Leveraging External Synchrony for Distributed Databases in Functions as a Service Workflows

Austin Li

Adviser: Professor Amit Levy

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Computer Science

Princeton University

April 2023

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

_____

Austin Li

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

Austin Li

Abstract

Functions-as-a-Service (FaaS) is a rapidly growing cloud computing paradigm. It promises efficient utilization of resources and removes the need for developers to manage their own infrastructure. FaaS workflows must use globally distributed databases to store data persistently across the workflow. Furthermore, to guarantee the consistency of data in the workflow, FaaS workflows must block for globally distributed database writes to become durable. This substantially increases the end-to-end latency for FaaS workflows. This paper takes a novel approach to addressing this issue by leveraging the concept of external synchrony. The external synchrony protocol is causally consistent and guarantees that the workflow computation will be indistinguishable from the baseline synchronous procedure to an external observer of the FaaS workflow, yet the external synchrony protocol also significantly improves performance. This paper finds that the external synchrony protocol empirically improves performance in synthetic single function microbenchmarks and also provides a 21.03% improvement in a real FaaS workflow. These results demonstrate that external synchrony has the potential to significantly reduce FaaS latency and thus provide tangible benefits for developers and users of FaaS.

Acknowledgments

I would like to thank my adviser, Professor Amit Levy, for his help with this project. This thesis would not have been possible without his input.

Many thanks to all of my friends for being there for me throughout this year. Thank you for the impromptu feedback and constant moral support. You have made this year fly by, and I could not have imagined it without you all.

Lastly, I would be remiss in not thanking my family for supporting me throughout this journey. Their unwavering belief in me has helped keep my spirits and motivation high during this process. Mom and Dad, I am forever grateful for your continued love and guidance in all that I do.

Contents

Chapter 1

Introduction

The expansion of cloud computing has provided access to powerful computer system resources that were previously inaccessible to most programmers. However, with this expansion of resources also comes an increase in complexity. Building a cloud application can require the involved process of creating and maintaining the infrastructure to support the application. This complexity can be a barrier for many developers looking to leverage cloud computing. Functions-as-a-Service (FaaS) is an increasingly popular cloud computing paradigm that addresses these issues. FaaS is designed to execute code in response to events. FaaS platforms, such as Amazon's AWS Lambda [2], Google Cloud Functions [7], Azure Functions [4], and OpenWhisk [1], provide users with the ability to develop, run, and manage application packages as modular executable functions in the cloud without the need to manage their own infrastructure. The FaaS platform will take care of the provisioning and management of cloud servers.

FaaS allows developers to focus on their application code and not have to worry about managing infrastructure. This can result in increased productivity and faster development times. In addition, the modular nature of FaaS allows for developers to build applications with an internal diversity of software stacks and hardware configurations. For example, as shown in Figure 1.1, a cloud photo sharing application

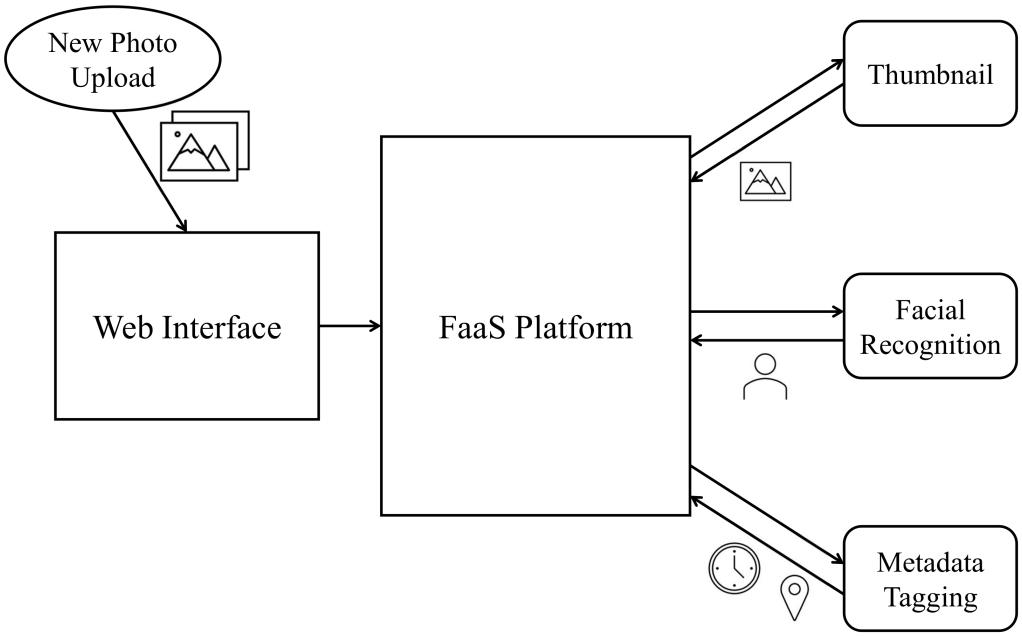

Figure 1.1: Example photo sharing application built using a FaaS workflow. The platform responds to a new photo upload and proceeds to invoke the thumbnail, facial recognition, and metadata tagging functions in the FaaS workflow.

could reasonably involve several components. First, it might have a thumbnail generation function, which could use a JavaScript library that requires few CPU's and memory [11]. Next, the application might also have a facial recognition function that utilizes Python's machine learning libraries and thus requires larger computational resources and GPU's. Finally, the application might have a metadata tagging function (time, location, etc.) that is very computationally light. This cloud photo sharing application can naturally be built as a FaaS workflow.

FaaS workflows can also enable better overall resource utilization. Functions only need to consume computational or hardware resources when they are actually servicing requests. Once the function is no longer needed in the workflow, the system can safely reclaim the resources allocated to the function. On the other hand, the system can quickly create new function instances to provide scalability when there is higher demand. This increased resource efficiency is another benefit of the modular nature of FaaS workflows. This translates to FaaS being priced in a granular fash-

ion and proportional to the resources utilized. Developers are thus only charged for their workflow's resource usage, unlike in traditional cloud applications where they are charged for all of the resources they reserve.

Applications designed using FaaS are likely to require robust data storage and access. With the potential volume of data that needs to be stored for a modern cloud application, a single computer is nowhere near enough. There is a clear need for scalability in the design of databases for such applications. Furthermore, even the best modern computers are not invulnerable to failure. From a single computer shutdown to a power outage in the entire datacenter, computer failure is an inevitable part of data storage. Despite this, users expect their data to be fault tolerant and durable. This demonstrates a clear need for data replication in the design of databases. These considerations are not unique to FaaS and have been present since applications started utilizing an ever-increasing amount of data. The solution has been to use a distributed database.

A distributed database is a database in which the data is stored across different physical locations. Modern distributed databases offer the capability to store data at scale, provide consistency guarantees, and be fault-tolerant to server failures. One of the primary tradeoffs that distributed databases make is sacrificing latency for consistency. User writes to the database need to be globally exposed in order to provide data durability and consistency. However, globally replicating changes to all computers in the distributed database is a relatively expensive operation. The roundtrip latency for Amazon Workspace network connections from Princeton vary from 27ms for Northern Virginia, 67ms for Oregon, and around 200ms for connections in Asia [8]. AWS S3 suggests users can expect latencies of around 100-200ms [5]. MongoDB average latency has been measured to range from less than 10ms to up to 150ms depending on the number of concurrent threads using the database [31]. These latencies accumulate as the number of operations increases and can result in

noticeable delays for users.

