds to the group controlled by the PGM and its corresponding control thread, the PGM continues instructing the execution threads to process more requests and to take checkpoints when needed. If instead the PGM finds that the current group is no longer its own, it simply waits until the PPM notifies it to continue. Providing the safety and liveness guarantees of SMR involves making sure that verification requests are made at all of the required times and that the responses to these messages are handled properly. More details about our verification modifications and how we handle rollback in this approach are described in Section 5.2.3.

We also make many changes to the execution threads to ensure that they only execute when it is their corresponding group manager's turn. For example, we ensure that the execution threads make the appropriate calls into the PPM to notify the PPM when they are ready to yield and that the threads ensure that it is their corresponding group's turn to execute in the pipeline before proceeding with their own executions.

5.2.3 Verification Stage

In Eve, the system is only required to send out a verification message at the end of the computation of a batch to ensure that all replicas converge [14]. In Adam however, the verification process must be performed before any execution thread makes a nested request as well as at the end of the batch because both of these situations expose intermediate state to potential clients.

In order to make this modification, we first add a second message processing thread to the system that can concurrently process the responses from a verification stage, while another thread is processing incoming client requests. When the verification responses are received from the verifiers by the execution replicas, the execution replicas process each response and determine whether they need to roll-

back, receive a state transfer, or can safely continue executing in the same way. This part of the verification stage works almost like Eve's which is described in Section 2.2.3.

5.3 Rollback and Correctness Guarantees

In this parallel pipeline system we must be prepared to rollback the state of an Adam replica to arrive at one that all replicas agree on in the event that the speculative execution causes a divergence. Note that because we always ensure that all replicas are in the same state immediately before making any nested request, this means that the state that we must rollback to in the case of divergence is either right before some execution threads make a nested request or right at the beginning of a batch. If it is the latter, rollback is trivial, we simply throw away all work performed so far and execute the requests again in a sequential deterministic manner to guarantee that progress is made and that all replicas converge. This deterministic execution can even be performed in a sequential pipelined manner to achieve better performance than a naive sequential approach.

If instead we rollback to right before the threads in a group make a nested request, our system restores the state of all execution threads to the state they had when the latest checkpoint was taken. After doing so and setting the new active group successfully, the system switches all threads into a sequential deterministic mode to ensure progress and convergence. We use Apache's Commons Javaflow library [12], specifically the Continuations objects in this library, to checkpoint the stack of each execution thread. In addition to the stack, the Adam protocol also requires that the objects referenced in the stack are deep copied to the Merkle tree, an operation that is automatically performed simply by annotating the prototype code. This allows us to restore this stack and all referenced objects during a rollback and guarantee that all execution threads are in the exact state they were in right before the nested request was made on each replica. These Continuation objects with the stack of each thread are stored in the same deterministic Merkle tree data structure that Eve uses which allows the replicas to make sure that they also have the same recorded stacks for each execution thread whenever a verification is performed. This also allows replicas to receive continuations in a state transfer and immediately start the execution threads at the same point in the execution.

After performing a rollback in the non-trivial case, the threads in the now current

group re-execute their nested request with the same sequence number that they used the first time which allows the remote service to respond from a cache and not perform operations multiple times. Since we know that all execution threads were in the same state before making this request, we know that the request that each execution thread makes during the re-exceution is exactly the same. This avoids the output-commit problem by ensuring that all outputs to the remote service are always the same, even if they must be redone.

Chapter 6

Evaluation and Results

When we finished implementing the multi-threaded and pipelined version of Adam, we were dismayed to find that the throughput of our system was much lower than we had hoped. For a variety of reasons, we first saw that using eight threads in Adam only resulted in less than a 4x throughput increase over the naive sequential implementation for very large requests when we expected something closer to a 16x improvement. We then took a methodical journey through several heinous bugs and the intricacies of this design until we arrived at much more satisfying performance results. For details on the approach we took to correcting our implementation, and more about the problem that we are trying to solve in general, please read Jim Given's thesis [11].

