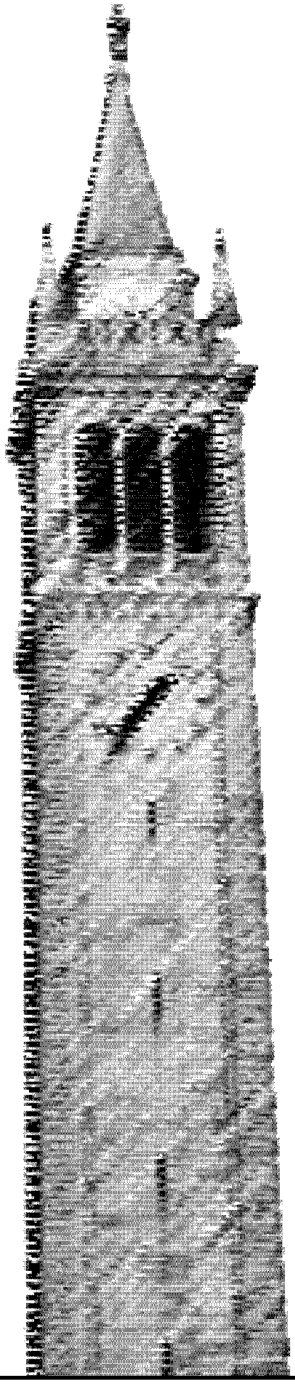
Understanding and Exploring Serverless Cloud Computing



Johann Schleier-Smith

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-273
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-273.html

December 30, 2022

Understanding and Exploring Serverless Cloud Computing

by

Johann Markus Schleier-Smith

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Natacha Crooks
Professor Heather Gray

Fall 2022

Understanding and Exploring Serverless Cloud Computing

Abstract

Understanding and Exploring Serverless Cloud Computing

by

Johann Markus Schleier-Smith

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

The past few years have seen a wave of enthusiasm for serverless computing, and we begin this work by analyzing the marketplace trends and underlying technical factors that have shaped the movement. We find that serverless computing addresses programming challenges in the same class as those that high-level programming languages address, suggesting that serverless computing may be viewed as high-level programming for distributed systems.

We next turn our attention to one of the key shortcomings of serverless: the lack of integration between compute and state. We develop FaaSFS, a distributed file system that is compatible with POSIX applications but uses a novel consistency model with relaxed real-time ordering constraints. We call this model externally consistent sequential consistency (ECSC) and use it to scale a pre-existing single-server application to 10,000 serverless processes. We also show that under reasonable assumptions ECSC is indistinguishable from linearizability, a widely accepted strong form of consistency.

Lastly, we explore whether serverless computing might lead to the demise of server hardware. By applying Amdahl's law and scaling rules for interconnect costs, we show that applications that rely on coordination protocols are particularly dependent on large servers for scalability. In contrast, those implemented with coordination-free protocols can run well on collections of small, low-cost servers or on disaggregated hardware. These approaches will likely continue to coexist, suggesting that a need for underlying server hardware will remain even as serverless abstractions thrive.

To My Mother

You gave so much yet expected so little.

Contents

List of Figures

List of Tables

Acknowledgments

Science, like so many things, is a collaborative pursuit. I am grateful to my collaborators, whom I thoroughly enjoyed working with on both projects described here and related ones. Nathan Pemberton has been my ally from day one, Leonhard Holz helped write much of the system implementation described in Chapter 3, Vikram Sreekanti and Chenggang Wu invited me to participate in the Cloudburst project, which turned out to be a lot of fun. Alexey Tumanov and Jose Faleiro are both sharp thinkers who shared their wisdom with me. My views on serverless computing were heavily shaped by conversations with a group that additionally included Eric Jonas, Neeraja Yadwadkar, Anurag Khandelwal, Joao Menezes Carreira, Chia-Che Tsai, Qifan Pu, Vaishaal Shankar, Karl Krauth, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. I am also grateful for the dissertation editing support that I received from Charles Kozierok and Ingeborg Schleier.

I owe a special thanks to Joseph Hellerstein, my advisor and collaborator, for steady guidance and supportive challenges. I also thank my dissertation committee members Heather Gray and Natacha Crooks for their enthusiastic support, careful reading, and thoughtful suggestions.

Life, too, is a collaborative pursuit. I am grateful to my wife, Ina, for her encouragement, and to my entire family for their support, understanding, and patience.

Chapter 1

Introduction

1.1 The Challenges of Scale

A simple question captures the motivation for this work quite concisely:

> *Why is creating software that works at large scale so much harder than creating software that uses only small amounts of resources?*

As an example, consider the scaling saga of Twitter, a popular social network. As we describe in Section 1.3, a small team built the first version of the product in just two weeks, but this implementation stopped working reliably as usage of the product increased. Keeping up with demand turned out to be surprisingly difficult—while it is natural to expect that supporting 100 million tweets per day would require more programmer effort than supporting 100,000, the effort required in this case was orders of magnitude greater. Why?

The question of what makes scale hard has puzzled us for well over a decade. It is also a question that many industry practitioners might jump to answer with conviction and authority. Ask one of them what makes scale hard and they will likely tell you about all the factors that they consider when designing a program to run at scale. Their response might start something like this: "You'll need a lot of servers because of the data volume and because you'll need to keep up with a high rate of incoming requests. You'll need to think carefully about how you split up the data and the different functions of the application to achieve performance, reliability, and security. Lots of things that could go wrong—for example, a server could crash, you might lose a network link, or a bug in one part of the system could knock everything down in a cascading fashion." The individual might continue on, sharing advice for writing an application as a collection of microservices, each resilient to service interruptions or changes in the behavior of components that it interacts with. The person might tell you about tools to use for monitoring or talk about how technologies such as container orchestration now make it easier to run lightweight virtual servers in the cloud. Table 1.1 shows several of the well known challenges of scale.

Table 1.1: Challenges of scale, adapted from Jonas et al. [216].

- Providing local redundancy to limit the impact of failures of individual compute, storage, or network components.
- Providing global redundancy to offer low latency in multiple geographies and to maintain service continuity in case of regional disasters.
- Implementing request routing or load balancing to ensure timely service responsiveness and efficient resource utilization.
- Monitoring to alert operators to any problems or potential problems with the service.
- Logging to support debugging and performance tuning.
- Upgrading system software, including keeping up with security patches.
- Migrating to improved hardware as new options become available.

It is abundantly clear that creating software that works at large scale is challenging today—the experts tell us so. However, those experienced in building scalable systems may be so familiar with the difficulties of their craft that they have come to take them for granted. We thus ask, are the challenges of scale fundamental and inescapable, or is it possible to create a platform that makes writing software for large-scale systems just as easy as writing software for small ones?

We want to be clear that when we talk about *scale*, we are referring to the amount of physical resources required to run the software—resources like CPUs, GPUs, and domain-specific accelerators as well as memory, storage, and network resources. We are not referring to the complexity or scope of functionality that the program provides—naturally, more complex software functionality requires more complex programs, and often larger ones as well. What we find surprising, however, is that simple applications such as social networks become vastly more difficult to implement at large scale than at small scale even when doing so adds no other features or functionality.

An industry trend known as "serverless computing," has greatly simplified writing some types of scalable applications. It provides abstractions that hide the underlying servers, allowing small and straightforward programs to run across thousands of servers. We found this intriguing, and set out to study it in hopes of understanding whether these promising developments might be generalized.

1.2 Higher Expectations

The need for something like serverless computing—a way of simplifying cloud programming, has been growing more acute over the past years. The underlying reason for this is an insidious trend towards higher expectations that can cause simple applications to demand distributed systems programming skills. Figure 1.1 illustrates an interlinked set of concerns that come to bear upon modern cloud application development. Scale and fault tolerance go hand-in-hand with cloud computing, meaning that the minimum expectations for applications are now greater than they were in the past.

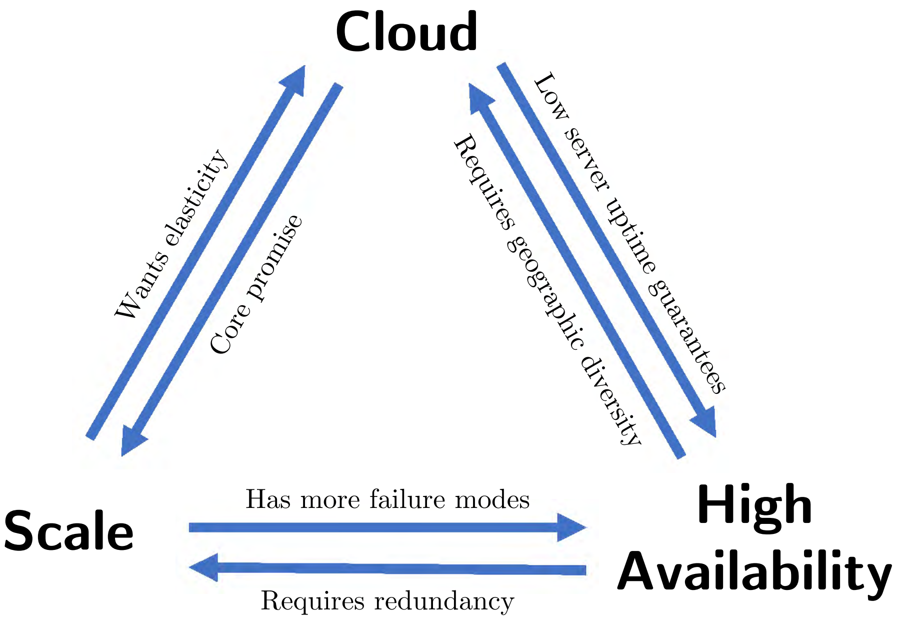


Figure 1.1: Intertwined concerns create higher expectations: scale, fault tolerance, and cloud computing are difficult to separate.

Scale and availability are intertwined because larger systems have both more potential failure modes and because maintaining service during failures requires redundancy, and so increases system scale. Scale also often brings a greater robustness requirement simply because failures can become more costly.

Cloud computing and scale are also obviously connected. Easy access to scale is a core promise of cloud computing [37], and the cloud is today the most practical way to access large amounts of computing resources.

Fault tolerance and cloud computing are intertwined as well. Cloud infrastructure traditionally used low-cost servers that were less reliable than those in enterprise data centers [59]. This is probably less true in today's public cloud, but the service level agreements (SLAs) published by major cloud providers all promise only 99.5% availability for any single instance [22, 108, 375]. This equates to up to 3.6 hours of downtime per month, and means that most applications require redundant servers to ensure acceptable reliability. Luckily, cloud computing can provide resources on demand in multiple geographies, exactly what is needed to achieve high availability.

Scale, fault tolerance, and cloud computing are intertwined concepts. The rise of cloud computing thus creates elevated expectations that software is still struggling to catch up with. We provide further support for this view in Section 2.8.2.

1.3 Social Networking Case Study

The early years of social networking put the challenges of scale into stark relief. Twitter wasn't originally a particularly complicated product, consisting of just the core functionality of today's product. Twitter allowed users to post short 140-character "tweets" on a time-ordered personal "feed." It also allowed users to view a consolidated feed that combined the posts from all of the users to whom they subscribed or "followed." This, along with generic account management features such as password management, comprised the functionality. As is reasonable for such a simple product, the initial implementation of Twitter was built by a small team in just two weeks [397].

The ease with which engineers first built Twitter contrasts with the difficulties they encountered in scaling it. As the product gained popularity, availability suffered [358] despite significant efforts to improve the platform [111]. Twitter eventually succeeded in scaling to support hundreds of millions of users, but it had to hire hundreds of engineers to do so. And while the product gained functionality (e.g., photos and videos, advertising, search), the company described requiring significant investments not just for these new features but also just to keep up with growing demand. When Twitter filed for its initial public offering, it disclosed both the size of these investments and the ongoing risk that scalability posed to the business, saying: "Managing our growth will require significant expenditures" and "Our ... results may be harmed by a disruption in our service, or by our failure to timely and effectively scale and adapt our existing technology and infrastructure" [412]. A review of Twitter's open source projects reveals over one million lines of code in platform-related projects, giving some sense for the magnitude of the investment in scalability (see Table 1.2). Several of these projects represented notable research contributions, e.g., to stream processing [238, 407].

Over time, a large team built an implementation that worked for 100 million users, but we can surmise that this application had perhaps 100× the amount of code as the version that supported 100,000 users. To a rough approximation, we guess that an increase in scale of three orders of magnitude required two orders of magnitude more code. It seems unlikely that this trend would apply in the other direction, that a version of Twitter supporting only 100 users would require two orders of magnitude less code.

Twitter's evolution reflects a middle path for social networks when it comes to scaling: a critical struggle that ultimately succeeded. Examples on either extreme are also instructive.

Friendster was the first social network to attract significant investment, yet it suffered a premature demise largely attributable to technical challenges posed by scale [155]. Instagram's story also offers a marked contrast: At the time it was acquired by Facebook, it had scaled to 30 million users and 150 million photos, all with only three engineers [197].

Instagram launched almost a decade after Friendster and benefited from the availability of cloud computing [37]. Instagram's engineers could rent servers on a minute's notice, upgrade to more powerful machines as needed, and take advantage of cloud services, e.g., saving photos using object storage. They also benefited from building on the experiences of the social networks that came before them; perhaps even benefiting from open source software projects that their predecessors had developed.

Despite Instagram's success, we believe that many companies still face challenges like those Twitter faced. In particular, those building pioneering products may be unable to take advantage of off-the-shelf solutions. It is clear that cloud computing vastly simplified access to scale, making it easier for talented small teams to do big things. While Friendster-like failures remain rare—or, at least, we don't hear about them—Instagram-like success stories appear to remain the exception rather than the rule. We have drawn these examples from social networking, however we believe that the parallels apply quite broadly to cloud software, especially because business applications have now commonly adopted design and technology approaches first established in the consumer sector.

Table 1.2: We reviewed Twitter's listing of open source software projects [413] and identified those related to back-end and infrastructure. The projects listed here contain over 1.2 million lines of code, and presumably represent only a portion of the systems needed to deliver the Twitter service at scale.

Project	Languages	Lines of Code	Description
finagle	Scala	163,084	Fault tolerant RPC system
pex	Python	127,243	Python executable generation
pants	Python, Rust	113,794	Build system
vireo	Bourne Shell	89,729	Video processing
util	Scala	70,389	Assorted utilities
scalding	Scala	67,230	Data API
pelikan	C, Rust	60,835	Unified cache backend
finatra	Scala	59,223	Services framework
rsc	Scala	57,461	Scala compiler
scoot	Go	47,412	Distributed task runner
scrooge	Scala, Java	40,795	Thrift RPC parser/generator
algebird	Scala	37,062	Abstract algebra
elephant-bird	Java	23,010	Compression and serialization
GraphJet	Java	18,997	Real-time graph processing
twemproxy	C	17,333	Cache and object storage proxy

hraven	Java	17,099	MapReduce job statistics
ccommon	C	16,197	Cache commons
twemcache	C	14,446	Caching and object storage
summingbird	Scala	13,266	Streaming MapReduce
rustcommon	Rust	11,072	Libraries
twitter-server	Scala	9,634	Services framework
bijection	Scala	9,555	Reversible computations
rezolus	Rust	9,315	Performance telemetry
cassovary	Scala	8,842	Graph processing
chill	Scala, Java	8,789	Serialization
torch-autograd	Lua	7,857	Auto-differentiation for ML
storehaus	Scala	7,648	Library for key-value stores
fatcache	C	7,392	SSD-based cache
torch-ipc	C, Lua	6,220	Parallel ML
sbf	Java	5,986	Graph processing
bazel-multiversion	Scala	5,802	Build system
netty-http2	Java	5,560	HTTP protocol implementation
rpc-perf	Rust	5,200	Benchmarking tool
iago2	Scala	5,128	Load generator
caladrius	Python	5,057	Performance modeling
libwatchman	C	4,580	File change monitoring
Serial	Java	4,321	Serialization
torch-decisiontree	Lua, C	4,087	ML algorithms
zktraffic	Python	3,923	Consensus algorithm analysis
torch-dataset	Lua	3,704	ML data loading
nodes	Java	3,625	Dependency graphs for services
joauth	Java, Scala	3,064	Authentication
groupcache	Go	3,011	Caching
hpack	Java	2,302	HTTP header compression
go-bindata	Go	2,231	Go build tool

1.4 The Path to Serverless Computing

The difficulties of programming for scale become even more puzzling in light of how integrated modern data centers are. It has become common to refer to the massive data centers that support the cloud and major online services as “warehouse-scale computers” [60]. This terminology reflects the idea that these systems, containing tens or hundreds of thousands of individual servers, are designed as a unit and have the potential to function as one machine. In practice, however, most programmers continue to focus on individual servers or server-like virtual machines while writing software, for this is what their tools are built to do.

Serverless computing represents an evolution of the programming model for the cloud toward abstractions more suitable for warehouse-scale computers [216]. The best known form of serverless computing is Function-as-a-Service (FaaS) [92], though as we will see in Section 2.5, serverless computing encompasses much more than FaaS alone.

FaaS provides a "stateless" computing model in which time-bounded tasks run in response to events such as web requests or the availability of items on queues. To deploy a FaaS program, the cloud customer provides code—often source files written in a high-level language such as Python or JavaScript—and specifies what triggers should cause that code to run. After that, the cloud provider takes care of everything else, including provisioning the necessary resources. The provider then bills the customer based on consumption, metering CPU and memory usage in intervals as small as 1 ms. By contrast, renting a cloud server requires paying for 1 minute of usage at a minimum.

For certain workloads, programmers using serverless computing and FaaS can access scale much more readily than they can with servers. Ideally they simply write the code, save it to the cloud, and let it run. In the often-referenced canonical example [421], a social network could write a simple FaaS function for resizing images to thumbnail size and configure it to run whenever new uploads appear in object storage. Doing this is much simpler than configuring a fleet of servers to scale up and down to meet the demand while also planning for various possible failure and workload scenarios.

That said, while FaaS works well in tasks like the image resizing example, there are many workloads for which it is currently unsuitable. For example, if we wanted to index and search the text captions of a large number of images, FaaS would do little to simplify a scalable implementation. Building the index would require coordinating a large number of functions, then piecing together their work. One experiment has shown that training and serving simple machine learning models can be 21× slower and 7.3× more expensive than a comparable implementation using servers [189]. Serving search results from a large index also would require partitioning, for which FaaS has no built-in support [355].

In addition to a lack of universal applicability, other shortcomings of serverless computing with FaaS have been documented [189]. It has limited, though configurable, execution time that must be configured when you create a function, offers weak performance guarantees, has restricted networking, and provides no support at all for persistent state. Still, the possibility of extending the benefits of serverless computing to more use cases remains tantalizing, as does the promise of general-purpose serverless computing.

We adopt a broad view of serverless computing that includes any cloud programming model that hides the servers behind an abstraction, scales automatically, and bills on a pay-as-you go model (see Section 2.4). For example, under this definition, object storage is a form of serverless computing, i.e., AWS S3, Azure Blob Storage, and Google Cloud Object Storage are all serverless. This perspective suggests that the success of Instagram relied in part on an early form of serverless computing.

Serverless computing solves the *challenges of scale* in limited settings and for limited applications. We set out to study it because it seemed to offer a glimpse into what might be possible, and because it had achieved a level of industrial adoption that we hoped would ensure a practical significance to any findings.

1.4.1 Overview of this work

At the risk of disappointing the reader, we acknowledge that this work will not present a solution that makes software scalability effortless. Our aim was more modest: to collect insights and lessons that might lead in that direction.

We begin with a study of serverless computing as it exists in industry today. As described in Chapter 2, we identify the essential characteristics of serverless computing, then survey how various cloud product offerings align with this definition. In our view the most important advance that serverless computing represents is a simplified programming model for the cloud. We explain how programming with servers really is what makes cloud programming complicated, and how programming models failed to keep up with the raised expectations of cloud computing described in Section 1.2. We also note how today's serverless computing has significant limitations, setting the stage for the narrower studies that follow.

The principal software artifact developed in this work is the FaaS File System (FaaSFS) described in Chapter 3. FaaSFS provides a serverless implementation of the Portable Operating System Interface (POSIX) file system interface in hopes of allowing existing software to run with serverless operating benefits. Integrating computation and storage in one service allows for interesting optimizations, and in some cases big wins. For example, we show that FaaSFS can scale to run an existing single-server web application well on 10,000 instances. However FaaSFS is not suitable for all applications, and demonstrates the limitations of sticking with the POSIX file system abstraction.

One of the enabling innovations for FaaSFS is a new form of strong consistency. In Chapter 4 we develop a formal theory of externally consistent sequential consistency (ECSC), a hybrid consistency model that ensures that programs produce the same outputs as they would when using linearizable storage, even though they violate linearizability in ways that allow them to run faster.

We close with a study that looks ahead to the possible implications of widespread serverless computing, considering whether removing servers from the programming model for the cloud might ultimately lead to alternative physical infrastructure, that is to "serverless hardware." In Chapter 5, we use a simple model of interconnect cost scaling together with Amdahl's law to show that the need for server hardware will likely continue even if software adopts serverless abstractions. The model shows that large server hardware, despite its high per-processor cost, is critical for scaling applications that rely heavily upon coordination, though less so for those that make greater use of coordination-free approaches.

It has been gratifying to see an explosion of research interest in serverless computing in the years since we started this work. Our hope is that the research we describe here proves to be a useful contribution to this now vast enterprise.

Chapter 2

Understanding Serverless Cloud Computing

2.1 Introduction

Modern computer systems can empower people by providing access to unprecedented amounts of computing power in convenient ways. The most visible proof of this is in the ubiquitous pocket supercomputers (mobile phones) that everyone carries with them. Invisible, yet equally important, are the millions of hidden computers that reside in commercial data centers. Since the advent of cloud computing, anyone has been able to rent these computers [37], which makes it quite affordable to obtain them even in large numbers. Renting 1,000 computers for one hour costs the same as renting one computer for 1,000 hours. Unfortunately, having many computers is of little use if they cannot be harnessed to work together effectively, and accomplishing that goal can be a serious challenge for programmers (see Chapter 1). As a result, despite the enormous success of cloud computing, much of its potential remains untapped.

We will show how serverless computing promises to make it easier to program the cloud by abstracting away those complicated and awkward details—the servers—that do not correspond to anything in problems that programmers are working to solve. Put one way, it can remove servers from the programming and operating model of the cloud. One might also say that it can make a collection of computers work more like one big computer.

A big caveat, as of now, is that serverless computing only delivers such benefits to a relatively narrow set of applications. The development that led to serverless computing was the introduction of Function as a Service (FaaS). It provides a simple cloud deployment model, but only for stateless programs with limited execution time. Furthermore, there are reasonable ways to argue that FaaS is a mundane development with ample precedent, that its commercial launch changes little. In this context, the idea of serverless computing having a revolutionary effect, like making it possible to use the cloud like a single massive computer, seems far-fetched. We adopt a broader view of serverless computing, from which its potential becomes apparent.

This chapter aims to clarify the gap between the conceptual promise of serverless computing and the reality of today's serverless computing offerings. It is important to realize that serverless computing encompasses more than just FaaS. FaaS was a tremendously exciting development because it brought a simplified programming and deployment model to programs written in a variety of popular programming languages. "Serverless" seems to have arisen as a catchy way to describe FaaS, and the name has stuck. However the idea of abstracting away the cloud's servers behind scalable services precedes FaaS. For example, cloud messaging or pub/sub services are abstractions that hide servers; the same is true of cloud object storage. These were some of the first offerings of Amazon Web Services (AWS), a cloud computing pioneer, and arrived years before AWS offered servers for rent outside of a beta program.

We also observe that the challenges of scale (see Chapter 1) result directly from the need to work with many servers. In Section 2.8 we make this case by drawing upon a celebrated result from the study of software engineering: the distinction between essential complexity and accidental complexity identified by Fred Brooks [79]. Essential complexity is that which is inherent in the functionality that the software provides, whereas accidental complexity results from the nuances and limitations of the machine that it runs on or perhaps the programming environment used to create it. In the case of cloud programming, programmers are funneled toward writing software for distributed systems. In many cases, they encounter complex challenges not because the problems they are solving are complicated, but because working with lots of servers introduces its own difficulties.

This perspective makes clear the role that serverless computing plays in the development of cloud computing—it directly addresses the problem that makes cloud programming hard. Furthermore, it allows us to draw straightforward connections to previous technological developments that also helped hide complexity in underlying machines, notably the development of high-level programming languages.

This chapter seeks to distill learnings from the emergence of serverless computing. We focus in some detail on FaaS, seeking to explain what made it successful. Even though the term *serverless computing* emerged to describe FaaS, it embodies a broader concept that now encompasses many cloud services that share a number of essential characteristics. We survey serverless cloud services, seeking to document their capabilities and limitations. We then explain why we see simplified cloud programming as the most useful interpretation of serverless computing—that which best tells us where the movement is headed. In closing, we touch on assorted topics that we found interesting and that helped us understand the emergence and implications of serverless computing.

2.2 Function as a Service

Function as a Service (FaaS) first appeared in AWS Lambda, which was announced in 2014 and reached general availability in 2015. The core idea was a simple one, easily described by the typical steps customers take to use it:

1. Write a small program, also known as a *function*, that takes some input, runs for a limited duration to complete some task, and optionally returns some output.

2. Save the function to the cloud, configuring it to run in response to specific events such as web service invocations or messages published on queues.

3. Pay for execution time, metered at a sub-second level, whenever the function runs.

With FaaS, the cloud provider assumes responsibility for provisioning the underlying servers and the secure execution environments in which the code runs. This includes making sure that there are enough instances available to meet execution demand and assigning resources to customers only when they are in use. A sample FaaS program appears in Figure 2.1, code for saving it to the cloud appears in Figure 2.2, and code for invoking it appears in Figure 2.3 and Figure 2.4.

Responsive scaling is one of the hallmarks of FaaS. When faced with sustained requests, AWS Lambda takes just a few minutes to create thousands of execution instances to run a function.

In addition to providing scalability, FaaS shifts other operational responsibilities to the cloud provider. A level of fault tolerance comes from automated retries, which protect against failures of various types; these range from hardware failures to intermittent bugs in the function code itself. The provider also deals with many security concerns, including the patching of operating systems, runtime environments, standard libraries, and other underlying technologies. Cloud providers can roll out fixes to serious vulnerabilities within hours—much faster than most customers are able to respond.

To the function code, the typical FaaS runtime environment looks much like the inside of a little server. There is a CPU with one or more cores and a popular instruction set such as x86 or Arm. A familiar operating system interface, such as Linux, is present as well. However, the environment has some important restrictions. One notable limitation is the timeout, which limits how long a function can run (typically up to several minutes). The operating system is also locked down significantly, so only unprivileged operations are permitted. The network allows outbound connections, but it has firewall rules intended to prevent inbound network requests [432].

FaaS has severe limitations regarding state management. It is often called *stateless*, but this label misses important nuances; it might be more precise to say that FaaS supports *ephemeral state* only. We identify two types of ephemeral state. The first one is the working memory used during function execution. This state expires when the function terminates, typically because it goes out of scope or becomes unreachable. The second is cached state that remains in the execution environment and thus continues to be available whenever the cloud provider reuses this environment to handle subsequent function invocations. In many cases, cached state is essential to performance. For example, a function that makes predictions using machine learning might load a model's weights at initialization time, then use them

```python
# AWS Lambda "Hello World" in Python
def say_hello(event, context):
    return 'Hello' + event['name'] + '!'
```

Figure 2.1: Hello World using Python and AWS Lambda. FaaS helps keep programming simple, even in the cloud and at scale.

```
# Copy the code to cloud object storage
aws s3 cp hello.py s3://code_bucket_path/hello.py

# Create the function
aws lambda create-function \
    --function-name=hello-world \
    --role=arn:aws:iam::... \
    --runtime=python3.9 \
    --handler=hello.say-hello \
    --memory-size=128 \
    --timeout=3 \
    --code=s3://code_bucket_path/hello.py
```

Figure 2.2: Installing Hello World function using the AWS command line interface. First we copy the code to object storage. Then we install it by specifying the name by which it will be called, the IAM role for security privileges, the Python environment required, the name of entry point within the Python program, the memory size (a proxy for all resources), an execution time limit, and the location of the code in object storage.

in many subsequent function invocations. Cached state also helps amortize runtime-specific overheads, such as library loading for Python [296], or JIT compilation for Java or JavaScript.

What FaaS lacks is any form of *persistent* state. Only the `/tmp` path of the local file system is writeable, and this is a repository for ephemeral state only. Its contents, along with all contents of the instance memory, may be lost when the function finishes executing. For example, the cloud provider may destroy an execution instance to reassign resources to another function, reoptimize placement, or perform upgrades.

Even when state still exists within a function instance, there may be no way to access it. When a function is invoked, the cloud provider may always choose to create a new instance to run it rather than to reuse an existing one. Also, there is no way to direct a function invocation at a particular instance, as the model assumes that they are unnamed and interchangeable.

```
# Invoke the function
aws lambda invoke \
    --function-name=hello-world \
    --payload='{"name": "Alice"}'
```

Figure 2.3: Invoking a FaaS function from the command line. The arguments are passed as a JSON object.

```
import boto3, json

client = boto3.client('lambda')
client.invoke(
  FunctionName='hello-world',
  InvocationType='Event',
  Payload=json.dumps({'name': 'Bob'}),
)
```

Figure 2.4: Invoking a FaaS function from a Python program.

FaaS functions may be invoked either synchronously or asynchronously. For synchronous invocations, the client makes a web service call to the cloud provider that blocks until the function has finished executing. At that point, the call completes, sending a response that includes any return value produced by the function. For asynchronous invocations, the FaaS service responds as soon as the invocation request has been stored on a queue. This allows the calling program to continue, confident that the invoked function will run at some time in the future.

2.3 About the "Serverless" Name

Prior to its use in the context of cloud computing, the term *serverless* appears in the literature with a distinctly different meaning. For example, the xFS file system was presented as a "serverless network file system" since it had no centralized entity and instead distributed all functionality across participating workstations [31, 430]. This use of the term, in an era where the client-server computing paradigm [373] dominated, was one way to indicate operation without computers dedicated to the server role, but this paradigm was very different from serverless computing as it exists in the cloud today.

