Building Replicated Systems for Performance

Jim Given

May 26, 2015
Contents

1 Introduction .................................................. 2

2 Background .................................................. 5
  2.1 State Machine Replication ................................. 5
  2.2 Agree-Execute ............................................ 6
  2.3 Execute-Verify ............................................ 7
  2.4 Eve .................................................... 8

3 Adam ...................................................... 10
  3.1 Motivation ............................................... 10
  3.2 System Model ........................................... 11
  3.3 System Design ......................................... 11
  3.4 Optimizations .......................................... 13
    3.4.1 Multithreading ................................... 13
    3.4.2 Sequential Pipelining ............................ 13

4 Multithreaded Pipelining in Adam ......................... 15
  4.1 Design Overview ....................................... 15
  4.2 Theoretical Predictions ................................ 16

5 Performance Improvements ................................. 18
  5.1 Experimental Setup .................................... 18
  5.2 Grouping ................................................ 19
    5.2.1 Predictions ....................................... 19
    5.2.2 Results ........................................... 20
  5.3 Batch Partitioning ..................................... 21
    5.3.1 Predictions ....................................... 21
List of Figures

3.1 Sequential Adam work flow ................................................................. 11
3.2 Multithreaded Adam work flow .............................................................. 13
3.3 Pipelined Adam work flow ................................................................. 14

4.1 Multithreaded Pipelined Adam work flow ................................................. 16

5.1 Performance of multithreaded pipelining with various pipeline groups ........ 20
5.2 Performance with respect to batch partitioning policy. ............................... 23
5.3 Performance of Adam based on running the mixer before and after batch creation. 25
5.4 Distribution of batch sizes with 512 clients. ............................................. 26
5.5 Latency-Throughput graphs for different Adam optimizations 10ms requests ... 27
5.6 Merkle tree overhead as threads increase ............................................. 28
5.7 Multithreaded pipelining performance improvement for various sized requests 28
Abstract

Building large replicated systems is a complicated task in which suboptimal implementation and configuration choices can potentially create performance bottlenecks. This thesis demonstrates an approach to making optimal choices by implementing and configuring a multithreaded pipeline in a novel state machine replication framework, Adam. Using this approach, the multithreaded pipeline is able to achieve near-optimal performance, significantly increasing resource utilization.
Chapter 1

Introduction

Because of the decreasing cost of hardware and the advent of cloud computing, distributed systems have become a prevalent aspect in today’s technology. Everything from e-commerce to social networking to data storage involves using a distributed system [1, 2, 3, 4]. These systems can provide better performance, availability, and a higher quality of service to users. Unfortunately, they come at a cost. They are large complicated systems that span many machines and naturally contain a large amount of non-determinism. Even worse, they can be thought of as highly heterogeneous systems in which data can shuffle through a myriad of different locations at varying speeds (e.g. servers, networks, disks). Forty years of research have shown that reasoning about these systems is a non-trivial task, especially when it comes to performance.

A lot of effort is often spent ensuring these large systems are correct and have low computational complexity. Much of the discussion about these systems is centered around their major subsystems or novel features and how these aspects affect the system as a whole. This often leaves very little room to discuss how these core aspects are brought together. [Give example to ground discussion]. Instead, choosing how these core aspects are brought together are left as an exercise for the implementer. While, they might not be the most revolutionary design decisions, these implementation and configuration choices have the ability to make or break the performance of a system. A poor choice in implementation or configuration can render even the smartest system crippled with respect to performance. The goal of this thesis is to show that, with great attention to detail, such poor choices can be avoided. This thesis demonstrates an approach for implementing, configuring, and reasoning about one of these systems with respect to performance.

The approach used is not complicated and when applied, can yield great improvements. The steps in the approach are the following:

1. **Define Expectations** - After implementing all the features of the system (functional, perfor-
mance, or otherwise), define the expectations of the system in terms of performance. Identify a maximum performance for the system. Note that theoretical maximum performance is almost never be achieved, so the expectations might instead be a set of acceptable performance values.

2. **Identify** - Identify an implementation or configuration choice, unspecified in the overall design of the system, that has the potential to affect the performance of the system. This choice can be arrived at during the implementation process when the overall system architecture does not specify a particular detail or found during experimentation when observed performance does not match the expected performance.

3. **Make Predictions** - Consider each option for the choice and predict its performance impact on the system. Develop verifiable predictions for the performance of the system under each choice. Reduce the set of all choices to a reasonably sized set of possibly optimal choices.

4. **Implement and Test** - Implement the most optimal choice or choices derived from the predictions. Run experiments to ensure that the predicted performance impacts are realized.

5. **Iterate** - Repeat steps 2 - 4 until the expectations defined in step 1 are achieved.

This approach should not be considered novel to this thesis. In fact, this approach remarkably resembles the scientific method, the guiding principles that are used to develop these systems. While this approach is used to develop their architectural pillars, it is sometimes forgotten when it comes to implementing and using these systems. Instead, we should apply the same rigor used to formulate these systems when implementing them. In order to demonstrate this, we use this approach to implement a highly optimized version of Adam, a state machine replication framework.

Adam is an architecture which allows replicated state machines to issue their own requests to other replicated services while still maintaining a high degree of performance. Not only is the structure of Adam naturally adapted for multithreaded services, but it also breaks the traditional client-server paradigm. Instead, Adam behaves as a server with respect to its clients and also as a client with respect to the services that it communicates with. These facts differentiate Adam from other replication frameworks. We leverage Adam’s unique position as both a server and client to implement optimization that traditional servers cannot perform.

This thesis starts where the thesis of David Wetterau [5] has left off. In David’s thesis, he describes implementing the most recent major optimization in Adam, multithreaded pipelining. This thesis applies the aforementioned approach to the multithreaded pipelined Adam system. Using this approach we are able to achieve 81% of the theoretical performance for large requests.
I highly recommend the reader to first read his thesis to gain a better understanding of the work involved to implement multithreaded pipelining in Adam.