We perform the evaluation we present here on a testbed of many Dell PowerEdge R200 4 x 2.4GHz Intel Xeon machines and three Dell PowerEdge R515 16 x 3.2GHz AMD Opteron machines. We use the 16 core (R515) machines as our execution replicas and the four core (R200) machines as our verifier, client, and filter machines. Because we only have three 16 core machines, we are only able to perform scaling studies up to 16 cores in an unreplicated setting (meaning we tolerate no faulty execution replicas) with one 16 core machine running as the local service and another as the remote service. For these unreplicated studies, only the execution nodes are not replicated, all verifier machines are still replicated in a fault tolerant way (four total to handle one commission failure). We perform the fully replicated study we describe in this chapter by using each 16 core machine as both an 8 core local replica and an 8 core remote replica which constrains us to only being able to test in a full setting (with $r = 1$) with up to 8 execution threads.

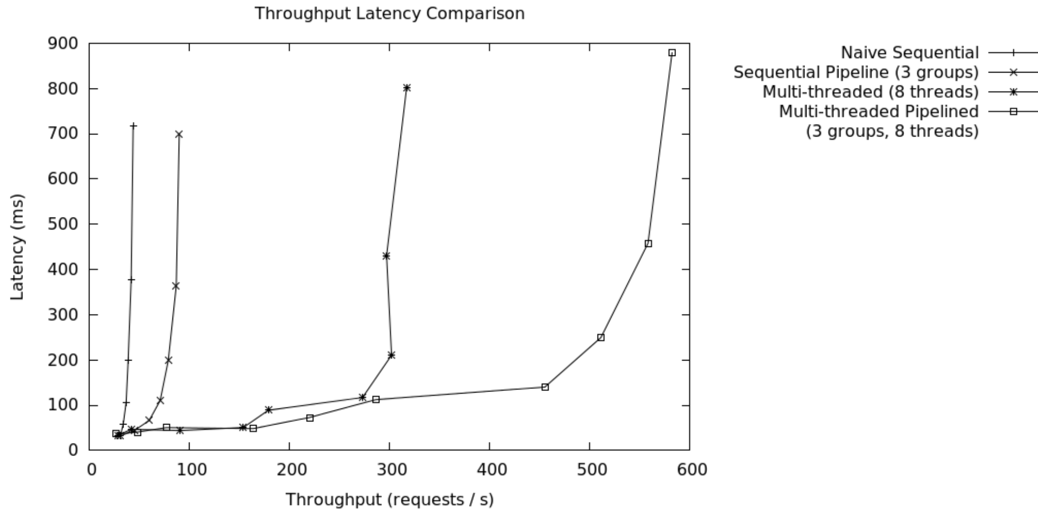

Figure 6.1: Latency-Throughput graphs of Adam for large requests with all four approaches.

We compare our Adam implementation head to head in a fully replicated environment with the previous three approaches described in Chapter 3: the naive sequential approach, the single-threaded pipelined approach, and the multi-threaded non-pipelined approach. Figure 6.1 shows latency throughput graphs for our implementation running with three groups of eight threads each in a fully replicated environment. We compare our implementation in this graph to the multi-threaded but non-pipelined approach running with eight threads, with the single-threaded pipelined approach running with three threads (which is equivalent to our implementation using three groups), and with the naive sequential implementation.

It can be seen from the graph that the naive sequential implementation achieves 45 requests per second which is close to the theoretical maximum throughput for 20ms requests (50 requests per second). The sequential pipelined approach achieves twice the performance of the naive implementation because of the fact that the pipeline is of depth two (the local service and the remote service). The original multi-threaded Adam approach achieves a throughput of just over 300 requests per second with eight execution threads. We expect that the multi-threaded approach achieves at most an 8x increase in throughput over the naive sequential approach which means a throughput of 360 requests per second. We miss this mark by a little more than one multiple of the naive sequential approach throughput because

of the added overhead of the Merkle tree operations and the verification stage. Our multi-threaded and pipelined implementation achieves a maximum throughput of at most 582 requests per second with this setup while the original multi-threaded approach achieves slightly over 300 at its saturation point. This shows that our pipelining improvements achieve close to the throughput doubling that we expect.