Other serverless file systems included Farsite [7, 70, 71], an attempt to provide distributed file system on networks of Microsoft Windows PCs. Additional examples of "serverless" de-

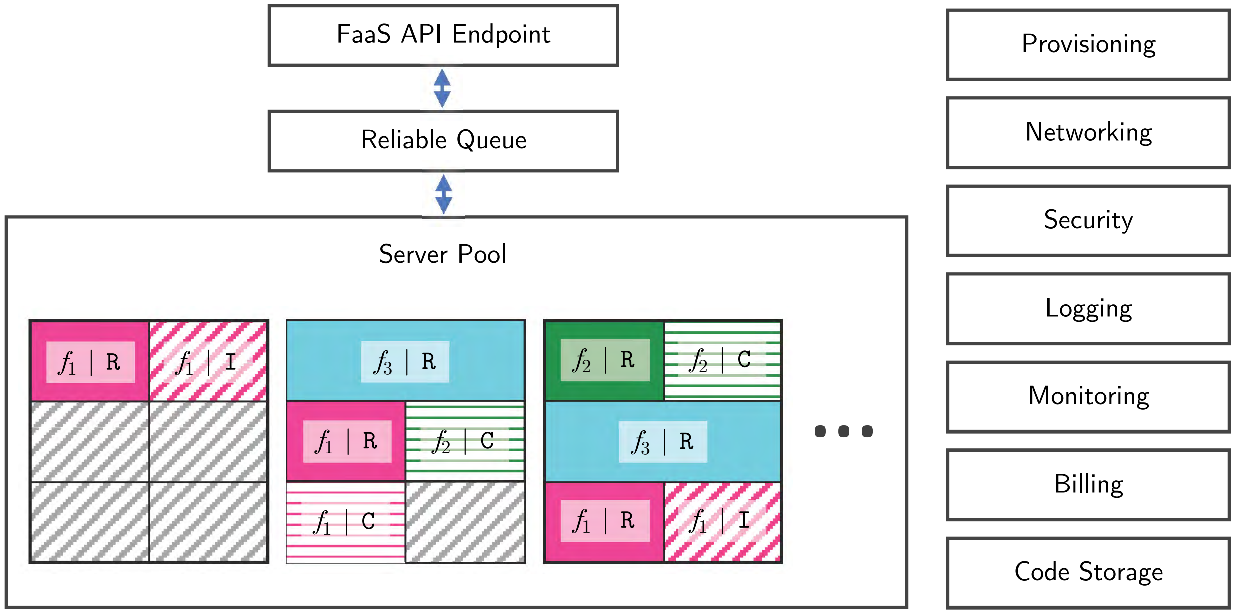


Figure 2.5: Components of a FaaS system. The application programming interface (API) Endpoint receives function invocation requests and configuration requests. A reliable queue holds invocations until execution resources become available and can ensure at-least-once execution. The server pool is a multi-tenant resource shared across customers and virtualization divides each server into a number of Virtual Machines (VMs). Lightweight virtualization allows one VM per function, though some serverless environments also may also pack several functions belonging to the same customer into a single VM instance. Each function is configured to run on an instance of a certain size, and receives a corresponding fraction of the compute, memory, and network resources in the machine. Supporting services include provisioning, networking, security, logging, monitoring, billing and code storage. In this example, f_1 and f_2 have small resource footprints, whereas f_3 has a larger resource footprint. Function instances also may have several states: (I) Initializing, (R) Running, and (C) Cached and idle.

centralization include applications in Voice over IP (VoIP) [80], Radio-Frequency Identification (RFID) [393], and mobile social networking [422]. All of these approaches fall under the umbrella of peer-to-peer (P2P) computing [279], and it is probably fair to say that serverless was once a synonym for P2P, which failed to stick.

Interestingly, the modern cloud traces its roots to the same network of workstations [30] used for the early experiments in serverless (in the P2P sense) file systems. To meet the computing needs of the internet boom, companies built networks of these relatively inexpensive "client" machines [59], reversing their role so that they functioned as servers to the millions of internet-connected devices [150]. As the approach matured, companies started to design data centers full of such machines, configured to operate as an integrated unit [60].

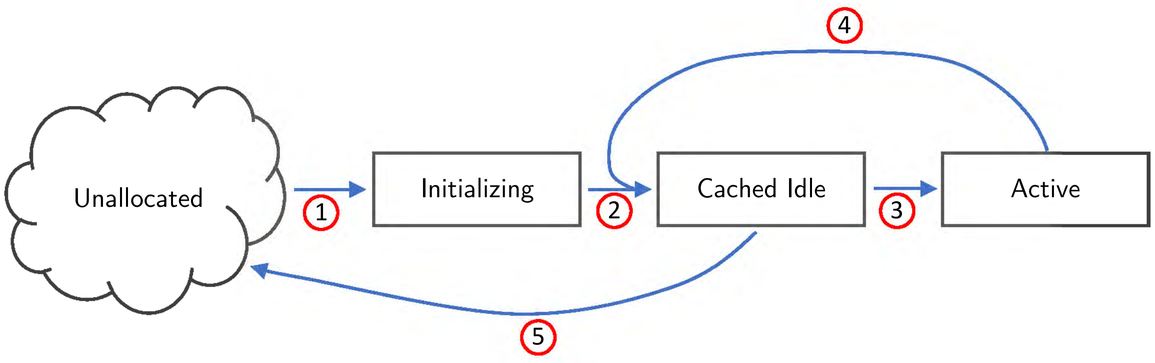


Figure 2.6: Life cycle of a FaaS execution instance. 1) a VM instance is created from unallocated server resources and the VM enters the `Initializing` state, 2) at the completion of initialization the function instance is ready to process requests, entering a `Cached Idle` state, 3) the function instance enters the `Active` state when it receives a request, 4) at request completion the instance returns to the `Cached Idle` state, 5) after a prolonged idle time the cloud provider reclaims function instance resources. The customer typically pays for `Initializing` time and `Active` time, but not `Cached Idle` time.

They also began to allow the public to rent these resources, which is now known as cloud computing [37].

When AWS Lambda first launched, the company did not call it FaaS or serverless: It was described simply as "event-driven compute." The other names came later, emerging from the ecosystem rather than the company.

FaaS fits neatly into the vocabulary of X as a Service, a categorization of cloud service offerings [37]. Services can be just about anything, just so long as that thing is provided over the network, and so long as the API hides the details of the implementation. Popular forms of cloud computing include Infrastructure as a Service (IaaS) for renting virtual servers, storage, and network resources; Platform as a Service (PaaS) for hosted software development frameworks; and Software as a Service (SaaS) for hosted application software. Another relevant category is Backend as a service (BaaS), which describes collections of service offerings that complement the frontend of an application, which might be a mobile app or code running in a web browser. BaaS typically includes some form of database for persistent state management, as well as services for authentication and messaging, e.g., via SMS or push notifications.

In a similar vein, FaaS accurately describes what AWS Lambda and similar products such as Azure Functions and Google Cloud Functions do: They provide an interface that allows users to define functions and then run them, all without worrying about what it takes to provide or manage the underlying infrastructure that makes this possible. Aside from this name being a bit dry, what prompted the need for another one?

To answer this, we draw upon a historical perspective assembled by Roberts [337]. The modern notion of serverless cloud computing predates FaaS by several years. In the 2012 article, “Why the Future of Software and Apps is Serverless,” Fromm describes a vision of elastic computing services and industrial-scale compute power [156]. He notes:

> Developers working in a distributed world are hard pressed to translate the things they’re doing into sets of servers. Their worldview is increasingly around tasks and process flows, not applications and servers—and their units of measures for compute cycles is in seconds and minutes, not hours. In short, their thinking is becoming serverless.

When AWS Lambda arrived in 2014, some developers were already craving a form of cloud computing that would free them from thinking about servers. Tapping into this thread of enthusiasm, cloud providers and other ecosystem participants adopted the name *serverless computing*, first for FaaS and later for other services.

There has been some controversy about whether serverless computing is a good name for the developments it represents. We believe it is and touch upon this in Section 2.8.5.

We emphasize that the notion of serverless computing in the cloud is conceptually distinct from its use to describe P2P technology. We can find some precedent for the idea of serverless cloud computing, however, by looking further into the past. Visions in the 1960s of “utility computing” imagined massive concentrations of computing power, drawing parallels to centralized electricity generation [57, 307]. It took decades of technological progress and market developments to get there, but with serverless computing, the cloud now appears poised to fulfill this vision.

Because the idea of serverless computing gained popularity due to the rise of FaaS, the distinction between the two sometimes gets lost; even authoritative works commonly make little distinction between the two (e.g. [92, 190]). Part of the reason for this is that FaaS is flexible and popular. Functions can be written in arbitrary programming languages, and despite some significant limitations [189], they work well in a reasonable variety of use cases [356]. Furthermore, FaaS is a concrete product that is straightforward to describe. Serverless, exciting as it may be, is a more abstract notion. We next distill the essential characteristics of serverless computing, which serve to cast its definition in greater relief.

2.4 Essential characteristics of serverless computing

A central tenet of serverless computing is that developers prefer not to think about servers: Other abstractions offer better targets for expressing program functionality. This point of view has a number of natural consequences, with both operational and business model implications. Taken together, they help outline a coherent and reasonably concrete definition of serverless computing.

We will review non-FaaS forms of serverless computing in more detail later in this section. For the time being, we focus on FaaS and also cloud object storage, as exemplified by AWS S3, Google Cloud Storage, or Azure Blob storage. Cloud object storage is backed by large pools of servers but provides a simple API allowing users to store and retrieve immutable objects. In some respects, it is like a simplified file system, offering a two-level naming scheme. Objects can be replaced in full but cannot be modified or extended. Users never provision capacity for cloud storage: They pay for reads and writes on a per-request basis and for storage based on the amount of data stored and the duration of that storage.

Hiding servers implies hiding certain details that are related to operating them. Cloud servers, which are actually virtual machines [377], already hide the details of how to provision servers, power them, and integrate them into networks. The serverless cloud goes a step further by introducing a layer of software that bridges individual servers, implements secure multiplexing of customer workloads, and provides an alternate computing abstraction. Figure 2.7 shows how serverless comprises a layer between applications and the underlying server-based cloud. The serverless cloud provider thus assumes a certain level of additional responsibility, including monitoring the availability of the underlying systems and ensuring that the workload is properly distributed over them.

This is where autoscaling comes in. When using servers, the developer typically configures a feedback mechanism that measures CPU utilization, the length of a queue, or some other application-defined metric, then adds or removes servers to maintain balance as the workload changes. In serverless computing, there are no servers to add or remove. The dominant pattern for serverless scaling is thus *implicit scaling*, where the cloud provider allocates resources in response to program execution rather than in response to an explicit request.

To make this more concrete, consider the following example: a customer could install open source FaaS software on a fixed number of servers; various alternatives exist [34, 190, 231, 236, 300]. This deployment offers programmers an abstraction without servers, but servers are still part of the operating model. If too few are provisioned, they will be unable to keep up with the workload; if too many, then idle resources will lead to unnecessary costs. This is FaaS, but is it serverless computing? To the programmer it may be serverless, assuming someone else in the organization is looking after the scaling. However to the administrator—who continues to manage servers—it is not.

We view cloud-provider-managed autoscaling as an essential feature of serverless computing. Not only does it eliminate one of the considerations that makes working with servers difficult, but the provider can be in a much better position to manage scaling than the customer. For one, the provider is better able to take advantage of statistical multiplexing. By drawing resources from a large pool shared across customers, it can achieve higher resource utilization than any one of them could alone. Furthermore, the time required to start up a server can be measured in minutes, whereas the virtualized environments used for FaaS can boot in under 100 ms [9, 180]. Faster start times mean resources can be more rapidly shifted between customers, making it possible to respond to load spikes more quickly or, alternatively, maintain a smaller amount of idle capacity.

The provider-managed autoscaling seen in serverless computing is unlike anything that

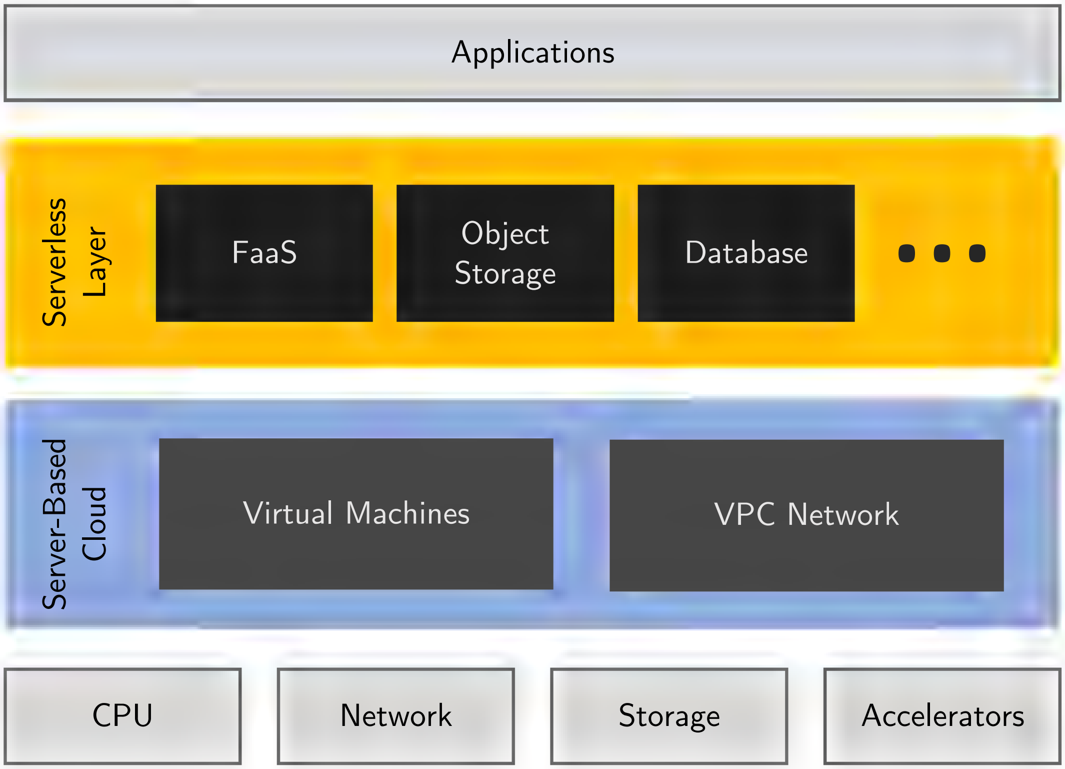


Figure 2.7: Serverless computing provides an abstraction over the underlying servers, networks, and other infrastructure that make up the cloud.

was possible with servers. At the high end, it allows scaling up to thousands of FaaS instances within just a few minutes [217]. At the low end, when no code is running, the customer pays nothing, or at most, just a small fee for storing the program code. This ability to scale down to zero while remaining ready to scale up again as needed is an important and distinctive feature of serverless computing—by contrast, a server-based service scaled to zero servers goes offline. In the context of cloud object storage, implicit autoscaling allows customers to upload petabytes of data without ever worrying about whether there will be enough capacity and without worrying about maintaining a comfortable buffer of free storage capacity. We view such high-quality autoscaling as an important characteristic, and key innovation, of serverless computing.

Pricing also plays a defining role in serverless computing. Its utility-style pay-as-you-go model fits naturally with provider-managed autoscaling. In the example of self-hosted FaaS, the customer pays for the servers rented from the cloud regardless of whether they are needed to satisfy the workload. Once the provider assumes responsibility for scaling, and with it the responsibility for managing idle capacity, it follows that the customer should never pay for this idle capacity. In the case of FaaS, this means paying when a function is executing, not otherwise. For object storage, it means paying from the time an object is stored to the time it is removed.

These considerations lead us to the three essential characteristics of serverless computing. These are the product of our work and are reflected in industry consensus as well [92, 216, 356]:

- Abstraction: Hiding the servers and the complexity of programming and operating them.
- Autoscaling: Automatic, rapid, and unlimited scaling of resources up and down to match demand closely, from zero to practically infinite.
- Pay-as-you-go: Eliminating the need for resource reservations and charges for idle resources.

In our view, the second and third go hand in hand and follow quite naturally from the first. When servers are abstracted away, so is the standard unit of provisioning and deployment. Serverless computing instead introduces a model where resource allocations and costs both follow from application behavior. There are also probably other characteristics that could be added to this list. For example, Roberts lists high availability as a defining characteristic of serverless computing [336]. In addition to accessing scale, availability and robustness are key reasons for linking together many servers, and some level of redundancy or fault tolerance is now expected in cloud computing (see Section 1.2 and Section 2.8.2).

When programmers seek an abstraction without servers, they also seek an abstraction without the myriad complex independent failure modes that server-based systems have. These need not hide all failures, but they must make recovery simple. For example, cloud object storage may use erasure coding and other techniques to hide problems with individual servers. However it may still return transient errors, e.g., the HTTP response code 500, meaning an internal error, to indicate that clients should retry a request later. The cloud provider is responsible for healing errors quickly, but it is not presently responsible for completely masking all failures.

When we review various serverless services in the next section, we will see that not everything marketed as serverless has all three of the characteristics of serverless computing identified above. That does not mean that cloud providers are wrong to call them serverless; after all, they do have some of these characteristics. What it does tell us is that serverless may not be a simple binary classification—that services can be serverless to varying degrees.

2.5 The Serverless Menagerie

Table 2.1 lists the services that cloud providers market as serverless at the time of this writing. We have also included a few products that are not marketed as serverless but that are similar to products from other cloud providers that are. This analysis has multiple purposes. First, it allows us to test how well the essential characteristics of serverless computing (see Section 2.4) match up with how the market uses the "serverless" term. Second, by comparing products across cloud providers and looking into how certain products have evolved over time, we can develop a sense for which aspects are important or have staying power. Third, examples let us test the boundaries of what it makes sense to label as serverless computing. Finally, by studying what forms serverless computing take today, we gain background understanding to inform future research.

This process allows us to confirm that serverless computing is about *much* more than FaaS. There is no doubt that serverless computing owes its fame to FaaS, but serverless object storage, messaging systems, key-value store databases, and big data query engines are also proven and popular serverless technologies. As these products have matured, they have come to align more and more closely with the essential characteristics of serverless computing. Improvements in scalability and fine-grained pricing are the most common enhancements. We also continue to see new serverless services emerge. These range from reimplementations of long-established APIs, like distributed file systems, to emergent technologies, like those for synchronizing data for mobile applications.

We divide our review by type of service. We touch, in turn, upon FaaS, managed compute, container services, application platforms, event services, workflow services, API management, object storage and file systems, database services and key-value stores, big data services, and machine learning services.

Service	Description	Serverless marketing	Abstraction	Autoscaling	Pay-as-you-go
	AWS				
Lambda	FaaS	✔	✔	✔	✔/IP
Fargate	Container service	✔	✘	>0	IP
Elastic Beanstalk	Managed application environments	✘	✘	>0	IP
EventBridge	Event-driven architecture	✔	✔	✔	✔
Step Functions	Low-code service orchestration	✔	✔	✔	✔
SQS	Message queues	✔	✔	✔	✔
SNS	Pub-sub, SMS, email	✔	✔	✔	✔
API Gateway	Web service endpoints	✔	✔	✔	✔
AppSync	GraphQL APIs	✔	✔	✔	✔
S3	Object storage	✔	✔	✔	✔
EFS	Distributed file system	✔	✔	✔	✔
DynamoDB	Key-value database	✔	✔	✔	✔/IP
Aurora Serverless	Relational database	✔	N/A	<1	IP
Glue	Data integration	✔	✔	✔	IP
Athena	Big data query service	✔	✔	✔	✔

Redshift	Big data query service	✔	✔	✔	✔
		Azure			
Functions	FaaS	✔	✔	✔	✔/IP
Kubernetes Service	Container service	✔	✘	>0	IP
App Service	Managed application environments	✔	✔*	✔	IP
Logic Apps	Low-code business workflows	✔	✔	✔	✔/IP
API Management	API Gateways	✔	✔	✔	✔
Event Grid	Event routing and management	✔	✔	✔	✔
Service Bus	Messaging service	✔	✔	✔	✔/IP
Cognitive Services	Natural language processing	✔	✔	✔	✔
Bot Services	Build intelligent bots	✔	✔	✔	✔
Machine Learning	Machine learning models	✔	✔	✔	IP
SQL Database Serverless	Managed database service	✔	N/A	<1	IP
Cosmos DB	Globally distributed database	✔	✔	✔	✔/IP
Blob Storage	Object storage	✔	✔	✔	✔
Files	Distributed file system	✔	✔	✔	✔
Stream Analytics	Real-time analytics	✔	✔	✔	IP
Data Lake Analytics	Big data query service	✔	✔	✔	IP
		Google Cloud			
Cloud Functions	FaaS	✔	✔	✔	✔
Cloud Run	Managed compute platform	✔	✔*	✔	✔/IP
API Gateway	Web service endpoints	✔	✔	✔	✔
App Engine	Application Platform	✔	✔*	✔	IP
Firebase	Application Platform	✔*	✔	✔	✔
Kubernetes Engine	Container services	✘	✘	>0	IP
Workflows	Workflow orchestration	✔	✔	✔	✔
Cloud Datastore	NoSQL database	✘	✔	✔	✔
Cloud Storage	Object storage	✘	✔	✔	✔
Cloud Pub/Sub	Messaging service	✘	✔	✔	✔
Cloud Dataflow	Stream processing analytics	✔	✔	✔	✔
BigQuery	Big data query service	✔	✔	✔	✔
Dataprep by Trifacta	Intelligent data preparation	✔	✔	✔	✔

Table 2.1: Serverless characteristics of cloud services. We included all services described as serverless on cloud provider web sites (as of September 2021), as well as other services with serverless characteristics. For abstraction, ✔* indicates weak serverless abstraction, e.g., on account of configurable per-server concurrency limits. Autoscaling ">0" indicates the service does not scale down to zero and "<1" indicates that autoscaling adjusts the size of a single server. "IP" indicates instance-based pricing where customer may pay for idle resources. ✔/IP indicates that the cloud provider offers customers a choice of pricing models.

2.5.1 FaaS Services

FaaS was the first to be called serverless computing and draws the green check mark (✔) across the board in Table 2.1. AWS Lambda, Azure Functions, and Google Cloud Functions all provide the abstraction of code running in response to events, with provider-managed autoscaling and a fine-grained pay-as-you-go pricing model. Despite these marks, FaaS is not a perfect fit for the essential characteristics of serverless computing.

We note that AWS Lambda and Azure Functions also offer alternate billing models. Lambda offers “provisioned concurrency,” a model introduced in 2019 [46], in which the customer pays a base rate on an ongoing basis to keep instances of a function running, even if they are not used. This means that when invocations do arrive, they can be processed without any initialization, which is helpful for ensuring consistent low latency and eliminating the problem of “cold starts” (see, e.g., [14, 190, 296, 362]). However, this model breaks pay-as-you-go pricing and also introduces a knob that might need manual configuration to ensure consistent low latency as a service scales. Azure Functions allows “App Service” deployment, which basically runs the FaaS software on a set of servers (described further in Section 2.5.2). The customer pays for the number of servers provisioned, and uses server autoscaling techniques. App Service deployment may be cheaper for some workloads, but it sacrifices some serverless benefits.

While FaaS can offer a clean abstraction that fully removes servers, users who dig down start to see what looks like a little server. Everything is there, including multiple CPUs, OS processes, network devices, etc., which is what makes compatibility with existing software possible. However, for programmers working in high-level languages such as JavaScript or Python, this is usually hidden, becoming visible only to those who dive into system programming.

We note that there are some server-like configuration knobs in all FaaS offerings. Memory configuration has been there from the beginning, allowing the FaaS execution instance to be sized in increments of 128 MB; scaling the memory also scales CPU performance proportionately. Over time, Lambda has offered ever larger memory configurations, and with the larger memory configurations, compute now scales to multiple CPU cores. This means that selecting a memory size seems quite a lot like selecting a server size. Misconfiguration can lead to idle resources, and this is another way that one can argue that FaaS provides a flawed serverless abstraction.

2.5.2 Managed Compute

A number of services aim to make it easier to run existing (server-based) applications in the cloud by standardizing and automating many operational matters. We call this class of products *managed compute.*

Azure App Service is one example of a managed compute service. Its unit of deployment is an *application service*, a program that listens on a port and responds to web requests. Users can configure the App Service with autoscaling, so that it adds more servers as it receives more requests. The App Service includes load balancing functionality, so that to the outside, a pool of servers appears as a single endpoint. By sitting on the path of incoming requests, the App Service gains insight into the responsiveness of each server. At present, this appears to be used to remove or restart unresponsive or failed servers, though in principle it could be used for autoscaling.

AWS Elastic Beanstalk offers functionality similar to Azure App Service. However, whereas Azure describes App Service as serverless, AWS does not describe Elastic Beanstalk as serverless. Is one cloud provider right and the other wrong? In our view both are justified—these managed compute services are both very much like server automation, but for some applications, they also work much like FaaS, so they can provide a serverless solution.

For example, consider a stateless web service that executes many short tasks, receives enough load to require a large pool of servers, and sees relatively slow load fluctuations. It will run similarly in a managed compute setting as in a FaaS setting. There are some minor differences in how one writes a function in a web framework and a serverless framework, but the programming model is similar: You write a snippet of code that runs for a short period of time somewhere in some big pool of compute. Also, operationally, the managed service can sometimes do basically the same thing as the FaaS service, even though it may be a little more complex to configure.

Operational differences start to appear when there are significant load fluctuations, as may be the case when a service receives sudden bursts of activity or tasks that are long-running or resource intensive, e.g., for heavy data processing. Here, the technology underlying FaaS can do a much better job of matching resources to the load than a server provisioning approach. FaaS is also much more suitable for services that run only intermittently, since it scales to zero whereas managed servers must run at least one instance to maintain availability.

One advantage that managed compute platforms presently have over FaaS is that costs can be lower when server utilization is high. We imagine that this could be corrected by innovations in the FaaS business model, e.g., by offering discounts to services that maintain a consistent or otherwise predictable level of usage.

We have classified Google Cloud Run as a managed service along with AWS Elastic Beanstalk and Azure App Service even though it has some characteristics that give it stronger serverless characteristics. Cloud Run was introduced after FaaS became popular, and the influence shows. We believe it was designed from the start to embody the features that make FaaS compelling, like scaling to zero and fine-grained pay-as-you-go billing. The underlying infrastructure of Cloud Run likely looks much like that of FaaS services, but instead of writing applications for a custom FaaS framework, users write applications using standard web service frameworks (e.g., Django for Python, Spring for Java, etc.). Compared to FaaS, Cloud Run offers improved compatibility with existing software ecosystems as traditional web applications can sometimes drop right in. Unlike other managed application services that require the customer to configure autoscaling by setting CPU utilization targets and selecting scaling intervals and strategies, Cloud Run removes the knobs, presumably replacing them with improved automation. Cloud Run also charges according to execution time, like FaaS, rather than by the number of instances provisioned. (Like AWS Lambda, it also allows customers to provision a minimum number of instances.)

Cloud Run exhibits all of the essential characteristics of serverless computing. However, we have placed an asterisk on abstraction (✓*) for it because Cloud Run instances may be processing multiple requests at the same time, and such requests may interact when they are handled by the same server. In some cases, the user may also need to configure a concurrency limit specifying how many such concurrent requests should be allowed on each instance.

Managed compute predates FaaS but has evolved to adopt its pricing and autoscaling characteristics. In the case of Cloud Run, it can be operationally indistinguishable. Azure App Service has adopted the serverless label, though it is not clear to us that it has evolved much to incorporate serverless characteristics. It remains similar to AWS Elastic Beanstalk, which is not presently marketed as serverless.

2.5.3 Container Services (Hosted Kubernetes)

A container is a form of lightweight virtual server. The concept gained popularity through OS-based virtualization, which allows an operating system to host multiple isolated execution environments, each of which looks much like an independent operating system to the application running on top of it. In contrast to traditional VMs, which can take minutes to boot, such containers can often start in under a second.

Container orchestration techniques help manage the process of running large numbers of such virtual servers on an underlying server pool. Google developed container orchestration technology for internal use for many years before launching Kubernetes, an open-source derivative [83]. Due to advances in system-level virtualization, container services can now also be provided with traditional VM isolation, which can provide both security and performance benefits over OS-based virtualization [9, 222, 265]. This is particularly valuable in the public cloud setting.

Both AWS Fargate and Azure Kubernetes Service are container orchestration platforms that present themselves as serverless. Meanwhile, Google Kubernetes Engine offers a similar service without calling it serverless. This is the sort of discrepancy that can make one wonder whether serverless is a meaningful technical concept or simply a marketing term.

Evaluated against our essential characteristics of serverless computing, container orchestration does not measure up particularly well. First of all, the abstraction of servers definitely remains. Autoscaling is provided, but not down to zero—a service must run at least one instance to remain available, and multiple instances are often necessary to meet high-availability needs. Autoscaling also requires configuration, similar to that required to scale servers in managed application environments (see Section 2.5.2). Billing is based on the number of instances provisioned. It is possible to provision fractional resources, e.g., a fractional CPU share, but the minimum billable increment period is one minute.

So what is serverless about container orchestration hosted by a cloud provider? Configuring servers to provide container orchestration can be an involved project. Not only is the software complex to configure, particularly with regard to networking, but scaling also requires managing resources in the underlying server pool. Hosted container orchestration thus relieves operators of significant administrative burdens, among them maintaining an up-to-date operating system kernel with the latest security patches. It also can allow statistical multiplexing across customers, which can improve utilization in ways similar to FaaS. AWS uses the same underlying Firecracker [9] virtualization platform for its Lambda FaaS service and Fargate Kubernetes services, suggesting a certain fundamental similarity between the two.

Kubernetes offers two ways to run containers: as applications and as jobs. Applications are instances of indefinite lifetime and thus closely mirror a traditional server. Jobs are instances that run for a period of time and then terminate. They can work, in effect, like a coarse-grained version of FaaS for workloads where startup costs comprise a small portion of the overall task execution time. This is common, e.g., in analytics.

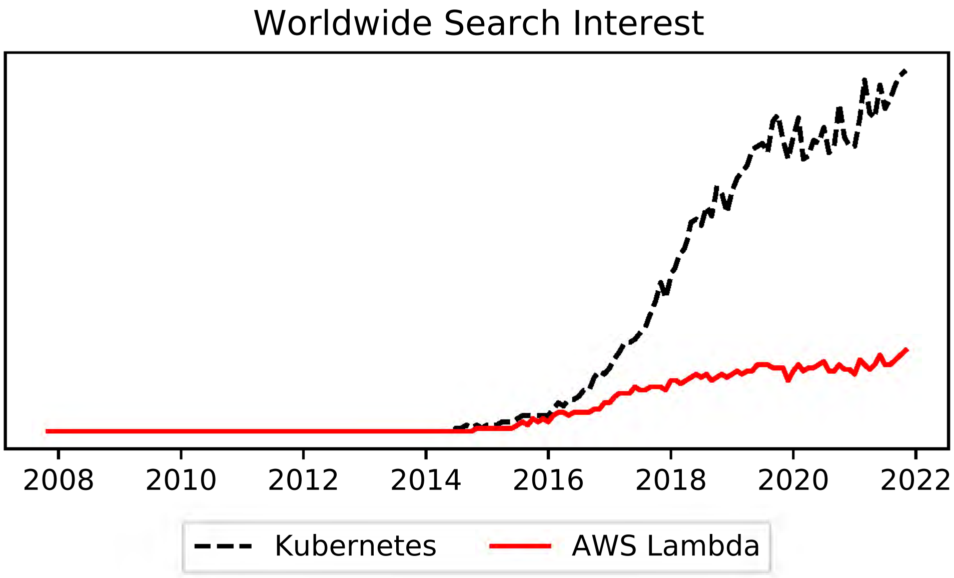


Figure 2.8: Google Trends data comparing the market-leading container orchestration and serverless FaaS technologies. Both Kubernetes and AWS Lambda started at around the same time, but Kubernetes shows higher sustained growth in search interest. We apply 150% scaling starting in October 2020 to account for a discontinuity in the data that appears to be an artifact.

Container usage has seen tremendous growth in the past few years. Figure 2.8 compares search interest in Kubernetes to AWS Lambda, showing that while both have grown, containers have gained in popularity more quickly. One reason may be that containers provide an easy migration path for existing server-based applications. These can be redeployed by an operations team, often with no code changes. FaaS, on the other hand, usually works best with new applications.

We also note that container orchestration can be complementary to serverless because it solves a somewhat different problem. While serverless technologies like FaaS provide an abstraction that hides servers, containers provide flexible, lightweight virtual servers that can make implementing serverless easier. This approach has been taken by many open source FaaS platforms (e.g., [34, 231, 236, 300]).

AWS and Azure call their hosted Kubernetes solutions serverless, whereas Google does not. Do these offerings differ from one another in some substantive way? In our view, the technologies are the same, and it is the marketing that differs.

Using our definitions, "Serverless Kubernetes" basically translates to "abstracting away servers with lightweight virtual servers," which is exactly what container services are doing. Traditional server management goes away, and applications may gain access to autoscaling benefits similar to those provided by FaaS, albeit with a programming model that involves servers. To those in operations roles, describing hosted container orchestration as serverless is useful for understanding it. The label is probably less meaningful to software developers, who stand to benefit more from an abstraction that removes servers and server-like concepts from the programming model, which container orchestration definitely does not.

2.5.4 Application Platforms

Application platforms can claim to have offered serverless computing before it became popular by that name. Rather than supporting arbitrary use cases, they are designed to make it easy to build and operate specific classes of them. Programmers write code using an application framework, e.g., a web application framework or a mobile backend framework. Such frameworks typically provide standard components for common needs, such as user authentication, mobile push notifications, or validating web form input. An application platform is, in essence, a hosted application framework. This type of offering is also known as Platform as a Service (PaaS).