Currently, in the context of FaaS, workflows can choose to perform database writes either synchronously or asynchronously. Asynchronous writes would mean sending the write operation to the globally distributed database but not waiting for it to finish before continuing on in the workflow. This has the benefit of avoiding the latency associated with a globally distributed database. However, often times, asynchronous writes are not a viable option for workflows that require data consistency throughout the workflow. Instead, such FaaS workflows must synchronously wait for the database to globally replicate every write to ensure the consistency and durability of the data throughout the workflow. This workflow perspective of choosing to synchronously perform all writes to the globally distributed database results in increased latency for FaaS workflows.

This paper addresses the latency issue created by the current synchronous protocol for FaaS workflows by leveraging external synchrony, an innovative data synchronization protocol originally developed in the context of operating systems [26]. External synchrony shifts the relevance of data durability from a workflow perspective to a user perspective, making improved performance possible. Under a workflow perspective, as described previously, data writes need to be synchronous to the globally distributed database at every point in the workflow. Under a user perspective, data need not be durable until it is exposed to the user. External synchrony exploits this gap by not syncing data updates to the globally distributed database until it is externalized to users. The user believes that they are receiving the same guarantees from the system as before. Meanwhile, the system can take advantage of the increased flexibility and provide greater performance. Thus, this paper introduces an external synchrony protocol in the context of FaaS workflows to reduce the latency of such systems.

Chapter 2

Background

2.1 FaaS Platforms

FaaS platforms use granular sandboxes, typically virtual machines or OS containers, to run functions in response to events. Developers simply write the code they want and the FaaS platform takes care of the infrastructure needed to run it. Developers can also specify the resources they want their functions to have access to. Once a request for a function comes, the platform will provision a new virtual machine with the necessary runtime environment, computational resources, and temporary storage. Upon completion, the platform may shut down the virtual machine instance or it may reuse the instance for a subsequent request. Developers must not assume that the function has any affinity for the underlying computing infrastructure. Additionally, functions are currently designed to be stateless because this allows for the platform to easily invoke other functions and reduce complexity. However, this means that any data that the developer wishes to be persistent must be written to a distributed database [22]. Functions have access to network connection and can thus utilize a globally distributed database.

2.2 Distributed Database Consistency Models

Within a distributed database, the data is stored across different physical locations. In addition, the data is sharded (divided) and replicated across computers. Sharding provides scalability and replication provides durability to computer failure. This results in an added layer of complexity when users attempt to read or write data from the distributed database. The distributed database must provide a consistency model so users know what guarantees can be expected of the system. There are many consistency models in the literature but for the sake of this paper, three examples will be presented. In all of the following figures, the notation for writing a key-value pair will be "Write(Key, Value)" and the notation for reading a key and obtaining a value will be "Read(Key) = Value". The line underneath each operation indicates the time the operation started to the time the operation completely finished.

Definition 2.2.1. Linearizability [21]
The distributed database guarantees that all replicas execute operations in some total order. This total order preserves the real time ordering between operations. That is, if operation A completes before operation B begins, then A is ordered before B in real time. If neither A nor B completes before the other begins, then there is no real time order between the two. However, there must still be some total ordering of the two events.

Linearizability is desirable as it makes the distributed database appear as if it were a single database. For example, as shown in Figure 2.1, if one client were to write a key-value pair ("X", 1) into the database, and then later, with no intermediate changes, another client reads from the key "X", the value will exactly be 1. However, to provide such a guarantee, the distributed database will give up some availability in order to replicate the data appropriately. This way, future reads cannot possibly access a stale value. In the example, this translates to the write taking additional la-

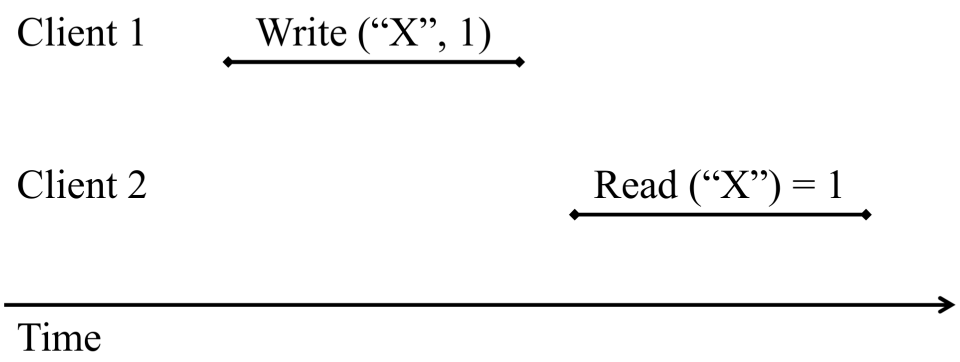

Figure 2.1: Example of a linearizable sequence of events.

tency because the data is being replicated. Popular databases such as Google Spanner provide linearizability [17].

Definition 2.2.2. Lamport happens before [23]

For two events A and B, we say that A happens before B in the following cases. First, if A and B are in the same process and A occurs before B, then we say A happens before B. Second, if A and B are in different processes but B receives a message from A, then we also say A happens before B. Finally, this happens before relation is transitive (i.e., if A happens before B and B happens before C, then A happens before C). All events not related by happens before are considered to be concurrent.

Definition 2.2.3. Causal consistency [13]

There is not necessarily a total ordering. However, for each process, there exists an order of all of the writes and that process's reads such that the order respects the Lamport happens before relation.

Causal consistency no longer guarantees that the distributed database will appear as if it were a single database. Consider the following example shown in Figure 2.2. Suppose there are two clients reading and writing to the database. Client one writes the key-value pair (“X”, 1) and then reads the key “Y” as 0. Client two writes the key-value pair (“Y”, 1) and then reads the key “X” as 0. There is not a total ordering

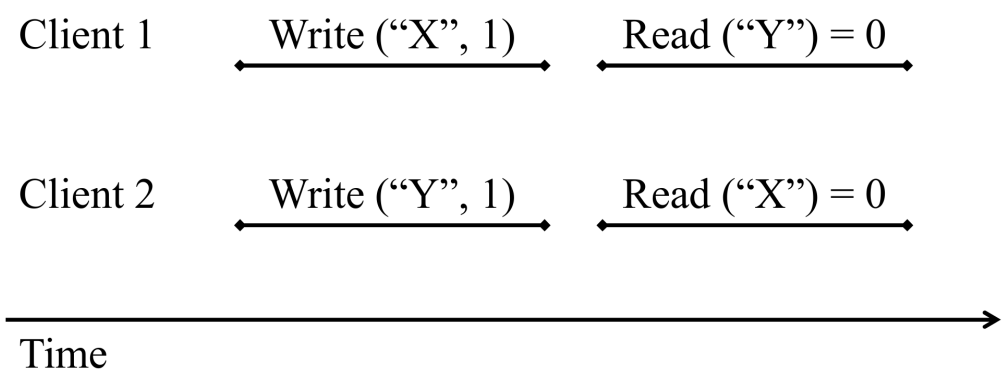

Figure 2.2: Example of a sequence of events that has causal consistency.

of the events in the two clients. However, for each client, it could see an ordering that respects the happens before relation. For client one, it could see its write of (“X”, 1), followed by the read of “Y” as 0, and then the write of (“Y”, 1). For client two, it could see its write of (“Y”, 1), followed by the read of “X” as 0, and then the write of (“X”, 1). Causal consistency is weaker than linearizability. Popular databases such as MongoDB provide causal consistency [6].