The rest of the thesis is as follows: Chapter 2 provides motivation for state machine replication and Execute-Verify architectures. It outlines of the Eve system, the logical predecessor to Adam. Chapter 3 overviews of the Adam system. Chapter 4 discusses in more detail the pipelining optimization in Adam and present theoretical predictions for the performance of the system. Chapter 5 lays out the experimental setup as well as discusses a couple of instantiations of the approach. Chapter 5 also presents the final results of the multithreaded pipelining optimization in Adam. Chapter 6 discusses some work related to Adam, highlighting some of their advantages and disadvantages. Finally, Chapter 7 concludes the thesis.
Chapter 2

Background

In this chapter, we describe the problem that Adam solves as well as previous approaches to solve that problem. First, in Section 2.1, we motivate state machine replication. In Section 2.2 we discuss a classical approach for state machine replication and its limitations. Lastly in Sections 2.3 and 2.4, we present a better paradigm for state machine replication, Execute-Verify, and describe Eve [6, 7], a system that implements this approach.

2.1 State Machine Replication

For the past 40 years hardware has gotten increasingly cheaper. Using this trend, enterprising researchers and companies have attempted to use more hardware to provide more available services by isolating and mitigating the effects of machine failures. One way to achieve this is to have multiple machines replicate the same service and mimic the behavior of a single machine. Clients of this service can therefore be agnostic to how the service is implemented and only interact with what they believe to be a single machine. This abstraction is called the single machine abstraction. When a replica, a machine replicating some service, fails, the failure can be masked by other correct replicas in the system and thus this service is more available than a service provided by a single machine.

This abstraction is so fundamental and important to distributed systems that there is a specification for it. Before presenting the specification, we need to outline the abstraction that these replicas provide. In essence, each replica in the system contains some local state. It processes requests from clients by changing its internal state and possibly responding with a message indicating so. Thus, these replicas can be thought of as state machines that perform deterministic state transitions in response to client requests. The goal of these state machines is to create the abstraction of a single
state machine, thus the specification of a replicated state machine is the following:

**Specification.** A Replicated State Machine (RSM) provides the abstraction of a single machine if all observable output (including messages) is the same across all correct replicas.

Crucially the abstraction of a single state machine allows clients to assume that remote services are deterministic and cannot partially fail. All the disadvantages and complications of a distributed system are kept internal to the replicated state machine.

Creating a system that provides the abstraction of a replicated state machine is a non-trivial task. Replicas must coordinate with one another to provide a service that is consistent with the behavior of a single machine despite network latencies, machine failures, or other sources of non-determinism. The classic technique for implementing an RSM is the Agree-Execute approach, which is the topic of the next section.

### 2.2 Agree-Execute

Traditional replicated state machines have used the Agree-Execute approach to maintain the RSM abstraction. In an Agree-Execute approach, correct replicas first agree on the sequence of requests that the system will process. Once the order is agreed upon, each correct replica executes each request sequentially. Since each request corresponds to a state transition, an ordering of requests is equivalently an ordering of state transitions. Since each correct replica is a deterministic state machine and they all start from the same initial state, then by applying the same sequence of state transitions they must all proceed through the same sequence of states, by definition. Thus, from any client’s perspective, any observable state is consistent across all correct replicas.

The Agree-Execute approach is a central concept and driving force behind distributed systems research. In fact, one of the first applications of asynchronous consensus was to create a replicated state machine using the Agree-Execute approach. Lamport’s presentation of the Paxos protocol presents an implementation of a replicated state machine in which replicas use the Paxos protocol to agree on the order of incoming client requests in an Agree-Execute fashion [8, 9]. Since then, a flurry of subsequent systems have provided the RSM abstraction using Agree-Execute architectures [10, 11, 12]. The Agree-Execute approach is so intertwined with the concept of a replicated state machine that agreement on the total ordering of requests is part of the specification of a replicated state machine in Schneider’s survey of replicated state machines [13].

However, the requirements that the Agree-Execute approach imposes are stronger than those of the specification for a replicated state machine. Crucially, the Agree-Execute approach requires that all replicas make the same state transitions and thus go through the same sequence of states.
On the contrary, the specification requires that only state observable by a client be the same across all correct replicas. The specification makes no restriction on intermediate state of the replicas. Furthermore, the Agree-Execute approach relies on sequential processing of requests for correctness. This requirement prevents the Agree-Execute approach from naturally achieving multicore performance inherent in today’s computing environments.

2.3 Execute-Verify

To contrast the Agree-Execute approach, there is another method of implementing state machine replication, Execute-Verify [6, 7]. In the Execute-Verify approach, correct replicas execute requests without first agreeing on any particular order of the requests. Before exposing state to entities outside the RSM, correct replicas attempt to come to an agreement on what is the correct state of the service they are providing. If the correct replicas agree, then it is safe to expose internal state. If the correct replicas do not agree, then steps must be taken to ensure that correct replicas come to an agreement. In practice, each correct replica’s state must be rolled back to a time when they all agreed on their internal state. Then those requests that were rolled back must be re-executed. This model imposes weaker requirements on local execution than the Agree-Execute approach. Again, the Agree-Execute approach guarantees that not only is all exposed state consistent, but also that the sequence of states that each correct replica goes through is exactly the same. Instead, Execute-Verify guarantees that all exposed state is consistent among the system, but it gives no guarantee about the intermediate states of each replica. Correct Replicas are allowed to temporarily have inconsistent states as long as they converge before they expose any state.

Consider the following scenario for which Agree-Execute is too strong. Suppose a replicated state machine needs to process two requests, the first stores the value 3 in X and the second stores the value 5 in Y. In the Agree-Execute approach, the order of these requests is decided amongst all machines and each machine processes these requests in that order. Thus, if one machine stores 5 in Y first, then all machines necessarily store 5 in Y first. Obviously, the order in which these requests get processed does not have any effect on the final state of each replica. If no state is exposed before both operations have completed, then the ordering of requests need not be deterministic across all machines. In the Execute-Verify approach, replicas are free to process these requests in whichever order they want so long as at the end of processing both requests all correct replicas agree that X contains the value 3 and Y contains the value 5.