Google Cloud stands alone among major cloud providers in offering application platforms. Other cloud providers offer similar pieces of functionality but do not package them in the same way. Firebase was started in 2011 and acquired by Google in 2014; it originally focused on supporting backends for mobile applications. Google App Engine launched in 2008 and offered a simplified way to build scalable web applications in the cloud.

Firebase meets all of the essential characteristics of serverless computing. However, it merely describes two of its components as serverless: its database and its FaaS implementation. It doesn't label the entire platform that way, even though all the included services abstract away servers, provide autoscaling, and have pay-as-you-go pricing. These include authentication, messaging, a content delivery network, machine learning, and storage.

App Engine has always had excellent autoscaling at the high end but added the ability to scale down to zero only later, likely based on the success of FaaS. This legacy is still visible in the pricing model, which is based on provisioned instance count. When instance utilization is high, which is typically the case at large scale, then provisioned instance count and execution time are closely proportional; at small scale, there can be significant discrepancies. In this way, App Engine seems to scale up better than it scales down. It seems that App Engine might have had the potential to spark the serverless movement, having launched well before AWS Lambda and with broadly similar aims. In Section 2.9.5, we discuss why this did not happen, and use the comparison to help support our characterization of serverless computing.

2.5.5 Event Services

FaaS is closely identified with event-driven programming, a well-established integration pattern [275]. Event-driven programming can be flexible, making software easy to extend with new functionality. It can also help achieve desirable execution characteristics, including robustness in the face of load fluctuations, high throughput, and good utilization. These practical observations are supported by various research works, including Click [233] and SEDA [434].

Distributed systems typically move events around using messaging infrastructure; the basic patterns include shared queues and publish/subscribe. Each cloud provider has one or more offerings, such as Azure Services Bus, Google Cloud Pub/Sub, or AWS SQS. All these have the essential characteristics of serverless computing: They offer a data-oriented abstraction, have excellent scaling characteristics, and are charged based on units transferred.

Additional event management services include Azure Event Grid and AWS EventBridge. These connect external event sources to cloud message services and can work with third-party services or a customer's own applications. Event Grid and Event Bridge both satisfy the essential characteristics of serverless computing.

2.5.6 Workflow Services

Workflow services are a complement to FaaS which all the major cloud providers have adopted. AWS Step Functions, Azure Logic Apps, and Google Workflows all offer "low-code" programming that allows users to specify a business process by drawing a graph of interconnected steps, each of which might be implemented by a FaaS function or some other cloud service. Under the hood of workflow services, one finds an interesting and general-purpose technology: state machines that are reliable, serverless, and fully programmable [25, 353, 392].

Even though their definitive forms have textual representations, the programming languages used to define workflows appear to be targets for visual tools rather than something that one would write directly as one does the code of a FaaS function. The underlying system interfaces for workflow languages are web services and cloud provider APIs. Unlike in FaaS, there is no underlying x86/Linux or other architecture and operating system interface. Workflow services are also autoscaling and charge for each state transition or step. Thus, they meet all of our criteria for serverless computing. Though their languages are quite general, their cost and performance characteristics probably limit them to linking together functionality written in other languages. Still, they show that FaaS is not the only model for writing serverless programs, and they offer a proof point illustrating that serverless programs need not be stateless.

2.5.7 API Management

Once created, a FaaS function can be invoked from anywhere using the cloud provider APIs. Many applications, however, also want to expose a web service of their own definition, apply their own security policies, or monitor usage and API health. This is where API management comes in. Google API Gateway, AWS API Gateway, and Azure API Management all provide these capabilities. They have all of the essential serverless characteristics and are often used with FaaS. However, they can equally well be interposed in front of server-based services.

An interesting emerging class of APIs is based on GraphQL [168]. GraphQL emerged as a response to the proliferation of APIs as well as their increasing richness and complexity. It provides a uniform abstraction that allows clients to bundle queries to multiple services in a single request and to perform filtering and certain kinds of join operations. GraphQL also supports subscriptions, allowing clients to receive incremental updates to query results in near real time as system state changes. As of this writing, AWS AppSync is the only GraphQL service offered by a major cloud provider. Its core functionality has the essential characteristics of serverless computing, but it also features a cache, which is priced in a non-serverless way based on instance size. We touch on the challenge of serverless memory in Section 2.9.3.

2.5.8 Object Storage and File Systems

Cloud object storage is one of the cloud's original product offerings. It is also a foundational building block, with variants offered by all major cloud providers: Azure Blob Store, Google Cloud Storage, and AWS S3. Even at its launch in 2006, S3 fit the essential characteristics of serverless computing perfectly. Other products, by imitation, do so as well.

The design requirements for AWS S3 were scalability, reliability, speed, low cost, and simplicity [26]. The first and last of these lead directly to serverless computing, whereas the others are generally aligned with it. Abstracting away servers is most directly connected to simplicity—a cloud storage API need know nothing about the underlying server infrastructure. Such abstraction also supports scalability, which is also supported by serverless autoscaling. Reliability, or robustness, is something we have referred to as a potential fourth essential characteristic of serverless computing (see Section 2.4).

Cloud object storage differs from file systems in a number of ways. It does not offer a hierarchical naming scheme but rather a simple two-level one. Objects are also immutable: Once written, they cannot be modified or appended to, only replaced in full. In addition, the standard interface for cloud object storage is via web services.

Another class of cloud storage, developed earlier and for Google's internal use, looks more like a traditional file system. The Google File System (GFS) [164] and its successor Colossus [195] were developed by Google for internal use. Colossus has a hierarchical name space, a client library interface, and a data model that allows mutations and file appends, including concurrent mutations and appends from multiple clients. The abstraction is influenced by the details of the implementation, specifically, replication across servers. Colossus files may contain blank records, duplicates, and undefined regions. The Colossus abstraction also leaks details about the data layout laid out on servers; for example, it allows clients to specify the replication factor used for redundancy. Colossus underlies Google Cloud offerings [195], but it is not offered as a cloud service itself. If it were, we would classify it as missing some elements of serverless abstraction while meeting its scalability promise.

AWS Elastic File System (EFS) and Azure Files are serverless implementations of distributed file systems that implement standards such as the Network File System (NFS) [185, 348] protocol and the Common Internet File System (CIFS) [107] protocol, which are broadly similar. Client support comes standard in major operating systems, and FaaS platforms such as Azure Functions and AWS Lambda now provide direct connectivity with these file system implementations. When we started this work, both EFS and Azure files exhibited significant deviations from the essential characteristics of serverless computing: Most notably, autoscaling was slow. Performance was also tied to space utilization or required purchasing a fixed amount of capacity. It appears that both platforms have evolved to remove limitations that made scaling awkward and have provided fine-grained billing models that distinguish between data storage and data retention. They have also both adopted the serverless label.

Chapter 3 details our studies of file systems in a serverless setting. This work shows that integrating a file system with a FaaS platform can provide performance greater than previous distributed file systems. This is possible while maintaining full compatibility with standard POSIX, which allows some existing server-based applications to run with all of the scalability of FaaS.

2.5.9 Database Services and Key-Value Stores

Serverless computing is closely associated with stateless computing, but serverless state management has a history in the cloud that precedes the emergence of FaaS. We focus this section on online transaction processing (OLTP) databases, leaving the discussion of data processing for analytics to Section 2.5.10.

AWS DynamoDB [132, 374] is a key-value database that launched in 2012 and today has all the essential characteristics of serverless computing. It was derived from Dynamo [122], a technology developed previously for Amazon's internal use, which featured scalability and robustness as key design criteria. Dynamo and DynamoDB were star examples for the NoSQL movement [389], which prioritized these operational imperatives over rich functionality. Autoscaling and pricing in DynamoDB have evolved over time. In 2018, as FaaS gained in popularity, DynamoDB added pay-as-you-go pricing based on the number and type of requests.

Google Cloud Datastore is a NoSQL database released in 2008 as a part of App Engine. It was later positioned as a separate product. Cloud Datastore meets all the essential characteristics of serverless computing, though its autoscaling can be slow. At the time of this writing, Google describes Cloud Datastore as a legacy product and directs interested users to instead try Firestore, a component of Firebase.

Azure CosmosDB is presented as a NoSQL database that offers multiple data models (confusingly, these include a dialect of SQL). Users see an autoscaling, provider-managed, globally-distributed database. CosmosDB meets most of essential characteristics of serverless computing: It offers various pricing models, including a "serverless" model with fine-grained pay-as-you-go billing. Azure CosmosDB launched in 2017.

These examples seem to show that NoSQL and "noservers" go hand-in-hand. These pre-existing stateful serverless services have proven to be an important complement to FaaS, and some of them have evolved to incorporate some elements from it. This includes more fine-grained pay-as-you-go pricing and likely also under-the-hood scalability improvements.

What about traditional SQL databases? All cloud providers now offer managed versions of the most popular open source databases: PostgreSQL and MySQL. These databases are designed to run on a single server and to scale "up" rather than "out" by migrating to larger machines rather than adding more machines. AWS and Azure have automated the process of scaling a database server up and down and describe the products that do so as serverless—Azure SQL Database Serverless and AWS Aurora Serverless eliminate server management for customers. However, they do not meet many of the essential characteristics of serverless computing. For example, autoscaling is available, but only up to the scale of the largest server instance. Pricing is also based on the size of the instance provisioned, rather than on, say, the time spent executing queries against it. This may make sense since users are paying for buffer cache, but it doesn't meet our criteria for serverless computing (see Section 2.9.3 for further discussion of serverless memory). It's hard to say whether these databases abstract away servers. Programmers interact with them via SQL, which has no concept for servers or anything similar. However, because these services are limited by the size of the largest server, programmers writing scalable software may need to reason about this limitation, potentially splitting their databases among multiple instances. We have categorized serverless abstraction as "N/A" for these databases.

Scalable OLTP SQL databases represent a notable gap in the cloud database offerings. While Azure CosmosDB supports a SQL API, its capabilities do not approach those of a relational database. Notably, joins are scoped to a single JSON document [215]. Google Cloud Spanner offers global scale in a relational SQL database [50, 113]. At the time of this writing, Cloud Spanner requires users to provision capacity directly, though Google also provides an open source autoscaling tool that customers can use to adjust this configuration automatically. We do not see a fundamental reason why SQL databases could not be offered as serverless products. As Stonebraker has opined [387], it has been easier to achieve scalability with NoSQL databases since they are simpler, but we can expect relational databases to catch up. However, as we discuss in Chapter 5, such databases will need to confront the reality that distributed databases have fundamentally different performance characteristics than those provided by large servers.

2.5.10 Big Data Services

Dremel [273], an internal tool that preceded Google BigQuery, provides another example of serverless computing from before FaaS arrived. The aim of Dremel was to provide interactive analytics on large "web-scale" data sets. By taking advantage of the extensive parallelism available in the data center, large volumes of data could be processed much faster than was possible on a single machine or even a traditional cluster. From the analyst's perspective, this looks like scaling up capacity instantly whenever a query needs to run, then scaling it back down again as soon as the query finishes. BigQuery includes its own storage and can also process files stored in object storage, such as Google Cloud Storage and even AWS S3. Query cost is proportional to the amount of data scanned, though capacity provisioning is also possible and available at a discount with committed spending. BigQuery hides the underlying servers behind a SQL-like interface, which completes the serverless abstraction.

Azure offers Data Lake Analytics, which is similar to BigQuery. It abstracts away servers and scales automatically. Its pricing model is based on Analytics Units (AU), which are server slices: as of this writing, an AU is 2 CPU cores and 6 GB of RAM. Since big data processing can often involve a great number of servers, the costs are probably similar in practice to other more clearly serverless models, such as pricing based on the amount of data scanned.

AWS offers three serverless analytics environments. AWS Athena uses the Presto distributed SQL query engine [325] to run queries over data sets stored in S3, charging according to the amount of data scanned. Athena appears to be targeted at interactive users, much like BigQuery. AWS Redshift started as a traditional column-oriented data warehouse and gradually added serverless features [85, 176, 302]. AWS Glue lets users spin up Apache Spark [457] in a serverless way. Glue is targeted at ETL (Extract, Transform, Load) uses but appears capable of running any Spark program.

Stream processing products round out the big data offerings. These allow data to be processed as it comes in, making it possible to update analytics over large data sets quickly: often within seconds. Google Cloud Dataflow and Azure Stream Analytics are examples of these services. While both provide SQL-like interfaces, Cloud Dataflow is interesting because it allows users to write data processing code implementations, such as transformations or custom aggregations, that are then embedded into a larger data processing pipeline. This is similar to the use of functional "map" and "reduce" transformations in MapReduce [121]. From this viewpoint, we can view big data stream processing and FaaS as two alternate embeddings of serverless functions.

This discussion has focused on services offered by cloud providers. In the big data space, it is noteworthy that third parties have also built successful services that layer on top of the public cloud: Examples include Snowflake [115], which provides an elastic data warehouse, and Databricks, which is based on Apache Spark [457] and has continued to develop innovative analytics products [38]. Both of these companies offer some of their products with serverless pricing and scaling [33, 378]. Dataprep by Trifacta [117] is a serverless tool used to explore, clean, and prepare data for analysis and machine learning. It is integrated with Google Cloud, and shows how serverless computing can encompass a variety of business models.

One interesting aspect of serverless big data processing is that it offers both latency-sensitive "interactive" workloads and less sensitive "batch" workloads. The two are complementary: Interactive workloads benefit from getting access to many resources for short periods of time, but keeping a cluster busy by statistical multiplexing of such workloads can be difficult; when the resource pool is shared with batch workloads, then these can be placed on hold, or preempted, to help accelerate the interactive workload. The batch workload can benefit in this scenario since it uses capacity that would otherwise be left idle in anticipation of spikes of interactive work. This principle can probably be extended to other areas, such as FaaS. Today's serverless platforms are siloed, i.e., to our knowledge, analytics services do not share underlying servers with FaaS. The Lambada [285] and Starling [317] research projects have already shown that FaaS can serve as a platform for analytics, suggesting that FaaS could perhaps evolve to support an optimized combination of interactive and batch processing.

2.5.11 Machine Learning Services

Azure markets several of its machine learning products as serverless. These include Azure Cognitive Services for natural language processing, and Azure Bot Services, which is used to power chat bots. Azure Machine Learning is a collection of hosted tools that is less explicitly described as serverless, though it is listed among Azure's serverless offerings. With the exception of Azure Machine Learning, which charges based on running instances, Azure's machine learning services all meet the essential characteristics of serverless computing.

Cognitive Services and Bot Services are examples of specialized software offered with consumption-based billing. They are also a form of SaaS aimed at developers. Google also offers many similar AI services for natural language processing and vision, including specialized variants targeting specific use cases, such as invoice parsing, or industry verticals, such healthcare or lending. Google does not describe these as serverless, instead simply presenting them as APIs with unit-based pricing. They all meet the essential characteristics of serverless computing, and they are so numerous that we have opted not to include them in Table 2.1.

These machine learning services provide the sort of high-level functionality that raises the question of where serverless computing ends and SaaS begins. Does a hosted cloud spreadsheet such as Google Sheets or Microsoft 365 Excel provide serverless computing? We do not think it is helpful to classify them as such. The serverless concept is more useful when scoped to those technologies that software developers use to build and deploy applications. We will elaborate on this perspective in Section 2.8. We view machine learning APIs as building blocks rather than end-user applications; to us, it seems reasonable and useful to call them serverless.

2.6 Limitations of Serverless Computing

Even though serverless computing is proliferating, many applications are still better suited to servers [189, 216, 356]. Commenting on the limitations of a rapidly advancing technology is a fraught endeavor, but we have attempted to do so here since it informs our research activities. We start out by reviewing the limitations of selected classes of serverless technologies.

2.6.1 Limitations of FaaS

FaaS is a restrictive model with a number of obvious limitations. Execution time is limited, though seemingly not in a very fundamental way. Functions initially could run for a maximum of one minute on AWS Lambda; this limit has increased to 15 minutes, and Google Cloud Run now supports FaaS functions with a timeout of up to one hour. We imagine that if customers want this limit to be still higher, cloud providers will comply.

Limitations on code size, memory size, and CPU power have all similarly been relaxed. For example, AWS Lambda was originally limited to a 250 MB zip file, but now supports 10 GB Docker images. Maximum memory started out at 1.5 GB and now is 10 GB; CPU cores were limited to two, and now can go to six. AVX2 instruction support, only sporadically present in the past, is now standard. For years, Lambda supported only the x86 instruction set, though it supports ARM now as well. GPU support remains missing and faces challenges because GPU virtualization is not yet well established. However, these limits may too be overcome.

Several more fundamental limitations of FaaS follow from the abstraction it provides. Functions have ephemeral state only and do not accept incoming network connections. Though workarounds for both these limitations exist [423, 426, 432, 463], in our view, they remain defining aspects of FaaS (though not of serverless computing). Adding networking to FaaS turns it into an on-demand server service, i.e., a container service (see Section 2.5.3). Especially since some of the underlying systems technologies have converged [9], it is ephemeral state and limited networking that sets FaaS apart.

The statelessness of FaaS represents two distinct limitations: FaaS itself has no durable storage, and cached state is not addressable from outside the function instance. Even when state exists, there is no way to access it on demand.

A lack of durable storage means that any state that needs to survive past when the function returns must be sent out of the FaaS system. Infinicache [426] replicates data across function instances and even uses erasure coding to reconstruct state if some of the instances are removed. Even so, data retention is not guaranteed. A cloud provider may remove any, or all, of a function's instances at any time, so it is impossible for FaaS alone to offer a reliable solution for long-term storage. Instead, FaaS can be used in combination with object storage, file systems, or databases, all of which can provide state to complement the stateless computation in FaaS. A cost arises, of course, from transferring data across the network to these other services.

The ephemeral state in FaaS functions is also trapped, in the sense that there is no way to find it should one ever desire to read or update it. This is because a new FaaS invocation may be routed to any available function instance and can also be served by a newly created one; state in FaaS is not only ephemeral but also unnamed and unaddressable. One way to remedy this situation would be through some form of session affinity. This could be an affinity to session state, where the platform attempts to route multiple requests from the same client to the same function instance. The affinity could also be based on function arguments, as we have previously proposed [355].

Another class of FaaS limitations relates to the overhead of invoking a function. There are two cases here: cold starts, where a new execution instance needs to be provisioned, and warm starts, where a cached execution instance can be reused.

Cold starts are required whenever the FaaS system creates new instances. After provisioning the execution environment, it must be loaded with the code, configured on the network, and configured with security privileges. The language runtime must be started, libraries loaded, and then user-provided initialization code may need to run. All of this consumes resources and takes time: Depending on the cloud provider and runtime environment, it requires between 100 ms and several seconds [429] before user code begins to run. Both cloud providers [9] and academic researchers [14, 296, 368] have made strides toward reducing these overheads, which allows higher utilization and lower response times.

Warm starts can still involve significant overheads, and function invocation on commercial FaaS platforms is still much slower than calling a web service running on a single server. For example, AWS Lambda invocations take about 25 ms [429], whereas a web service can often respond in under 1 ms. We believe that the queuing mechanisms are responsible for these overheads, though there could be other factors as well, e.g., whatever mechanisms are used to route work in a multi-tenant environment. Since the details of these mechanisms are proprietary, we have not been able to analyze them. It does appear, however, that warm start overheads translate directly to customer costs, which can make FaaS uncompetitive for short-running functions. To see this, consider that AWS Lambda bills at \$0.2 per million invocations plus \$0.0000166667 for every GB-second of runtime. Since the smallest unit of runtime is one millisecond, and the smallest unit of memory is 128 MB, the smallest billable increment of execution is $\$2.13 \times 10^{-9}$. This is \$0.0021 per million units. For any request taking less than 93 units (e.g., anything less than 93 ms with a 128 MB function), Lambda will charge more for the invocation than for the execution time. While FaaS looks very fine-grained in comparison to VMs, each function invocation still incurs a cost equivalent to hundreds of millions of CPU cycles. From this perspective, FaaS still incurs a huge cost overhead. Researchers have proposed a number of solutions [14, 383], but it is not clear to what extent they meet the needs of commercial deployments.

2.6.2 Limitations for Object Storage, Key-Value Storage, OLTP Databases, and File Systems

The stateful serverless offerings each have distinct strengths and weaknesses. Object storage, key-value storage, and file systems all have limited ability to perform logical operations on the data they store, which means they must be used in combination with some compute service, such as FaaS or some server-based service.

Object storage offers low costs for long-term data retention as well as low-cost throughput. However, access costs for storing and retrieving objects are high, and the data model is relatively simple. Some key-value stores support richer data models through structured values such as maps, but they have a relatively high cost for both data access and retention. The same is true, in broad terms, of file systems.

OLTP databases have a richer set of capabilities but more limited scalability. Serverless implementations of PostgreSQL and MySQL are offered as part of AWS Aurora [200], but scale for these products is measured in fractions of a server—there is no way to scale to multiple servers. The recently released CockroachDB Serverless [385] product appears to overcome this limitation with its distributed SQL implementation.

The landscape of serverless state management solutions has clear gaps—some functionality is simply missing, while other functionality is just much cheaper to provide using servers. We have already seen key-value stores and file systems improve in recent years, quite possibly in response to the needs of FaaS workloads. While this category includes some of the earliest and most ubiquitous products, such as object storage, it seems that it also offers ample opportunities for innovation.

2.6.3 Limitations for Big Data Analytics Systems

Serverless big data processing systems are competitive with their server-based variants, and in this sense, they have no serious limitations. One potential architectural downside is that serverless systems may physically separate compute and storage; for example, data in object storage may need to be copied over the network before being analyzed. Server-based solutions may be better equipped to analyze data near to where it is stored. Some serverless systems, like Big Query, also have integrated storage. Another alternative is for storage to support code execution; ZeroVM [330] does this and also allows operator push-down.

In multi-tenant settings, guaranteeing resource availability can also be a problem. Google Big Query allows customers to reserve capacity, ensuring that it will be available any time they need it. While paying for idle capacity is at odds with a key characteristic of serverless computing, it seems to be a feature that fulfills a business need.

2.6.4 Implications

Serverless computing works well today for certain sorts of applications [309], but is still beset by numerous limitations. For example, it suits needs like event-driven processing or glue code between services well. It also works for general-purpose web services as long as they have modest demands for performance or efficiency. Applications need not be stateless, but state management needs must fit the profile for object storage or key-value storage. Those hoping to obtain high-frequency and low-cost access to in-memory data are out of luck. Similarly, those hoping for a mature and full-featured OLTP database may need to turn to a server-based product.

An area where serverless shines is analytics and big data processing. Here it benefits from an ecosystem of competing solutions that have evolved over the years to meet the needs of demanding users.

2.7 Serverless Computing Research

Our approach to understanding serverless computing has focused on analyzing the industry trend. Industry developments have also inspired a great deal of academic research, and in this section, we provide an overview of some it. We divide our research survey into several topic areas and highlight selected work within each.

As we have stressed, the industry serverless trend is about much more than FaaS (see Section 2.5). However, serverless computing research remains closely associated with FaaS, and so FaaS is the main theme of the survey that follows. Time will tell whether other serverless cloud products attract as much attention as FaaS has.

2.7.1 Analysis and Surveys

A number of authors have analyzed the emergence of serverless computing, seeking to explain the trend, as we have done here. Baldini et al. [55] and Lynn et al. [259] provide early surveys, van Eyk et al. [416] place serverless in a historical context, and Castro et al. [92] provide a more recent survey that includes a comprehensive analysis of FaaS. Jonas et al. describe serverless in terms of a new programming model for the cloud [216], and Schleier-Smith et al. emphasize the role of serverless computing as the next phase of cloud computing [356]. Hellerstein et al. [189] focus on the limitations of FaaS, thus highlighting targets for innovation.

2.7.2 Economics

Several of the high-level articles discussed in Section 2.7.1 touch on the economics of serverless computing. Adzic et al. [8] use a customer case study to look at the costs of serverless computing. Eivy et al. [136] warn of hidden costs of serverless computing. There is also nascent work on building economic models of provider and customer incentives [254] as well as utility-based pricing [178].

2.7.3 System Improvements for FaaS

A large category of serverless research involves system improvements to FaaS. There is a tension between providing isolation, efficient multiplexing, and low-latency performance. OpenLambda [190] and McGrath et al. [268] both developed early prototype FaaS systems that mirrored the inner workings of FaaS platforms and helped illustrate this research challenge.

One manifestation of the tension is in cold starts. SOCK [296] uses various systems techniques to reduce these, particularly for FaaS applications that use libraries with high initialization costs. Catalyzer [131] takes on the same challenge, using checkpoints to start functions instead of executing their initialization code. Xanadu [119] provides techniques for mitigating cascading cold starts, and Mohan et al. [281] discuss techniques for preallocating resources such as network interfaces to reduce cold start times.

FaaS also incurs cold start latencies and other overheads from the underlying operating system and hypervisor. Firecracker [9] is a lightweight microVM technology developed by AWS that reduces the startup times and memory requirements of VM isolation. Koller and Williams [234] have suggested using unikernels with FaaS instead of traditional operating systems in a further bid to improve efficiency. There are also alternatives to using VMs for isolation. Faasm [368] provides lightweight isolation based on WebAssembly [181]. Alto [245] generalizes lightweight virtualization to other managed runtime environments.

Isolation not only creates startup costs but ongoing runtime costs as well. Young et al. [449] study the performance overheads of gVisor [180], which is used by Google's serverless products. Anjali et al. [32] compare serverless isolation mechanisms, including Linux containers, gVisor, and Firecracker microVMs.

Even when no cold starts are involved, the latency of FaaS function invocation can be too high for some applications. Contributing factors include overheads of passing data, queuing overheads, and scheduling overheads or delays. Sonic [262], SAND [14], SEUSS [84], and Cloudburst [384] all address various aspects of these slowdowns.

Work on scheduling includes that by Kaffes et al. [220], which uses a centralized scheduler with a global view to mitigate imbalances. FnSched [391] offers another scheduler that aims to improve latency and utilization, and Caerus [460] provides scheduling for serverless analytics. Work by Mahmoudi et al. [263] describes an algorithm for adaptive function placement.

A diverse assortment set of other work seeks to improve FaaS. Shredder [462] embeds FaaS computations with object storage. Faa$T [338] provides a provider-managed cache for serverless applications. Particle is a network overlay suited to the burstiness of serverless computing [402]. Gupta [177] et al. demonstrate straggler mitigation using error-correcting codes. Kappa [463] provides fault tolerance and extended execution times by checkpointing and restarting FaaS applications. InfiniCache [426] shows how to use erasure coding to build a cache from idle FaaS instances. Harvest VMs [464] allows FaaS to run using resources momentarily left idle by traditional server VMs.

2.7.4 Stateful Serverless

Augmenting FaaS with state has been the subject of considerable research. This includes Chapter 3 of this work, which describes the FaaSFS distributed file system.

One application with specific state management requirements is analytics, which requires ephemeral storage to pass intermediate results between functions [230]. Pocket [229] provides a solution to challenges in this area. Though managing state for analytics can be challenging on account of the volume and transient nature of the data, its simple and well-defined usage patterns lend themselves to optimized solutions.

A more general challenge arises in managing changing application state, which is often subject to certain consistency requirements. Serverless computing gives coordination-free techniques an opportunity to shine because they have provable advantages at scale [187]. Cloudburst [384] is a stateful FaaS system that integrates with the scalable Anna [443] key-value store. It provides local caches in function instances and transactional causal consistency [442]. FaaSTCC [256] is another system that provides similar guarantees.

An alternative approach is to use an underlying logging infrastructure to represent state. Logging involves coordination, but it can provide strong consistency and better throughput scaling than distributed protocols such as two-phase commit [2]. Beldi [459] and work by de Heus et al. [193] both provide transaction mechanisms that integrate FaaS and underlying storage. Boki [212] and Retro-λ [271] also make use of an underlying log to manage state.

Azure Functions [48] includes “durable functions” in its production offering. Durable functions use a checkpoint mechanism to allow long-running execution on top of a FaaS runtime. In this programming model, state can be maintained reliably and for long periods of time within the functions themselves. Burckhardt et al. [82] provide a formal model of durable functions and show that various implementations are possible.

Stateful serverless must reckon with faults. AFT [383] provides a fault tolerance shim that can be interposed between a FaaS environment and underlying storage, providing atomicity guarantees. Ray [284] is not derived from FaaS but offers similar scalability and fits under the broader definition of serverless. The platform has served as a proving ground for various novel fault tolerance approaches [431, 465]. Other work on stateful serverless computing includes SFL [76], a compiler for generating stateful serverless applications.

2.7.5 Autoscaling, Optimization, and Quality of Service

Autoscaling is a defining characteristic of serverless computing, so a great deal of research touches on it in some way. Autoscaling must balance quality of service and cost, and the work we highlight here relates directly to this tradeoff. Even with FaaS, customers are still required to configure some resources, notably the “memory size,” which serves as a proxy for instance execution resources. Sizeless [134] and COSE [11] analyze functions as they run, attempting to find optimal resource configurations. Winzinger and Wirtz [439] also provide a model for FaaS execution. There are multiple approaches to quality of service: Sequoia [396] targets policy goals, whereas Atoll [371] focuses on latency objectivs.

An eclectic mix of work rounds out the early autoscaling-focused efforts. Yussupov et al. [456] study how to reengineer existing applications for scalability, introducing the notion of "serverless parachutes" that are used only under exceptional load conditions. Spock [175] uses both server VMs and serverless functions to meet elasticity and cost goals. Anna [441] provides autoscaling tiered storage, seeking to optimize for both cost and performance goals.

2.7.6 System Abstractions

Even popular serverless abstractions such as FaaS are not standardized across cloud providers. SPEC-RG seeks to address this and proposes a reference architecture for FaaS [417]. Adopting a broad view of serverless computing also invites proposals for standardized abstractions of storage and communication, which round out the core features of an operating system. Pemberton et al. [316] outline this need, whereas ServerlessOS [16] offers a concrete proposal. LegoOS [363] provides operating system abstractions for hardware disaggregation. Though not positioned as serverless, it addresses the same core concerns.

2.7.7 Monitoring and Debugging

Many of the tools traditionally used for debugging and monitoring applications do not carry over to FaaS, which creates a need for new solutions. Watchtower [18] monitors runtime invariants for FaaS applications. Borges et al. [72] design and evaluate multiple approaches to distributed tracing. Manner et al. [266] provide a combined monitoring and debugging solution for FaaS.

2.7.8 Formal Methods

Several authors have proposed formal models of FaaS. Jangda et al. [211] and Obetz et al. [297] both introduce formal models of FaaS and event-driven computation. Gabrielli et al. [157] propose the Serverless Kernel Calculus, which is similar and includes a stateful extension. Burckhardt et al. [82] analyze durable functions in the context of a formal model, providing one demonstration of the value of these techniques.

2.7.9 Security

The transition to a serverless model has many implications for security (see [251] and Section 2.9.2). Established techniques such as secure enclaves can be used with FaaS, but doing so requires overcoming various obstacles [167, 329, 409].

Fine-grained isolation in FaaS offers potential security benefits, but it will be difficult for programmers to take advantage of this without supporting tools and techniques. Information flow control [343] provides the basis for some approaches, including Valve [118] and work by Alpernas et al. [19]. Will.iam [349] produces more robust permission boundaries through workflow integration, and Hong et al. [198] suggest a collection of design patterns that can help develop secure serverless applications.

Researchers have found that serverless computing is susceptible to novel forms of attack. For example, Kelly et al. [69] describe “denial of wallet” attacks that exploit the scalability of serverless computing to exhaust the victim’s budget. The Warmonger attack [444] is a type of denial of service attack that exploits multi-tenant infrastructure to introduce abusive activity on a victim’s IPs, leading other services to block them.

Work has also focused on analyzing the security of specific applications, e.g., the OmniBallot online voting system [380].