One of our chief concerns when we evaluate the performance of our system is to see how the system scales with additional parallel execution threads. Figure 6.2 shows the factor of throughput increase that we achieve by adding more execution threads to our system in an unreplicated setting.

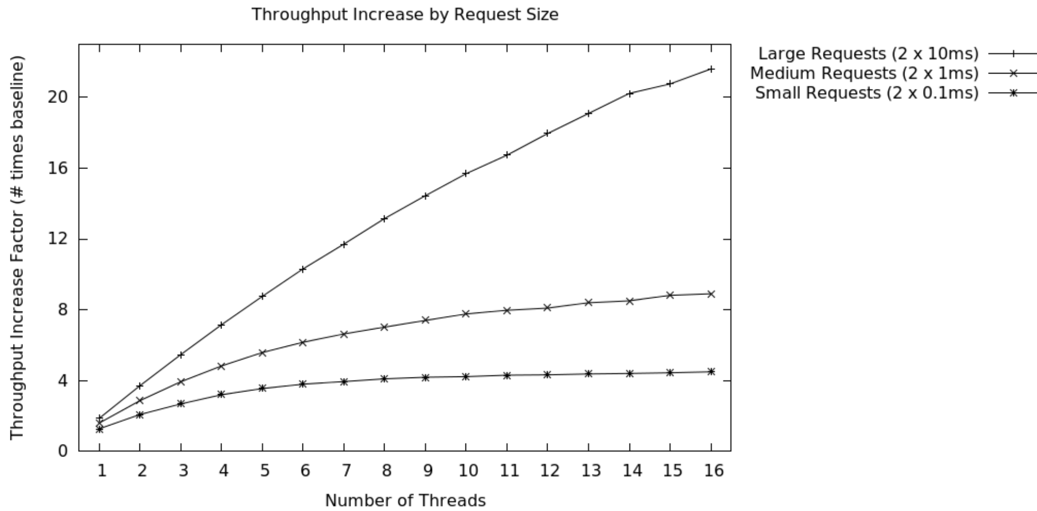

Figure 6.2: Adam speedup over sequential performance for small, medium, and large requests.

These numbers are obtained by running our system with three groups of x threads each where x is the number found on the horizontal axis. We then take the saturated throughput of the system for each number of threads and divide it by the throughput that a naive sequential implementation achieves for each size of request. As mentioned in Chapter 4, we expect the theoretical maximum benefit of our system to be 32 times the throughput of the baseline, gaining a factor of 16 improvement from utilizing 16 cores and then doubling the throughput because of the two stage pipeline. We come closest to this theoretical maximum with large (10 ms local computation and 10ms remote computation) requests because the increased computation time helps mask the overhead introduced by the checkpointing

and the verification stage. We deviate from the theoretical benefit even more for shorter requests as the overhead becomes more pronounced and possibly because of bottlenecks in the system when processing larger numbers of client requests that we have not yet identified.