Because machines are allowed to temporarily disagree on state, each replica is able to execute requests in different ways. This allows replicas to naturally execute requests in parallel. While
executing the requests in parallel, replicas might temporarily diverge, but as long as they converge in the end, the system can still provide the RSM abstraction. Such an optimization would not be possible in an Agree-Execute architecture since the architecture assumes that the replicas makes the same sequence of state transitions. The following section describes Eve, an implementation of an Execute-Verify architecture whose primary focus is multicore performance.

2.4 Eve

Eve [6] is an implementation of the Execute-Verify approach that attempts to leverage the weakened requirements of Execute-Verify in order to gain multicore performance. The basic flow of Eve is based upon the idea of Execute-Verify, although much attention is spent to avoid permanent divergence, ensure progress, and reduce the overhead of the verification step.

What follows is a high-level overview of the Eve system. To begin, clients send requests to a primary replica in Eve. The primary replica gathers up requests into what are called parallel batches and sends these parallel batches out to all the replicas in the system. Each correct replica then executes the parallel batch in parallel collecting but not releasing client responses. When all requests have been executed, each correct replica computes a hash of its state and all potential client responses and sends that hash along with the hash of the previously committed state to a set of verifiers. The verifiers then run an agreement protocol on these messages. The output of the agreement protocol can be commit, rollback, or state transfer. This output determines the next stage of execution in the following way:

1. Commit - The verifiers decide to commit if all correct replicas agreed upon their state and it is safe to send out client responses. Correct replicas can commit their local state, send out client responses, and continue normal execution.

2. Rollback - The verifiers decide to rollback if there is no majority of correct replicas that agree on the state of the system. At this point, all correct replicas roll their internal state back to the last committed state. The requests that have been rolled back are then re-executed sequentially to ensure forward progress of the system.

3. State Transfer - The verifiers decide to perform a state transfer when some minority of replicas do not agree with a majority of correct replicas about their final state. The replicas who have diverged then request a state transfer from replicas who are believed to be correct.

Divergence is expensive in an Execute-Verify architecture and should be avoided as much as possible. Not only does rollback require that all replicas re-execute a parallel batch of requests,
it forces the re-execution to happen sequentially. Since Agree-Execute architectures traditionally don’t have to rollback, this has the potential to be a major performance bottleneck in Execute-Verify system that Agree-Execute systems don’t have to deal with.

In order to mitigate this problem, Eve employs a request mixer. Before executing each parallel batch, each correct replica runs a deterministic mixer over the entire parallel batch. Using application specific information, the mixer attempts to find requests that might cause divergence when executed in parallel. Once found, these requests are moved to separate parallel batches to reduce the chance of replica divergence.

Execute-Verify requires correct replicas to agree on application state rather than a sequence of requests. Since application state can often be several orders of magnitude larger than requests, without clever thinking this operation can be unwieldy. To mitigate this effect, Eve stores all application specific data in a merkle tree. By design, the merkle tree efficiently computes a hash of its contents and it is this hash that is sent off to the verifiers. Furthermore, the merkle tree used in Eve is versioned between commits. Updates in the merkle are tagged with a version number corresponding to the number of times that the system has performed verification. During rollback, the merkle tree removes all updates to the merkle tree that have a version number larger than the last committed version of the tree. Furthermore, during state transfer, a correct replica needs to only transmit the updates tagged with a version number or numbers that another replica disagrees on. The replica receiving a state transfer can then apply these updates to the agreed upon version of the merkle tree. This eliminates the need for correct replicas to send the entire application state across the network during state transfer.

Eve’s results demonstrate that it is able to effectively utilize its multicore environment. When processing requests with heavier payloads (10ms of execution per request) on 16 threads, Eve’s throughput is 12.5x over an Agree-Execute implementation. Furthermore, for lighter requests (1ms of execution per request) Eve is able to achieve a 10x performance increase. These results and others give hope that the weakened requirements of the Execute-Verify approach can yield an increase in performance for replicated services.
Chapter 3

Adam

In this section, we present Adam [7], which provides the platform for instantiations of the methodology. In Section 3.1 we discuss the limitations of Eve and present motivations for Adam. In Section 3.2 we present the system model for Adam. In Section 3.3 we present Adam as a modification to Eve which allows for external communication within a request while still maintaining correctness. Finally, in Section 3.4 we explore possible performance improvements for Adam.

3.1 Motivation

The environments for replicated state machines have changed significantly since their formulation. Originally, replicated state machines would only communicate with clients as they process their requests. In modern data centers, it is often the case that, while processing a client request, a service needs to issue its own request, a nested request, to some other service. For instance, a replicated webserver might need to access a replicated database in order to serve out a webpage. Alternatively, an online shopping might need to make a request to a bank when processing a client’s checkout. In fact many modern day services and data centers consist of many intercommunicating systems [1, 4, 2].

The Agree-Execute architecture is already suited to handle this kind of environment. It guarantees that all state transitions are the same across all replicas, thus the system is assured not to expose any inconsistent state to any party outside of the system. Like before, this approach is unable to leverage modern day multicore environments. Instead, this problem is a ripe candidate for the Execute-Verify approach. The Execute-Verify approach handles possible divergence by making correct replicas agree on their state before outputting state to any entity, client or otherwise. In Eve, this meant agreeing on internal state before sending messages to the client. For this new
3.2 System Model

Adam is able to be configured to work in a variety of system conditions. Adam can tolerate a lossy network in which arbitrary messages get dropped. It can work in synchronous networks, where message delivery time is bounded, or asynchronous networks, where message delivery time is unbounded. Since general consensus is impossible in an asynchronous system where replicas can fail [14], Adam requires periods of synchrony to make progress but stays safe no matter the network connection. Adam stays correct given a configurable number of omission and commission failures. We assume that processes are unable to break cryptographic primitives, for example generate another process’s MAC. Emmanouil Kapritsos’s thesis [7] contains an in depth discussion on the fault tolerance of Adam. For the purposes of this thesis, the precise nature of the faults that Adam can tolerate is not related to the performance of the system during failure-free execution. All that is necessary to know for this thesis is that correct replicas must perform verification to detect and repair divergence or faults in the system.