2.7.10 Analytics

There have been several efforts to apply FaaS to analytics workloads. We touch upon a few examples here and refer the reader to Werner et al. [435] for an overview and comparison of serverless data processing frameworks.

PyWren [217] demonstrated the benefits of simplified cloud programming with a simple FaaS-based framework geared at analytics tasks. Subsequent work by IBM [346] extends it with additional constructs, and Locus [326] showed how to implement shuffling, an important analytics primitive, in a FaaS environment.

Wukong [90, 91] focuses on optimizing analytics tasks, enhancing locality by minimizing data movement across tasks. In a similar vein, HASTE [35] focuses on optimizing serverless DAG execution.

In the database literature, Lambda [285] and Starling [317] both use FaaS to operate on data stored in S3. Flint [227] tackles the same problem using Apache Spark [457].

2.7.11 Benchmarks and Data Sets

Serverless computing stands to benefit from broadly accepted benchmarks. A number of these have been proposed, though a leader has not yet emerged. Contenders include FunctionBench [226], FaaSdom [264], and Serverlessbench [451]. DeathStarBench [159] is targeted at microservices as well as FaaS applications. Work by Martins et al. [267] also proposes a benchmark and uses it to compare cloud providers. Scheuner and Leitner [354] provide a literature review of various FaaS performance evaluations.

The need for new benchmarks is especially evident because serverless computing emphasizes autoscaling. The quality of this autoscaling is often referred to as “elasticity,” a metaphor that suggests it might be described by a simple number or perhaps a relationship between two variables, as is the case in physics or engineering. So far no such metric has emerged, though work by Kuhlenkamp et al. [237] moves in this direction.

2.7.12 Serverless in Practice

Understanding how serverless FaaS platforms work is challenging because the leading products are either partially or entirely proprietary. Early efforts to innovate on FaaS devoted significant effort to understanding how FaaS platforms might be implemented [190, 217]. More recent work by Wang et al. [429] has thoroughly analyzed the major FaaS platforms, documenting their isolation mechanisms, elasticity, coldstart latencies, and container recycling policies. Lee et al. [248] provide another evaluation of FaaS providers.

Other reports and analyses of real-world experiences are valuable as well. Shahrad et al. [362] describe the production workload of Azure Functions as well as policy optimizations that improve efficiency and quality of service. The Wonderless Dataset [140] contains open source serverless applications extracted from GitHub. Eismann et al. [135] review various serverless applications and attempt to explain why and when they are successful. Mohanty et al. [282] evaluate open source serverless computing frameworks.

2.7.13 Machine Learning

FaaS can support machine learning in both training and inference applications. Projects that focus on training include MLLess [347] and LambdaML [213]. Cirrus [89] and Stratum [67] address end-to-end machine learning workflows, which include both training and inference. Inference-focused projects demonstrate serving deep learning models [209] and automatic model partitioning for cost optimality and SLO compliance [450].

GPUs and other accelerators [219] are commonplace in machine learning but are not presently supported by commercial FaaS offerings. Research that addresses this shortcoming includes work on efficient GPU sharing for serverless workflows [350]. Another project, PyPlover [447], is a serverless framework that allows the deployment of GPU code directly to a FaaS environment.

2.7.14 Interactive Work

PyWren [217] popularized the notion that serverless computing could empower end-users by simplifying access to cloud computing resources. Whereas PyWren focused on analytics tasks, gg [149] demonstrated how to offload heavy software build jobs. It also provided a framework for scaling interactive tasks in the cloud. Another example of using FaaS for interactive work is sshell [261], which makes it possible to run shell scripts in the cloud in much the same way as one runs them on a local computer.

2.7.15 Edge and IoT

Serverless computing has generated enthusiasm [42] in the areas of edge computing [366] and the Internet of Things (IoT) [44]. IoT envisions embedded computing and communication in sensors, actuators, and everyday electronic items. IoT devices are often resource-constrained, so they may benefit from offloading computation over the network. Edge computing makes it possible to do this while maintaining low latency: It augments the cloud resources in centralized data centers with compute, storage, or other resources placed at the "edge" of the network, i.e., near devices. Combining edge computing and IoT presents challenges since devices may move and because the resources available at a particular edge location can become oversubscribed. These are the sorts of challenges that serverless computing is equipped for.

This is an active area of research that includes numerous works. Hall et al. [182] suggest an execution model for FaaS at the edge. Gand et al. [160] describe a containerized management solution for deploying serverless code. Pinto et al. [321] propose dynamically moving functions between an IoT device and the edge. Apollo [376] provides a system for runtime function composition and flexible placement, whereas Costless [138] describes an approach to optimization that includes function fusion. LaSS [427] focuses on meeting the needs of latency-sensitive edge applications. Aske and Zhao describe work on supporting multi-provider serverless computing at the edge [41]. In addition to processing data generated at the edge, serverless models can be applied to disseminating information sourced from centralized data centers, as Facebook does with Bladerunner [58].

2.7.16 Network Function Virtualization

Network function virtualization (NFV) [277] decouples network functionality from its physical embodiment in hardware. In some ways, it is the equivalent of VMs for network equipment. There have been multiple proposals for serverless NFV [5, 372], which can be viewed as a logical evolution of NFV. Potential applications for serverless NFV include improved quality of service for 5G networks [96]. Work also suggests that it may be practical to combine serverless NFV and edge FaaS deployments [461].

2.7.17 Other Applications

Serverless computing, like cloud computing, is a general-purpose technology that can be deployed in many contexts. Use cases that attracted attention early on included chatbots [446] and video encoding [148]. Serverless autoscaling also makes sense for disaster response, and several such applications have been studied [154, 272, 312].

Serverless robotics involves motion planning and could occur in the cloud or at the edge [29, 272]. There are applications in the oil and gas industry [204] and in geospatial computing [62]. Virtual environments, including games, are another application area [129]. DevOps, which involves things like software testing, is a bursty workload that stands to benefit from serverless computing [210]. There have also been proposals to use FaaS to enable blockchain applications [99] and to execute FaaS on blockchain infrastructure [163].

Various scientific applications may benefit from serverless computing. In the high-performance computing space, high invocation rates and short deadlines could challenge existing technologies, but FaaS might provide useful benefits [289, 381]. Other examples of scientific applications of FaaS include federated function serving [95], serverless linear algebra [364], and a distributed parallel analysis engine for high-energy physics [239].

2.8 Simplified Cloud Programming

In describing the challenges of scale, we explained how creating software that uses many computers is much more complicated than writing software for a single computer (see Chapter 1). This observation leads us to conclude that simplified cloud programming is the most compelling benefit of serverless computing [216]. Serverless hides the complexity of cloud programming by abstracting away, to varying degrees, the underlying servers, thus promising improved programming productivity.

In this section, we delve deeper into this claim. We find that serverless computing attacks a class of problems that previous improvements in programmer productivity also tackled. This parallel suggests a path toward realizing its full potential.

2.8.1 No Silver Bullet

Among the seminal works of Fred P. Brooks is his 1986 essay "No Silver Bullet" [79]. Writing at a time when programmer productivity had seen tremendous improvements in recent years, Brooks made a point that many people didn't want to hear: future improvements would likely be harder to come by.

To build this argument, Brooks first distinguished between two forms of complexity in programming: essential complexity and accidental complexity. Essential complexity is that which is inherent to the functionality that the program provides, whereas accidental complexity results largely from the complexity and limitations of the underlying machine or programming abstraction.

The idea of distinguishing essence from accident goes back to Aristotle. He described as essential those attributes without which one type of thing would become another type of thing. Accidental attributes, in contrast, could be changed without changing what the type of thing an object is. For example, Figure 2.9 shows four different stools. These come in various shapes and colors; have four legs, three legs, or just one leg; their heights vary, and while many of them have a bar where one might rest a foot, not all do. These attributes are all accidental, however: Altering them does not change the stool into something else. In contrast, if we were to squash one of these stools to two inches tall, compromise its ability to support a person's weight, or remove the horizontal surface for sitting on, then that object would cease to be a stool—it would then be something else.

In programming, essential complexity arises when a program does something complicated. For example, operating systems are generally complex because they have large interfaces and offer many features. A program used by a large insurance company may be complicated because of the various types of policies offered, the variety of alternative terms that may

Figure 2.9: Comparing essential and accidental properties. These stools all share the same essential attributes, but differ in their accidental attributes.

apply to individual policies, and the disparate regulatory regimes that the company must comply with.

Software may involve a lot of essential complexity even when its ultimate functionality affords a simple description. For example, a medical imaging machine like a CT scanner produces pictures of a person's insides. This is easy to say, but complex mathematics govern the reconstruction of images from its sensors, and its software thus is inevitably complex as well.

At the time Brooks wrote his essay, he could claim, fairly in our view, that much of the accidental complexity in programming had been eliminated through a combination of advances. He listed high-level languages as first among these, saying that "[s]urely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility." High-level languages allow the programmer to express functionality using abstract data types and operations on them, without thinking about details such as registers or how values are encoded in memory.

Brooks identified several other sources of accidental complexity that had been mitigated recently. He pointed to hardware limitations, including processing speed and memory capacity, noting that programmers had been investing a great amount of energy and effort into writing highly efficient programs to make the most of limited hardware resources, and that this became less necessary as the cost of computing rapidly declined. Brooks also gave credit to time-sharing for creating an immediate feedback loop, removing the cognitive burden that arises when returning to a programming task after waiting many hours for a batch job to run. Finally, he credited Unix and Interlisp with improving program interoperability, which again eliminated a burden having nothing to do with the functionality the software needed to provide.

Brooks built his core argument on the observation that while accidental complexity might have accounted for over 90% of program complexity in 1966, so that reducing it could have created the 10× gains the industry had experienced, no single cause could account for so large a fraction of programming complexity in 1986. The rest of his essay thus focuses on ways to make essential complexity easier to manage, and in the intervening years, we have seen the benefits of some of the approaches that he advocated come to pass.

2.8.2 The Return of Accidental Complexity

In the decades since Brooks wrote, accidental complexity has been creeping back into programming practice. This might seem surprising since computers keep getting faster and programming languages keep getting better, but as we discussed in Section 1.2, we simply expect more from our programs today. Moreover, we often expect more in ways that are easy to specify but hard to implement. There are numerous examples of this phenomenon. We expect social networks to scale to support millions of members (see Section 1.3). We also expect cloud software to be highly available and accessible at any time (see Section 1.2). We expect to be able to answer simple business questions quickly even when they involve massive data sets. Web search engines, which remain feats of engineering, have conditioned us to think that it is normal to find whatever we are looking for among tens of billions of pages in just a fraction of a second. We expect mobile app software to work on the go and to synchronize itself across our devices and with the devices of others. Some of these expectations involve additional essential complexity, but many do not: We ask for the same functionality, just bigger and better.

Is scale ever a matter of essential complexity? Brooks makes it clear that he views the limited processing speed and memory size of early computers as accidental complexity. We agree with this view and believe that the same logic applies when needs exceed the capabilities of today's computers. *If a problem goes from hard to easy once an improved computer becomes available, then the difficulty was never inherent to the problem and is better attributed to the limited capabilities of the computing system.*

While scale is generally a matter of accidental complexity, there are exceptions. In some situations, physical realities independent of the computer system govern the functionality of the product. For example, if globally distributed users participate in a single financial marketplace, then rules must govern how to prioritize orders that arrive with various delays. Some degree of delay is inescapable on account of the finite speed of light, which is a feature of the world the software runs in. If all users were located close together, such concerns might not be relevant, so we see that in this case scale, or more accurately distance, has introduced essential complexity to the problem.

We outline some drivers of the new accidental complexity during the past 35 years as follows:

- Larger N: Computers have been getting faster but data production is growing even more quickly.

- Higher Availability: In 1986, many services could run 9 am - 5 pm, Monday through Friday. Today's users generally expect always-on service and availability is measured in "nines," e.g., 99.99%.
- Faster Response Times: We often want products used by people to respond "instantly," i.e., faster than the threshold of human perception. Sometimes delays of a few seconds are acceptable, but we live in a competitive environment where faster is better. Other uses of data have varying timeliness requirements, ranging from seconds to days, but in general, the trend is toward speed.
- Low and Proportionate Costs: The cost of computing has continued to fall, as it has for many years. The cloud has also created the expectation that you should pay only for what you use, even if it does not fully make good on that promise.

What tools do we have available for handling the new accidental complexity? Setting aside for the moment developments in serverless computing, there has been progress in a number of other areas.

Traditional cloud computing offers infrastructure innovations that remove a great deal of complexity related to provisioning servers. It also offers the cost benefits that come from renting infrastructure in a multi-tenant environment.

Hardware also continues to advance. Servers are now available with TB-scale memories. Despite the slowing of Moore's Law [400], this is roughly 10× larger than was possible a decade ago. Network speeds are racing past 100 Gbps, representing an even faster rate of improvement.

On the software side, microservices architectures have gained in popularity. Though microservices were introduced primarily to improve the effectiveness of software development organizations, splitting applications into independent services can make scalability easier. It means that scaling challenges can be addressed one service at a time and that some services may never need scaling attention at all. Innovation in microservices also includes management tools such as container orchestration (discussed in Section 2.5.3) and service meshes (e.g., Istio, Consul, and Linkerd).

These developments all represent progress, but, in our view, they are not enough. In Section 1.2, we described how scalability, fault tolerance, and cloud computing represent interlinked concerns. Taken together, they embody the higher expectations we have of software. They also present difficulties that fall almost entirely into the accidental complexity bucket.

It is true that as hardware gets more powerful, some problems get easier. A server with 1 TB of memory can manage much or all of the operational data for many businesses. However, geographic redundancy might still be required, as might cost proportionality. As a result, the problem gets easier, but only so much.

A serverless system like BigQuery, in contrast, offers a SQL-like interface and can scale out to thousands of nodes. This is possible in part because it uses a programming abstraction (SQL) that effectively hides the servers and an execution model that expands resources to meet the need. It also hides hardware failures to provide reliability, as pioneered by other big data processing techniques like MapReduce [121] and Spark [457]. Chapter 5 explains how serverless computing still benefits from powerful servers, but powerful servers are alone not the solution to the new accidental complexity.

Our view is that keeping servers in the programming model for the cloud is like keeping registers in the programming model for servers. Named variables and abstract data types provide a much more natural way of expressing program functionality than registers do. In a similar way, the problems programmers are solving are not represented readily in terms of servers. This is fundamentally what makes programming with servers complicated. Any time we see a server, or something that looks like a server, we should flag that as accidental complexity that will hinder programmers, practically regardless of what problem they are working to solve.

Put another way, we see a direct and literal parallel between the challenges that cloud computing faces today and the difficulties that high-level programming languages had recently overcome when Brooks wrote "No Silver Bullet." In both cases, accidental complexity demands that developers direct a lot of time and energy toward mapping application functionality to the structure and behavior of some underlying machine. In the past, these details revolved around making good use of registers, designing memory layouts, or picking optimal instruction sequences. Today, they have to do with inferring when to add more servers to a subsystem, how to design interfaces between microservices for optimal efficiency, or how to structure data in caches and storage. How to keep the system working if anything goes wrong is another ever-present cross-cutting concern. We can say that serverless computing, at its core, aims to simplify distributed systems programming.

2.8.3 Anticipated Objections

We imagine that some readers may be inclined to raise objections to our characterization of serverless computing. Some might question whether simplified distributed systems programming is an achievable aim. They might also question whether it is desirable, perhaps fearing that simplification might obscure important details, or they might point to previous attempts that ended in failure. We will outline some of these concerns before reviewing developments that enhance our confidence in Section 2.8.4.

2.8.3.1 We Have Failed Before

One prominent effort to simplify distributed systems programming was distributed shared memory [292]. In distributed shared memory, there is no distinction between access to data structures stored on the local computer vs. those stored on remote computers. Waldo and collaborators break down the problems with this approach [425]. Remote objects look the same as local objects to the programmer, but their access latency is much higher and this causes performance problems. Shared memory also offers inadequate solutions for mitigating partial failure and for managing concurrency. In the high-performance computing space, where much of the work on shared memory took place, message-passing systems came to dominate. In contrast to shared memory, message passing provides only a thin abstraction over the underlying resources. It provides a simple and uniform API for communication between processes as well as some basic building blocks for coordination. However, it is firmly rooted in the model of a collection of servers, whereas distributed shared memory, which failed in the marketplace, looks much more like what we would now call serverless computing. Successful attempts to generalize serverless computing will need to avoid the pitfalls that distributed shared memory encountered.

Another technology for simplified distributed systems programming that has a reputation for disappointing impact is automatic parallelization. While the field has chalked up robust technical accomplishments [276], automatic parallelization has not become a mainstream programming technique. We believe this is because automatic parallelization worked well only for certain algorithms—it generalized less well than other compiler techniques. As a result, it could be outperformed by specialized domain-specific tools such as ATLAS [436] or Halide [331].

2.8.3.2 Fundamental Trade-Offs

The CAP theorem [166] is a famous result that reinforces the notion that distributed systems programming is fundamentally more complicated than programming a single machine. The interpretation of CAP has changed over the years [77], and its implications are perhaps most clearly outlined by Abadi [1] under the less memorable PACELC mnemonic. PACELC makes it clear that system designers can choose one set of trade-offs during normal operation and another during failures. It asks two questions. First, during a network partition, does the system favor availability or consistency? Second, during regular operation, does the system favor latency or consistency? The trade-offs posed by these two questions arise from the same basic theorem, as distance can be viewed as a form of temporary partition that lasts only for the duration of the latency [77]. However applications may still want to navigate the trade-off differently during rare network failures than they do during routine operation. For example, an application may favor low latency over consistency during routine operation, knowing that this represents bounded staleness [53], but favor consistency over availability during rare network events, to prevent state from diverging too far. Applications may also navigate the trade-off differently depending on the type of operation. For example, they may allow stale reads but force consistent updates or ensure certain consistency guarantees, e.g., causal consistency when security privileges are involved [303].

Could serverless computing adopt a “safe” default that makes programming easy? This would imply starting with strong consistency, favoring simplicity over availability and latency, and allowing programmers to override this when necessary. Such an approach may be possible, but we have to be careful—a correct program that cannot keep up with workload still fails to meet the need. As we explore in Section 2.8.4.1 and Chapter 5, strong consistency is fundamentally less scalable than weak consistency. This is because strong consistency usually creates *contention* [152], which in general gets worse, not better, as we add resources to a system. Serverless computing is thus stuck with a certain inescapable tension: the simple programs sometimes simply do not scale.

2.8.3.3 End-to-End Considerations

The end-to-end argument states that it is often redundant to implement robustness functionality at low levels of a system [345]. There are multiple prominent examples in communication systems, including data integrity checks, duplicate suppression, and encryption. Similar end-to-end considerations can also apply to crash recovery [87].

The end-to-end argument is stated not as an absolute rule but as an important design consideration. In some cases, low-level robustness can improve application performance, but in some cases, it may simply contribute unnecessary overhead because such functionality needs to be implemented at the application layer anyhow. Redundancy can even work against the aims of an application, such as when retry attempts introduce delays or jitter into real-time communications.

In the context of serverless computing, end-to-end arguments imply that low-level platform infrastructure should not force high availability or strong consistency on applications. Some of them will simply not need it, some will be served better by weaker guarantees, and some will want to implement variants of these properties that are specifically geared to their needs.

Still, serverless computing needs to offer solutions to robustness. After all, the details of providing it are precisely the sort of accidental complexity it should eliminate from the programming model. Also, end-to-end arguments often arise in communication between servers, whereas serverless aims to remove servers from the programming abstraction. Perhaps some end-to-end arguments will vanish along with their endpoints.

Sometimes end-to-end considerations tell us that accidental complexity inside the system corresponds to essential complexity outside it. For example, in a two-person chat application, it is possible that two messages are physically concurrent. This occurs when their senders are separated by a greater spatial distance than light travels in the time that elapses between when messages are sent. Specifying how the software handles concurrent messages is a requirement that is independent of the implementation technology. In cases like this, it may make sense for programmers to reason about communication between servers. Abstracting them away might have little benefit and could even make it more difficult for programmers to construct a mapping from the problem to the implementation.

2.8.4 Reasons for Hope

Research on simplified distributed systems programming predates both serverless computing and cloud computing. There are a number of advances that give us reason to believe that much of the accidental complexity of cloud computing will be hidden from the programmer, just as the registers, low-level memory layouts, and other details in a single computer have been hidden.

2.8.4.1 CALM Theorem

Hellerstein's Consistency and Logical Monotonicity (CALM) Theorem [187] is a fundamental possibility result. In contrast to CAP and other results that tell us what we cannot do, it describes classes of programs that will always lend themselves to no-compromise distributed implementations: These are the programs that can achieve both consistency and low latency at the same time. A program with these properties, or any composition of such programs, will scale indefinitely. Interestingly, CALM also has a converse, which states that if a scalable solution to a problem exists, then it necessarily satisfies certain properties. This also means that if an implementation does not satisfy CALM, then it will fail to scale at some point.

CALM is based on the idea of "logical monotonicity," which describes programs that can derive new facts as they make progress but never take them back. To take some examples from relational algebra, filter, join, and projection are all monotonic operators. In contrast, aggregation (say, to produce a sum) is non-monotonic because its output would, in general, be invalidated by additional input. Monotonic programs are also known as being "coordination-free," which basically means that they avoid certain distributed systems protocols, particularly those that create contention.

Software requirements do not generally lend themselves to implementations that are purely monotonic, but developers can be encouraged to write programs that are largely monotonic. When using logic languages such as Bloom [20] or Daedalus [21], it is also possible to pinpoint those parts of a program that are not monotonic. In some cases, programmers can do this by inspection; in other cases, automated tools can help.

For serverless computing, logical monotonicity can probably be used in a couple of ways. For one, we know that logically monotonic programs provide both consistency and low latency, so we know they will scale. In addition, the implementations are actually rather simple because we do not need to worry about order when sending messages between parts of the program—we just need to provide reliable delivery. Those parts of the program that are non-monotonic need to be implemented using coordination protocols. That will result in scalability bottlenecks, but at least we will know precisely where to look for them, so we can focus on making them run as fast as possible.

The use of logical monotonicity can give programs structure. There are portions that are scalable and portions that are not. For those portions that are scalable, the serverless implementations should be sure to allow for elastic scalability and distributed processing. For those that are not, it should do just the opposite, maximizing scale by concentrating these bits, ideally on a single server or even a single CPU core. Mixt [278] is one system that shows how to combine weak and strong consistency in a unified programming abstraction. While Mixt is not serverless, Cloudburst [384] has demonstrated the benefits of integrating monotonic data structures with serverless FaaS and has overcome some of the challenges of doing so.

Programmer focus on extracting monotonicity can drive changes to the specification. Sometimes one can improve program performance and scalability without changing functionality, i.e., without changing the allowed behaviors. In general, however, improving scalability means adding new behaviors that would not have been allowable under the requirements originally specified. For example, if we allow remote replicas to return slightly stale data, this opens up many valid executions that would not have been possible otherwise.

While using CALM involves abstraction, its benefit goes beyond hiding accidental complexity. CALM analysis can help steer the features of an application to avoid certain limitations of the underlying infrastructure—i.e., to avoid accidental complexity. To appreciate this, we must view the programmer's responsibility as not merely to implement a specification but also to help the team decide how the software should function. Such decisions are quite naturally informed by a combination of business needs and available technology. In the words of Brooks, "the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. For the truth is, the clients do not know what they want. They usually do not know what questions must be answered, and they almost never have thought of the problem in the detail that must be specified." [79]. By using CALM, programmers can help teams define requirements so that software can scale.

2.8.4.2 Actors, CRDTs, and Other Distributed Programming Models

FaaS has seen commercial success, but its limitations mean that it works well only in specific applications (see Section 2.6). Most notably, it is a poor fit for programs that manipulate state in demanding ways. We believe that there are established programming paradigms that could be implemented as serverless services, providing an alternative to FaaS for applications with different needs.

In the actor model of computation [10, 194], a number of stateful entities, the actors, interact via message passing. Each actor maintains its own state internally, and such state can only be accessed by the actor code, which runs in response to messages. This encapsulation is very similar to that offered by object-oriented programming languages.[1]

Modern actor implementations include Orleans [65], Akka [12], and Erlang [39]. Erlang matured in the 1990s and showed that actors could be a practical way of building distributed systems with very good availability and scalability characteristics. Akka brought actors to the Java and Scala ecosystems and introduced a simplified approach to providing high availability. Orleans innovated by introducing the virtual actor concept. Actors in Orleans do not need to be memory-resident at all times: Their internal state can be serialized to secondary storage with the framework instantiating them on demand. Orleans innovates in other areas as well, for example, by simplifying the messaging model to a request-response pattern. Orleans is available as open source software, but it is not offered as a serverless product, so users cannot benefit from provider management. Its scalability is proven—Microsoft has used Orleans as the backend for several popular games, and other companies have adopted it as well [301]. We imagine that cloud providers could offer "actors as a service" as a complement to FaaS, and briefly look at how that might work.

[1] In fact, many object-oriented languages use a messaging metaphor to describe object method invocation.

Actors and FaaS have interesting similarities as well as differences. One way to compare them is by analogy to object-oriented programming languages. Object methods are often implemented as functions that simply take a reference to some data, "the object," as an argument. Actors are an implementation of distributed objects, and we can think of each actor method as a function that always takes a stateful object as one argument. Could that function be implemented as a FaaS function? Perhaps. Suppose we wanted to build virtual actors, like those in Orleans, on top of a commercial FaaS platform. Virtual actor state is cached state, so keeping it in FaaS is fine (durable state lives elsewhere, e.g., serialized transparently to key-value storage). We would have to add a routing mechanism of some sort to ensure that each actor invocation goes to the execution instance at which it is cached. This routing layer needs to be reliable and strongly consistent (linearizable). Orleans uses a distributed hash table for this purpose [386], and presumably the same mechanism could be added to a FaaS system. Note that, in this vision, groups of objects would be mapped to each execution instance, just as in other actor systems. It would not be efficient to create a distinct runtime instance for each individual actor, at least not without a more lightweight isolation technology.

The Ray [284] system incorporates an actor model, distributed remote functions, and distributed object storage. Remote functions are similar to FaaS but accept arguments in the form of either values or references to data in object storage. These references may be futures, that is, they may reference the results of calls that remain pending. Ray programs can build up graphs of futures, which allows construction of a directed acyclic graph (DAG) of the computation. Ray is offered both as open source software and as a commercial service. While it is not positioned as a serverless service, it offers a programming model that abstracts away the servers and could presumably be hosted as a serverless service.

Akka Serverless [13] is a hosted platform that offers a number of state models on top of an underlying actor and stream processing framework. Since its state models are durable, its creators say it allows "database-less" applications, i.e., no separate state tier is required. This sort of unification of state and computation seems to be a hallmark of stateful serverless systems. Akka Serverless offers three different state models: value entities, event sourced entities, and replicated entities. Each value entity is a strongly consistent (linearizable) object. Event source entities have an underlying log of updates [66]. Replicated entities provide Conflict-Free Replicated Data Types (CRDTs) [365], which allow for eventual consistency via monotonicity [187].

Cloudburst [384, 442] offers another approach to stateful FaaS. It integrates a high-performance key-value store with the FaaS execution engine and offers the ability to execute DAGs representing compositions of functions. Cloudburst uses lattice types [110] (providing CRDTs) to enforce coordination-free consistency guarantees (see Section 2.8.4.1). One consequence is that each function instance can cache key-value store state locally. In place of traditional cache coherence protocols, it substitutes causal consistency, which is the strongest form of coordination-free consistency [260]. This means that if some state update is seen by a program, then all downstream computation will see it as well. This is true whether such access occurs later on in the same function or downstream of it in a computation DAG.

There are clearly movements afoot to bring alternate programming models to the cloud. These include proven programming models like actors or event sourcing, as well as more nascent approaches like Ray's remote functions or CRDTs. These models all offer abstraction layers over the underlying servers, and a cloud provider might offer any one of them as a serverless service.

2.8.4.3 Deterministic Databases

Strong consistency is easy to reason about both for programmers and, perhaps more importantly, for the whole team involved in designing an application. However, strong consistency can be problematic in distributed systems because the coordination algorithms typically used to implement it introduce contention (i.e., waiting). For example, two-phase commit ensures that separate partitions agree on whether a transaction commits or rolls back, but it means that locks are held while instances communicate over the network.

Deterministic database systems [2] improve the scalability of strong consistency, particularly with regard to throughput. In traditional (non-deterministic) databases, transaction order emerges from the concurrency control mechanism and is subject to various unpredictable system behaviors, such as operating system thread scheduling, network delays, and I/O timing. Deterministic databases, on the other hand, establish a transaction order up-front, before any transactions start executing. Various timing delays still occur during execution, but they no longer influence what state the database reaches, what results it returns, or whether a transaction succeeds or aborts. Deterministic databases have shown benefits at various scales, ranging from a multi-core server [142, 143], to the scale of a data center [404], to cross-region scale [335]. The result is strong consistency, and the simplified programming it allows, without the throughput limitations that contention otherwise induces. Deterministic databases have some limitations: The entire transaction must be provided up-front, and implementations may rely on pre-declared read and write sets [404].

2.8.4.4 Language Techniques

Cheung et al. describe a research agenda that brings together techniques from programming languages, databases, and distributed systems to improve the cloud programming experience [100]. They present a four-component model, termed PACT, that emphasizes declarative programming and separates multiple concerns:

- Program Semantics: Programmers should specify the correct behavior of their code. By doing so declaratively, they avoid reasoning about the complex interleavings of sequential code, as required today.

- Availability Specification: Rather than reasoning about failure domains or redundancy mechanisms, an availability specification allows programmers to describe the requirements in terms of metrics that matter such as percentile response times.

- Consistency Guarantees: Programmers usually build up application consistency using the features of an underlying database, programming language, and perhaps their own protocols. In this vision, consistency guarantees, specified as invariants, would describe the desired common-case semantics as well as the allowable deviations from those semantics.
- Targets for Dynamic Optimization: There are trade-offs between cost, availability, and latency. Programmers should specify how the underlying system prioritizes each factor.

This vision is akin to high-level programming languages for distributed systems. It also includes a proposed implementation that translates between a broad range of established programming models and a common declarative intermediate representation that can be analyzed and optimized. It also envisions an underlying runtime that might sit on top of FaaS, serverless storage, or other cloud services. The vision is both comprehensive and encouraging. It makes it clear that efforts to tackle the new accidental complexity are already underway and that an important part of the solution, this time as in the past, will come from advances in programming languages.

2.8.5 Servers Really Represent the Problem

Even though the technology of serverless computing would be the same under any name, the focus on abstracting away servers is appropriate because it focuses on a prominent aspect of distributed systems that makes cloud programming difficult. Some have criticized the name [183, 214, 286], pointing out, for example, that the existence of underlying servers has caused some confusion. We respect these arguments but counter that "serverless" still describes what is happening better than any other single word would.

Consider alternative descriptions for the developments now underway. Instead of using the term "serverless," we could describe the developments now underway as the "ops-free cloud" or "fine-grained pay-as-you-go" or emphasize the "focus on your own business logic." We could talk simply in terms of X as a service, where X includes functions, storage, databases, queues, platforms, or backends. All these descriptions are meaningful but none of them captures the movement toward simplified cloud programming and its implications the way that "serverless" does.

"Simplified cloud programming" [216], "new directions in cloud programming" [100], or "removing the accidental complexity from distributed systems programming" all capture the important shift that is going on. All of these names implicitly recognize the new accidental complexity described in Section 2.8.2, and *servers* are the most obvious manifestation of it. Hiding the servers is precisely what leads to the benefits in FaaS, cloud object storage, and other services that we call serverless. It is what will lead to the benefits of new, more general, approaches to cloud programming. Would "registerless computing" have been a good name for high-level programming languages? Probably not, and perhaps we will talk about serverless computing in other terms in the future. For today, the term suits the need quite nicely.