Definition 2.2.4. Eventual consistency [32]
The distributed database guarantees that if there are no more writes, all replicas will eventually agree.

Eventual consistency provides minimal guarantees to the user. Consider the following example shown in Figure 2.3. One client writes a key-value pair (“X”, 1) into the database and then writes a key-value pair (“Y”, 1). Immediately after, another client reads from the key “Y” and sees 1. The second client then proceeds to read the key “X” and obtains 0. By Lamport happens before, the write of (“X”, 1) must happen before the write of (“Y”, 1), which in turn must happen before the read of “Y” since the read return value of 1 matches the write value. However, due to the final read of “X” being inconsistent with the initial write, there is not an ordering of the events for client two that respects the happens before relation of the events.

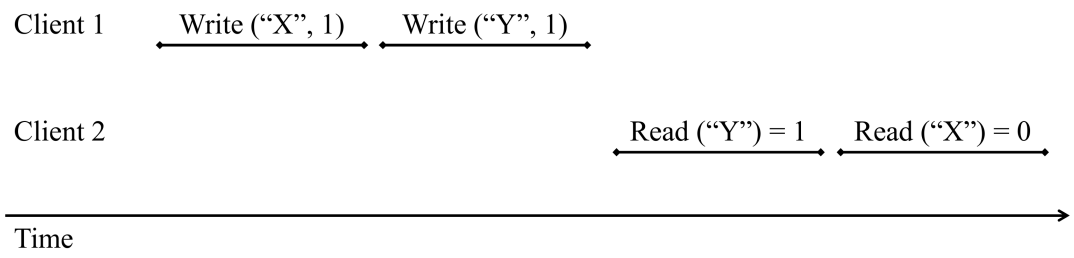

Figure 2.3: Example of a sequence of events that is eventually consistent only.

An eventually consistent database does not ensure that client two will read “X” to be 1; it could be some stale value for the key from before. The value of 1 is only guaranteed eventually. Eventual consistency is the weakest of the consistency models. The primary benefit of an eventually consistent database is that it is highly available. Because it does not have to provide the same guarantees as linearizability, an eventually consistent database can respond to requests much faster. Popular databases such as Amazon DynamoDB are eventually consistent only [18].

When programming in a FaaS workflow, developers must make a choice of what database they will use. Many applications will use a distributed database that guarantees linearizability. This is because developers often require the consistency of data accesses throughout their workflows. A common pattern in a FaaS workflow is for some function early in the workflow to write to the database, and then a later function in the workflow will read this value for its computation. Thus, linearizability is the only possible way for the database to guarantee that the workflow has the user's expected behavior.

2.3 Related Work

2.3.1 FaaS Research

Researchers have been increasingly interested in FaaS systems because it provides a new, easy-to-use abstraction for leveraging cloud computing. By saving developers the need to implement their own cloud server infrastructures, FaaS promises to become the most accessible and popular cloud computing paradigm. However, FaaS platforms still have many limitations today, thus motivating researchers to try and improve them.

One of the primary areas of FaaS research lies in making functions stateful in a more efficient manner. Distributed databases provide an easy way to make functions stateful and allow data transfer between functions. However, accessing a globally distributed database can be a costly operation. Thus, there is ongoing research into creating systems to enable stateful FaaS [14, 16, 22, 29]. Much of this research relies upon creating a new distributed solution for storing shared state and data. However, by working around distributed databases, these solutions are actually less accessible to developers who already rely upon established distributed databases. These efforts do not look to directly improve the performance of FaaS with a globally distributed database. There has also been prior research on making FaaS systems causally consistent [25, 33]. This work demonstrated the latency improvements that causal consistency can provide to FaaS workflows. However, external synchrony was not used and thus no guarantees regarding external observers was made.

Other FaaS research has also examined reducing the system overheads and cold-start time of FaaS platforms [22, 28, 30] or orchestrating workflows more effectively [3, 15, 24]. All of this previous work has aimed to make FaaS workflows more efficient. Despite this, the approach of leveraging the external synchrony protocol for FaaS functions and globally distributed databases has not been studied.

2.3.2 Rethink the Sync

The idea of external synchrony was originally developed for local file systems by Nightingale et. al. in their seminal paper "Rethink the Sync" [26]. For local file systems, data is made durable when it is written to disk. However, writing data to disk is an expensive operation compared to saving the data in memory or volatile cache. Synchronous I/O would have the file system write to memory and then also commit to disk immediately. This process will block the application from continuing. The primary tradeoff that synchronous I/O makes is that it provides data reliability but sacrifices performance. On the other hand, asynchronous I/O would write to memory but not commit to disk immediately. Instead, the OS will wait for some later point in time to persist the write to disk. This provides performance but sacrifices data reliability.

Nightingale et. al. created a new model of external synchrony that provides a similar set of guarantees to the user as synchronous I/O but does so with greater performance. Nightingale et. al. pointed out that synchronous I/O takes an application-centric view of the computer system. By committing to disk immediately, the system provides a set of strong guarantees for applications. However, they argued that users should be the focus of the system, not applications. In particular, users can only observe application state when data is sent to external devices. Thus, external synchrony relies upon only writing data to disk when its effects would be externalized to a user of the system. Any external output (i.e., printing to the console, passing data over a network, etc.) will trigger writing to disk. The OS will block the externalized data from being viewed until it is committed to disk. This results in the user being unable to tell the difference in output between external synchrony and a strictly synchronous system. Furthermore, the performance of an externally synchronous file system closely approximates that of an asynchronous file system.

The idea of external synchrony has clear adaptability for the context of FaaS work-

flows and distributed databases. Moreover, the significant performance improvements for local file systems suggest that similarly substantial gains could be made for FaaS workflows.

Chapter 3

Design

3.1 System Overview

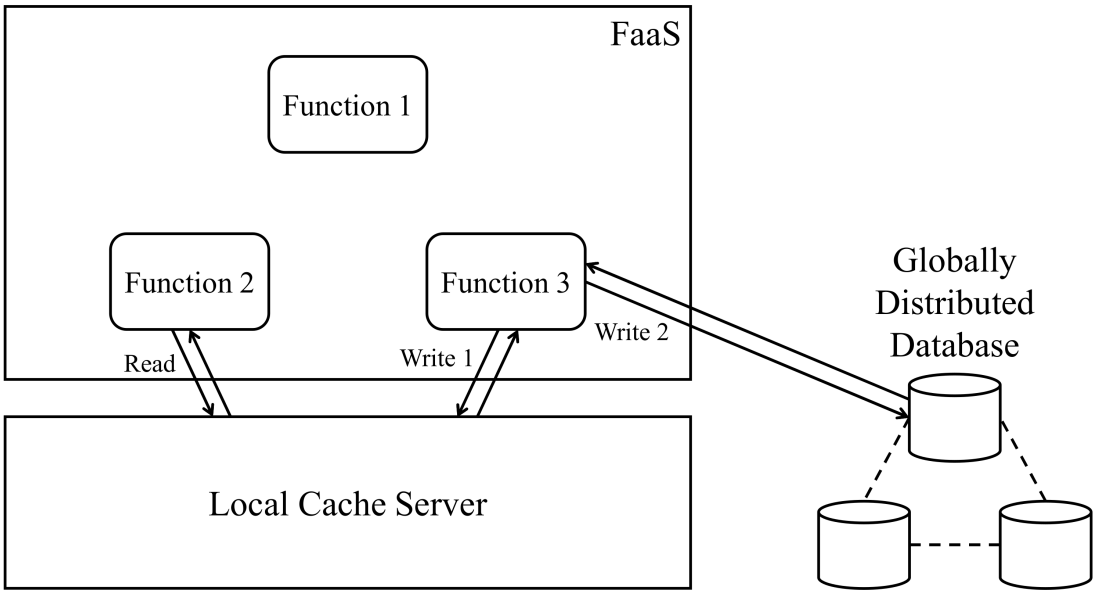

Figure 3.1: Overview of FaaS system with databases. All functions have access to a shared local cache and a globally distributed database. Write 1 to the local cache server is always synchronous. Write 2 would be synchronous in the baseline procedure but asynchronous in the external synchrony protocol.