3.3 System Design

Adam is a system which allows services to issue nested requests in an Execute-Verify fashion. Adam builds off of the system model of Eve. Client requests are collected together at a primary replica into parallel batches which are then sent out to all replicas in the group. Correct replicas then begin processing the requests in the parallel batch and verify computation before outputting internal state. The major difference between Adam and Eve is that Adam allows requests to generate nested requests while being processed.
As stated before, a nested request poses the same problem that client responses posed in Eve, namely they expose internal RSM state to external entities. Therefore, these nested requests must satisfy the constraint presented in the specification for state machine replication. If these requests were not subject to this constraint, then external services may be irrecoverably altered by the speculative state of Adam. For instance, if an online shopping center issued a payment request to some bank, then before responding to the client, the shopping center decided to rollback, the payment request might be reissued to the bank. In the best case, these requests would be identical and the bank will be able to drop the request, however in the worst case, the first request already altered the state of the bank and cause a series of other operations (overdraft fees, money transfers, etc.) that cannot be rolled back.

Thus, in Adam nested requests are treated the same as client requests. Before nested requests are sent out, replicas must perform verification to ensure that their internal states as well as nested requests agree with each other. In Adam, any operation that causes verification to happen is considered a wall. When execution reaches a wall, replicas compute a hash of their local state and previous state, which is sent to a group of servers for verification. The result of verification is exactly the same as it is in Eve. Replicas can either be told to continue execution and send out their message, rollback to a previously agreed upon state, or request a state transfer from a set of replicas who are believed to be correct. Execution of batch proceeds from one wall to the next, in units called slices.

Unlike in Eve, checkpointing in Adam is done much more frequently. Eve was guaranteed to checkpoint exactly once in a batch while Adam can checkpoint arbitrarily often due to the amount of nested requests an application sends while processing a client request. Since Adam can be rolled back to a state in which a request is partially executed, Adam needs to store application context for each checkpoint. Adam achieves this by storing thread continuations in the merkle tree along with application state. In order to rollback in Adam, previous application state is extracted from the checkpoint and worker threads resume execution from a continuation stored in the checkpoint. These continuations are part of the application state and therefore are included in the hash that is sent off to verification.

This naïve implementation of Adam is able to correctly issue nested requests to other services. Unfortunately, this design is limited to single threaded performance and, moreover, a large chunk of execution time is spent waiting on backend computation. Since the assumptions that Adam uses for correctness specify nothing about temporary divergence of replicas or a linear order of requests in the system, this system is ripe for optimizations.
3.4 Optimizations

3.4.1 Multithreading

Since Adam follows the same basic structure as Eve, multithreading is a natural optimization for Adam. In multithreaded Adam, once a replica receives a batch of requests, requests are assigned to worker threads. These worker threads begin executing their sequence of requests until they come to a wall, at which time they pause execution. Once all threads reach a wall, a hash of the internal state of the system is computed and sent off for verification. In the case that verification fails and replicas have to rollback, replicas rollback to the last agreed upon slice and execute the remainder of the batch sequentially to ensure liveness in the system.

Multithreading allows Adam based services to achieve greater performance than a naïve sequential implementation, however there is a significant amount of idle time on the Adam system while nested requests are being serviced. This optimization efficiently uses the physical resources of modern systems, but is unable to stay saturated due to the constant switching between local and remote computation.

3.4.2 Sequential Pipelining

Normally when a nested request is issued to a service, execution in Adam is paused. This leads to unnecessary idle time in Adam. As an alternative, Adam can pipeline requests. First, requests from a batch are partitioned into groups which each act like a batch. Execution of the overall batch starts by executing the first request in the first group. Once execution reaches a wall and local state is verified, the nested request is sent out. While the nested request is being executed remotely, the first request of the second group executes its request until it hits a wall. Again if verification succeeds then it sends out the nested request and execution of the pipeline is passed to the third group and so on and so forth. When the last group issues its nested request, execution is passed...
back to the first group in the pipeline. At this point, it is likely that the response for the first request has already returned and thus there is no idle time while the nested request is being processed.

Using sequential pipelining, Adam is able to reduce idle time and perform better than naïve sequential Adam. While this performance is better, it is also unable to utilize the multicore environment found on most commodity hardware. Since pipelining and multithreading are not exclusive optimizations, there is a chance that these two optimizations can be combined, with great attention to detail, to create a system which not only utilizes multicore environments but also reduces idle time while waiting for nested requests.
Chapter 4

Multithreaded Pipelining in Adam

While Adam allows replicated services to do more than they could in Eve, namely issue nested requests, the system can benefit from two major optimizations, multithreading and pipelining. In this chapter we give a high level overview of the multithreaded pipelining optimization in Adam. We also establish a theoretical peak throughput for the Adam system using this optimization. This is used as the overall goal when using approach established in Chapter 1 to make implementation and configuration choices in Adam.

While the implementation details of the pipelining optimization are noteworthy, they will not be discussed in great detail here. Instead, I recommend reading the partner thesis to this thesis, Communicating Replicated State Machines by David Wetterau[5], which gives a detailed discussion into the implementation of multithreaded pipelining in Adam.

4.1 Design Overview

Multithreading and pipelining are orthogonal optimizations. Pipelining gives the abstraction that there are multiple independent machines executing since exactly one group in the pipeline is executing at any given time. On the other hand, multithreading gives the abstraction that a single machine can have multiple threads of execution at a given time. Thus, these optimizations can naturally compose into a system such that at any given time one group of multiple threads is executing.

To implement this, the following changes were made to the single threaded pipelined Adam execution flow. Instead of creating \( n \) worker threads or \( k \) pipeline groups per Adam replica, multithreaded pipelined Adam creates \( k \) groups of \( n \) worker threads per replica. Requests from a parallel batch are then handed out to all \( kn \) threads. When execution of the batch starts, the 1st group of
Figure 4.1: Execution pattern in multithreaded pipelined Adam.