2.9 Additional Topics

We close our review and analysis of serverless computing with a series of assorted topics that have helped us hone our understanding. Since serverless implies an absence of servers, it makes sense to begin by checking that we understand precisely what a server is. We then touch on security, the puzzle of missing serverless memory, and the comparison between microservices and serverless computing. Finally we look at why Google App Engine did not spark a serverless movement even though it could do much of what FaaS offers years earlier.

2.9.1 What Is a Server?

We are not aware of any substantial debate about what a server is, but since we are studying serverless computing, thoroughness demands that we define clearly the inverse that its name implies. We caution that the goal here is not to find a definition of serverless computing by analyzing its name. As a concept, it must have a meaningful technical basis independent of what we choose to call it. In other words, the essential characteristics of serverless computing, introduced in Section 2.4, define serverless computing regardless of its name. Just the same, we know that serverless is a reaction to some things about servers, and we want to make sure to explore all the possibilities for what those might be.

Early in the computing literature, the term "server" first appears as a modifier to describe a host attached to the network [361]. RFC-5 distinguishes between the "user-host" and the "server-host" [341]. In this context, hosts are computers or other devices connected to the network [199].

Other early literature uses the term "server" to describe networked programs, not necessarily the computers that they run upon. For example, an early ARPANET article defines users as "programs desiring service" and servers as "programs providing service" [401].

Use of the term "server" exploded with the rise in popularity of client-server computing. In a prominent review article, Sinha writes: "A Server provides a service to the Client" [373]. This leaves open the question of whether the server is a computer or perhaps just a program running on a computer. Luckily, this distinction is inconsequential—computers must run programs to provide a service, and with virtualization, programs routinely emulate computers. In either case, servers are long-lived stateful entities on a network, which is what matters. Given the prevalent modern usage, we prefer to think of servers as either computers or virtual computers. Furthermore, when we think of servers, we do not think of distributed systems—those are collections of servers.[2] Sinha makes it clear that servers can cooperate and communicate behind the scenes. He says: "It is advisable, and desirable, that in a multiserver environment, the Servers communicate with one another to provide a service to the Client without its knowledge of the existence of multiple Servers or intra-server communication."

[2] We do sometimes think of the components inside a powerful computer as a distributed system.

So, if servers are computers that provide services, what is a service? Despite the widespread use of the term in computing, we have been unable to identify a universal authoritative definition. However, there seems to be a consensus that a service is some well-defined functionality that is available on demand, accessed via a network, and abstracted behind an interface. This last feature, abstraction, ensures that any number of implementations can be developed to provide the service.

We conclude that servers are what everyone thinks they are: computers, or their virtual embodiments, attached to the network, providing some useful functionality, "a service," behind an abstract interface. While a service can be provided by a collection of computers working in concert, that collection is not a server but something larger.

If servers are what you get when you have both computers and services, then "serverless" could describe removing the computers or removing the services. Computers without servers are everywhere, in all sorts of electronics and devices, but in the data center they are not very useful. In this context, serverless can only mean removing the computer, not the service.

Keeping in mind that serverless computing is a metaphor, we believe that the "-less" can mean removing anything that we do not like about servers, i.e., any of those things that make it awkward to build services with computers. In the case of abstractions that mirror and maintain compatibility with the architectural and system interfaces, e.g., container orchestration, it means removing the startup time, complexity, and monitoring burdens traditionally required to run VMs. In the case of FaaS, there is also robust autoscaling and fine-grained pricing. In the case of dataflow computing, everything having to do with piping information in and out of the computation gets taken care of. SQL systems provide a programming paradigm that allows developers to operate on large data sets with simple expressions, even ones much larger than any computer could store. With storage and other stateful serverless services, developers do not need to worry about how to create a reliable service from a collection of unreliable computers. These are a few examples, and the reader can doubtless suggest many more.

2.9.2 Serverless and Security

Is serverless computing more or less secure than traditional forms of cloud computing? This is a natural question to ask, but it is not the best way to understand how serverless computing impacts security.

We believe there is a strong case that serverless computing can make it easier for programmers to secure their applications. However, this does not derive from any inherent property of serverless computing; in fact, serverless applications may be more exposed to certain kinds of attacks than traditional cloud applications.

We first consider the ways that serverless computing could make an application more vulnerable to security threats. Several potential concerns arise from increased sharing of hardware, which can expose applications to bugs in isolation mechanisms as well as side-channel attacks. While traditional cloud VMs may coexist within a shared server, CPU and memory resources are often partitioned. This means that separate VMs largely use separate parts of the machine even though they reside in the same enclosure. Multiplexing is primarily spatial, and resources are reassigned from one customer to another only on long timescales. Customers can also specify that VMs must be provisioned on machines dedicated to their organization and not shared by others.

With serverless computing, hardware resources can rotate among customers much more quickly, i.e., there is more temporal multiplexing. This is how FaaS achieves improvements in utilization and efficiency. It also creates greater exposure to any vulnerabilities involving shared hardware or a shared hypervisor. These could include privilege escalation bugs, or side-channel attacks such as those involving speculative execution [232] or Microarchitectural Data Sampling (MDS) [88, 352]. The potential for vulnerability exists in other serverless services, even those such as object or key-value storage that run little customer code. These threats are real and hard to eliminate [269].

There are several other security vulnerabilities that are specific to FaaS. For example, since functions have an internal cache that persists from one invocation to the next, there is the potential for information leakage if user code has bugs. FaaS is stateless, in the sense that it has only ephemeral state, but it does not guarantee a clean slate at every function invocation. Also, FaaS applications may expose a larger service interface due to fine-grained application decomposition. Properly securing a large interface could be burdensome, and some users may not take the time to do it properly. There are also hidden risks that come with making software easy to run. Some software that would have better been retired may be left up and running even when it is no longer needed. Extra effort may be required to cull unused or unnecessary FaaS functions that increase an application's attack surface.

We also stress that whenever there is a new model, risks arise out of the transition. For example, an infamous attack against Capital One exposed a tremendous amount of data stored in AWS S3. In this case, the actual source of the vulnerability was *server* misconfiguration, which allowed attackers to gain access tokens granting them privileges on the serverless object storage infrastructure [235]. Once in possession of these tokens, serverless scalability allowed the attackers to download a large amount of sensitive data very quickly. This is a vulnerability that does not apply to serverless any more than it does to servers—the core problem is that administrators failed to understand a new security model.

To appreciate just how different the serverless mindset can be for a security professional, consider that so many security tools and techniques focus on controlling, locking down, and monitoring servers. With serverless, all of this vanishes. While the cloud provider continues to secure the machine, other access control mechanisms become purely logical. In principle, this creates the opportunity for better security controls that are designed around the application and the business model rather than the deployment on servers. However, there is a learning curve, and therein lies risk.

The transfer of low-level security responsibilities to the cloud provider is a clear benefit of serverless computing. A lot of work is needed to ensure that operating systems, language runtimes, and standard libraries always have the latest security patches. With serverless, the cloud provider takes on responsibility at the operating system level, and possibly above it.

The scalability of serverless computing can also provide protection against certain types of denial-of-service (DoS) attacks. An attacker generating a large volume of requests to overwhelm installed server capacity can be stymied by an implementation with serverless autoscaling. Of course, the attacker may be targeting a company's budget, so other defenses are still required, but it is more difficult for the attacker to compromise the availability of a service.

We expect the most compelling security benefit of serverless computing to come from the abstraction it provides. Replacing privileges on physical resources with privileges on logical resources allows fine-grained controls that can be designed to meet the needs of the application irrespective of its physical deployment. This is already evident in today's emphasis on IAM configuration for FaaS, object storage, key-value databases, and other serverless services. We expect that it will become easier for organizations to secure their serverless applications and that they will do a better job of it.

2.9.3 The Missing Serverless Memory

An essential characteristic of serverless computing is utility-style pay-as-you-go pricing with no charge for idle resources. Another is autoscaling. When it comes to memory resources, serverless sometimes falls short on both accounts. For example, the FaaS billing model is based on ($time \times memory$). While *time* is the measured execution time, *memory* is the configured amount, not the amount actually used by the program. AWS AppSync prices queries, modifications, and real-time updates in cost per million operations. In order to have caching, however, users configure and pay for a fixed amount of memory. Stateful serverless services such as object storage and key-value stores all use memory for caching, but none has a pricing model that reflects the underlying costs. Serverless memory is a missing concept and an open challenge; we now explore why that is.

2.9.3.1 The Cost Disconnect

When AWS Lambda returns after a function invocation, it logs not only the execution time but also the maximum memory consumption. However, the billed cost is not the reported amount of memory used but rather the configured amount. Why not bill on actual usage? A complicating factor is that configured memory is a proxy for the share of machine resources allocated to a function. Allocating more memory also provides more CPU time and network bandwidth. Billing for actual memory usage would require metering all of these resources, and so far, no FaaS platform provides such a model. As a further wrinkle, some applications may benefit from more memory for the operating system page cache, which is not accounted for in the memory used by application processes.

Another potential mismatch occurs with FaaS because customers pay for memory only while the function is running, but the cloud provider incurs resource costs for cached function instances even when they are idle. Say a function runs only intermittently—for 1 s every 100 s. The instance likely stays cached, but since the customer pays only when the function is running, the cloud provider ends up billing for only 1% of the memory used. This might not be a problem if memory were a small part of the machine cost, but it is not. By analyzing Google Compute Engine custom instance prices,[3] and by comparing AWS instance prices,[4] we conclude that memory accounts for between 32% and 48% of the cost of a server.

This sort of disconnect is problematic and may be causing market distortions. It means that the cloud provider may be charging disproportionately more for applications that run frequently relative to those that run intermittently. Exploiting this fact to obtain memory at a discount is an explicit aim of Infinicache [426].

Another example of a pricing model that fails to align the costs and benefits of cache memory is object storage. Object storage bills have two components: a cost for retaining the object (billed according to size and duration) and a cost for accessing the object (billed for every PUT or GET request). For retention, this model reflects cloud provider costs well, but for access, it can diverge. Objects that are accessed frequently can be retained in memory, which can dramatically lower the cost of delivering them. Yet the business model does not pass the cost savings on to customers.

Cloud providers offer a number of caching products, but none that autoscale memory. For example, in the AWS ecosystem, there is ElastiCache, which offers the Redis and Memcached APIs but only with a fixed cache size. AWS also offers DAX, a caching accelerator for DyanmoDB, a serverless key-value store. Somewhat surprisingly, DAX requires customers to configure a fixed amount of memory. It appears to be a server-based accelerator for a serverless database. The Anna key-value database is a research system that addresses these limitations, providing autoscaling and storage tiering [441, 443].

The previous examples all suggest that something is hard about serverless memory. They also point to the root of the problem: Applications provide clear indicators of their need for compute, storage, or network resources, but not always for memory, much of which is used for caching. We know that a program needs the CPU from the time it starts running to the time it finishes. We know that disk space is needed from the time an object is created to the time it is deleted. Whenever a program sends or receives data over the network we can count the bytes, then allocate resources and costs accordingly.

With cache memory, the situation is less clear. Providing more cache memory often lets applications run faster and more efficiently, and it reduces the utilization of other system resources. However, it is difficult to determine automatically how much cache memory should be provided to meet the application's needs. In FaaS, for example, the cloud provider typically caches function instances to reduce cold starts, but this behavior appears unpredictable and varies from one provider to another [429]. While this solution seems to deliver reasonably good results, it is hard to imagine billing a customer based on the amount of memory held by opaque caching mechanisms.

[3] The on-demand price of Google Compute Engine custom instances is $0.02289 per vCPU hour and $0.003067 per GB hour. A large standard instance has 32 vCPUs and 128 GB memory, suggesting that memory corresponds to 35% of the cost of a server.

[4] See Section 2.9.3.2

2.9.3.2 Cache Memory Simulation

We conducted simulations to assess the potential benefits of cache autoscaling. The input for our simulation is a workload trace collected while running a TPC-C database benchmark on a SQLite database. We simulate an LRU cache interposed between the database and its storage, then measure the impact of cache size on performance and cost. Our test database is 800 MB in size and has a 1 kB block size.

We extrapolate per-unit compute, memory, and I/O costs based on published AWS prices. The general-purpose `m5.24xlarge` instance costs $4.608 per hour and provides 384 GB memory, whereas the high-memory `r5.24xlarge` instance costs $6.048 per hour and provides 768 GB of memory. Both machines have 96 cores, so we calculate the unit cost of memory as $\$1.042 \times 10^{-6}$ per GB-sec and the unit cost of compute as $\$9.167 \times 10^{-6}$ per core-sec. AWS sells provisioned IOPS on EBS for $0.065 per month per IOPS. Based on this rate, if a cache miss results in one I/O operation, the per-miss I/O cost is $\$2.508 \times 10^{-8}$.

Figure 2.10 shows how the per-transaction cost varies with cache size for a constant 1/s transaction rate. In this model, the minimum cost is achieved with a 64 MB cache. For smaller caches, adding memory reduces both `CPU` and `Read I/O` costs because we reclaim time spent waiting on I/O. There is no impact on `Write I/O`, and we assume that writes are buffered and do not consume CPU time.

In Figure 2.11 we again hold the transaction rate steady at 1/s and now plot transaction latency against transaction cost as the cache size varies. We include three latency percentiles: the 50th, 90th, and 99th. Below the cost-optimal cache size, increasing memory both improves performance and cuts costs. After reaching the lowest-cost point, there may still be value in increasing cache memory if an application benefits from faster performance.

Figure 2.12 shows how the cost-optimal cache size varies with the transaction rate. At the low end, for transactions that arrive once every 3 s to 10 s, 32 MB of cache memory is optimal. As the query rate rises, the cost-optimal memory size increases, reaching 512 MB for rates in the hundreds of queries per second. We conclude that this workload would benefit from an autoscaling cache if the transaction rate is variable.

A key consideration when providing an autoscaling cache is that scaling down the cache when optimizing for cost can result in latency increases. Figure 2.13 illustrates this, showing comparatively high latency at low transaction rates.

2.9.3.3 Possible Solutions

Automatic cache sizing does not exist in the cloud today, likely because of the complexities discussed in Section 2.9.3.1 and Section 2.9.3.2. Some research has begun to address this challenge in the serverless context [339]. Also relevant is classic work such as Gray's "five-minute rule" [171] for database caches. The five-minute rule reproduces the cost-optimal policy that we explored in Section 2.9.3.2, but it does not address latency targets.

Providing a solution is beyond the scope of our work, but we briefly consider possible policies that might be considered in addition to cost optimization when sizing a cache:

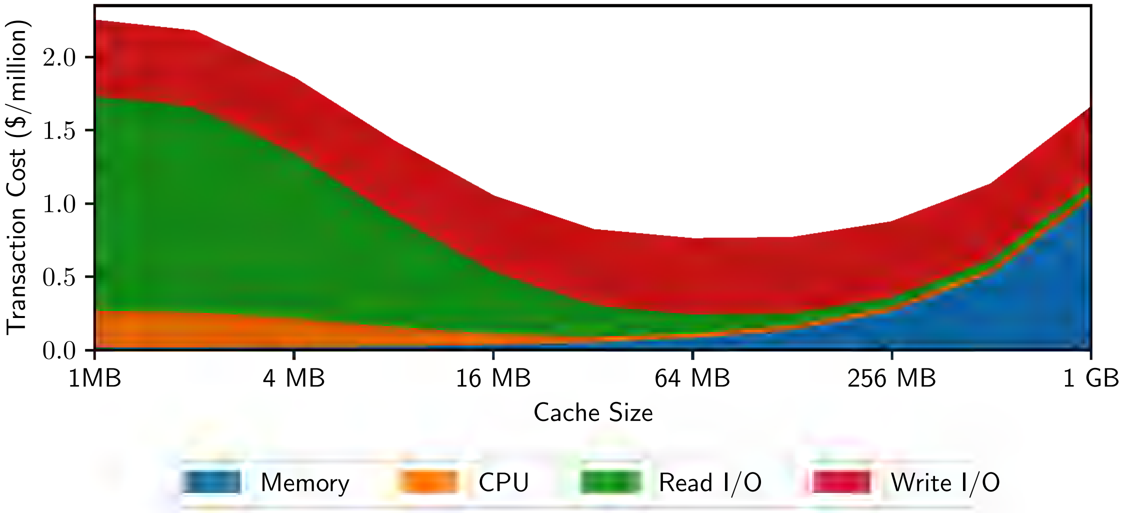


Figure 2.10: TPC-C cache simulation: Breakdown of transaction cost at various cache sizes. Transaction rate is held constant at 1/s.

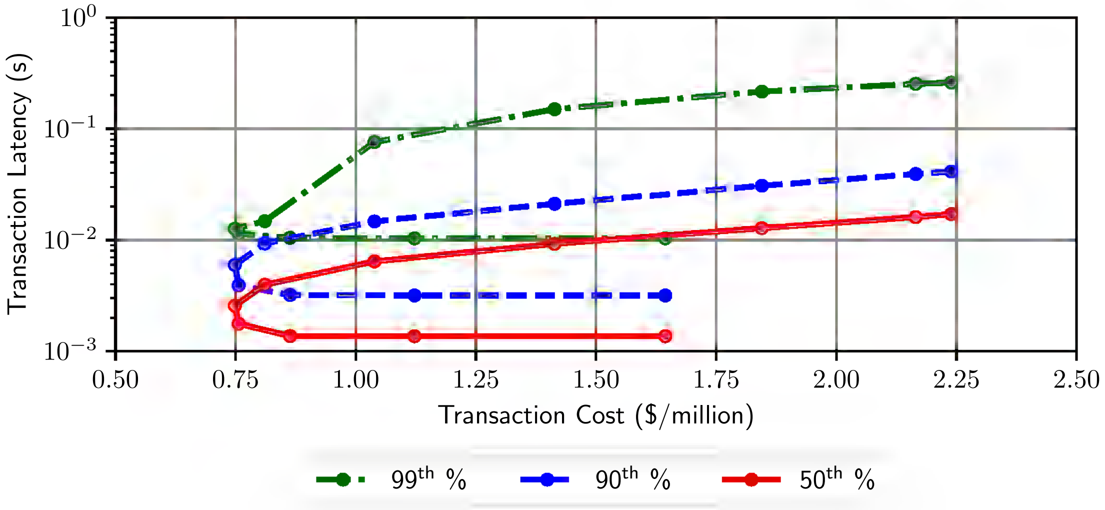


Figure 2.11: TPC-C cache simulation: Query latency vs. query cost shown for cache sizes varying from 1 kB to 1 MB. 50th, 90th, and 99th latency percentiles are shown. Transaction rate is held constant at 1/s.

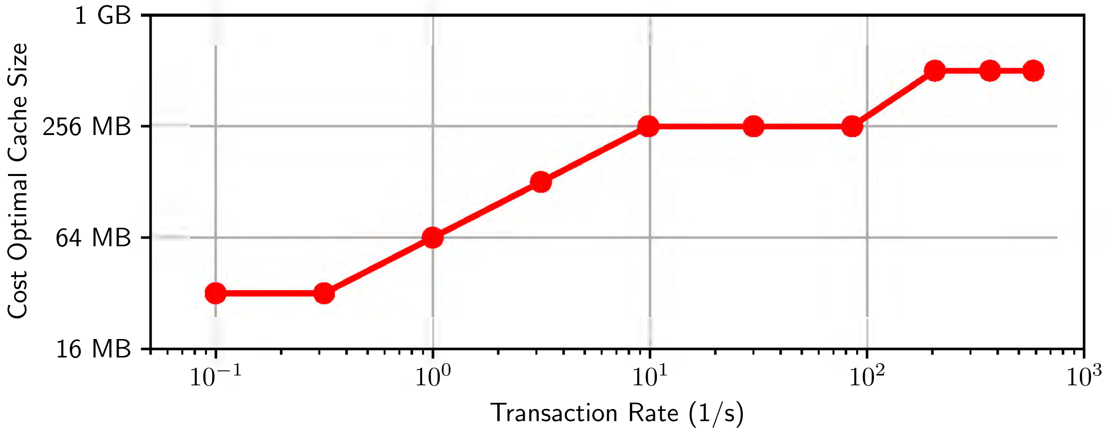


Figure 2.12: TPC-C cache simulation: The cost-optimal cache size varies according to the transaction rate.

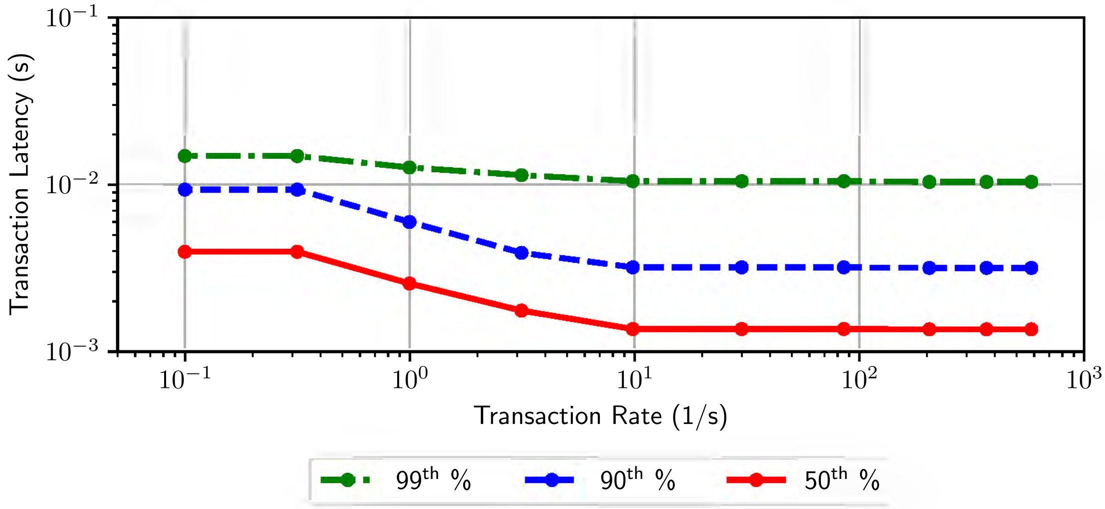


Figure 2.13: TPC-C cache simulation: The latency at the cost-optimal cache size varies according to the transaction rate.

- *Target latency*: The cache size adjusts automatically to ensure that the application latency remains under target latency bounds. This approach is appealing because it is a direct measure of application performance.
- *Maximum idle time*: An object will be kept in the cache until it has not been accessed for longer than the maximum idle time. Customers can set this time based on how they expect the application to be used, e.g., if customers who return once a week should have a fast experience, then the maximum idle time should be longer than one week.

A problem with both of these approaches is that systems often have many caches [63]. Appealing as it may be to connect their configuration to externally visible consequences, this may not always be practical. Other internal goals, like a target miss rate, are possible but also problematic. Cache auto-sizing is the key missing ingredient for serverless memory; we expect it to present a fruitful avenue for research.

2.9.4 Serverless vs. Microservices

Serverless computing with FaaS is sometimes likened to microservices or posited to be an evolution of them. This is perhaps because both involve breaking up programs into smaller units of functionality. However, though there is a superficial similarity, the technologies solve different problems.

The microservices architectural pattern aims to ensure loose coupling between components so that they can evolve independently [288]. It places a particular emphasis on deployment, helping to ensure that one team can upgrade part of a running system, usually without requiring changes, collaboration, or support from teams responsible for other components. This operational lens goes a step beyond traditional software modularity [308], requiring, for example, that a service protect itself from availability problems, transient errors, or unexpected responses from other services.

Microservices stand in contrast to “monolithic” application design patterns where the entire application functions as one piece of code and is typically deployed as a unit. Such software monoliths can still comprise collections of services. For example, in traditional service-oriented architecture (SOA), applications are split into services, but they are tightly coupled and so typically need to be upgraded all at once, i.e., the collection of services may be monolithic even though it runs on many servers. This can slow down and complicate the development process because it means that problems in one team can hold up other teams.

Even though FaaS and microservices are independent concepts, FaaS can still be a good way to implement a microservice. Proliferating microservices can lead to operational headaches, say if each requires custom autoscaling policy configuration or custom monitoring. This is just the sort of problem that serverless computing is designed to solve. However, serverless computing is also set up to support software monoliths. Platforms such as AWS Lambda allow function invocations to request specific versions, which makes it possible to ensure compatibility when upgrading many related functions. Other forms of serverless computing, such key-value databases, provide value for both microservice and monolithic applications.

In summary, microservices and serverless computing with FaaS represent compatible but independent innovations. There is a superficial similarity because FaaS and microservices both involve cutting up a code base into pieces, but the aims are different. In FaaS, a function represents a unit of program functionality. With microservices, the units (services) are also governed by the organizational structure of the team. Serverless computing can make it easier to adopt a microservices approach because it reduces the effort required to build and deploy a large number of services. Yet serverless computing also provides benefits for monolithic applications.

2.9.5 Comparing Google App Engine and AWS Lambda

Why did serverless computing take off with the introduction of AWS Lambda in 2014 rather than with Google App Engine in 2008? Both technologies possess all of our characteristics of serverless computing, so comparing the two could provide insight into key features of serverless computing.

The difference in industry impact between the two products is significant. The search trend data shown in Figure 2.14 indicates that App Engine initially received a strong level of developer interest, but it waned over time. AWS Lambda, in comparison, shows more sustained growth in developer interest over time. Furthermore, no major cloud provider launched a direct App Engine competitor, whereas all of them launched FaaS products inspired by AWS Lambda.

App Engine and Lambda are similar in many ways. Both abstract away the underlying servers, scale automatically, and allow users to deploy code written in high-level languages. App Engine users write stateless web services that are quite similar to FaaS functions. Stateless web services have ephemeral state only and must generally complete within a bounded period of time. Both also support event-driven programming. Even though Lambda is more closely associated with this style, App Engine has always supported event-driven programming through its integration with Google Pub/Sub.

App Engine differs from Lambda in several technical considerations, and the differences become particularly notable when comparing the products as they were originally released. While it now supports autoscaling down to zero, each App Engine deployment originally required a minimum of one or two instances at all times, depending on availability requirements. This difference has now been closed, but it seems that as launched, both were good at scaling up, but Lambda was better at scaling down. The pricing models also still reflect this difference. App Engine continues to charge according to the number of instances that the autoscaling mechanism provisions, with a minimum billing increment of 15 minutes. Lambda, on the other hand, launched with a minimum billing increment of 100 ms (this is now 1 ms, but for small instances, a per-invocation request fee makes the minimum cost similar). At large scale, when many instances are needed, the difference between these models may be small. Instances will generally be well utilized, and single-instance steps in scaling

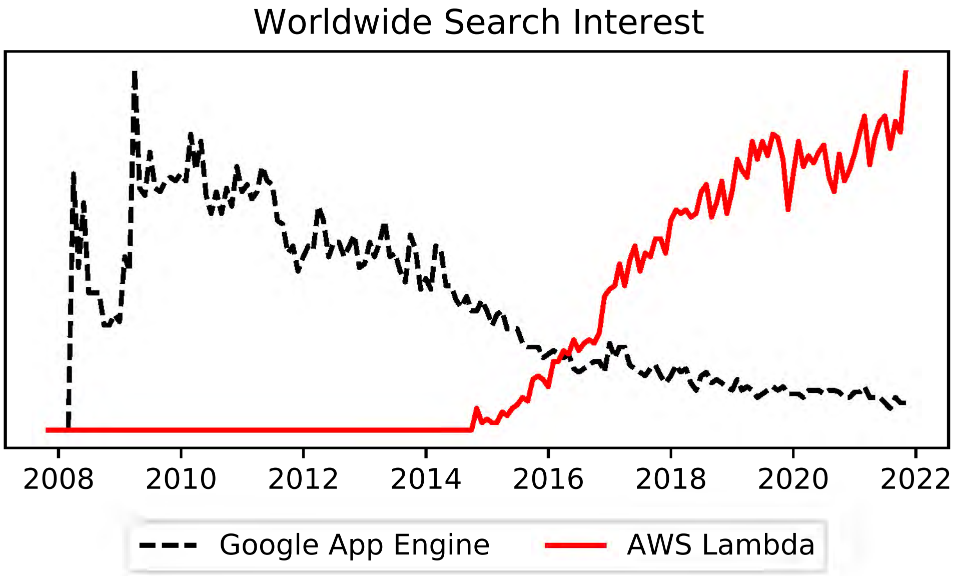


Figure 2.14: Trends compared: Google App Engine and AWS Lambda. Global search traffic reported by Google Trends. We apply 150% scaling starting October 2020 to account for a discontinuity in the data that appears to be an artifact.

make little difference in a large pool. At small scale, however, and when demand is highly variable, the Lambda model matches costs to usage much better. This reinforces the notion that autoscaling down (including down to zero) and fine-grained pay-as-you-go pricing were important to the popularity of serverless computing.

App Engine takes a more prescriptive approach than Lambda, which is consistent with its positioning as an application framework. Frameworks strive to include a broad range of functionality, and developers hoping make the most of their benefits generally expect to sculpt their applications to fit the mold. In the case of App Engine, this means that developers write their web applications in a particular way, e.g., following App Engine conventions for binding a URL to corresponding code. Early versions also restricted developers to using a specific version of Python and a specific set of libraries. App Engine also provides several related services, such as object storage, caching, and search. These components are designed to work together and appear to be intended to offer web developers everything that they need from a cloud provider.

In contrast, Lambda has taken a more open approach from the start. At launch, its creators emphasized that an important design consideration was allowing developers to bring their own libraries, including native-code libraries [424]. They also positioned Lambda as a sort of connective tissue or “glue” between the many services that AWS already offered in 2014 [421]. App Engine launched much earlier, in 2008, and the positioning that Lambda used might not have been successful at the time because cloud provider ecosystems had fewer offerings.

A perceived lack of openness may have led App Engine to face a greater fear of lock-in. Such fears have always been a problem for cloud computing [37]. Lambda is not immune to concerns over lock-in [356], but since all cloud providers support FaaS in some form, it can be more acceptable than using a cloud provider's proprietary web application framework.

In summary, a few key differences help explain why AWS Lambda led to the serverless movement, while Google App Engine did not. Foremost, Lambda embodied a simple and seemingly general-purpose concept, a function in the cloud, whereas App Engine was more explicitly tied to a particular use case, web development. Lambda's positioning surely aligned with a more inspiring vision and perhaps also stoked fewer lock-in fears. While neither technology allowed users to run existing applications, Lambda launched with good support for arbitrary libraries, allowing developers to embed more or less any code that they wanted to. App Engine only added such capabilities later on. App Engine also seems to have added the ability to scale down to zero only after FaaS introduced it. Its pay-as-you-go model remains tied to a coarse-grained measure of instance provisioning, meaning it is still better at scaling up than it is at scaling down. Lastly, Lambda launched into a rich ecosystem of cloud services and was designed to be complementary to them. We conclude that general-purpose positioning, integration with the cloud ecosystem, support for existing code in the form of libraries, and autoscaling—particularly, scaling down well—all contributed to the success of FaaS and the subsequent emergence of serverless computing.

Chapter 3

A FaaS File System for Serverless Computing

3.1 Introduction

In this chapter, we describe our efforts to make serverless computing practical for more applications. In Section 2.9.5, we noted that ensuring compatibility between FaaS and existing software ecosystems was a deliberate design decision, and that this likely contributed to its success. One of the most obvious limitations of FaaS is its "statelessness," and we decided to investigate whether one could address this limitation in a similar way, i.e., by using standard system interfaces.

FaaS functions have ephemeral state only, meaning that they must externally save any state that requires preservation past the point of function return (see Section 2.2). There are some advantages to this solution, which are illustrated in Figure 3.1—notably, it allows the compute tier to scale independently of the storage tier, helping ensure cost proportionality and supporting pay-as-you-go. There are drawbacks, however. For one, accessing remote storage services incurs a performance penalty [189]. Programmers also must conform to object storage and key-value storage APIs that are proprietary to each cloud provider.[1]

The file system abstraction offers an enticing alternative. POSIX [218] provides a standard API, which helps make software that uses file systems pervasive. File systems also offer attractive performance characteristics. Most implementations benefit from operating system caching and buffering, allowing them to hide the latency of storage access from applications [394].