The key idea of this project was to leverage external synchrony. As shown in Fig-

ure 3.1, the functions in the FaaS workflow all have access to a single local/nearby cache server and the linearizable globally distributed database. A baseline procedure would be to have all writes sent to both the local cache server and to the distributed database, synchronously waiting for the distributed database to complete each write. This would block the workflow from continuing until the globally distributed database finishes the write. All reads would be from the local cache server. The external synchrony protocol maintains synchronous writes to the local cache server but does not require that the writes to the globally distributed database be synchronous. However, the writes to the globally distributed database must be in the same order as to the local cache. Additionally, upon externalization, the system will ensure that the writes to the globally distributed database have completed before returning to the user.

3.2 External Synchrony Consistency Model

The baseline procedure of having all writes be synchronous to both the local cache server and to the distributed database is linearizable. Because the local cache is shared by all functions, and is synchronous with the distributed database, it is guaranteed that there will be a total order of reads/writes that respects the real time ordering between operations.

The external synchrony procedure will not provide linearizability from the workflow's perspective as writes to the globally distributed database are asynchronous. However, since all writes to the local cache server are synchronous and reads are from the local cache, which is shared by all the functions, the operations to the local cache will be linearizable. Although the globally distributed database is not synchronized with the local cache, as long as workflow reads are from the local cache server and not the globally distributed database, this will not be an issue in terms of the workflow's computations. Furthermore, because the asynchronous writes to the globally

distributed database respect the ordering of the writes within a function, the state of the globally distributed database will be causally consistent. Upon user externalization, external synchrony guarantees that all asynchronous writes to the globally distributed database will be complete. Thus, when data is externalized, the state of the globally distributed database will be causally consistent. Workflow computations after externalization will also be correct. If each function writes to distinct keys, then this causal consistency will become linearizability. Additionally, if externalization happens at the end of every function, and all functions occur in a total order, then this will also give linearizability.

3.3 Externalization

One of the key components of "Rethink the Sync" lies in its ability to define what externalization of data means in a user-centric view of the operating system [26]. Doing so allowed for the greater performance of the external synchrony protocol. Because this project aims to leverage external synchrony, this concept of externalization must be adapted for the context of FaaS workflows.

Under the baseline procedure of having all writes be synchronous to the globally distributed database, FaaS workflows are implying that every data write ought to have its effects immediately observable to the user of the workflow. If the user were monitoring the workflow, then under the baseline procedure, they can query the globally distributed database at any time and see the progress the workflow is making. However, this is often unnecessary. One of the most common use cases of FaaS is in response to events that can happen at any moment. The user of the workflow simply does not have the time nor the need to monitor the workflow outputs at every moment. Thus, outputs of the FaaS workflow do not need to be synchronous at all times. Instead, the globally distributed database only needs to be correctly synced

when the user requires it. Thus, externalization can be defined abstractly to be whenever an entity outside of the workflow needs to view the state of the globally distributed database. Let externalize mean the process in which the protocol ensures that the globally distributed database state is synced with the local cache state.

It is reasonable to assume that externalization always happens at the end of a FaaS workflow. The user should expect the state of the globally distributed database to be finalized by the time the workflow is complete. Thus, the protocol must have all data externalized by the end of the workflow. Although this is the perhaps the most basic and common case of externalization, it is also possible that the globally distributed database needs to be externally accessed in some intermediate stage of the workflow. This means that externalization could also happen in the middle of a function, but only as a response to a request for viewing the globally distributed database. One such situation where this could arise is if there are two workflows which are dependent on each other. If workflow A relies on a particular intermediate function of workflow B having put some data in the globally distributed database, then B must externalize that data before moving on in the workflow. Otherwise, there is no guarantee of the data remaining constant later on in workflow B. Therefore, the external synchrony protocol for FaaS workflows must be flexible in terms of when to externalize data.

3.4 External Synchrony Protocol

The next step is to formally define the external synchrony protocol. All writes need to be synchronous to the local cache server. However, writes can be asynchronous to the globally distributed database. The asynchronous writes to the globally distributed database must respect the ordering of writes within the function. That is, although the writes to the globally distributed database are asynchronous, the order in which

the writes occur must be the same as what was in the function. This guarantees the correctness of the writes in the globally distributed database as they will respect the causal order of the function.

Anytime output is required to be externalized, all data written in the local cache server must also be in the globally distributed database. The protocol will block the execution of the workflow until all asynchronous writes to the globally distributed database that occurred before externalization was required are complete. Thus, after externalization all data is written in both the local cache server and the globally distributed database. All reads are still from the local cache.

3.5 Theoretical Performance Improvements

The runtime of a function is dependent on both the actual time complexity of the function execution and the time it takes to make writes to the globally distributed database. Call the total time a function takes in waiting for writes to the globally distributed database to complete to be the write time, W, of the function. Call the total time a function takes in doing its computations to be the execution time, E, of the function. From a theoretical standpoint, the baseline procedure of always synchronously writing to the globally distributed database results in a runtime that is linear in the execution time and write time, $O(E + W)$. On the other hand, with external synchrony and externalization at the end of the function only, the runtime is linear to the maximum of execution time and write time, $O(\max(E, W))$. This is because external synchrony essentially parallelizes the function execution with the writes to the globally distributed database.

Thus, the external synchrony protocol will be expected to give the most improvement in workflows that take a similar amount of time in executing the functions and writing to the globally distributed database. Such workflows will best be able to

take advantage of the parallelization of function execution and writes to the globally distributed database. For workflows in which either execution time dominates write time, or vice versa, the improvements of external synchrony will be marginal. When execution time dominates write time, external synchrony will only be able to save a small amount of time relative to the baseline in writing to the globally distributed database. When write time dominates execution time, external synchrony is similar to the baseline in that the majority of the time will be spent waiting on writes to the globally distributed database to complete. Figure 3.2 shows the theoretical expected performance improvement of the external synchrony protocol when varying the function execution time. As discussed above, the improvement is maximal when the execution time is the same as the write time. When the execution time is much less or much more than the write time, the performance improvement falls off.