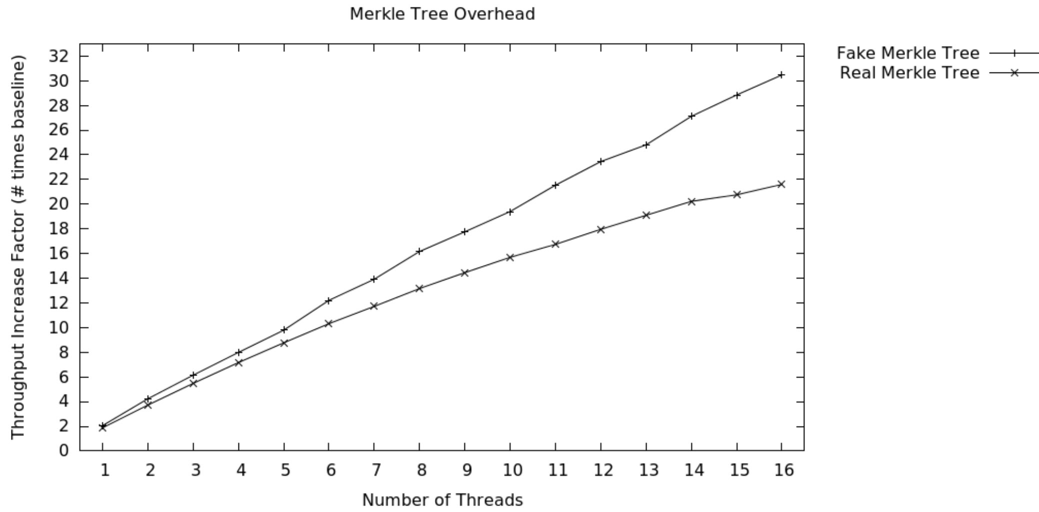

Figure 6.3: Merkle tree impact on throughput relative to the sequential execution baseline for large requests.

As Figure 6.3 shows, the Merkle tree implementation that we use appears to be a major source of overhead in this system. In this graph, we compare the scaling behavior of the unreplicated system when using a real Merkle tree implementation to when we use a fake Merkle tree. With the fake Merkle tree, we perform all of the identical operations that we perform with the real tree but all hashing and Merkle tree versioning operations complete instantly. With the fake tree in this unreplicated setting we achieve a maximum of just over 30 times the throughput (1371 requests per second) of the naive sequential baseline execution (45 requests per second) out of a theoretically possible 32 times the throughput. For the real tree, we achieve a speedup of almost 22 times the baseline (972 requests per second) with all 16 cores, meaning that we are losing nearly 400 requests per second because of the overhead of the Merkle tree's operations. As a result, we believe that there is still room for improvement through optimizations of our Merkle tree implementation. This could be achieved through better detection of new objects in the replica that must be hashed or more intelligent space allocation within the tree itself.

Chapter 7

Future Work

One proposed improvement to the described design that we did not implement is an optimization for the verification process. The idea is to combine the nested requests with the verification requests, eliminating the need for the verifiers and the extra round trip they require. This approach does however require that the remote service is aware that the service making nested requests might have failures of commission while currently the remote service just has to treat Adam replicas as clients that might make repeated requests with the same sequence numbers. In practice, we find that this round trip is not the bottleneck of the system, but this improvement can still easily be made.

As shown in our evaluation of Adam, we find that our system has much better performance gains on longer client requests than shorter ones because of the overhead that the additional checkpoints incur in the form of additional Merkle tree computations. We believe that the system can be modified to minimize the total number of checkpoints needed by making much larger groups and not checkpointing until all of the requests in these larger groups are ready to make the nested request.

Another concern with our current implementation is the need to have large batch sizes. With three groups running in the system with 16 threads each, at least 48 concurrent and non-conflicting requests are needed to assign each thread a single client request to perform. To achieve the performance benefits from pipelining that we describe in this paper, each execution thread must have multiple requests to process which drastically increases the needed batch size. One proposed solution is to design the system so that the pipeline is always active, thereby eliminating the need to restart the pipeline with new batches and instead just adding the work

to the execution threads' queues. We believe this is possible since we are already taking checkpoints for each request, though it complicates the rollback and work assigning procedures that we describe here.

Chapter 8

Related Work

This work built off of existing work from the University of Texas at Austin Laboratory of Advanced Systems Research from both the Eve project and initial proof of concept approaches to Adam. Aside from this already described related work, there has been some work in the field of replicated remote procedure calls (RPC) and work on passive replication systems.