Unfortunately, in a distributed setting, file systems lose some of the benefits we have described. While they maintain the same interface as local POSIX file systems, they have different behaviors. Due to network overheads, all of them can exhibit degraded performance when accessing shared data. A number of implementations attempt to improve performance

[1] Examples include S3 and DynamoDB on AWS, Storage and CosmosDB on Azure, and Cloud Object Storage and Firestore on Google Cloud.

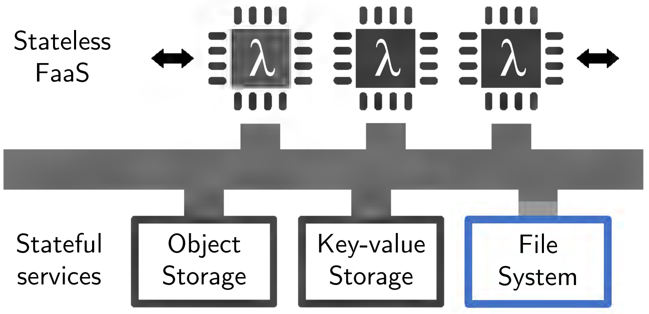


Figure 3.1: Serverless applications often use stateless FaaS together with stateful services such as object storage or key-value storage. FaaSFS is a distributed file system designed to offer full POSIX compatibility and scalability to match that of FaaS.

by providing weakened consistency guarantees, but this means that they do not preserve the functionality described by POSIX. Distributed file systems thus implement the API of a local file system, but all of them work differently to varying degrees.

FaaSFS integrates a POSIX file system with a serverless FaaS environment. We find that doing so allows us to overcome some of the performance challenges that impact other distributed file systems. We show that this allows us to scale existing applications to 10,000 processors, far beyond what even a large server can handle.

Key to this result is a new consistency model: Externally Consistent Sequential Consistency (ECSC). POSIX provides a linearizable [192] consistency guarantee, which is difficult to reconcile with scale and performance. Linearizability establishes a correspondence between the real-time[2] order of operations and a global logical order of operations. This is a useful property but one that is not always necessary to guarantee program correctness. ECSC provides a weaker guarantee, one closer to sequential consistency [241]. Sequential consistency also ensures that all operations correspond to a global order, but it describes this correspondence in terms of program order, rather than in terms of real time. We show that ECSC produces results that are almost indistinguishable from those obtained using linearizability—the only difference is the timing of operations. ECSC makes it possible to rearrange the real-time order of certain operations, without relying on commutativity, by maintaining logical precedence relationships instead.

ECSC is particularly effective when combined with speculative execution, a technique best known for its role in processor architecture [191]. Speculative execution has also been used to speed up distributed file systems in Speculator [290], and we build upon that approach.

[2] Linearizability is defined in terms of a precedence relationship and not in reference to wall clock time. However, the definition uses a global time model, and so this precedence relationship is an ordinal measure of real (i.e., physical) time.

Serverless computing creates obstacles to implementing the techniques traditionally used to improve the performance of distributed file systems. Scale itself poses challenges, and the FaaS environment also creates a number of specific difficulties (see Section 3.2.3). Luckily, FaaS has some useful characteristics as well: It provides an isolated execution environment and a built-in restart mechanism, both of which make it easier to implement ECSC with speculative execution.

The rest of this chapter is organized as follows. We describe background context on distributed file systems in Section 3.2. We motivate and define ECSC in Section 3.3. In Section 3.4, we describe the implementation of FaaSFS, then evaluate it in Section 3.5. We outline next steps in Section 3.7 and then summarize in Section 3.8.

3.2 Background

3.2.1 POSIX File System Behavior

The POSIX specification [218] is widely recognized as an authoritative description of how a file system should behave. It is not, however, a formally precise specification but rather a plain English one. This means that when it comes to details, POSIX is defined as much by its implementations as by the language in the specification. Developing new implementation approaches, as we seek to do here, can be challenging as a result.

An example of the non-formal language in the POSIX specification is the description of the `write` command, which includes the language, “Writes can be serialized with respect to other reads and writes” and “If a read() of file data can be proven (by any means) to occur after a write() of the data, it must reflect that write()...A similar requirement applies to multiple write operations to the same file position.” Clearly, this text specifies an important consistency guarantee, but we wonder, is it one that we are familiar with? Does it have a name? And what exactly does “after...by any means” include?

Luckily, a number of authors have developed formal models of file system behavior [36, 98, 283]. The work of Ntzik et al. is recent and thorough [294, 295], reflecting careful study of both the text of the specification and the behavior of actual file system implementations. Though it does not cover distributed file systems, the work provides a reference point for how a file system should behave.

It is clear from Ntzik et al. and other works that *linearizability* works well for modeling POSIX file systems. Linearizable operations are atomic, i.e. indivisible, and must correspond to a global total order that is consistent with the real-time precedence relationship among operations. This helps us understand what “after...by any means” actually means. We say that a linearizable operation $\mathcal{V}$ occurs after $\mathcal{U}$ if $\mathcal{U}$ ends before $\mathcal{V}$ begins. For more detail on this definition, see Section 3.3.1.

Additional complexity arises because some POSIX operations are implemented as a sequence of atomic operations and are not atomic as a whole. For example, name resolution can involve multiple sub-operations, one for each directory along a path. As a result, operations like `rename`, which is widely understood to be atomic, can still produce unexpected behavior when concurrent operations manipulate the directory hierarchy [294]. Behaviors resulting from interleaving the atomic sub-operations in POSIX, while permissible, are generally undesirable; Programmers do not write code that relies on race conditions, and would almost certainly prefer an implementation providing larger units of atomicity, say one that executes each API call atomically.

Many of the intricacies of POSIX relate to directories and metadata, whereas our work focuses on file operations. In this context, the key operations are `open`, `close`, `read`, `write`, `seek`, `sync`, `truncate`, and `flock`. We assume that the reader is generally familiar with POSIX but describe its finer points here.

`open` operates only on metadata but serves as a herald of data access to come. It verifies access permissions and translates a name to an object identifier, or *inode*. The inode is an internal file system identifier, and the calling process receives a reference to it called a *file descriptor*. `close` releases this file descriptor and destroys the reference. Flags passed to `open` can specify what sort of access is possible through a file descriptor, e.g., whether the process holding it may write anywhere, merely append, or only read. A number of performance optimizations may be taken based on file descriptor state. For example, if only one program has an open descriptor for a file then it can be granted exclusive access—linearizability is guaranteed even when it operates on a local copy. Similarly, if there are multiple open file descriptors but they are all read-only then clients can safely cache file content locally.

The `read` and `write` operations are fairly intuitive, but there are some details that are interesting. While these operations are guaranteed to be atomic, the implementation gets to choose the size of the transfer: A read may return fewer bytes than requested and a write may store fewer bytes than offered. This behavior probably originates in a desire to break up operations when encountering a delay in accessing storage, such as one encountered performing a disk seek.

Other subtleties arise from how the file length the file length interacts with reads and writes. It is tempting to imagine that reads or writes are independent and commutative when they operate on non-overlapping byte ranges within a file, but this is not the case. A write automatically extends the length of the file; If the write position is beyond the end of the file then the gap is filled with zeros. A read beyond the end of the file returns a zero-length result (indicating EOF), but a write that comes before it can change this to a zero-filled result even when the bytes written do not overlap with those read. A single-byte write can thus potentially impact reads on a broad range of offsets, limited only by maximum file size. The `truncate` operation can have similarly global effects, and it may produce both increases and decreases in the file length.

Work on improving file system performance on multiprocessor systems has shown that non-POSIX interfaces with greater commutativity also can scale better [139]. Clements et al. [103] introduce the scalable commutativity rule to describe such behavior and provide a toolkit for state-dependent interface commutativity analysis. In FaaSFS, we prioritize compatibility with existing applications. Instead of achieving scalability by using commutative operations, we instead relax the real-time correspondence imposed by linearizability.

Most of the POSIX specification describes failure-free operation. It does provide the `sync` operation, which flushes data and metadata to disk, so they will survive system crashes or power failures. For crashes that occur prior to flush completion, there are no guarantees, and the atomicity provided during failure-free operation can even be violated [73, 320].

3.2.2 Distributed File Systems

There are several file systems that aim to provide shared storage to multiple machines while adhering to POSIX, or something close to it. All of them are faced with the same challenge: Processes on the same machine all have access to the same physical memory, which allows them to use a shared cache and makes low-latency coordination possible; however, in a distributed setting, local caches are independent, and all coordination involves network communication. There are several approaches that distributed file systems have taken in response to this difference.

3.2.2.1 Weakened Consistency

The original Network File System (NFS) [348] made few guarantees about consistency. Writes could be buffered at the client, then propagated to the server at some time in the future. Reads could be served from cached data. Applications had no way of obtaining the latest data and no way to influence which version would prevail in case of write conflicts. The term "eventual consistency" would not be defined until later [398], but it describes this situation well.

The Andrew file system introduced close-to-open consistency [224], which was later incorporated into NFS Version 3 [313]. Close-to-open consistency ensures that if process `A` closes a file before process `B` opens the file, then any updates made by `A` will be visible to `B`. While this guarantee can be a useful building block, applications generally must coordinate access to the file in order to achieve well-defined behavior. If multiple file descriptors are open concurrently, and one or more of them is used for writing, then the results are just as unpredictable as in early versions of NFS. Applications can coordinate by communicating with one another directly over the network. They may also coordinate using file system locks, which can be provided by via a dedicated lock manager (e.g. [68]) or integrated into the distributed file system protocol, as in NFS Version 4 [185].

These forms of weakened consistency introduce behaviors that local POSIX file systems do not allow. They can be confusing for users and developers because they present the same POSIX-interface, yet behave differently.

3.2.2.2 Coordinated Local Processing

The simplest way to fully conform to POSIX is to process all requests at a centralized server, but this adds latency and creates a scalability bottleneck. Lustre [75, 359] is a scalable and fully POSIX-compliant distributed file system that uses several techniques to improve on this performance. It is representative of the state of the art.

Lustre uses separate servers to manage metadata (directories) and data (file content). Clients communicate directly with these servers, e.g., first communicating with a metadata server while opening a file, then with an object storage server while accessing its content.

Lustre uses an internal locking mechanism to coordinate client access to file and directory data, guaranteeing what it describes as "cache coherent" behavior. Such locks are distinct from the file locks exposed through the POSIX API (via the `flock` and `fcntl` calls). Their purpose is to allow client-side caching to coexist with atomic and linearizable execution. However, they work in basically the same way, with shared and exclusive states.

Lustre also supports *intents*, which are an extension to the locking mechanism. Intents contain operations that the server may choose to execute instead of granting a lock [428], which can offer greater performance in cases of heavy contention.

Lustre's locks have a parallel to the *delegation* technique available in NFS Version 4 [185]. Delegations are also an internal locking protocol of the file system implementation. When used in NFS, they can allow the client to process open, close, lock, and unlock operations locally.

We note that any lock-based concurrency control technique will encounter problematic scenarios such as deadlock and unresponsive clients. Distributed deadlock detection [94] and leases [169] can help, but these problems may be exacerbated by scale.

3.2.3 The FaaS Environment

Chapter 2 provides a broad overview of serverless computing, including descriptions of FaaS (Section 2.2 and Section 2.5.1) and its limitations (Section 2.6.1). In this section, we focus on specific aspects of FaaS that are relevant for the design of FaaSFS.

In designing FaaSFS, we needed to account for a number of ways that FaaS differs from a traditional server environment. We focus on AWS Lambda but note that these restrictions are broadly similar across major cloud providers.

FaaS characteristics relevant to the design of FaaSFS include:

- *Ephemeral function state.* Cloud providers create execution environments when functions start running and can destroy them when not in use. In general, cloud providers keep instances cached because it is costly to create and initialize them [296, 429], and this allows applications to cache state inside them as well. However, any cached state will be lost if the cloud provider chooses to reclaim the instance resources.
- *Limited system privileges.* Lambda offers a controlled environment that prohibits operations such as loading kernel modules or mounting NFS shares (except via the EFS [24] integration; See Section 3.5). Thus, in our FaaSFS implementation processes communicate directly with a user space file server, rather than using the established FUSE user space file system interface [147]

- *Function instances freeze between invocations.* Lambda instances run only when they are processing a request invoked through the API. In the frozen state, no processing occurs, even if data arrives on the network or a timer is scheduled to raise a signal. Function instances can retain cached state while frozen, but there is no way to update that state until the instance begins processing a new request. This also makes it impractical to use delegations [201] or leases [169], which are proven distributed file system optimizations, because we may be unable to process revocations on demand.
- *No inbound network connections.* Cloud functions live behind a NAT layer that prohibits inbound network connections. While some workarounds have been demonstrated [423, 432], direct communication between functions is not part of the programming model. This is helpful for our approach because functions instead interact through shared state, where FaaSFS can mediate those interactions.
- *No names for function instances.* While Lambda may create many instances of a cloud function, as dictated by load, there is no way to route an invocation to a particular instance (see Section 2.6.1); each invocation could go to any function instance. Partitioning a cache is possible by defining one copy of the function for each partition. This is straightforward, though not scalable or elegant.
- *Limited execution time.* In many practical applications, cloud functions run for as little as a fraction of a second. Run times of minutes are supported, but there is always some bound on their execution time. Short function duration is helpful for speculative execution because it limits the amount of work that might need to be repeated when speculation fails.
- *Function-grained fault tolerance.* FaaS usually provides an at-least-once execution model. Functions must be safe to retry, a requirement that usually leads programmers to write idempotent code. Transactions can make it easier to provide idempotence, as previous work on serverless computing has shown [383, 459].

3.3 Externally Consistent Sequential Consistency

This section provides a relatively informal introduction to externally consistent sequential consistency (ECSC), the consistency model implemented in FaaSFS. A more formal definition and proofs of its properties are found in Chapter 4.

ECSC sits between two well known strong consistency guarantees: linearizability [192] and sequential consistency [241]. All results produced by ECSC are consistent with linearizable execution, which allows it to serve as a drop-in replacement where linearizability is required. As we will see in Section 3.3.3 and in the evaluation of Section 3.5, ECSC permits implementations that have better performance than is possible with linearizability.

ECSC relies on one key insight: We can relax linearizability's real-time correspondence requirements. During intervals where programs run in isolation and communicate only with storage but not otherwise with the outside world, the real-time order of operations becomes irrelevant. In its place, we can substitute a precedence rule based on program order, just as sequential consistency does. Sequential consistency allows programs to "read from the past" or "write to the future," which allows distributed systems to achieve improved performance at scale by using caching and buffering.

3.3.1 Linearizability and Sequential Consistency

Linearizability [192] and sequential consistency [241] are similar since they both require all operations to behave as if they were executed in sequence. However, they make different assumptions and enforce different properties.

Sequential consistency merely assumes that we can determine the order of operations at each program and requires that these orders be respected by some global order. In contrast, linearizability models separate invocation and response events for all operations. It assumes that these events exist within a well-defined global order that induces a global precedence relationship (a partial order) across all operations.

Linearizability is described using a *global time assumption* [4]. Time is defined in an ordinal sense, and while the model makes no reference to clocks, it defines a real-time precedence order that is what one would measure using clocks.

Sequential consistency is described using a local sense of order only—it considers the order of operations at each processor and ensures that it is possible to construct some corresponding global order. For example, let $\mathcal{H}_1^{\mathtt{A}}$ and $\mathcal{H}_1^{\mathtt{B}}$ denote the history of operations at two processors, `A` and `B`. Consider the following pair of histories:

$$\begin{aligned}\mathcal{H}_1^{\mathtt{A}} &: \mathtt{W_A(X;1)W_A(Y;1)}\\ \mathcal{H}_1^{\mathtt{B}} &: \mathtt{R_B(X;0)R_B(Y;1)}\end{aligned}$$

In this notation, $\mathtt{W_A(X;1)}$ denotes a write of the value 1 by process `A` to the variable `X`. Similarly, $\mathtt{R_B(X;0)}$ denotes a read by process `B` of the variable `X`, resulting in the value 0.

These histories satisfy sequential consistency because it is possible to construct a global history in which all of the operations appear in the same order as they do at each processor:

$$\mathtt{R_B(X;0)W_A(X;1)W_A(Y;1)R_B(Y;1)}$$

Consider, on the other hand

$$\begin{aligned}\mathcal{H}_2^{\mathtt{A}} &: \mathtt{W_A(X;1)W_A(Y;1)}\\ \mathcal{H}_2^{\mathtt{B}} &: \mathtt{R_B(Y;1)R_B(X;0)}\end{aligned}$$

There is no way to construct a global history consistent with these processor histories, so $\mathcal{H}_2^{\mathtt{A}}$ and $\mathcal{H}_2^{\mathtt{B}}$ do not satisfy sequential consistency.

To define linearizability, we need to extend our notation to denote the beginning and end of each operation, which we define in terms of *invocation* and *response* events. For example, we can break down the operation $\mathtt{W_A(X;1)}$ into the sequence $\textsc{In}(\mathtt{W_A(X;1)})$ $\textsc{Re}(\mathtt{W_A(X)})$. We also must define a global history, which we will consider instead of per-processor histories. The implicit assumption in doing this is that each invocation or response represents an event that occurs at a unique moment and that all such events can be placed in a total order. This is not precisely the same thing as a measure of time, e.g., there is no measure of the duration between events, but it takes for granted an underlying reality where everything happens in order. This is known as a global time model [4].

For example, consider the history

$$\mathcal{H}_3 : \textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{X};1))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{X}))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{X}))\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{Y};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{X};0))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{Y};1))$$

Figure 3.2 illustrates a set of intervals that correspond to these events.

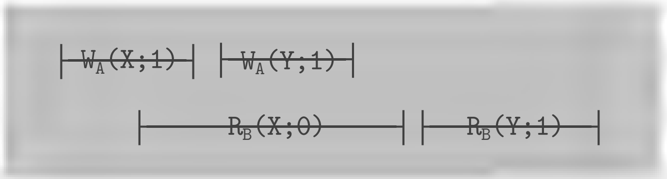


Figure 3.2: Diagram for $\mathcal{H}_3$.

An operation $\mathcal{U}$ is said to precede an operation $\mathcal{V}$ in $\mathcal{H}$, written $\mathcal{U} \prec_\mathcal{H} \mathcal{V}$, if $\textsc{Re}(\mathcal{U})$ appears before $\textsc{In}(\mathcal{V})$, written $\textsc{Re}(\mathcal{U}) <_\mathcal{H} \textsc{In}(\mathcal{V})$. A history is linearizable if it can be rearranged to a valid *sequential history* while respecting $\prec_\mathcal{H}$. In such a sequential history each invocation must be followed immediately by its corresponding response and the values returned must respect the behavior of the storage object (a read returns the value most recently written).

The precedence relationships in $\mathcal{H}_3$ are

$$\begin{gathered}
\mathtt{W}_\mathtt{A}(\mathtt{X};1) <_{\mathcal{H}_3} \mathtt{W}_\mathtt{A}(\mathtt{Y};1)\\
\mathtt{W}_\mathtt{A}(\mathtt{X};1) <_{\mathcal{H}_3} \mathtt{R}_\mathtt{B}(\mathtt{Y};1)\\
\mathtt{W}_\mathtt{A}(\mathtt{Y};1) <_{\mathcal{H}_3} \mathtt{R}_\mathtt{B}(\mathtt{Y};1)\\
\mathtt{R}_\mathtt{B}(\mathtt{X};0) <_{\mathcal{H}_3} \mathtt{R}_\mathtt{B}(\mathtt{Y};1)
\end{gathered}$$

A sequential history equivalent to $\mathcal{H}_3$ is

$$\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{X};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{X}))\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{Y};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{Y}))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{X}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{X};0))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{Y};1))$$

Another is

$$\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{X}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{X};0))\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{X};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{X}))\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{Y};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{Y}))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{Y};1))$$

In contrast, the history $\mathcal{H}_4$ is not linearizable, even though it is sequentially consistent, because it contains the precedence relationship $\mathtt{W}_\mathtt{A}(\mathtt{X};1) \prec_\mathcal{H} \mathtt{R}_\mathtt{B}(\mathtt{X};0)$, which cannot be reconciled with a valid state transition history for $\mathtt{X}$:

$$\mathcal{H}^4 : \textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{X};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{X}))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{X}))\textsc{In}(\mathtt{W}_\mathtt{A}(\mathtt{Y};1))\textsc{Re}(\mathtt{W}_\mathtt{A}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{X};0))\textsc{In}(\mathtt{R}_\mathtt{B}(\mathtt{Y}))\textsc{Re}(\mathtt{R}_\mathtt{B}(\mathtt{Y};1))$$

Past work has compared the theoretical performance of linearizability and sequential consistency [43]. In situations that are either write-heavy or read-heavy, sequential consistency can be provided at much lower latency.

One nice property of linearizability is its composability—if two storage systems or objects each satisfy linearizability, then when combined, they produce a system that also satisfies linearizability. This is a nice property for building large systems because it allows components to be designed and implemented independently. In contrast, maintaining a sequential consistency guarantee while composing systems requires some form of shared implementation, e.g., a cross-component logical clock or other mechanism to allow components to agree on a common sense of order.

3.3.2 Defining Externally Consistent Sequential Consistency

Figure 3.3 shows the example that we will use to explain ECSC. We consider a system with three end-users: Alice, Bob, and Charlie. We assume that Alice and Bob are sitting in the same room, where they are able to talk to one another, while Charlie is in a remote location. Each individual interacts with a cloud-based computing system using their own personal devices.

In this example, Alice executes a FaaS program f_1, and after it returns, tells Bob to go ahead and execute f_2. Concurrently with these two operations, Charlie executes f_3. The functions operate on three variables $(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$. f_1 increments $\mathtt{X}$, f_2 increments $\mathtt{Y}$, and f_3 sets $\mathtt{Z}$ to $2 \times \mathtt{Y} + \mathtt{X}$, evaluating the expression from left to right. We assume all variables are initially zero.

Figure 3.4 shows several scenarios for how these functions might execute. Time runs from left to right, and we show the interval corresponding to each underlying read and write operation, denoting the invocation event with "$\vdash$" and the response event with "$\dashv$". Figure 3.4 (a) shows an example linearizable execution, whereas (b) shows a very similar execution that is not linearizable. The two are equivalent to all outside observers as well as to the functions themselves. The difference is that in (a) $\mathtt{W_B}(\mathtt{Y}; \mathtt{1})$ is concurrent with $\mathtt{R_C}(\mathtt{Y}; \mathtt{0})$, whereas in (b) $\mathtt{W_B}(\mathtt{Y}; \mathtt{1}) \prec_{\mathcal{H}} \mathtt{R_C}(\mathtt{Y}; \mathtt{0})$, which is not compatible with linearizability.

Why would a system allow the execution of Figure 3.4 (a) but not that of Figure 3.4 (b)? Linearizability knows nothing about f_1, f_2, or f_3—it is defined purely in terms of the invocations and responses of individual read and write operations. ECSC incorporates information about the communication between processes, and takes advantage of sequences of operations that occur in isolation.

In Table 3.1, we present a simplified definition of ECSC that assumes that all programs represent function invocations and that each function executes in isolation. Communication comprises only 1) storage invocations, 2) storage responses, 3) input at the beginning of the function invocation, and 4) output at function return. ECSC respects the order of operations at each program, like sequential consistency. However, instead of relying on the real-time order between operations, as linearizability does, it uses the real-time order defined by their enclosing functions (see Figure 3.5). We give a more complete and formal definition of ECSC in Chapter 4.

	$\mathcal{U} \prec_{\mathcal{H}} \mathcal{V}$ when
Linearizability	$\textsc{Re}(\mathcal{U}) <_{\mathcal{H}} \textsc{In}(\mathcal{V})$
Sequential consistency	$\textsc{Re}(\mathcal{U}) <_{\mathcal{H}} \textsc{In}(\mathcal{V}) \wedge \textsc{Func}(\mathcal{U}) = \textsc{Func}(\mathcal{V})$
ECSC (isolated FaaS)	$\textsc{Re}(\mathcal{U}) <_{\mathcal{H}} \textsc{In}(\mathcal{V}) \wedge \textsc{Func}(\mathcal{U}) = \textsc{Func}(\mathcal{V})$ $\vee\ \textsc{Re}(\textsc{Func}(\mathcal{U})) <_{\mathcal{H}} \textsc{In}(\textsc{Func}(\mathcal{V}))$

Table 3.1: Precedence relationships that define various consistency models. $\mathcal{U}$ and $\mathcal{V}$ denote operations such as reads and writes to storage. $\textsc{In}(\mathcal{U})$ and $\textsc{Re}(\mathcal{U})$ denote events corresponding to the invocation and response for operation $\mathcal{U}$. $\textsc{Func}(\mathcal{U})$ denotes the function invocation (i.e., execution instance) that issued operation $\mathcal{U}$. $\textsc{In}(\textsc{Func}(\mathcal{U}))$ and $\textsc{Re}(\textsc{Func}(\mathcal{U}))$ are the corresponding invocation and response events for that function execution instance; i.e., the delivery of a message containing the input and the sending of a message containing the output.

Figure 3.4 (c) shows an example that is sequentially consistent but does not satisfy ECSC. We note that there is no linearizable execution that results in $z = 2$.

3.3.3 Simulating ECSC

To demonstrate the benefits of ECSC, we developed a simulation derived from the contention microbenchmark described in Section 3.5.3. Each client reads 10 blocks at random from a file containing 100 blocks, then writes one block. At this point, it communicates with the outside world, as FaaS instances do between function invocations.

We simulated clients with local caches large enough to hold the entire working set. We evaluated both linearizability and ECSC, and we simulated each consistency guarantee with and without speculative execution.

In the absence of speculative execution, we implemented caching for linearizability with a standard cache coherence protocol [306]. This models the behavior of distributed file systems such as NFS [185], which can provide shared or exclusive client delegations on file regions. For ECSC, we use an adaptation of the Tardis [452] cache coherence protocol, which ensures sequential consistency using logical leases (see Section 4.5).

With speculative execution, clients proceed using locally cached state but validate their assumptions before producing output. For linearizability, we compare read versions to the

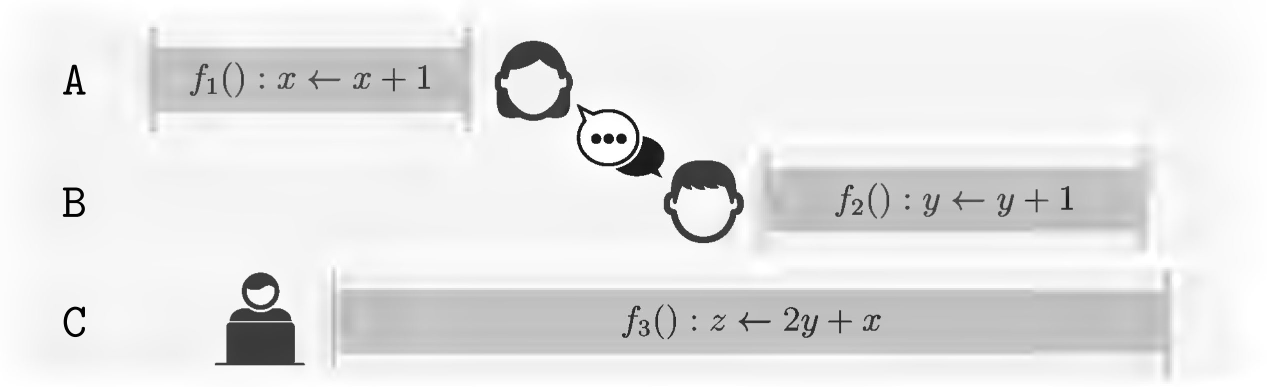

A
$f_1() : x \leftarrow x + 1$
B
$f_2() : y \leftarrow y + 1$
C
$f_3() : z \leftarrow 2y + x$

Figure 3.3: ECSC example: Alice, Bob, and Charlie run FaaS functions f_1, f_2, and f_3. Alice and Bob communicate, creating an external dependency and the precedence relationship $f_1 \prec_{\mathcal{H}} f_2$. Charlie accesses the system from a remote location, which adds some latency. Perhaps as a result, f_3 appears to run concurrently with f_1 and f_2. Time runs from left to right.

latest versions at a central server, which is similar to Speculator [290]. For ECSC, we speculatively extend the logical leases, as Sundial [455] does, then validate these lease extensions at the central server. If we fail to validate, we reset the cache and retry.

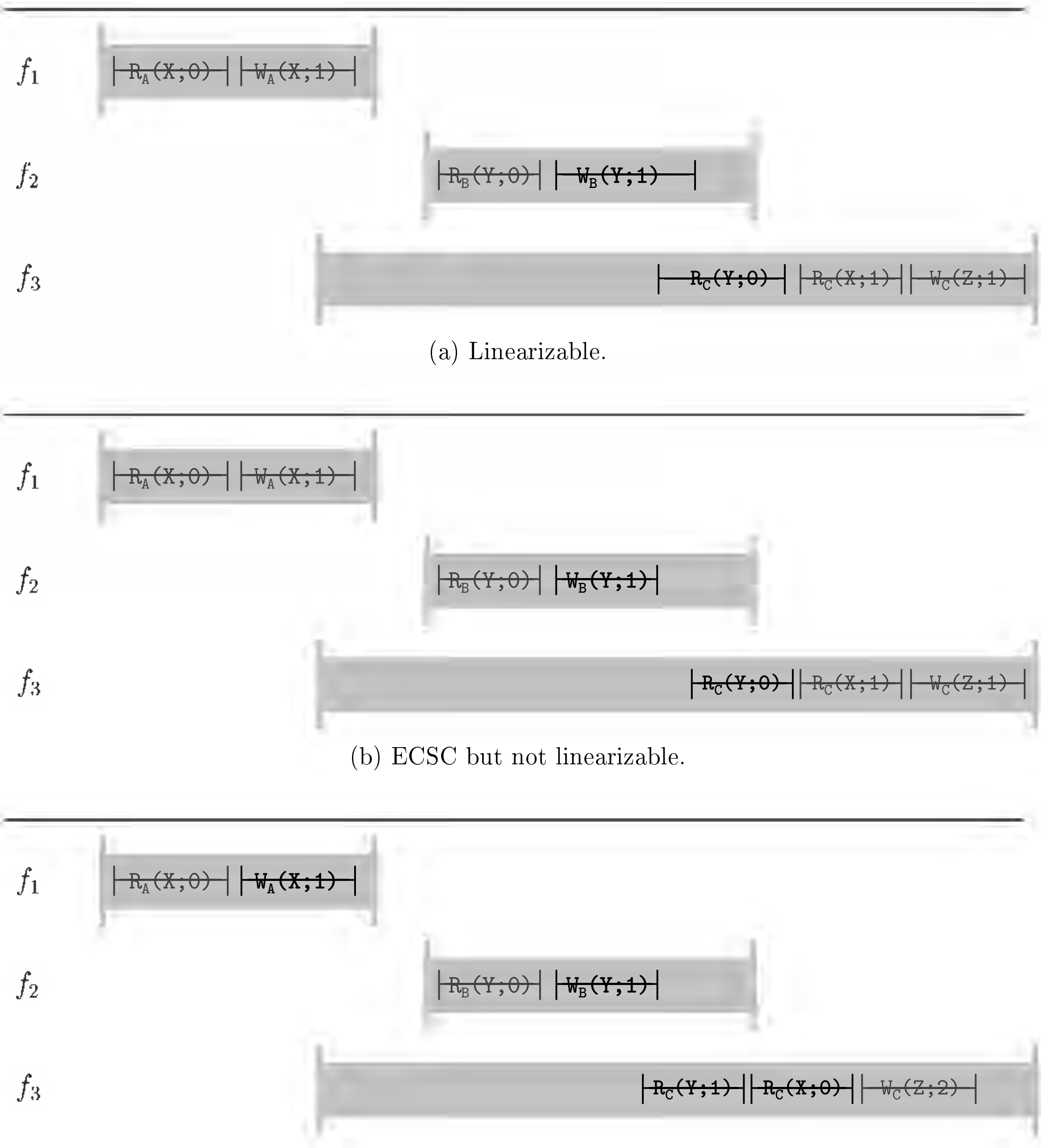
As shown in Figure 3.6, without speculative execution, both ECSC and linearizability provide similar performance. For this workload, that can be worse than an implementation where each operation is handled individually at a central server (denoted "No cache"). With speculative execution, ECSC provides significantly improved scalability over linearizability. In this model, the one-way network delays are drawn from a uniform distribution representing the range of 100,000 to 150,000 μs, a range we chose to be roughly representative of the delays within a commercial data center. In this example, the writes are all at the end of the transaction, which means that ECSC experiences no aborts. Applications that can buffer their writes can generally achieve this effect.