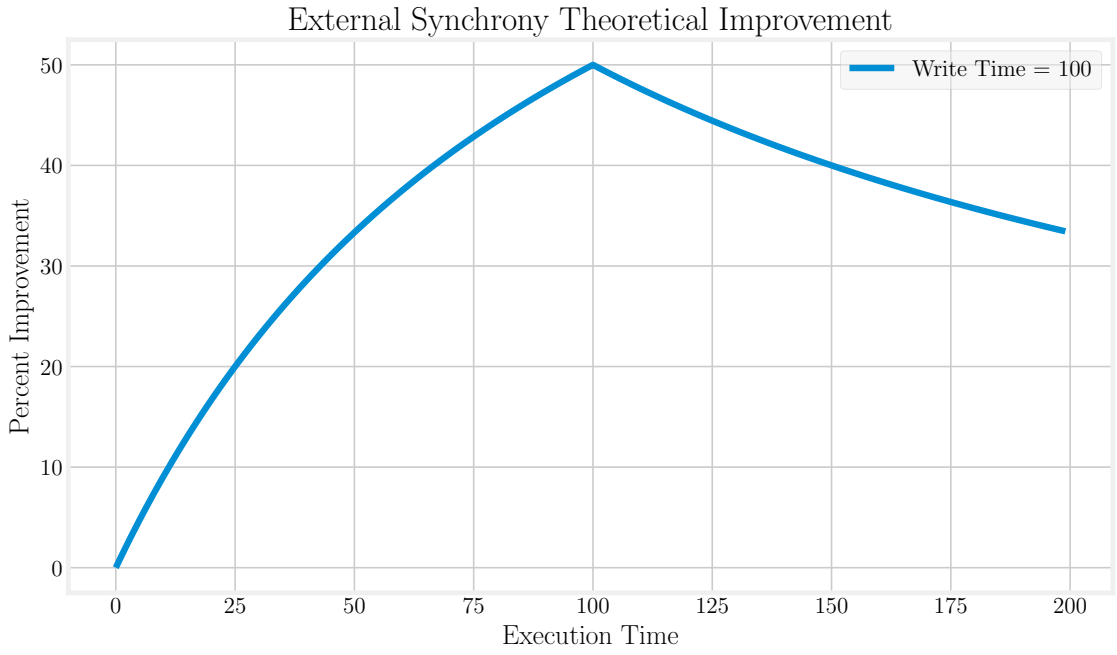

Figure 3.2: Theoretical performance improvement of the external synchrony protocol within a single function. The write time is fixed as 100 and the execution time is varied from 0 to 200.

Chapter 4

Implementation

4.1 FaaS Architecture

The FaaS architecture is implemented in Rust and is based off of the Faasten project [9]. The Faasten project provides a FaaS prototype with the capability to dynamically allocate VM's to service functions. Functions are coded in Python and can interact with the system via predefined protobuf-based RPC's. Additionally, Faasten provides a file system built on top of a key-value store. The primary innovation of the Faasten project was to incorporate security policies into the FaaS file system; however, for this project, the security policies were not the focus and thus not utilized.

4.2 Distributed Database

No production level distributed databases were used. The local cache key-value store was built using Lightning Memory-Mapped Database Manager (LMDB) [10]. LMDB is a simple database that provides full transactional semantics. In using LMDB, the local cache server would thus be on the same computer that the FaaS platform was running on. To simulate the fact that even a local cache server for a FaaS platform must be accessed via network, the LMDB database could only be accessed

via network as well. The LMDB database had a server that would listen for protobuf based RPC's. Each VM allocated by the FaaS architecture had a database client that would send these requests through TCP. Buffering of requests was turned off for TCP. No sharding or replication was implemented for the local cache server.

The globally distributed database was implemented using TiKV [12]. TiKV was chosen because it is used in production by a number of companies including Dailymotion (video sharing platform), JD Cloud (cloud computing branch of Chinese e-commerce giant Jingdong, or JD), and Bank of China. Additionally, TiKV provides a development stage Rust client with a simple key-value API with full transactional semantics, making it easy to integrate into this project. A local TiKV cluster can be created and subsequently accessed through the network. In order to simulate the latency of a globally distributed database, the TiKV cluster was placed on a computer in a geographically distinct region from the FaaS platform. The network latency provides a fairly reasonable approximation of the latency users would experience in utilizing a globally distributed database.

4.3 External Synchrony

To implement the external synchrony protocol, each database client would run a background thread for sending requests to the globally distributed database. For any write operation, the database client would first synchronously write to the local cache database, and then asynchronously use its background thread to send the request to the globally distributed database. The client would use a channel to pass the request to the background thread. The background thread continuously listens on its channel for requests from the client, processing them each in turn and making writes to the globally distributed database. Because the database client had only a single background thread and used channels to pass the requests, the order of the requests

received by the client would be preserved in the background thread.

The database client also needed to be able to react to an externalization event. For flexibility, an externalization event would be triggered simply by a request to write to the key "EXTERNALIZE". This way, a function can easily simulate an externalization event simply by making this write request. Upon receiving the externalization event, the database client would synchronously send this request, along with a notification channel, to the background thread. Once the background thread processed the request, it would notify that it had received it through the channel and then unblock the database client. This implementation guarantees that all requests before the externalization event have been written to the globally distributed database.

Chapter 5

Evaluation

5.1 Experimental Setup

All evaluation experiments were conducted using the CloudLab Server [19]. The FaaS platform was deployed using the CloudLab Massachusetts cluster. The local cache server was thus also on the Massachusetts cluster deployment. The TiKV server was deployed on the CloudLab Wisconsin cluster. Using a simple ping test, the roundtrip latency from Massachusetts to Wisconsin is on average 23.2ms. This setup mirrors what a production FaaS platform might use. The globally distributed database can be located far away from the computers used for the actual function execution.

To provide context for the results, some simple tests were conducted to measure the average latency of operations. The relevant operations were the time for an LMDB write operation and the time for a TiKV write operation. On the Massachusetts cluster, the average latency across 100 LMDB writes with no network communication involved was 0.105ms per write operation. In setting up the TiKV server on localhost on the Massachusetts cluster, the average latency across 100 writes to the TiKV server was 15.15ms per write. Because there is negligible network latency for using localhost, this value gives context for how long TiKV itself takes to complete its oper-

ations. Finally, the average latency across 100 writes from the Massachusetts cluster to the TiKV server in Wisconsin was 99.95ms per write operation. One thing to note is that the first write to the TiKV server takes much longer, around 178.66ms. Every write following the first one stays around 99-100ms per write operation. Although this greatly exceeds the roundtrip latency of the ping operation, it is actually somewhat reasonable as TiKV uses a distributed transaction protocol based on Google's Percolator [12, 27]. This involves a 2-phase commit algorithm of prewrites and then commits. For the commit stage, the protocol does a primary commit and then a secondary commit. In the case of writing a single key-value pair per transaction, the secondary commit requires communicating with the server but there will be nothing to do on the server's end. This totals 3 communications with the server per transaction. Assuming 15ms for the TiKV server to do the write, this adds up to $3(23)+15 = 84$ms of latency. Additionally, the TiKV Rust client is still in development so there could be further latency contributions there.

The metric examined in all evaluations was latency as this was the area in which external synchrony provided a theoretical performance improvement. The evaluation of the external synchrony protocol was measured relative to the baseline procedure. The protocol was tested in two contexts: a single function microbenchmark and a real serverless grading system used for two Princeton computer science courses. Each set of parameters in each experiment was tested for 30 trials. Because unexpected network delays occurred, the trials where this happened were dropped from the final results. The number of dropped trials per 30 was at most 3. The graphs presented all plot the average percent improvement with single standard deviation error bars.