$n$ threads begins executing their work while all other threads remain idle. When the 1st group reaches a wall, threads checkpoint, and send a verify message to the verification cluster. After the verification cluster responds with *commit*, the 1st group issues nested requests to other services. At this point, the 1st group of threads is idled while the 2nd group begins executing. The sequence continues until the $k$th group issues their nested requests at which point the $k$th group is idled and the 1st group begins executing again. Hopefully, by the time the 1st group resumes, the responses from the 1st group of nested requests have returned and the 1st group does not have to wait before continuing execution. The expectation is that the Adam cluster will, at all times, be executing in parallel some set of client requests while nested requests get serviced.

Interestingly enough, this optimization has no bearing on the correctness of the system. Note that before any nested requests or client responses are made, all replicas perform verification. This ensures that all exposed state is the same across all correct replicas. Furthermore, since only the requests from a single batch are divided up into groups, all guarantees provided by the mixer hold across groups. Pipelining creates dependences between groups, since one group could depend on partial modifications from some previous group. Because of this, when application state is rolled back due to divergence, each thread in each group must also be rolled back. Each thread has to restore their continuation from a committed version of the merkle tree and re-queue the requests which have been rolled back. Again, Adam finishes execution of a rolled back batch deterministically and sequentially to ensure liveness.

### 4.2 Theoretical Predictions

In this section we derive a theoretical maximum throughput for the pipelined Adam system. For this thesis, we consider the smallest environment that Adam could be deployed. This consists of a set of clients that send requests to a service, called the middle service. This middle service processes client requests and, while doing so, sends nested requests to another service, the backend
service. The backend service simply processes requests from the middle service and responds. The middle service is a group of machines that use the Adam framework to implement a replicated state machine. In general, Adam is agnostic to the configuration of the service that it communicates with, but for the purposes of testing the performance of Adam, we need a backend that performs as well as the middle service to ensure that measurements accurately reflect Adam and not the backend service. For this reason, we utilize the success of Eve and replicate the backend service with the Eve framework. Also, for the sake of verifying performance, all requests will be homogeneous and consist of some configurable time spent on the middle Adam group as well as some configurable time spent at the backend.

First, we consider the benefit from pipelining only. In order to derive a prediction for the performance improvement, network latencies and other overheads are considered negligible in this derivation. We predict that, when middle and backend workloads are configurable, the theoretical performance improvement is 2x over a non-pipelined implementation. An outline of the proof is given below.

**Sketch of Proof.** The optimal outcome of pipelining is that each computation site is constantly active. Thus, the optimal ratio of work between the middle and backend is 1:1 since any other ratio would create a buildup of requests at one site and idle time at the other. If each computation site is performing work at all times, twice as much work is being performed when compared to the non-pipelined case in which only one site is executing at a time.

Note that this derivation is based upon a theoretically perfect system. In practice, no such system exists. Instead, in some cases, many of the assumptions made here are broken and in such cases one might be able to achieve more than a 2x performance. Having said that, we use this prediction for the purposes of this thesis and some initial evaluations of the system. In order to evaluate Adam in an environment more similar to the one presented here, we initially evaluate Adam with larger requests that are able to mask many of the overheads assumed to be negligible.

Next, we note that in the optimal case, allowing \( n \) threads to perform simultaneous work can at most increase performance by a factor of \( nx \). Since each pipelined group is multi-threaded, these two optimizations can be considered composable. In the most optimal case, they can be considered as \( n \) independent machines each of which is pipelined. Thus, in the absence of overhead, the maximum performance should be \( 2nx \) greater than the single threaded non-pipelined implementation. Furthermore, if each request takes \( c \) seconds to be fully processed then the maximum throughput should be \( \frac{2n}{c} \).
Chapter 5

Performance Improvements

In this chapter we apply the methodology presented in Chapter 1 to the implementation and configuration of a multithreaded pipelined Adam. First in Section 5.2, we look at how the number of groups in the pipeline affect performance and determine the optimal number of groups for the system. Next in Section 5.3 we look at how the demands of pipelining and multithreading interact when partitioning requests. We look at how to balance the limited amount of requests in a parallel batch between these orthogonal optimizations and derive an optimal policy for the partitioning of requests in a batch. In Section 5.4, we look at different possible locations for the mixer during execution of the batch and how these locations affect performance. Lastly, in Section 5.5 we present the overall performance of multithreaded pipelining and identify major causes of sub-optimal performance.

5.1 Experimental Setup

To analyze the effectiveness of the Adam framework and the multithreaded pipelining optimization, we implement a simple key-value store. Unless otherwise noted, all tests involve the following set up. The tests consist of a variable number of clients sending requests to this test server replicated through the Adam framework. Each request causes the server to work for 10ms to mimic application specific work that would be done by the server. After processing the request for 10ms, the server writes a value to the local key-value store and issues a nested 10ms request to another service. While it is not critical what the service is, for testing purposes, the backend service runs that same application replicated with the Eve framework.

All performance evaluations are performed on the following system. The Adam middle-end group and Eve backend group consists of one replica that is located on one Dell PowerEdge R515
Each server is equipped with 2 8-core 3.2GHz AMD processors, 64GiB of physical memory. Each server contains either one Adam replica or one Eve replica. Tests are performed with at most 16 threads per Adam or Eve replica. Verifier groups for both Adam and Eve groups are located on four Dell PowerEdge R200 server blades. Each R200 is equipped with one 4-core Intel Xeon processor running at 2.4GHz and 8GiB of memory. The Adam group, Eve group, and verifier groups run Ubuntu Linux 12.04. Clients run on four R200s identical in hardware to the verifiers but run Ubuntu Linux 14.10 instead. Each client server can host up to 256 clients at a time thus the system can have a maximum of 1024 clients in any experiment.

Though testing, we found that the performance difference from an Adam group of 3 replicas and 1 replica incurs negligible overhead in failure-free execution. In order keep the backend and frontend homogeneous and utilize all 16 cores per replica, we chose to perform our tests with a replication factor of 1. The system still performs all the same actions that it would have otherwise performed however verification trivially passes with a quorum size of 1.