3.3.4 External Consistency and Other Transactional Guarantees

We have borrowed the term *external consistency* from the language of transaction systems. Originally defined in Gifford's doctoral dissertation [165], external consistency was later popularized by Spanner [113], Google's globally distributed database.

External consistency is similar to *strict serializability* [192], which is equivalent to linearizability when each operation is a serializable database transaction (see Section 4.4). What sets external consistency and strict serializability apart from other transactional guarantees is that they enforce real-time correspondence at the transaction boundaries.

The situation is similar with ECSC. Communication with the outside world leads to the enforcement of real-time precedence relationships. Between episodes of communication with



(a) Linearizable.

(b) ECSC but not linearizable.

(c) Sequentially consistent but not ECSC or linearizable.

Figure 3.4: ECSC example scenarios. (a) $\mathtt{W_B(Y;1)}$ is concurrent with $\mathtt{R_C(Y;0)}$, as is permissible under linearizability; (b) $\mathtt{W_B(Y;1)} \prec_{\mathcal{H}} \mathtt{R_C(Y;0)}$, violating linearizability but not ECSC because f_2 and f_3 are concurrent; (c) f_3 contains $\mathtt{R_C(Y;1)} \prec_{\mathcal{H}} \mathtt{R_C(X;0)}$, which is permitted by sequential consistency but not ECSC, which must maintain $\mathtt{W_A(X;1)} \prec_{\mathcal{H}} \mathtt{W_B(Y;1)}$. We assume all values are initially 0.

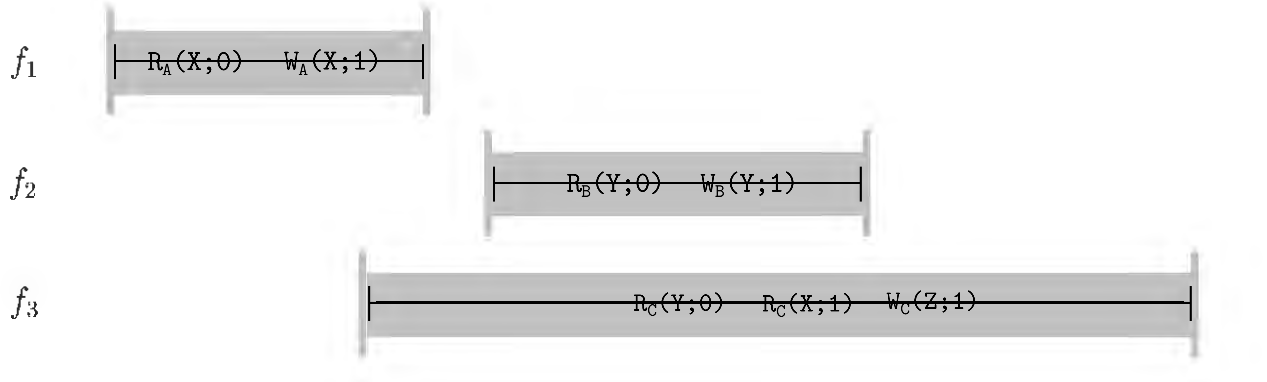


Figure 3.5: Whereas linearizability enforces real-time correspondence on the basis of individual operations (see Figure 3.4), ECSC enforces real-time correspondence at times of external communication.

the outside world, logical consistency (logical rather than real-time precedence) suffices to ensure correct behavior.

3.4 Implementation of FaaSFS

3.4.1 Overview

In a departure from the general approach to ECSC presented in Section 3.3, our implementation uses a multiversion transaction mechanism. Section 4.4 shows that doing so provides a valid implementation of ECSC when all communication occurs at the end and beginning of the function. We adopted this approach mainly because it allows us to use proven techniques for implementing distributed databases.

Transactions in ECSC are created transparently by the FaaS environment and execute with strict serializability [192]. They begin when a function starts executing and commit when it completes. We deploy this mechanism as a custom runtime layer in AWS Lambda, so no changes to application code are needed. Should a transaction abort because some speculative assumption fails, we re-execute the function.

Transactions also make it easier to write programs that maintain correctness, including for security, under concurrency. They can also be added to the file system specification with a simple extension [323, 357], and they could allow the FaaSFS approach to be used outside FaaS.

We chose to implement FaaSFS from scratch rather than by modifying an existing file system or building on top of an existing database. We hoped that this would help us achieve our aims because it allows integration of the caching mechanism and the concurrency control mechanism. Prior work, such as Sundial, shows that this is valuable [455]. Also, applications expect file systems to provide latency and throughput performance that database systems

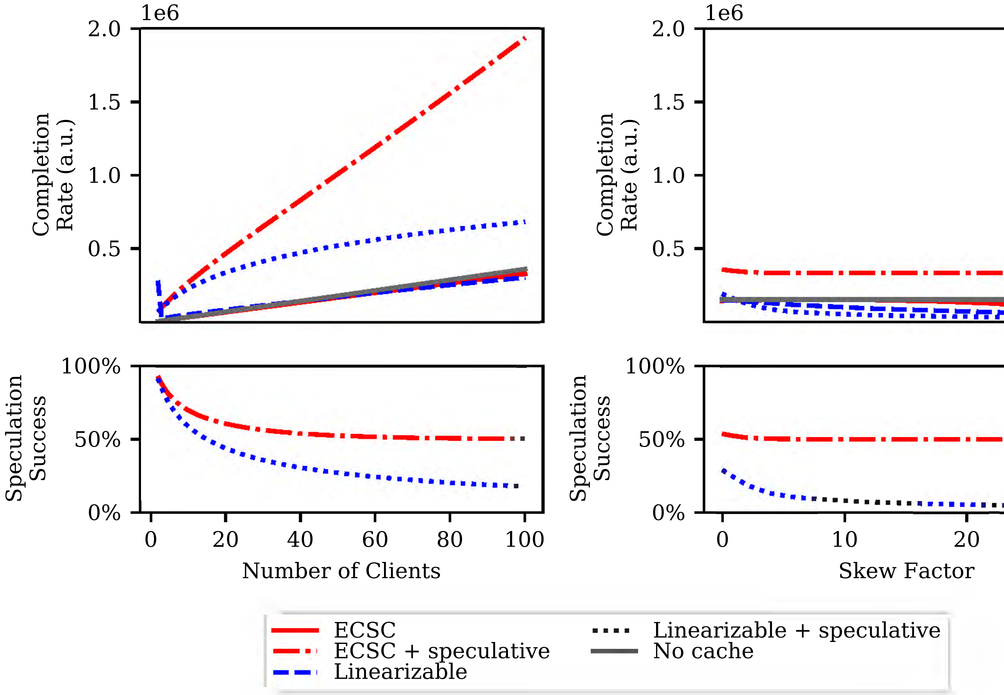


Figure 3.6: ECSC simulation on a read-intensive workload. On the left blocks are selected from a uniform distribution while the number of clients varies. On the right the number of clients is fixed at 50 and the skewness of the access distribution is varied. Here a skew factor of n means that $1/(2+n)$ of the blocks receive $(1+n)/(2+n)$ of the requests. Speculative execution scales better when used with ECSC than with linearizability because ECSC is more permissive about the real-time ordering of storage operations.

often cannot match. FaaSFS comprises roughly 4,000 lines of C and 40,000 lines of Go. Table 3.4 lists the POSIX operations that we implemented and their completeness.

3.4.2 FaaSFS Client Components

Figure 3.7 shows the components of FaaSFS alongside an application. In the discussion that follows, we work our way down the stack from the application level to the FaaSFS backend and the underlying object storage.

3.4.2.1 System Call Intercept

Operating in the AWS Lambda environment, we have no ability to modify or configure the kernel (see Section 3.2.3). The `ptrace` system call is also blocked, so we resort to binary

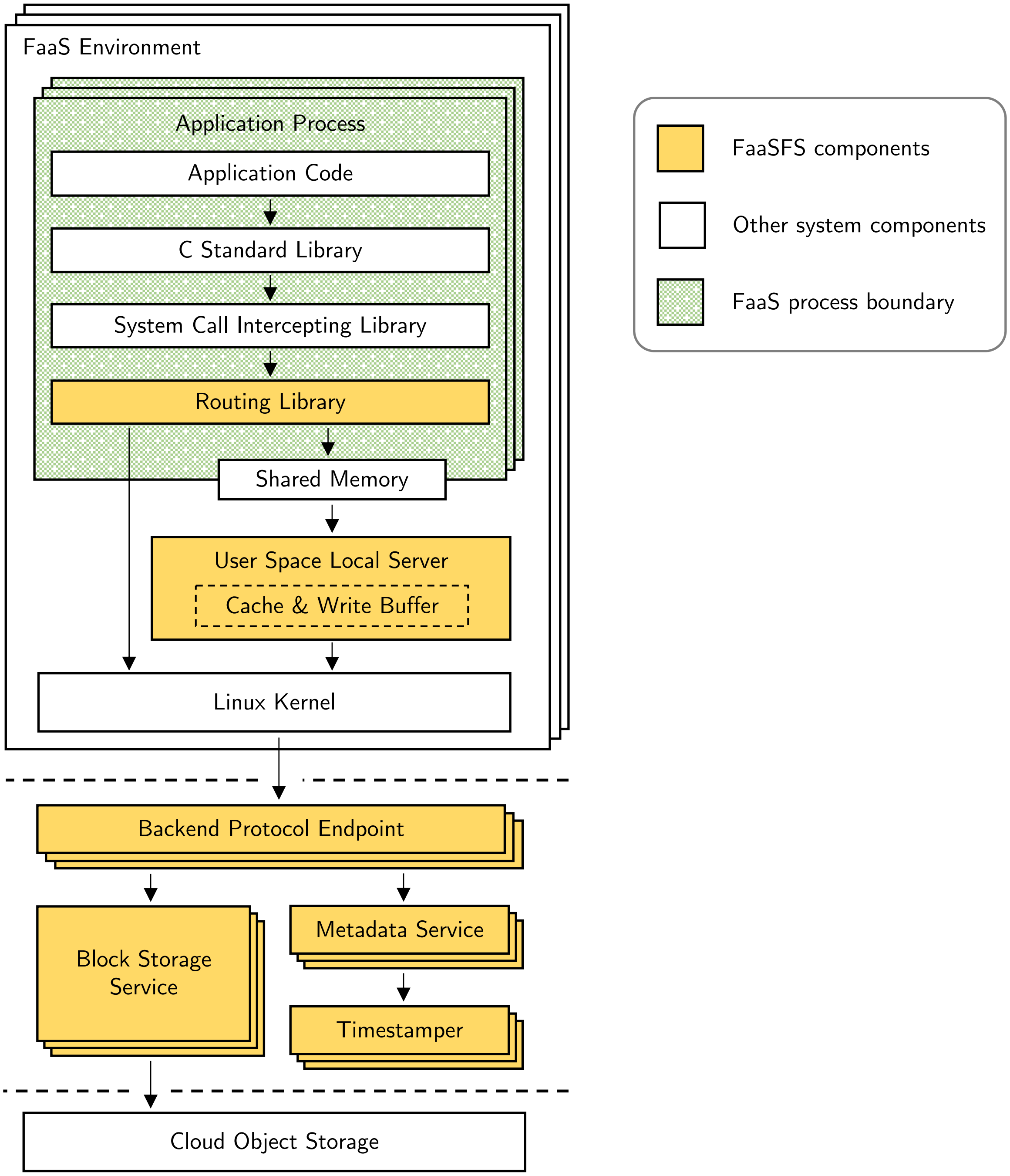


Figure 3.7: Overview of FaaSFS. Yellow indicates FaaSFS components (software that we wrote). Dashed lines delineate the interfaces between clients, the backend services, and cloud object storage. The Metadata Service, Block Storage Service, and Timestamper are all replicated stateful services. The Block Storage Service and Metadata Service scale by partitioning, whereas the Backend Protocol Endpoint and FaaS Environment scale as stateless services.

modification via hot patching to intercept system calls. We do this using the System Call Intercepting Library [399] provided as part of the Persistent Memory Development Kit [351]. We use `LD_PRELOAD` [327] in an AWS Lambda custom runtime [114] to install this intercept routine before the Python interpreter starts up. In our workloads, all system calls originate in a limited set of shared libraries—the C standard library, the pthread library, or the dynamic linker library—so these are the only binaries that require patching.

3.4.2.2 Routing

The FaaSFS Routing Library runs in the address space of the application and registers a handler with the System Call Intercepting Library. This handler gets invoked ahead of all system calls and can either let them pass unchanged or substitute an alternative implementation (just like `ptrace`). For those calls corresponding to the POSIX file system, we must perform a routing decision using arguments such as the path name or the file descriptor to determine whether the operation should go to FaaSFS or the underlying operating system for local file system access. For paths, we test the prefixes, e.g., `/mnt/faasfs`, normalizing them to account for relative paths. Some delicate bookkeeping is required. If a FaaS function forks child processes, we pass this critical bookkeeping information along in environment variables. The routing library is written in C.

3.4.2.3 Shared memory IPC

The Routing Library and the Local Server communicate using a shared memory area. We maintain a set of buffers, configurable in number and size, to allow for concurrent requests. By default, we provide 10 buffers, each 2 MB in size. A client, which may correspond to either a thread or a process, first checks out a buffer (it attempts to reclaim the last-assigned buffer using an atomic compare and swap operation to verify ownership and set a busy bit, but if the buffer has been reassigned this operation fails and the client falls back to IPC with the server). It then writes the request data and marks it as ready for processing by the server. The server will busy-wait, spinning for up to 16 μs before falling back to wait on a semaphore. The response works the same way, with the client first spinning in hopes of receiving a low-latency response before turning to operating system support for coordination. We adapted elements of this technique from the Intel Thread Building Blocks [319]. We evaluate the efficiency of the IPC mechanism in Section 3.5.1.

3.4.2.4 User Space Local Server

The FaaSFS User Space Local Server (LS) runs in a separate process from the application, and each instance of a cloud function runs one such process. The LS serves as a shared cache for all processes in the function and buffers their writes as well. It intermediates all network communication with the FaaSFS Backend Service. Both the LS and all backend services are written in Go, and we use gRPC [173] for communication between them.

FaaSFS supports several cache update policies. LS may check for updates on all cached objects when a function begins executing, it may check each object on first use, or it may rely solely on the optimistic concurrency control's commit mechanism to identify outdated local state. In our evaluation we use only the first approach, as we found that checking all versions when the function begins executing works best for our workloads.

3.4.3 FaaSFS Backend

The FaaSFS backend is designed with scalability in mind and comprises a number of separate components. A key design principle is to keep to a minimum the amount of work that gets done in an ordered sequential context. Our prototype is a single-tenant implementation, but we imagine that a cloud provider deploying FaaSFS to its customers would develop a multi-tenant implementation.

State management is divided between two components: a Block Storage Service (BSS) and a Metadata Service (MDS). BSS is responsible for handling the bulk of the bytes stored in the file system and for moving them in and out of object storage with low overhead. Unlike cloud object storage, where the client provides a name for the key by which it may be retrieved, BSS generates keys and provides them to the client. MDS is responsible for recording directory information, and for tracking which stored blocks comprise each version of a file. At a high level, this design is similar to the pattern used in other scalable file systems [86, 195, 359, 369, 403]. However, the FaaSFS backend is targeted specifically to the cloud environment, as is especially evident in the design of BSS (see Section 3.4.3.4).

3.4.3.1 Backend Protocol Endpoint

The Backend Protocol Endpoint (BPE) provides a unified access point for clients and coordinates the tasks of the underlying services. As illustrated in Figure 3.9, assembling an API response can require interacting with multiple services and their partitions, and BPE hides this from the client.

BPE is stateless in the same sense as FaaS functions are, i.e., it has ephemeral state only (see Section 2.6.1). This makes it easily scalable. Clients maintain a persistent TCP connection to amortize connection overheads and allow request pipelining. The BPE and its associated client are fate-sharing: If a BPE instance fails, the associated FaaS client fails as well.

3.4.3.2 Metadata Service

The Metadata Service (MDS) maintains a current and historical record of file content, represented by references to data stored in BSS. The history allows reads of the file at past timestamps (the retention policy is configurable, with 10 versions retained by default). MDS also maintains the content of directories. State in MDS is partitioned among several Raft logs [299], each of which serves as the transition history of a replicated state machine responsible for a subset of inodes. We use the Dragonboat library [130], which provides a high-performance pipelined implementation of Raft that amortizes the overheads of leader election and replication across many individual logs. In practice, we set the number of logs to 12× the number of cores. MDS is partitioned and uses two-phase commit for distributed transactions.

3.4.3.3 Timestamp Service

In its present implementation, FaaSFS relies on a centralized timestamp mechanism provided by the Timestamp Service (TSS). This is a single Raft state machine that merely implements a counter. We have found that the throughput of TSS is limited to roughly 500,000 requests per second. This is presently the only global resource for which we have not provided a scalable solution. We imagine that this could be replaced with a more scalable replicated state machine, such as Compartmentalized MultiPaxos [437], or perhaps by wall clocks as in Spanner [113].

3.4.3.4 Block Storage Service

The Block Storage Service (BSS) is designed to provide an efficient and scalable mechanism for storing and retrieving data from underlying cloud object storage. Our implementation works with AWS S3, preserving its advantages and layering on the versatility needed to support FaaSFS. We view BSS as a new general-purpose component that could be used to implement other stateful serverless systems, and so we describe its design in some detail.

Cloud object storage provides low-cost, long-term, highly durable data storage (see the comparison of cloud storage alternatives by Jonas et al. [216]). It provides high-throughput for large transfers, but small operations are particularly costly in terms of both price and performance; whereas a single large operation can saturate the client's network interface, latency, which ranges from 10 ms to 20 ms, is much higher than the network latency. Access is also billed on a per-request basis, regardless of the amount of data transferred, which makes small transfers expensive. In addition, cloud object storage offers a restrictive interface: objects are stored as sets of immutable versions, and any update to an object requires uploading a new version.

BSS forms a caching and buffering layer that enables low latency and high throughput, even for small-block operations, while preserving the durability and cost-effective long-term retention of the underlying cloud object storage. BSS is not without trade-offs, however. Its interface is even more restrictive than that of cloud object storage, which makes possible strong performance along all these measures.

The BSS API, shown in Table 3.2, is similar to key-value storage—however, it uses object references (keys) that are generated by the system, rather than ones provided by the client. By generating references using a scalable UUID mechanism, BSS achieves write-once semantics without using coordination protocols. For each block $\mathcal{B}$ provided via `put`, BSS creates an access reference $\mathcal{R}$ that can be used to retrieve it via `get`. These references are

Table 3.2: Block storage service API.

`put`$([]\mathcal{B})$: $[]\mathcal{R}$
`get`$([]\mathcal{R})$: $[]\text{Option}\langle\mathcal{B} \mid \textsc{Unknown} \mid \textsc{Dropped}\rangle$
`drop`$(\mathcal{R}_{seg})$

only partially opaque, as they comprise two parts $\mathcal{R} = (\mathcal{R}_{seg}, \mathcal{R}_{index})$. BSS combines many blocks into a single *segment*, which corresponds to an object in the underlying storage. $\mathcal{R}_{seg}$ is a reference to this object whereas $\mathcal{R}_{index}$ is an offset within it. Data can only be removed one segment at a time, via `drop`.

Whereas `put` provides strong consistency for subsequent `get` operations, `drop` operates with eventual consistency. We summarize the BSS consistency guarantees as follows:

1. Before BSS returns $\mathcal{R}$, `get`$(\mathcal{R})$ must return UNKNOWN.
2. After `put`$(\mathcal{B})$ returns $\mathcal{R}$, `get`$(\mathcal{R})$ must return $\mathcal{B}$, unless `drop`$(\mathcal{R}_{seg})$ was issued.
3. After `drop`$(\mathcal{R}_{seg})$ succeeds, `get`$(\mathcal{R})$ may return DROPPED.

Figure 3.8 illustrates the internal architecture of BSS and illustrates how it achieves durability. Each BSS server accumulates writes independently, constructing a segment and replicating it up to n times for fault-tolerance. Once the primary receives m replies, it acknowledges `put` completion to the client. Once a segment reaches either the size limit or age limit, BSS stops adding blocks to it, at which point we say the segment is *finalized.* BSS then proceeds to save it to cloud object storage. If a replica fails to receive a heartbeat or segment updates from the primary, it will independently finalize the segment (rejecting future updates) and write it to cloud object storage. It is thus possible for multiple versions of a segment to exist in cloud object storage, but once acknowledged, a block written to BSS has at least m paths to cloud object storage.

Table 3.3: Time window to save to cloud object storage, as given by T_{store} in Equation 3.3.

m	In-Memory Replication	On-Disk Replication
2	3.1 μs	31 ms
3	5.6 s	5,600 s
4	680 s	316,447 s

We aim to ensure that BSS provides durability no worse than that of cloud object storage. AWS S3, Azure Blob Storage, and Google Cloud Storage all presently advertise an “11 nines” durability design aim [49, 116, 293]. This means that there is a 99.999999999% likelihood that a stored object survives any one-year period. For short intervals T, the survival probability

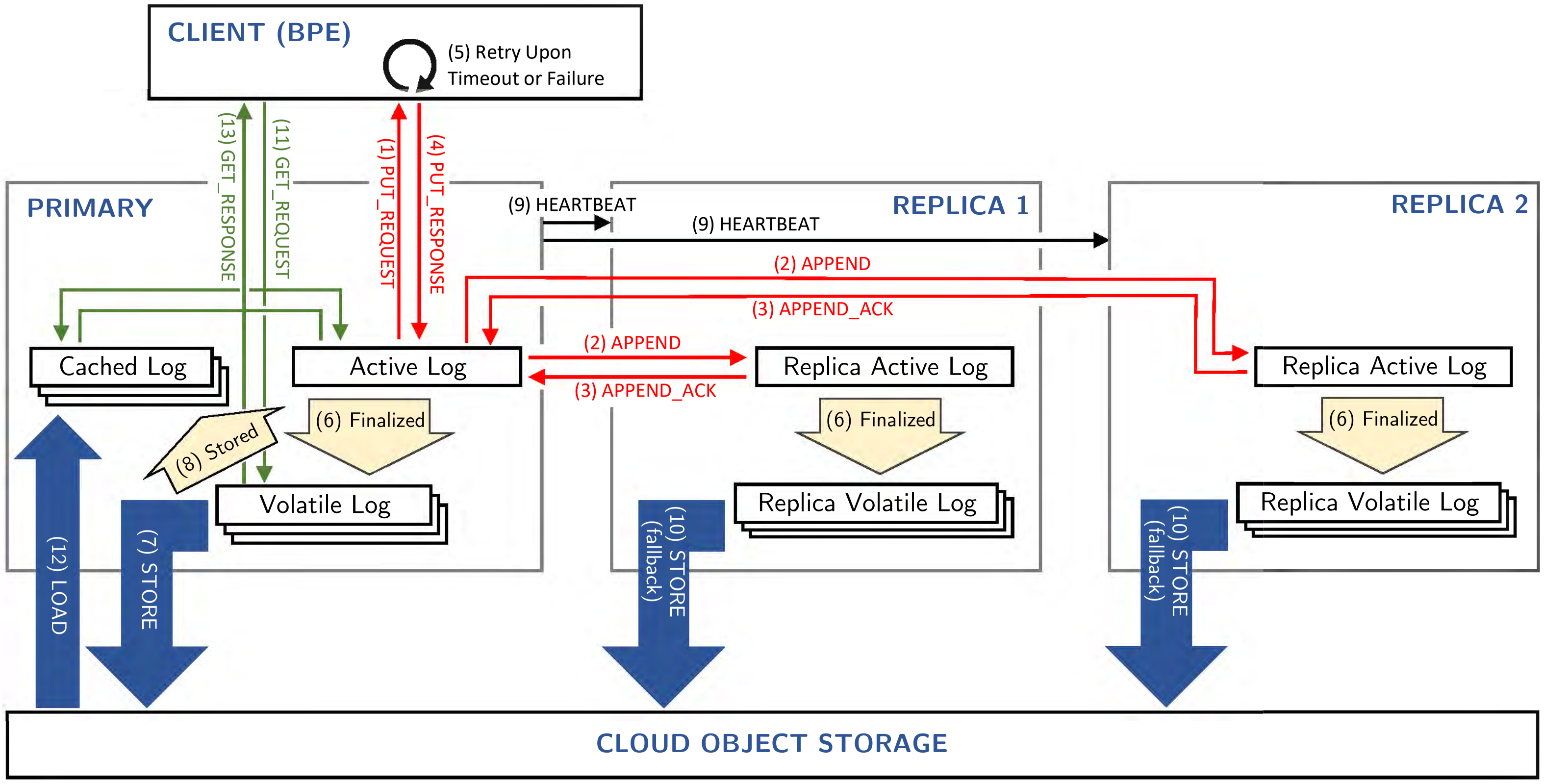


Figure 3.8: Replication in BSS. Replication of writes to an active log, (1)-(4) are shown in red. A retry loop at the client (5) re-initiates the storage request, which may be directed to the active log on another server due to load balancing. (6)-(8) indicate the failure-free storage path whereby active logs are saved to cloud object storage once they become full. If replicas fail to receive messages or a heartbeat (9) to indicate that primary is operational and successfully saving logs to storage, then they will independently finalize their copy of the log and save it to cloud object storage (10). (11)-(13) show the process of retrieving data from BSS, which may involve loading and caching data stored in cloud object storage. We discuss this design and its fault-tolerance characteristics in Section 3.4.3.4.

is approximately

$$1 - 10^{-12}\frac{T}{1 \text{ year}} \tag{3.1}$$

Studies of warehouse-scale computers suggest that servers fail 2-3× per year [60]. We thus conservatively estimate that BSS servers fail at a rate $R_f = 10^{-7}$ per second, erasing any state in process memory when they do. The same studies also report that 99% of failures are recoverable, resulting in loss of volatile state but allowing state on local disk, or instance storage, to be recovered. An implementation of BSS that writes blocks to local nonvolatile storage before acknowledging them would lose state at a rate of $R_f = 10^{-9}$ per second. Assuming uncorrelated failures, if T is the interval between when a block is acknowledged and when it is stored in the cloud, the survival probability in BSS is approximately

$$1 - (R_f T)^m \tag{3.2}$$

Taken together, Equation 3.1 and Equation 3.2 suggest that BSS offers durability greater than or equal to that of cloud object storage when

$$T_{store} \leq \left(\frac{10^{-12}}{{R_f}^m}\frac{1}{1 \text{ year}}\right)^{\frac{1}{m-1}} \tag{3.3}$$

where T_{store} is the mean time required to save a block to cloud object storage.

Table 3.3 lists T_{store} for various values of m, and for both in-memory and on-disk storage. Maintaining three in-memory copies allows up to 5.6 s, on average, to save a block to cloud object storage, which we achieve by saving a 1 MB block every 10 s.

We now summarize the architectural features and assumptions that informed our design of BSS. In FaaSFS, consistency and durability are separate concerns. BSS is focused on the latter, making possible a narrow API with write-once semantics and system-generated access references. BSS provides a buffering and caching layer on top of cloud object storage that provides low latency and high throughput for both reads and writes so long as the cache can hold the working set [126]. The working set can be cached efficiently when it contains blocks that were written together, providing a form of locality [125]. BSS always preserves the low-cost and high-durability for long-term data retention, just like the underlying storage. Our calculation shows that BSS can buffer writes for 10 s while still matching the durability of cloud object storage, so long as it replicates blocks 3× before acknowledging write success. The calculation assumes that failures are uncorrelated, which is a reasonable assumption for servers located in distinct availability zones. It also assumes that the BSS application is itself stable, i.e., that it does not fail unless the underlying system fails. While this has been true in our experience, maintaining such reliability for an actively-developed commercial product may be difficult, and in such environments disk-backed replication may be advisable.

3.4.4 Transactional Implementation

As we noted in Section 3.4.1, FaaSFS implements ECSC using transactions. This approach has allowed us to benefit from a variety of well established techniques for ensuring consistency and scalability. FaaSFS provides strict serializability, which requires that a transaction be able to see the results of any transaction that committed before it started. We enforce strict serializability using timestamps. We choose not to rely on the availability of synchronized clocks (as systems like Spanner [113] do), and instead rely upon a centralized timestamp service: TSS. At the beginning of each transaction, the client communicates with TSS to obtain a read timestamp T_R, which corresponds to the most recently committed version across the file system. Each file is represented as a collection of blocks, each of which has an associated timestamp T reflecting the commit time of its last change. By maintaining a history of versions, we can reconstruct the file system state at any given timestamp T_R.

Throughout the course of a transaction, LS maintains a read set $\mathbf{R}$ and a write set $\mathbf{W}$. Each read occurring during the course of a transaction adds a record of the form (*blocknum*, T) to $\mathbf{R}$, where $T \leq T_R$ represents the last modification time of the actual version read. Similarly, write records of the form (*blocknum*, *changed data*) are added to $\mathbf{W}$, where *changed data* is of the form (*offset*, *byte*[]) and can represent a partial update to the state of the block. Any end-of-file encounters during read operations are also recorded.

Read-only transactions do not require any backend processing at commit time. For transactions involving writes, LS sends $\mathbf{R}$ and $\mathbf{W}$ to the BPE. Figure 3.9 details the interactions that occur next. Blocks are first written to BSS, where they are replicated three times. Once replication is complete, BPE contacts a Transaction Manager (TM) in the MDS. The TM is responsible for coordinating a two-phase commit protocol across the multiple metadata partitions. Each of these partitions is itself a replicated state machine that runs the Raft protocol [299]. During the first phase of the transaction commit, MDS validates reads. It goes through the entire read set and acquires a read lock on every block in $\mathbf{R}$ and a write lock on every block in $\mathbf{W}$. It then checks to see that the read version of every block in $\mathbf{R}$ is still the current version. If any reads encountered the end of the file, it also checks that the file length has not changed since T_R. We use a centralized deadlock detector that runs periodically and aborts transactions in the prepare phase whenever it encounters a waits-for cycle.

After validating the transaction at all partitions, TM contacts TSS to obtain a commit timestamp. It then enters the second commit phase, where MDS records the new BSS reference and commit timestamp at each block in $\mathbf{W}$. It then releases locks.

We note that our commit protocol does not take full advantage of the available multiversioning. Our validation technique is well established for optimistic concurrency control [64], but it is conservative in this context. It would be interesting to try to attempt more aggressive validation approaches, perhaps by extending Sundial [455] to support strict serializability. The commit protocol is also conservative when it comes to changes in the length of a file and changes to directories.

FaaSFS implements a form of speculative lock elision [332], an optimization previously demonstrated in the processor architecture context. We implement the POSIX `fcntl` lock and unlock operations as no-ops, which is safe since the entire function is wrapped in a transactional context, ensuring that no interleaving of read and write operations of different function invocations occurs.