5.2 Single Function Microbenchmark

First, the impact of the protocol needed to be measured at the most granular level of a single function. Because the protocol is not expected to give the same performance improvements in all contexts, this single function microbenchmark should evaluate the performance improvement of the protocol when varying the complexity of the function. A function can be thought of as some combination of database reads/writes and useful computations. Since the external synchrony protocol does not make any changes to the baseline read method, this evaluation focused on the writes. Thus, the function should be repeatedly writing data to the database and performing useful computation. In particular, every useful computation was performed in between writes to the database. For the sake of testing, the function's writes were to a unique key each time with random bytes as the value, and the function's useful computation was substituted with sleeping the system for a given amount of time. Externalization only happened at the very end of the function.

The two independent variables for the single function benchmark were how many iterations of writing and computation to do (call this the number of repetitions), and how long the computation should take (call this the interoperation delay time).

For each experiment, one of the independent variables was fixed while the other was varied. For the external synchrony protocol, the system will quickly return after completing the write to the local cache server. The time for the writes to propagate to the globally distributed database can be partially masked by the interoperation compute time. That is, while the function does its computation, the external synchrony protocol is propagating the previous writes to the globally distributed database. On the other hand, the baseline protocol writes to the globally distributed database synchronously and thus does not have any parallelization benefits.

For a fixed number of repetitions, as the interoperation compute time increases, there is more time for the writes propagating to the globally distributed database to

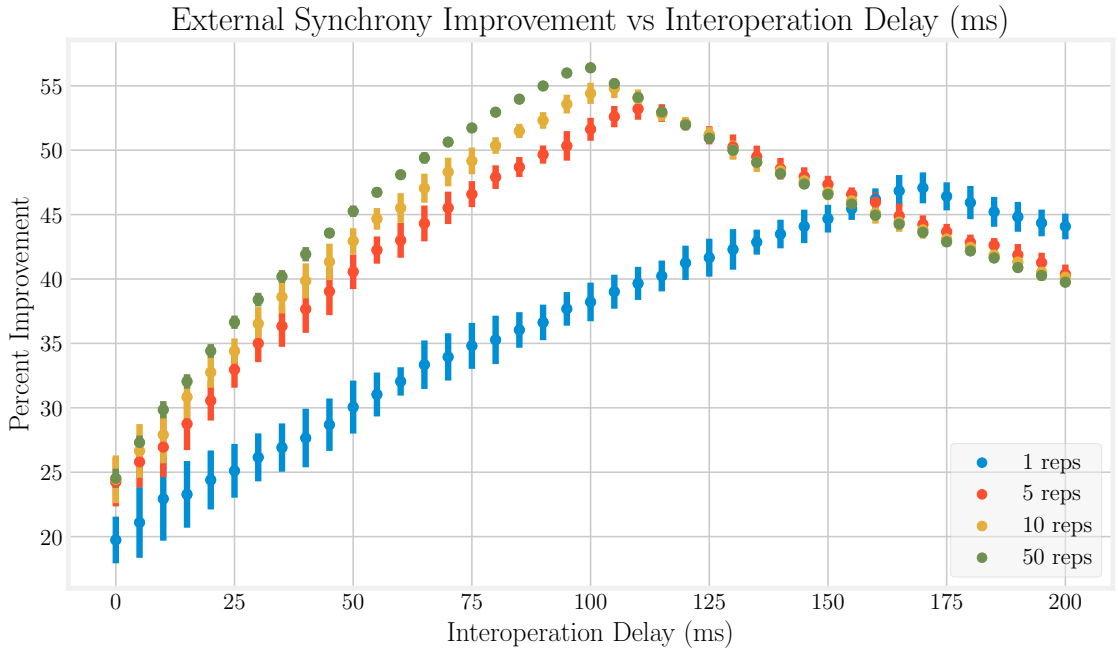

Figure 5.1: Single function microbenchmark varying the interoperation delay time for four fixed repetitions.

complete in the background. This should give an increasing amount of improvement over the baseline. However, once the interoperation compute time is too high, the writes will finish before the interoperation compute is done, meaning that in both the external synchrony and the baseline procedure there will be long periods of time when both are waiting on the interoperation compute to finish. Thus, the maximum performance improvement should occur when the time required to complete a write to the globally distributed database exactly equals the interoperation compute time.

In Figure 5.1, the percent performance improvement of the external synchrony protocol relative to the baseline procedure was measured against varying the interoperation delay time for four fixed repetitions. The interoperation delay time ranged from 0 to 200ms in steps of 5ms. The number of repetitions were chosen to be 1, 5, 10, and 50. The results match up with what was expected. As detailed in Section 5.1, the average latency for writes to the globally distributed database was 100ms. For 5, 10, and 50 repetitions, the peak in improvement is exactly around 100ms in inter-

operation compute time. For 1 repetition, the peak in improvement is around 170ms in interoperation compute time. Again, this matches the expectation since the first write to the TiKV server was profiled to take 178.66ms. In the case of a single repetition, this is also the only write to the globally distributed database. Thus, these results show that in all cases the peak in improvement is when the interoperation compute time matches the write time. Furthermore, the error bars generally become smaller as the number of repetitions increases. This is simply because as the number of repetitions increases, the overall time increases and thus random fluctuations have less proportional impact on the runtime. This translates to the percent improvement being more consistent between trials.

Next, experiments were conducted with a fixed interoperation compute time and a varying number of repetitions. For lower repetitions, the performance improvement of the external synchrony protocol should be mitigated by things such as the slow first write to the server or general system overheads. Additionally, at lower repetitions, external synchrony's cumulative effect of having background writes queued to the globally distributed database should be reduced. As the number of repetitions increases, the average write time to the globally distributed database normalizes to 100ms and the impact of things such as system overhead are relatively reduced. Once the number of repetitions reaches a certain point, the protocol's parallelization is the main contributing factor to the performance improvement. Because there is only one background thread completing writes to the globally distributed database, the performance improvement cannot scale indefinitely. Furthermore, the fixed interoperation delay being too low or too high will also dampen the performance improvement of the external synchrony protocol.

In Figure 5.2, the percent performance improvement of the external synchrony protocol relative to the baseline procedure was measured against varying the number of repetitions for four fixed interoperation delay times. The number of repetitions

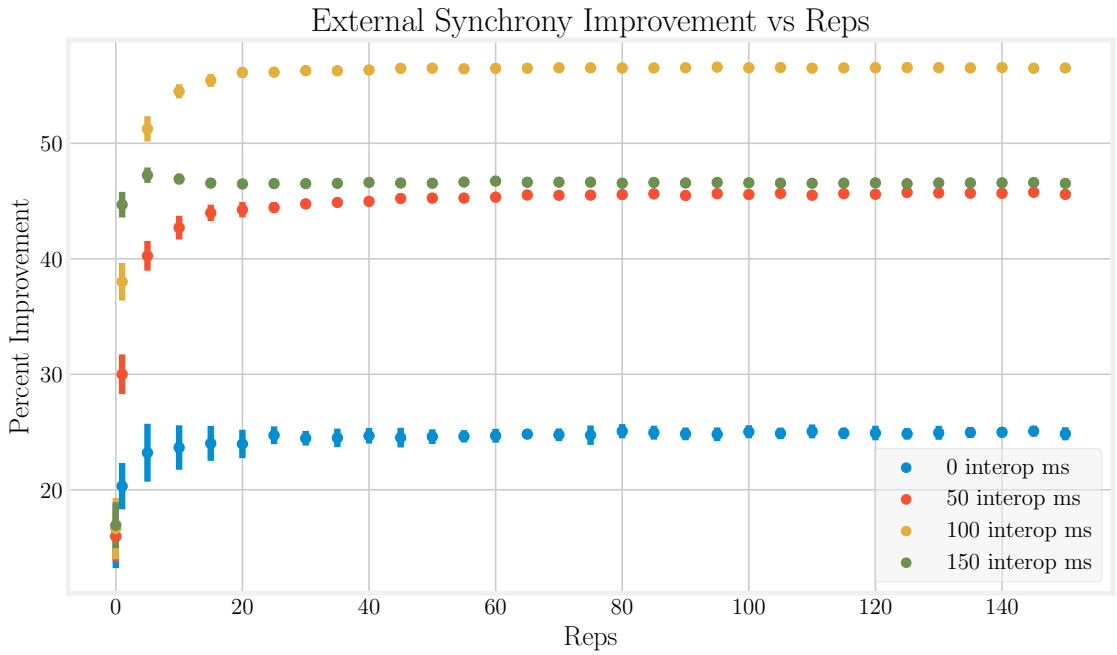

Figure 5.2: Single function microbenchmark varying the number of repetitions for four fixed interoperation delay times.