5.2 Grouping

Recall that pipelining is an optimization for any intermediate service, like the services that Adam can replicate. The goal of pipelining is to reduce the amount of idle time spent at each computation site. To achieve this in Adam, once a set of requests are sent to another service for a nested request, a different set of request are allowed to execute in Adam while the first set waits for their response. These sets of requests are called groups and are a configuration option when using Adam.

Choice. What is the optimal number of pipeline groups?

5.2.1 Predictions

If the goal of pipelining is to minimize idle time at all sites of computation, then the optimal number of pipeline groups should minimize idle time at each computation site. Obviously having only one pipeline group is sub-optimal. One pipeline group ensures that only one site of computation is occupied at any given time and since Adam makes nested requests to other services, there is guaranteed to be more than one site of computation in the system. In fact, having only one pipeline group should reduce the system to exactly the behavior of a non-pipelined system. Thus, Adam should use at least as many pipeline groups as there are sites for computation in order to saturate each site.
Figure 5.1: Performance of multithreaded pipelining with various pipeline groups.

For the moment, disregard network latencies and assume that each replica group begins working on an available group as soon as it is ready. Then it would be sub-optimal to have more pipeline groups than sites for computation. If there were, then there would always be some groups that are not being processed which would not help the system perform any better. Since each group incurs some amount of overhead, it is expected that performance of Adam with more groups than sites would be slightly worse than Adam with as many groups as sites, but still better than the single group case.

Now factor in network latencies and the fact that the execution speeds of these services are slightly noisy. It can be the case that backend might not finish and deliver responses to Adam exactly when waiting groups are scheduled back onto the system. In this case, there is some time that the middle server will be waiting for the backend to respond. Thus, in practice it might actually be optimal to have slightly more pipeline groups than sites for computation.

In our experimental setup we have two sites for computation: the middle replica group running Adam and the backend replica group running Eve. Thus, we would predict optimal performance with three pipeline groups, however we expect and number of pipeline groups greater than one to approach the expected 2x speedup from pipelining.

5.2.2 Results

In order to verify the most optimal number of groups for the pipeline, the same workload was run for different numbers of pipeline groups. The test consists of 32 to 1024 clients each sending
400 requests each of which takes 10ms on the Adam system and 10ms on the Eve system. As expected, having only one pipeline group reduced the system to non-pipelined performance. As a bonus, the performance of the non-pipelined system is roughly half that of the performance of the pipelined version with two or more groups, which helps verify our pipelining prediction. Furthermore, performance does not drastically increase past two pipeline groups which is expected. As long as the number of pipeline groups is greater than two, the amount of requests that can be processed at any given time is the same. Finally, we note that the performance of three and four groups does marginally better than two groups. As predicted, this is because a two group system is noticeably affected by small perturbations in the timing of backend responses. If the backend is slightly perturbed by network latencies or other effects out of its control, Adam will idle waiting for a response for a brief time. To avoid this, three pipeline groups are used in practice since these ensure that some set of requests is always on deck to be computed next. By the time the third pipeline group finishes their slice, the responses from the backend will have definitely returned.

5.3 Batch Partitioning

Multithreaded pipelining in Adam has given Adam two powerful optimizations at the same time. Although during execution these optimizations can be considered orthogonal, each optimization depends on the amount of requests that need to be serviced. Unfortunately each optimization organizes work differently. Typically, multithreading partitions requests into queues of requests for each thread then each thread processes their queue concurrently. Pipelining, instead, partitions requests into groups which are run disjointly to increase resource utilization. Despite both methods of partitioning requests, the methods do exactly the opposite of each other during execution. Thus, the method of partitioning requests into pipeline groups and parallel work queues can effect the performance of each optimization and the system as a whole.

**Choice.** *What method of partitioning requests in a batch will achieve the best performance?*

5.3.1 Predictions

Obviously, any partitioning policy that completely ignores one optimization will not perform well. Suppose that pipelining is ignored and all requests are distributed amongst \( n \) parallel work queues. Since there is effectively only one pipeline group in the system, we have trivially reduced the system to solely multithreaded performance. Similarly, we can implement a policy that completely ignores multithreaded execution and just assigns work into \( k \) pipeline groups. This policy will
create only a single thread of execution in each pipeline group thus reducing the system to single threaded pipeline performance. Any sensible policy will be sensitive to both optimizations.

It may help to imagine that one Adam replica contains $k$ machines of $n$ threads such that at any time only one machine is running. Obviously the optimal division of requests assigns each machine the same number of work and has all threads executing requests when their machine is running. Thus, in the optimal case, batch sizes should be a multiple of $kn$ such that each of the $kn$ worker threads gets an even amount of work. What about the case when the batch size is not a multiple of $kn$, which could be caused by many factors (e.g. timeouts or under-saturation)?

In the case where a batch size is not a multiple of $kn$, request partitioning should still try to be as even as possible in order to waste the least amount of time. Therefore, requests should be assigned in units of $kn$ in which each of the worker threads gets one requests. Eventually there will be a remainder of requests, $c$, smaller than $kn$. We can consider two policies: Prefer threads such that a request is given to the first thread of each group then the second thread of each group and so on or prefer groups such that a request is given to each thread in the first group then each thread in the second group and so on.

We first note that each of these policies wastes the same amount of time on Adam groups. Each policy has to handout $c$ requests where $c < kn$ thus there will always be $kn - c$ threads that will not have work and instead be idled. Of course if it is the case that an entire group contains no more work, it will yield the pipeline immediately. However, that group is now unable to mask remote computation that would otherwise not be noticed by the system. Effectively it still wastes the same amount of time because now some group has to idle while waiting for backend responses.