8.1 Replicated Remote Procedure Calls

The concept of a replicated RPC has been around for decades [4,19]. Such a system allows multiple replicated modules to interact through a procedure call interface presented to the system designer. Replicated RPC was explored before advancements were made in the area of speculative execution and as a result the systems in literature rely on sequential execution (or at minimum serialization of requests) to maintain consistency between replicas.

8.2 Passive Replication

Passive replication is a technique employed by systems such as Remus [7] that allows for multi-threaded SMR. These techniques work by allowing the primary to perform all execution speculatively (which can be multi-threaded and therefore non-deterministic) and then having the other replicas passively absorb this state from the primary. One downside of this approach is that it does not handle errors of commission while systems such as Adam and Eve do.

In addition, Remus relies on aggressive checkpoint pipelining for performance that would not work in an environment with inter-service communication. Remus performs speculative execution past checkpoint times, which allows it to buffer client outputs until the state of the primary becomes visible. Outputs of the system originally occur only when the primary releases the responses to client requests. In this new setting however, Remus also has to block until a checkpoint is performed to make a nested request. This additional blocking leaves the system idle until the checkpoint has been performed and is received by other replicas. It is possible for Remus to be adapted to perform similar pipeline techniques to Adam but it would still suffer from its more expensive checkpointing operations and higher network bandwidth usage than Eve [14].

Like Eve, Adam collects only the subset of the state that determines the operation of the state machine and ignores values that vary over the replicas such as the state of communication devices or other unimportant configuration parameters [13]. As a result, Adam has to transmit less over the network in the case of a state transfer than a modified version of Remus. In the common case, replicas in the Adam protocol only has to transmit signed hashes to guarantee consistency compared to the entire state that Remus must transmit before any nested request.

Chapter 9

Conclusion

Traditional approaches to state machine replication use deterministic single-threaded execution to guarantee that replicas agree on the same final state. Although these approaches are straightforward, they fail to take advantage of the multicore nature of today's machines. In addition, these approaches are ill-suited to an environment with communicating services. This is because communication with other services requires a sequentially executing thread to block while waiting for a response.

Newer techniques such as the Eve protocol [14] leverage speculative execution to allow SMR systems to be multi-threaded. Even systems like Eve however do not immediately perform well in an environment with communicating services and known pipelining techniques can be effectively used to improve performance. We present a novel approach to provide the replicated state machine abstraction in an environment with communicating services that leverages both the multicore nature of today's machines and pipelining techniques to improve resource utilization.

We find through the use of microbenchmarks that our design and implementation allows a workload of heavy requests to scale over 21 times the throughput of a naive sequential implementation in a non-replicated setting with 16 core machines. In a fully replicated and fault tolerant setting, we achieve 12.9x the throughput of a naive sequential approach with just eight cores per execution server in our microbenchmarks. Although our current results on smaller request sizes do not scale as well with the number of cores in the system, we believe that the current issues with our system are implementation specific and are not general issues with the design of the system. We demonstrate that speculative execution based multithreading techniques can be used with communicating replicated services to scale the system

with the number of cores of the execution replicas and that pipelining can be applied to these multi-threaded techniques to nearly double the performance of such a SMR system.

Bibliography

[1] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[3] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM, 2009.

[4] Eric C Cooper. Replicated procedure call. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 220–232. ACM, 1984.

[5] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[6] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[7] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual

machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[9] Dropbox. `http://www.dropbox.com`.

[10] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[11] Jim Given. Putting it together: Building state of the art replicated systems for performance. 2015.

[12] Apache Commons Javaflow. `http://commons.apache.org/sandbox/commons-javaflow`. Accessed: 2015-04-22.

[13] Emmanouil Kapritsos. *Replicating Multithreaded Services*. PhD thesis, University of Texas at Austin.

[14] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. *OSDI*, 12:237–250, 2012.

[15] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[16] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[17] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[18] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[19] Kiam S Yap, Pankaj Jalote, and Satish Tripathi. Fault tolerant remote procedure call. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 48–54. IEEE, 1988.