ranged from 0 to 150 in steps of 5, and also included 1. The interoperation delay times were chosen to be 0, 50, 100, and 150ms. The results match up with what was expected. For each fixed interoperation delay time, the performance improvement increased until around 20 repetitions, after which it leveled off. Comparing across interoperation delay times, the 100ms delay provided the most improvement, followed by the 50 and 150ms delays, and the 0ms delay was the lowest. The 100ms delay closely matches the 100ms write time to the TiKV server and thus should provide the most improvement. One thing to note is that the percent improvement exceeds 50%, which was the maximum percent improvement reasoned in Section 3.5. This is likely due to the overhead for each write associated with the FaaS protobuf implementation. This overhead can be hidden by the external synchrony protocol's parallization but will still be present in the baseline. The 50 and 150ms delays are about the same difference from the 100ms write time. Thus, for large repetitions, they should provide similar improvements. For small repetitions, the 150ms delay performs better because

the average write time will be larger than 100ms. This is due to the slow first write time, which will pull the average write time upwards and thus closer to 150ms. Finally, for 0ms delay, the percent improvement is naturally the smallest as there will be much less opportunity for the external synchrony protocol to complete parallel writes.

The single function microbenchmarks demonstrate the potential efficacy of the external synchrony protocol. The experimental results from varying the interoperation compute time largely match with the theoretical expectation. As reasoned in section 3.5, when the write time and execution time of the function are comparable, the performance improvements are greatest. Furthermore, the experimental results from varying the number of repetitions show that functions need to have sufficient runtime for the maximal gains of the external synchrony protocol to be realized. However, these results were obtained in a synthetic context. The potential benefit is clearly present, but the achievability of these results also needed to be tested in a more realistic setting.

5.3 Grading System

The Princeton computer science courses COS316 and COS326 both use a FaaS-based application for automatically grading student programming assignments on every push to the student's GitHub repository. In response to a new student push to their repository, the system would trigger the workflow shown in Figure 5.3. First, the system calls the `go_grader` function, which runs all the tests relevant for the current assignment (Go is the language used for COS316). Next, the system will invoke the `grades` function, which takes the test outputs and computes the grade of the student. Then, the system will invoke the `generate_report` function, which creates a markdown formatted report detailing the submission's results for each test. Finally, the system will create a comment on GitHub consisting of this report. In one pass

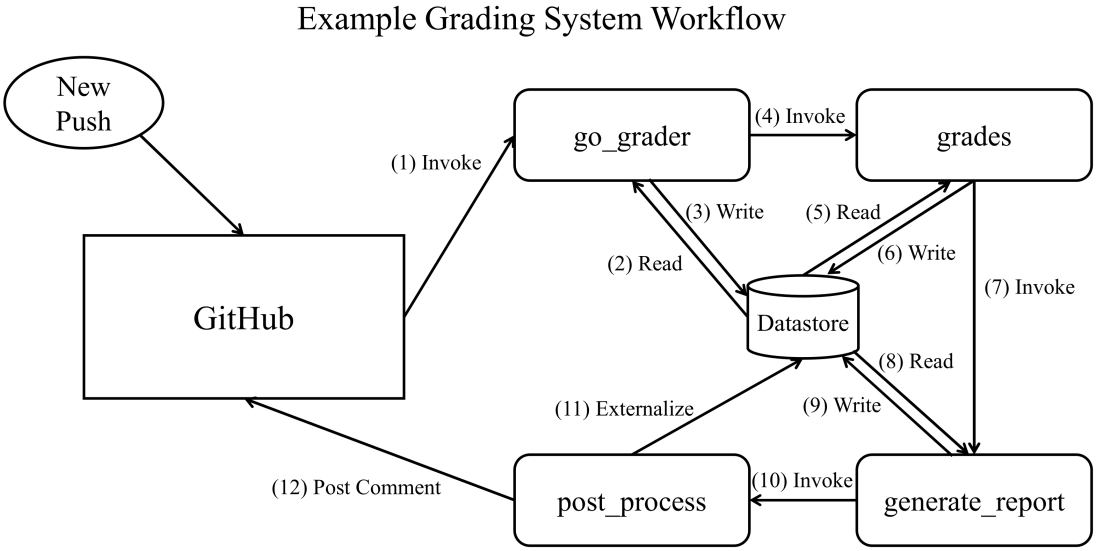

Figure 5.3: Example workflow for Princeton's COS316 grading system. Each function will read in the relevant data - (2) student submission, (5) graded submission, (8) summary of grades - and process it. Then the function will write its output back to the datastore - (3) graded submission, (6) summary of grades, (9) markdown report ready for GitHub comment. Finally, the `post_process` function will externalize all data and post the markdown comment to GitHub.

through this workflow, there are 3 calls to write.

For this evaluation, the grading system used a sample COS316 submission along with a grading script that only contained a single test. Before the workflow was run, the database was populated with the relevant metadata regarding the submission and grading script. Rather than creating a comment on GitHub at the end of the workflow, the final function simply made an externalization. Additionally, the first run of the workflow was not counted for the evaluation as the system was not yet "warm."

In Table 5.1, the cumulative latency of the workflow is measured for both the baseline procedure and the external synchrony protocol. The table shows that as the workflow progressed, the external synchrony protocol's margin of improvement became larger. The first function `go_grader` involves the most computation as the grad-

Table 5.1: Cumulative latency for example grading system workflow.

	Cumulative Latency (ms)		
Function	External Synchrony	Baseline	Difference
`go_grader`	1499.29	1647.15	147.86
`grades`	1501.06	1779.51	278.45
`generate_report`	1503.02	1915.07	412.05
`post_process`	1617.65	2048.50	430.85

ing script must be run on the submission. The external synchrony protocol removes the latency of making the write in the function. This approximately corresponds to the difference in latency with the baseline procedure. Next, the external synchrony protocol completes the `grades` and `generate_report` functions very quickly because those two functions involve one read and write each but contain very little computation. As such, the external synchrony protocol only needed to do the read/write to the local cache. The additional difference in latency with the baseline procedure again approximately corresponds to the latency of making a write to the globally distributed database. On the final `post_process` function, the external synchrony needs to externalize. This externalization accounts for the time the final function takes. By the end of the workflow, the external synchrony protocol resulted in a 21.03% improvement over the baseline procedure.

The results of the grading system evaluation showed that the external synchrony protocol produced a substantial improvement over the baseline procedure. However, the percent improvement is not as large as the maximum in the single function microbenchmark. This is quite reasonable since the workflow only has 3 writes, meaning its write time is much less than the execution time of the workflow.