The difference between these two policies lies in their use of the backend. For the policy that favors threads, Adam delays issuing nested requests unnecessarily and is therefore wasting backend computation time. In the worst case, only the first thread of each group has a request to process. Thus, it takes $k$ time units to release $k$ nested requests, where $x$ is the amount of time each request takes on the frontend. Consider the same $k$ requests distributed with preference to groups over threads. In this case, each $x$ time units, Adam is able to release at most $n$ nested requests. Thus, the time to release all nested requests in this case is $\lceil \frac{k}{n} \rceil$. Optimally we would like to release as many nested requests as fast as possible since we assume that the backend service is multithreaded, batching, or has some other optimization which is optimized for large workloads. Thus, we would expect that a policy which favors groups to be more optimal than a policy that favors threads.
5.3.2 Results

For our evaluation, we have both the middle Adam group and the backend Eve group perform the same amount of work. For this experiment, we are using eight threads and three pipeline groups. To exacerbate the scenarios in which these two policies differ, we artificially look for a batch size that is a multiple of 12 but not 24. With this ideal batch size, the difference between partitioning policies is maximized. As expected, the policy that favors groups performs better than the policy that favors threads. Since the backend system is multithreaded and is batching 8 requests at a time, the policy that favors threads on average releases less than 8 requests to the backend per slice. This under-saturates the backend and hurts overall performance. Instead, the policy that partitions a batch with respect to groups releases 8 requests and saturates the backend on every slice except the last.

5.4 Mixing

Recall that both Eve and Adam use a deterministic mixer to reduce the possibility of divergence within a parallel batch. This mixer uses application specific knowledge to identify possibly conflicting requests and separate those out into different batches.

As stated in Chapter 4, pipelining achieves its theoretically maximum speedup as the number of requests in the pipeline approach infinity. Since Adam uses batching, the number of requests that can be in the pipeline at a time are not only finite, but a small fraction of the number of requests
that the system processes in total. Thus, to achieve the best performance, batches must be as big as possible within these confines.

In the original presentation of Eve and Adam [6, 7], the mixer is run after a batch is created. A primary server gathers up requests into a suitable sized batch and sends that batch out to all replicas. Each correct replica then runs the deterministic mixer over the batch to find possibly conflicting requests. Because the mixer is deterministic, all replicas are guaranteed to find the same set of conflicts and as a result create the same set of batches.

This choice of mixer, while good at detecting potential conflicts, creates unintended consequences with regards to performance. There is another choice for when to run the mixer, before sending out batches to all replicas. We will consider the ramifications of each choice.

**Choice.** *When is the optimal time to run the mixer, before or after the batch is sent out to all replicas?*

### 5.4.1 Predictions

Suppose the mixer was run after the batch is sent out. Since all replicas eventually execute all requests in that batch before executing another batch, the system is live and every client is guaranteed service. While this property helps satisfy liveness, it can drastically hurt performance. When a potential conflict is detected in the batch, the two requests are separated and if needed a new batch is made for the conflicting request. Thus, if there is a conflict then all resulting batches, both the one sent from the primary and those created at the replicas are guaranteed to have fewer requests than desired. Furthermore, in the case of a few non-commutable requests of which there are more than one in a batch, the mixer reduces the size of the original batch and creates a series of small batches to hold these conflicting requests. The size of these small batches can severely impact performance since they most likely are not large enough to mask the overhead of batching or pipeline nor are they able to benefit from multithreading.

Suppose instead that mixing is done at the primary only, before batches are sent out to all replicas. In this case, the primary can wait and only send out batches of a desired size that have been cleared by the mixer. This approach eliminates the occurrences of these small batches from normal execution. With the average batch size much larger than what it would have been if the mixer was run after batch creation, the overall performance of the system should increase. Furthermore, since the mixer is run at the primary, it is theoretically possible to implement any mixer, not just a deterministic one. While this is an interesting side effect, it will not be explored.

This choice raises two potential problems, first the liveness property for the system has to be reproved, and second the mixer can now be run multiple times on one batch. With this mixing
Figure 5.3: Performance of Adam based on running the mixer before and after batch creation.

To verify that mixing before batch creation is the most optimal choice, we implement both options and run the same workload on both implementations. Every test attempts to collect a batch size that is $\frac{3}{8}$ of the number of clients. As figure 5.3 shows, mixing before batch creation improves overall performance. This can be attributed to the distribution of executed batch sizes by each policy. Figure 5.4 shows the distributions of batch sizes with 512 clients. Each client is picking a random key from a key space of 1000 values thus the probability that no two requests modify the same key in a batch of size 192 is about 1 in 10 million. Thus, by leveraging the intelligence of the mixer at batch creation time, the system is not forced to make a series of small batches due to collisions. The leader is able to preemptively avoid putting conflicting requests in the same batch.
Figure 5.4: Distribution of batch sizes with 512 clients and a preferred batch size of 192. This was gotten from the experiment in figure 5.3.

and instead wait a little longer for a larger batch of non-conflicting requests.

5.5 Overall Results

The choices above represent just some of the most interesting and important considerations made while implementing multithreaded pipelining. With the methodology laid out in Chapter 1 our implementation of multithreaded pipelining is able to achieve a 13x improvement in throughput with 8 threads over single threaded performance as showed in figure 5.5.

Furthermore, we are able to demonstrate that our predictions for each separate optimization are correct. Sequential pipelining yields almost exactly a 2x performance gain over the naïve approach. Multithreading approximately yields a 7x improvement over sequential execution. We expected to see an 8x improvement over sequential in the best case, however in practice there are a couple of reasons to expect slightly less than linear speedup. Recall that in order to be able to rollback to a partial execution of any request, execution threads each store a continuation before verification is performed. Therefore, amount of data that is managed by the merkle tree grows linearly with the number of execution threads. For this reason, we expect some performance degradation as the number of execution threads in the system increase.

For multithreaded pipelining, we expect a 16x increase over the naïve sequential performance. Instead, our improvement looks more like a 2x increase over multithreading. While the amount of
metadata kept in the merkle tree is low, it does grow linearly with the number of threads. Note that in multithreaded pipelining, we create $kn$ threads. Since there are three pipeline groups in this test, this overhead is magnified. Instead, if we remove the overhead of the merkle tree and put in place a "dummy" merkle tree that neither stores any information nor performs any computation, we see that we get the $2^n x$ speedup that we predicted shown in 5.6. We can see that without a merkle tree, we can achieve the theoretical maximum speedup over sequential non-pipelined performance.

Overall, this optimization allows the throughput of Adam to scale near linearly with the number of threads. Figure 5.7 shows this fact for 10ms requests. Unfortunately, we are not able to get the same speedup with smaller requests. We suspect that with smaller requests, overheads created by continuations, the merkle tree, and even network latencies become an overwhelming factor. We know that batch sizing and collection timings play a large role in performance. Most of the final days leading up to this thesis were spent discovering these parameters for 10ms requests. At such small scales where Adam and network overheads are the same order of magnitude as the requests themselves, some assumptions previously made to derive these optimal configuration parameters are broken. We plan to extend this tuning work and use the methodology to find the optimal parameters for these smaller sized requests in the days to come.
Figure 5.6: Merkle tree overhead as threads increase.

Figure 5.7: Multithreaded pipelining performance improvement for various sized requests.
Chapter 6

Related Work

In this chapter we discuss some work that is related to Adam. Remus is an active-passive replication framework that replicates at the virtual machine (VM) level by using speculative execution and pipelining. Zyzzyva is a replication framework that uses speculative execution to optimize failure-free execution for Byzantine fault tolerance.

6.1 Remus

Remus[15] is a system which implements Active-Passive replication at a VM level. A Remus group contains one primary and at least one backup. The primary speculatively executes for one time quantum. During this quantum, outgoing network messages are not sent out, but instead pooled at the VM layer. When a time quantum finishes, Remus takes a checkpoint of the system from the VM and begins to transfer the checkpoint to the backup. Once the backup acknowledges the checkpoint transfer, previously batched messages are released since it is assured that some other replica knows about the internal state that created these requests. If the primary crashes, the backup can start execution from last checkpoint and take over as the primary.

Replication at the VM layer has many advantages. It allows unmodified application code to be replicated. Since state is replicated at the VM layer, multithreaded applications can be replicated without modification as well. Furthermore, since network messages are batched until a successful checkpoint transfer, services can safely communicate while processing a request. The optimizations made in Adam can be made in Remus with no modification to the underlying Remus framework.

Unfortunately, Remus does not attain the performance of Execute-Verify systems[6]. The replication granularity is too coarse for many applications. Remus assures that the entire operating sys-

29
tem is replicated. This includes application data that does not need to be replicated for correctness as well as any other process on the system. The overhead involved in replicating these parts of the replica is unnecessary. Furthermore, since Remus implements an active-passive replication model, it is only able to tolerate fail-stop crashes. In contrast, Adam tolerates a configurable amount of omission and commission failures by having every replica execute each request and performing verification on the output of these requests. Lastly, time quanta in Remus are too large for even the largest requests in Adam. The shortest time quantum in Remus is 25ms[15]. In Adam, the largest requests takes 20ms to execute across both Adam and the backend. In practice, most requests would take much less time to process than even this request, thus Remus would require a large amount of requests in the pipeline to mask the message pooling and backend computation.

6.2 Zyzzyva

Speculative execution is not a novel idea in state machine replication. Zyzzyva[12] uses speculative execution to perform efficient Byzantine fault tolerance. Replicas speculatively execute and reply to client requests after they are broadcasted by the primary. Each reply contains the response to the client request as well as the sequence of all executed requests. Correct clients then try to find different quorums of replicas who agree on response and history. Depending on the size of these quorums, clients inform the group to continue, accept a new request history, or re-order the request. Like Eve and Adam, Zyzzyva uses speculation to optimize for the failure-free case of execution.

Despite the speculative approach, Zyzzyva still imposes requirements on replicas that are stronger than those imposed by the replicated state machine specification. It speculates and verifies the history of requests instead of the application state, thus in some respects it resembles classic Agree-Execute architectures. Zyzzyva is able to only verify the sequence of requests because it requires them to execute sequentially. Instead, Eve and Adam speculate on the application state and only require that replicas converge on state not history. As a result, this prevents Zyzzyva from naturally utilizing multicore performance.
Chapter 7

Conclusion

The main goal of this thesis is twofold. First, on the practical side, the goal of this thesis is to make Adam as performant as possible, particularly, by integrating the new multithreaded pipeline mode of execution. Reasoning about performance and this optimization’s interactions both within itself and to the outside world is no small task. The second goal of this thesis is to aid this process and set a precedence for future work of this kind. While often skipped over, the exact details on how to implement a system like Adam can often make or break the system with regards to performance. We therefore outline a principled approach to reasoning about and deciding on these exact details. Using this approach, we have effectively navigate our way through implementing and configuring the Adam system. We are able to achieve the theoretical maximum speed up for pipelining while still maintaining multithreaded performance.

In Chapter 1 we outline a rigorous approach to making implementation and configuration decisions in a large and unwieldy system. In preparation to apply this approach, Chapters 2 and 3 provide insight to the problem of state machine replication and the approach Adam uses to solve this. Adam breaks the client-server paradigm that previous replicated state machines assume. It is thought of not solely as a server, but as both a client and server simultaneously. Because of this, we are able to add an optimization unique to its position, pipelining. Chapter 4 discusses this optimization and provides bounds for the maximum benefit that the optimization can achieve. The approach laid out in Chapter 1 is instantiated in Chapter 5. Chapter 5 contains just a small, but important, selection of applications of the approach to some implementation and configuration choices made in Adam. Chapter 5 also contains overall performance characteristics for multithreaded pipelining in Adam. Most importantly it shows that for large requests, the theoretical maximum throughput is achievable. Finally 6 discusses work which is related to Adam, outlining some of their advantages and limitations in relation to Adam.
Adam and its predecessor Eve represent a paradigm shift in how we implement replicated services in today’s computing environment. This is a great moment to demonstrate that no matter how novel or unique a system can be, rigorous attention to detail is a necessity to develop a performant system. While we demonstrate some initial success, there is still much more work to be done. We plan to continue to apply this approach to Adam to extend the performance gains to smaller sized requests. More importantly, this approach is not unique to Adam. There are plenty of opportunities to achieve the same success seen here. We hope that in the future this is just one in a long line of success stories using this or a similar approach.
Bibliography