Chapter 6

Discussion and Future Work

6.1 Discussion

The results show that the external synchrony protocol provides substantial latency improvements over the baseline procedure. The single function microbenchmark results showed that the improvements the external synchrony protocol provides closely match the theoretical expectations. This exhibited the clear potential external synchrony has in providing significant performance improvements in FaaS workflows. The grading system evaluation then demonstrated the efficacy of the external synchrony protocol in a real application.

However, there were limitations to the evaluation. As mentioned in Section 5.3, the grading system application involved relatively few writes throughout the workflow. Completing further evaluation on a wide variety of FaaS applications would give more insight into the external synchrony protocol's efficacy. An application with a larger number of database writes would provide a more robust evaluation of the external synchrony protocol. One such set of applications is the DeathStarBench Suite [20]. The DeathStarBench Suite is an open-source benchmark suite for cloud microservices such as FaaS. This set of applications is realistic and has a fair amount

of workflow complexity. Additionally, the applications within the DeathStarBench Suite are heterogeneous from each other, providing a diverse set of applications to test on.

Another limitation of the current evaluation was that there was not an experiment consisting of multiple workflows running concurrently. As discussed in Section 3.3, in order to accommodate multiple workflow dependencies, the design of the external synchrony protocol requires that it is flexible and able to externalize at intermediate points in the workflow. Although the implementation supports this, there was not an evaluation of it. Such an evaluation could involve many workflows running concurrently with dependencies on each other. This would require externalization of writes in the middle of workflows. Additional externalization could theoretically diminish the improvements the external synchrony protocol provides as the system would need to block more often. Testing this would give further understanding to the scalability of the external synchrony protocol to concurrent workflows.

6.2 Future Work

The external synchrony protocol was only tested on simple key-value reads and writes. More complex abstractions such as directories and files are often used in real applications and can be built on top of basic read/write primitives. Furthermore, such abstractions can also have additional layers of security. The Faasten project creates a novel security protocol for FaaS systems [9]. Integrating file abstractions with such a security protocol into external synchrony would allow for the exploration of how security might impact latency improvements.

Finally, this project also assumed the use of a single shared local cache server for all functions in the workflow. This simplifies the consistency model greatly by guaranteeing the correctness of reads. However, a more realistic system would have one

local cache per function, thus creating the need for a more complex protocol to ensure the correctness of the local caches with respect to the globally distributed database. Implementing external synchrony in a multi-cache setup would enable researchers to obtain a more complete understanding of the benefits of external synchrony in a fully distributed setting.

Chapter 7

Conclusion

In this paper, the application of external synchrony to the distributed database protocol of FaaS workflows was examined. The external synchrony protocol enabled the workflow to make writes to the globally distributed database asynchronous while preserving each function's ordering of operations to the database. The external synchrony protocol also guaranteed the correctness of the workflow computation when compared with the synchronous baseline procedure. Upon externalization, the system would block until all previous function writes to the globally distributed database had completed. This ensures that the globally distributed database is causally consistent. Thus, external synchrony was hoped to provide relatively strong consistency guarantees for users of FaaS workflows while improving performance.

These theoretical improvements were supported by the evaluation. The single function microbenchmark experiments demonstrated the latency improvements that the external synchrony protocol could provide in a range of synthetic contexts. The grading system workflow proved that the external synchrony protocol could provide substantial end-to-end latency improvements in a real FaaS workflow application. Further evaluation on a more diverse and complex set of real applications would strengthen these conclusions. Despite this, this paper has shown that external syn-

chrony can be used to provide significant latency improvements in FaaS workflows while maintaining relatively strong guarantees for external observers.

Appendix A

Code

All code and results for this project can be found on GitHub at `https://github.com/ATLi2001/faasten`. Much of the code was forked from the original Faasten repository with modifications made. The original Faasten repository can be found at `https://github.com/faasten/faasten`.

Bibliography

[1] "Apache Open Whisk," https://openwhisk.apache.org/.

[2] "AWS Lambda," https://aws.amazon.com/lambda/.

[3] "AWS Step Functions," https://aws.amazon.com/step-functions/.

[4] "Azure Functions," https://azure.microsoft.com/en-us/products/functions/.

[5] "Best practices design patterns: optimizing Amazon S3 performance," https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html.

[6] "Causal Consistency and Read and Write Concerns," https://www.mongodb.com/docs/manual/core/causal-consistency-read-write-concerns/.

[7] "Cloud Functions," https://cloud.google.com/functions.

[8] "Connection Health Check," https://clients.amazonworkspaces.com/Health.html.

[9] "Faasten," https://github.com/faasten/faasten.

[10] "Lightning Memory-Mapped Database," https://www.symas.com/lmdb.

[11] "sharp - High performance Node.js image processing," https://sharp.pixelplumbing.com/.

[12] "TiKV," https://tikv.org/.

[13] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal Memory: Definitions, Implementation, and Programming," *Distrib. Comput.*, vol. 9, no. 1, p. 37–49, mar 1995. [Online]. Available: https://doi.org/10.1007/BF01784241

[14] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with Crucial," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, mar 2022. [Online]. Available: https://doi.org/10.1145/3490386

[15] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient Execution of Serverless Workflows," *Proc. VLDB Endow.*, vol. 15, no. 8, p. 1591–1604, apr 2022. [Online]. Available: https://doi.org/10.14778/3529337.3529344

[16] C. Cicconetti, M. Conti, and A. Passarella, "On Realizing Stateful FaaS in Serverless Edge Networks: State Propagation," in *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2021, pp. 89–96.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, aug 2013. [Online]. Available: https://doi.org/10.1145/2491245

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *SIGOPS Oper. Syst. Rev.*,

vol. 41, no. 6, p. 205–220, oct 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[19] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, jul 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[20] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[21] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, jul 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[22] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, Feb 2019. [Online].

Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[23] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[24] D. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing More with Less: Orchestrating Serverless Applications without an Orchestrator," in *Proc. 20th Symposium on Networked Systems Design and Implementation*, 2023, p. to appear.

[25] T. Lykhenko, R. Soares, and L. Rodrigues, "FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 159–171. [Online]. Available: https://doi.org/10.1145/3464298.3493392

[26] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the Sync," *ACM Trans. Comput. Syst.*, vol. 26, no. 3, sep 2008. [Online]. Available: https://doi.org/10.1145/1394441.1394442

[27] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[28] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075. [Online]. Available: https://doi.org/10.1145/3352460.3358296

[29] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2438–2452, aug 2020. [Online]. Available: https://doi.org/10.14778%2F3407790.3407836

[30] Y. Tan, D. Liu, N. Li, and A. Levy, "How Low Can You Go? Practical cold-start performance limits in FaaS," 2021. [Online]. Available: https://arxiv.org/pdf/2109.13319.pdf

[31] A. H. Tortosa *et al.*, "Performance Benchmark PostgreSQL / MongoDB," OnGres, Tech. Rep., https://info.enterprisedb.com/rs/069-ALB-339/images/PostgreSQL_MongoDB_Benchmark-WhitepaperFinal.pdf.

[32] W. Vogels, "Eventually Consistent," *Commun. ACM*, vol. 52, no. 1, p. 40–44, jan 2009. [Online]. Available: https://doi.org/10.1145/1435417.1435432

[33] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional Causal Consistency for Serverless Computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 83–97. [Online]. Available: https://doi.org/10.1145/3318464.3389710