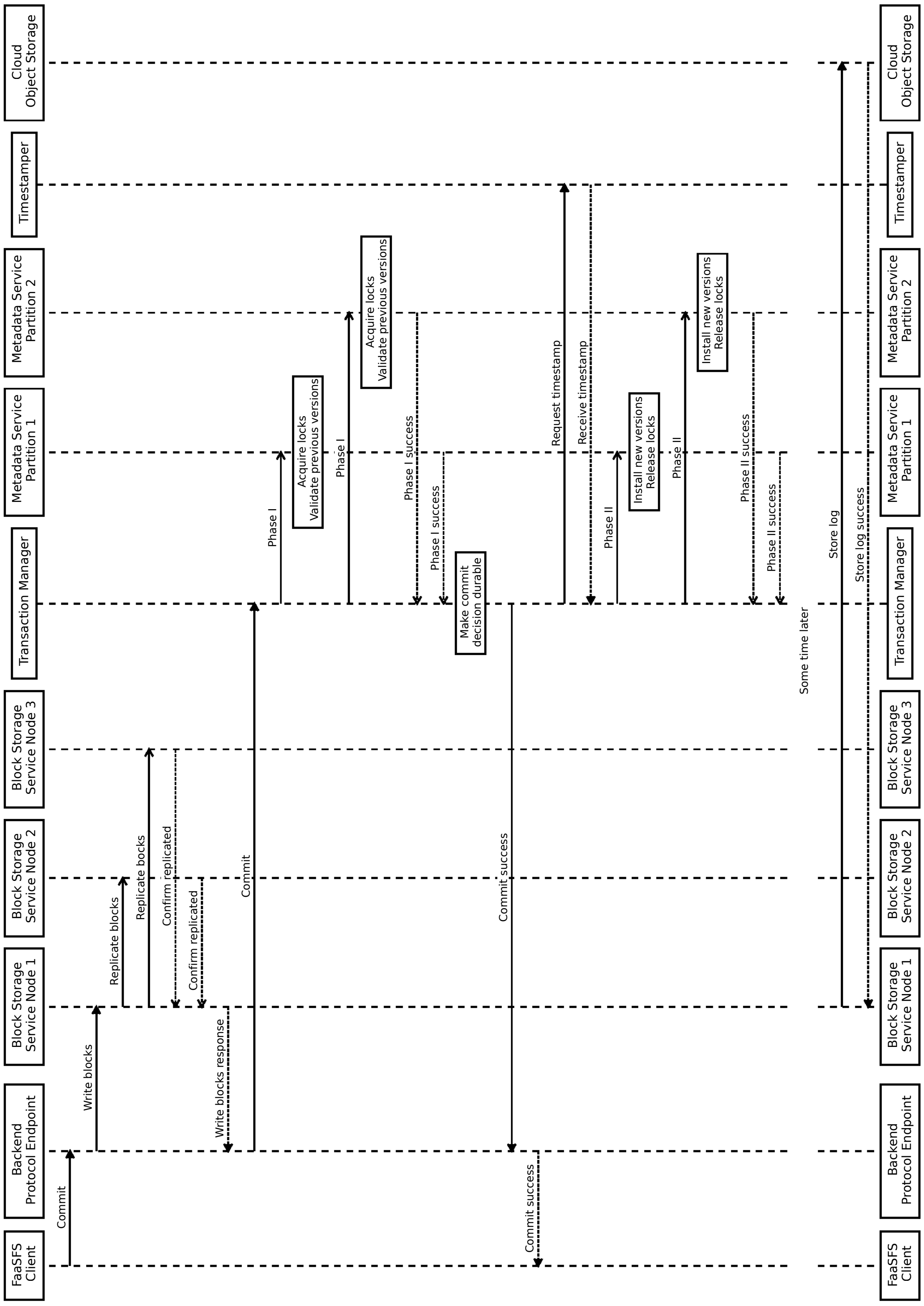


Figure 3.9: Sequence diagram for failure-free transaction commit.

Table 3.4: Listing of POSIX operations and status of implementation in FaaSFS. With the exception of `mmap`, for which we support a read snapshot only, there is straightforward path to a fully POSIX compliant implementation of all of the below operations.

Status	Operation	Description
Complete or Nearly Complete	`open`	Get new descriptor for file system object
	`close`	Close descriptor
	`write` / `pwrite`	Write / positioned write
	`read` / `pread`	Read / positioned read
	`stat`	Get size, permissions, last modified, etc.
	`getdents`	Read directory
	`sync`	Ensure updates are durable
	`seek`	Set descriptor position
	`dup` / `dup2`	Copy descriptor
	`truncate`	Set file size
	`flock`	Byte range lock and unlock
	`mkdir`	Create directory
	`rename`	Rename file system object
	`unlink`	Delete file system object
	`chmod`	Set access permissions
	`chown`	Set ownership
	`utimes`	Update modified / accessed timestamps
	`clock_gettime`	Get current time
	`chdir`	Set working directory
	`getcwd`	Get working directory
	`exec`	Load and run a program
Partially Implemented	`mmap` / `munmap`	Map file to memory
	`access`	Check access rights
	`mknod`	Create a special or ordinary file
Extensions	`begin`	Start transaction
	`commit` / `abort`	End transaction

3.5 Evaluation

Our evaluation of FaaSFS consists of several parts. We begin with a basic performance characterization that uses three benchmarks: one that measures the latency of various common operations, one that focuses on throughput, and one that evaluates performance under contention. We then describe experiments with a synthetic file system benchmark, Filebench [395], which demonstrates the potential for gains across a variety of workloads patterns. Finally, we run a real-world blog application on FaaSFS, demonstrating the ability to run traditional server-based software at serverless scale.

We chose AWS EFS [24] as the primary point of comparison for FaaSFS. EFS implements NFS version 4 [185], which is the latest version of one of the most widely deployed distributed file system protocols. EFS lacks some NFS features, notably delegation, but it appears to be one of the most scalable NFS implementations available. EFS is also the only distributed file system supported by AWS Lambda. Since Lambda is a controlled environment (see Section 3.2.3), it is not possible to access the widely available Linux kernel NFS client or install kernel modules for Lustre [359] or Ceph [433] clients. Previous research has documented the performance of EFS with AWS Lambda under a variety of workloads and client configurations [101].

When creating a new EFS file system, the user can select between two modes: a `Max I/O` mode that offers greatest scalability and a `General Purpose` mode that provides the lowest latency but is limited to 35,000 operations per second. We chose to compare FaaSFS against the EFS `Max I/O` mode since it appears to be internally partitioned, as FaaSFS is. We note that even in `Max I/O` mode, EFS also has various undocumented limits, e.g., on per-file and per-directory operations per second. We designed our experiments to avoid them.

Both FaaSFS and EFS replicate file system state synchronously for redundancy. EFS replicates state to multiple Availability Zones (AZs) within a region. According to AWS, "AZs are physically separated by a meaningful distance, many kilometers, from any other AZ, although all are within 100 km (60 miles) of each other" [3]. To match EFS, we configured FaaSFS to replicate data to three separate AZs.

3.5.1 Latency Microbenchmark

To assess the quality of our implementation, we designed a microbenchmark to measure the latency of some of the most common file system operations. The test program uses one client FaaS function to repeatedly execute a sequence of operations on a single file. It opens, reads at a random location, writes at a random location, syncs, and closes the file. The block size used is 4 KB and the file is 1 MB, which is small enough to be fully cached.

In the case of FaaSFS, these operations are wrapped in a transaction using `begin` and `commit` operations. The backend consists of three `m5dn.2xlarge` instances. Figure 3.10 plots the resulting median latency and Table 3.5 shows a more detailed breakdown that includes distribution percentiles.

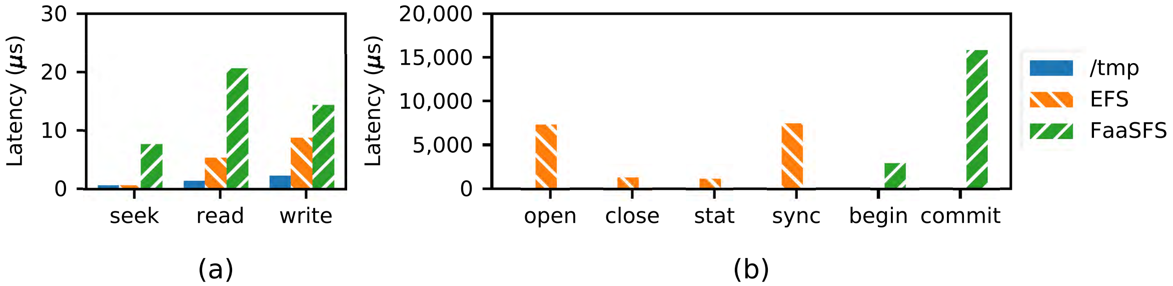


Figure 3.10: Microbenchmark results showing median latency for different operations. (a) the scale of 0-30 μs mainly highlights the performance of the FaaSFS IPC implementation (b) the scale of 0-20 ms shows the performance of network requests. Please refer to Table 3.5 for further detail on these measurements; It includes p95, p99, and maximum latencies and lists values that are too close to zero to show up on the scale of (b).

The fastest operation is `seek`, which has a median latency of about 0.55 μs for a local `/tmp` file system, 0.61 μs for an EFS target, and 7.67 μs for FaaSFS. Seek is a trivial operation, and its latency is a dominated by system call overhead. In our case, it tells us that the FaaSFS IPC implementation is about 10× slower than a system call. We have found the performance of this mechanism to be architecture-dependent, i.e., the cost varies between machine types and between EC2 instances and AWS Lambda. Furthermore, the performance on AWS Lambda changed over time—getting significantly worse—presumably due to some changes in the underlying platform (hardware, hypervisor, or operating system). AWS is known to be innovating in these areas [9], and we imagine that efforts to optimize the environment for multi-tenancy may run counter to the needs of fast IPC. The main takeaway from this measurement is that our implementation incurs a cost of 5-10 μs on each operation relative to what an in-kernel implementation would achieve. We accept this penalty as the price of integration with a production serverless FaaS service. We imagine that a future version of FaaSFS could be deployed as a kernel module to bring this overhead in line with that of EFS and other kernel-based file system implementations.

Both EFS and FaaSFS show significantly higher `read` and `write` latency than `/tmp`. We focus on the median latencies (shown in Figure 3.10 (a)). These are all much less than the network round-trip time and so reflect the overheads of accessing data cached at the client. Our implementation has an IPC overhead similar to that of `seek` and also needs to copy the payload in and out of a buffer area. Additionally, we can encounter overheads resulting from the language runtime environment. For example, Go must allocate memory and may encounter garbage collection pauses. In FaaSFS, writes are slightly faster than reads because writing merely involves appending to a log, whereas reading requires constructing the latest version of the data from that log.

Figure 3.10 (b) is scaled to show the latency of operations that communicate with the backend. Since it is an implementation of NFS, EFS provides close-to-open consistency semantics [224, 313]. This means that each `open` operation contacts the server to learn latest version of the file. The `close` operation sends a notification that informs the server that any subsequent `open` operations must now reflect modifications made to the file. The `sync` operation requests that data be stored durably. We do not know how EFS guarantees durability, but we presume that durable data must be replicated to multiple AZs and perhaps also flushed to stable storage such as SSD. The `stat` command returns file metadata, which includes not only the size but also access and modification times. Since network communication dominates the latencies in Figure 3.10 (b), we infer that EFS communicates with multiple AZs during `open` and `sync` operations but only with the local AZ for `close` and `stat` operations. EFS does not support transactions and skips the `begin` and `commit` operations. FaaSFS executes `open`, `close`, `sync`, and `stat` operations speculatively, and it incurs no network latency in doing so. Instead, it incurs latency during the `begin` and `commit` operations. The `begin` operation obtains a timestamp, which involves reading from a replicated state machine. The `commit` operation involves sending both read and write sets to the server, validating the transaction, and replicating the state. We note that the median commit latency in FaaSFS is 15.8 ms, compared to a median sync latency of 8.52 ms for EFS. We believe that our implementation may be at a further disadvantage because it treats all commits as distributed transactions.

3.5.2 Throughput Microbenchmark

We performed a measurement to assess the throughput of the FaaSFS backend. Since AWS Lambda performance can be unpredictable, in this experiment we used clients on EC2 and perform reads and writes using a test program integrated with BPE that communicates with BSS and MDS directly. Our cluster uses 12 `m5dn.2xlarge` instances and provides 3× replication.

We looked for a benchmark that would exercise the transactional capabilities of our system, and one that would highlight the performance for basic read-write operations. We thus chose to model this experiment on YCSB+T [127], which is a transactional variant of the popular YCSB benchmark [112]. For simplicity, we used the CoreWorkload generator, which is shared with the original YCSB. We considered block sizes of 1 KB and 1 MB and transactions ranging in size from 1 to 1,000 blocks per commit. Blocks are accessed according to a Zipfian distribution, and we maintained a 50-50 balance between reads and writes. We used one file per worker. As a result, there were no conflicting transactions and speculative execution always succeeded.

Figure 3.11 shows that throughput scales past 6.4 GB/sec using 1 MB blocks and 10 blocks per transaction. We achieve over 24,000 transactions per second using 1 KB blocks and 10 blocks per transaction. When grouping 1,000 1 KB blocks in each transaction, we achieve over 2,400,000 blocks per second. For comparison, AWS EFS by default imposes throughput limits of 5 GB/s or less and promises operation rates of 500,000 per second for reads and 100,000 per second for writes [23].

FS	Op	min	p50	p95	p99	max
/tmp	seek	0.52	0.55	0.57	0.62	12.43
	read	0.93	1.38	1.88	4.17	24.48
	write	1.56	2.27	2.95	4.22	168
	open	1.29	1.45	1.68	2.05	133
	close	1.00	1.20	1.39	2.39	120
	stat	1.06	1.22	1.44	2.32	18.85
	sync	63.08	72.28	121	245	1,190
EFS	seek	0.56	0.61	0.64	0.67	9.64
	read	3.88	5.33	6.17	10.63	87.19
	write	6.27	8.73	10.64	16.26	207
	open	6,220	7,250	8,240	9,940	15,900
	close	8.66	1,230	1,560	1,880	9,260
	stat	941	1,140	1,440	1,760	11,000
	sync	5,450	7,430	9,190	12,500	91,800
FaaSFS	seek	1.17	7.67	21.64	40.26	2,340
	read	10.73	20.57	55.46	168	1,550
	write	5.08	14.31	36.29	77.71	2,290
	open	6.27	17.47	90.47	221	4,030
	close	2.32	9.12	20.89	39.95	3,810
	stat	16.36	50.68	182	260	2,970
	sync	1.82	8.52	25.95	46.65	1,110
	begin	2,140	2,860	3,600	4,660	29,300
	commit	13,800	15,800	20,200	23,700	38,800

Table 3.5: Comparison of latency distributions for the microbenchmark workload of Section 3.5.1. All times in μs.

BSS thus demonstrates the flexibility to perform well on several measures of performance. We view this as a key requirement of a file system implementation and attribute it to the separation we maintain between the linearizable MDS and the immutable BSS. The metadata store, which is partitioned by file, is the bottleneck for the smaller block sizes. At the largest block sizes, we saturate the capacity of the block storage. While the TSS, which is a single global resource, would become a bottleneck in larger clusters, its capacity exceeds the transaction rates seen in these experiments by more than an order of magnitude.

3.5.3 Contention Microbenchmark

We developed this experiment to understand the performance of FaaSFS under concurrent access. We use a 1 MB file comprising blocks of 1 kB each. Clients run in a loop in which they select a block, read that block, occasionally update that block (5% of the time), then move on to another block. We protect these operations using file system range locks (`fcntl`), setting a read lock before reading, upgrading it to a write lock if updating, and then releasing it when finished with the block. The client operates on 10 blocks in quick succession (as a transaction in FaaSFS), then sleeps for 5 ms before repeating the cycle. We use a non-uniform block selection policy that places 20 of the blocks into a "hot set" that receives 20% of the accesses (12.55× the rate of access that other blocks receive). Our FaaSFS server configuration is three `m5ad.xlarge` instances, each in a separate AZ. The clients are on AWS Lambda.

Figure 3.12 shows the results from two experiments. In (a), we show strong scaling, where all clients share one file. In (b) we show weak scaling, where we create additional files as we add clients so that each client operates on a separate file. The difference between these lets us see the impact of contention. We use up to 1,024 clients, equal to the number of blocks in the file. Significant contention is evident in this workload even for many fewer clients, as evidenced by the abort rate under strong scaling (see Figure 3.12 (a)).

FaaSFS has an advantage even with only one client because lock elision (see Section 3.4.4) reduces the time spent waiting on network communication. For the strong scaling case, it scales well up to 128 clients, at which point internal contention in the FaaSFS backend limits capacity. For weak scaling, both EFS and FaaSFS scale well up to 1,024 clients, but FaaSFS demonstrates the benefit of lock elision and the lower latency it provides.

3.5.4 Filebench

To demonstrate the ability of FaaSFS to execute a variety of simulated applications, we used the Filebench [395] test suite. We ran six of the standard "personalities": file server, network file server, mail server, video server, web proxy, and web server. These classic workloads represent a variety of I/O patterns and thus provide a flavor for the diversity of applications that FaaSFS can support. In adapting Filebench to the FaaSFS setting, we wrap each iteration of the workload in a transaction. This modification seems reasonable

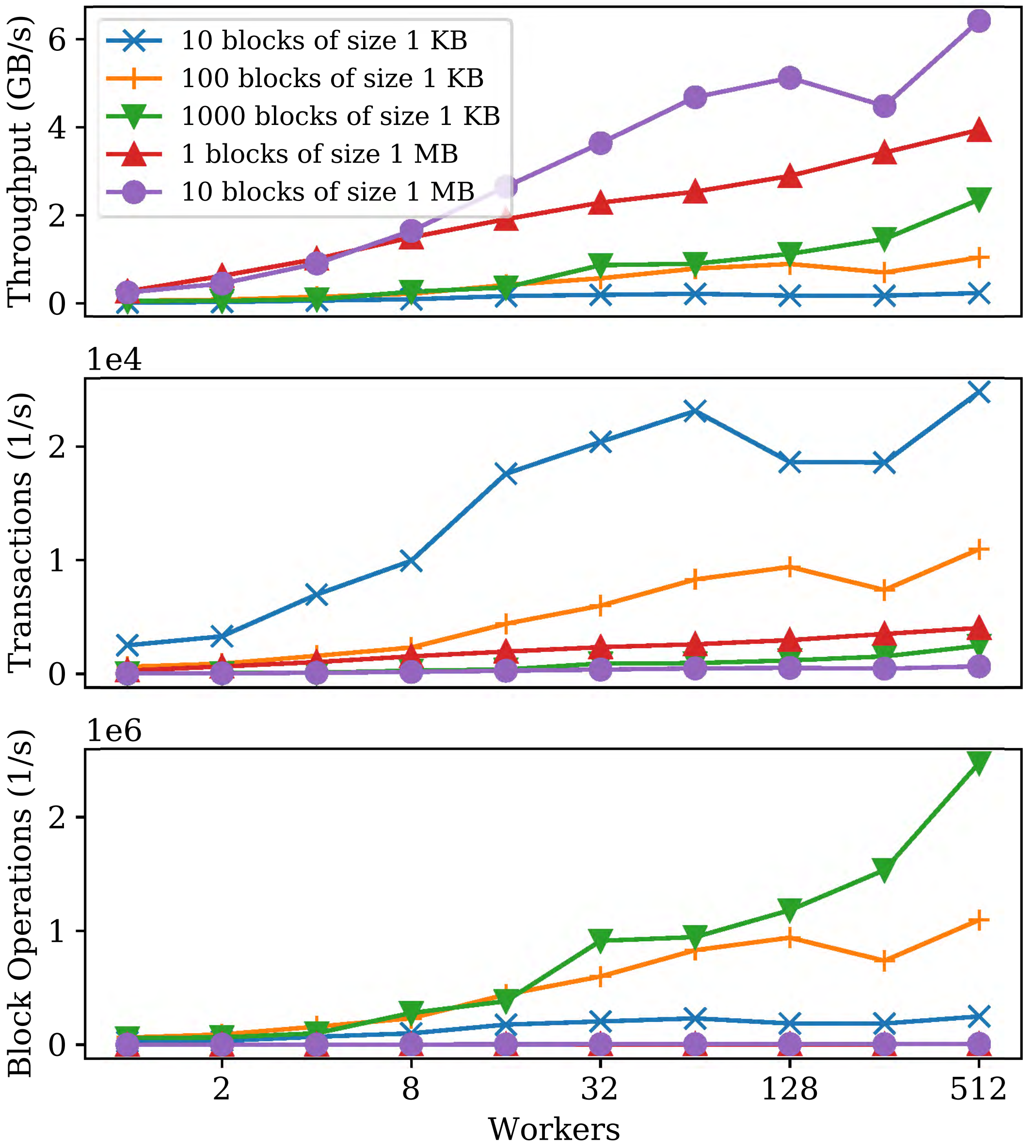
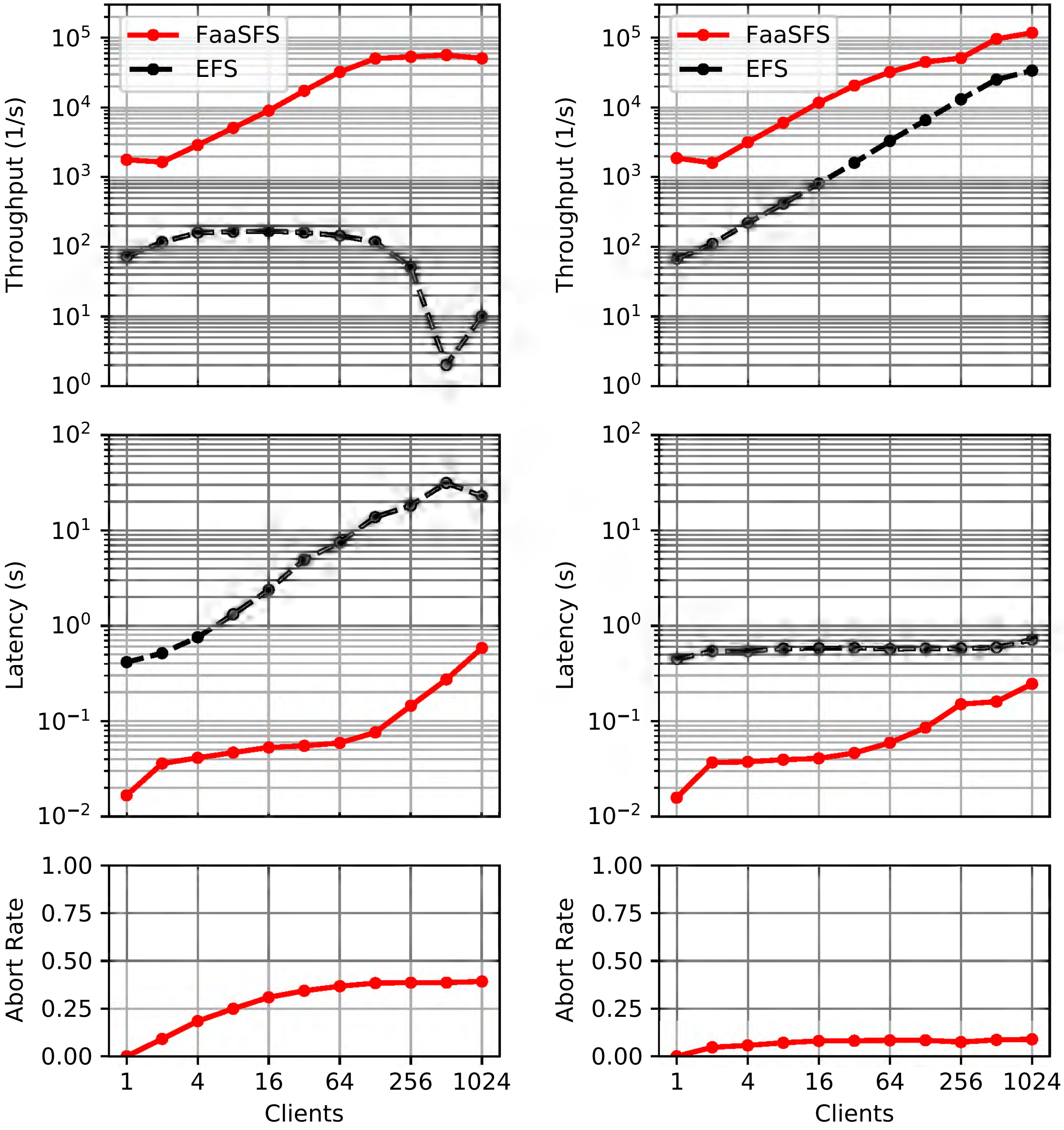


Figure 3.11: FaaSFS backend throughput scaling. We test multiple block sizes and transaction sizes in a 50-50 read-write workload based on YCSB+T [127].



(a)

(b)

Figure 3.12: Update-intensive contention benchmark: (a) strong scaling, (b) weak scaling.

Table 3.6: Average duration in μs of individual operations in the Filebench workload of Figure 3.13.

System	Workload	O_f	C_f	R	W	O_d	C_d	S	Z	B	C
FaaSFS	File Server	13.25	3.67	4,170	773	7,180	8.50	-	-	1,580	50,500
	Network File Server	11.10	2.03	6,430	18.73	5,520	10.06	-	134	1,570	28,400
	Mail Server	25.39	2.74	3,150	878	8,110	19.28	5.10	-	1,890	34,500
	Video Server	-	-	88.69	-	2,770	2.75	-	687	1,350	4.00
	Web Proxy	24.56	2.46	1,860	997	9,070	18.92	-	-	2,170	30,500
	Web Server	13.02	2.15	990	1,840	7,160	8.50	-	-	1,530	35,400
EFS	File Server	8,010	8,470	6,160	4,370	3,290	4.73	-	-	0.99	3.49
	Network File Server	7,480	4,690	25,500	3,250	3,320	3.77	-	1.00	0.99	2.77
	Mail Server	8,030	1,190	5,900	4,500	1,260	3.00	8,580	-	0.75	3.00
	Video Server	-	-	51.94	-	3,510	3.50	-	1,180	0.88	1.00
	Web Proxy	7,990	3,530	5,520	6,060	1,280	3.00	-	-	0.87	3.75
	Web Server	8,030	1,160	4,670	171	2,750	3.00	-	-	1.25	2.00

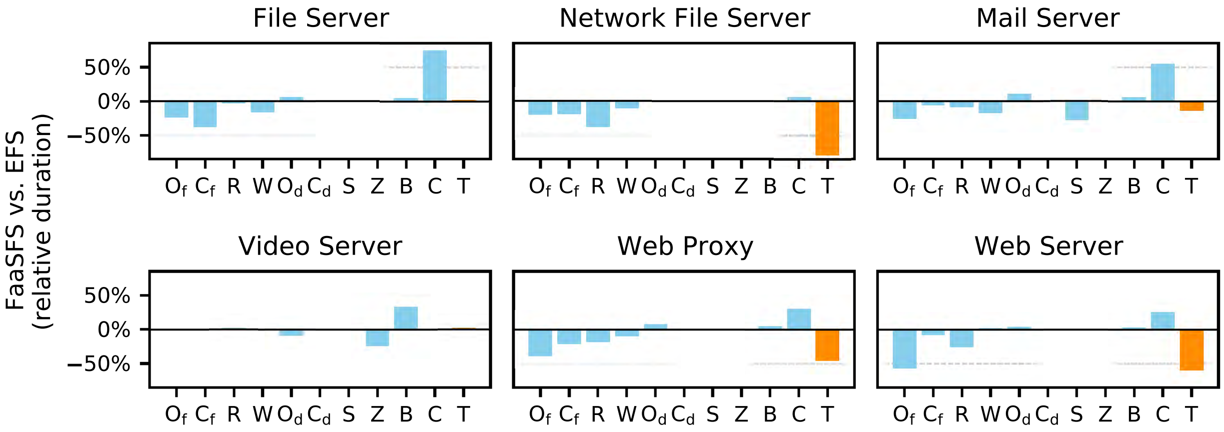


Figure 3.13: Filebench workload. Difference in average time elapsed between FaaSFS and EFS. Lower means FaaSFS performs better and higher means EFS performs better. Columns are (O_f) open file, (C_f) close file, (R) read file, (W) write file, (O_d) open directory, (C_d) close directory, (S) fsync, (Z) rate limit, (B) begin, (C) commit, and (T) total overall. Table 3.6 lists the corresponding values.

since the outer loop of the benchmark workloads appears to describe iteration over a logical unit of work.

Figure 3.13 compares FaaSFS to EFS with four concurrent Filebench clients. To understand performance difference for each type of operation, we calculate a fractional contribution to the overall benchmark performance; i.e., we calculate the difference in time spent on the operation, then divide by the total time spent:

$$\Delta_T = \frac{\textit{FaaSFS op time} - \textit{EFS op time}}{\textit{EFS all ops time}}$$

Put another way, we define Δ_T, the fractional difference in performance attributable to operations of type T, as

$$\Delta_T = \frac{\sum\limits_{\pi \in \mathcal{H}^{\text{FaaSFS}} | type(\pi)=T} duration(\pi) - \sum\limits_{\pi \in \mathcal{H}^{\text{EFS}} | type(\pi)=T} duration(\pi)}{\sum\limits_{\pi \in \mathcal{H}^{\text{EFS}}} duration(\pi)}$$

Where $\mathcal{H}^{\text{FaaSFS}}$ and $\mathcal{H}^{\text{EFS}}$ are sets of the operations from each experiment, and where each experiment has the same number of operations. For each operation π, $type(\pi)$ indicates the type of the operation and $duration(\pi)$ indicates how long it took to complete. This value indicates how changes in the performance of each operation impact overall benchmark performance. In the last column, in orange, we also show this overall performance difference, which is just the sum of the differences indicated by each of the blue bars. It can also be expressed as

$$\Delta = \frac{\sum\limits_{\pi \in \mathcal{H}^{\text{FaaSFS}}} duration(\pi) - \sum\limits_{\pi \in \mathcal{H}^{\text{EFS}}} duration(\pi)}{\sum\limits_{\pi \in \mathcal{H}^{\text{EFS}}} duration(\pi)}$$

Our implementation of FaaSFS outperforms EFS on some workloads. When it falls short, it does so by a small amount. For example, in the file server workload, FaaSFS gains significant advantages from faster file open and close operations ($\mathrm{O_f}$ and $\mathrm{C_f}$) but pays a small penalty when opening directories ($\mathrm{O_d}$) and beginning transactions (B) while incurring a significant cost during commit (C). Overall, it is 2.5% slower. In contrast, the web server workload has wins and losses on the same operations, but runs about 2.4× faster overall.

The discrepancy highlighted above is driven in part by the number of operations executed in each transaction, which is 3× greater for the web server than it is for the file server. The network file server gains a significant advantage for read operations (R), file opens and closes ($\mathrm{O_f}$ and $\mathrm{C_f}$), and writes (W), leading to an overall 4.6× gain in performance. We attribute this to more effective caching in FaaSFS.

With the web proxy, FaaSFS sees the greatest advantage when opening files ($\mathrm{O_f}$) but also has an advantage across a range of other operations. FaaSFS performs 1.8× faster.

For the mail server, we note the reduction in time spent in sync operations (S), though this does not outweigh the added cost of time spent in begin (B) and commit (C) by much. FaaSFS performs 1.1× faster.

In the video server, Filebench implements a per-client rate limit. In this read-mostly workload, the cache update time added in begin (B) gets absorbed by the rate limit (Z). FaaSFS is slower than EFS by a small amount: less than 2%. Table 3.6 shows a breakdown of the time spent in each operation for the Filebench workloads tested.

FaaSFS makes more effective use of caching than EFS, even though it provides a stronger consistency guarantee. Since it uses cache state speculatively, it can sometimes contact the server only at the beginning and end of each transaction. In contrast, EFS requires server communication on every file open ($\mathrm{O_f}$) and close ($\mathrm{C_f}$). FaaSFS also may require less server communication during reads (R) and writes (W) because it supports fine-grained cache updates and uses aggressive write buffering. The reduction in network traffic leads to significant improvements in performance for three of the six Filebench workloads: Network File Server, Web Proxy, and Web Server. On the remaining three, File Server, Mail Server, and Video Server, the increased time spent in commit (C) and begin (B) balance out these other improvements, leading to performance that is similar to that of EFS. We see no fundamental reason why FaaSFS cannot outperform EFS in all these scenarios. As described in Section 3.5.1, our implementation has various overheads that a more mature implementation could eliminate.

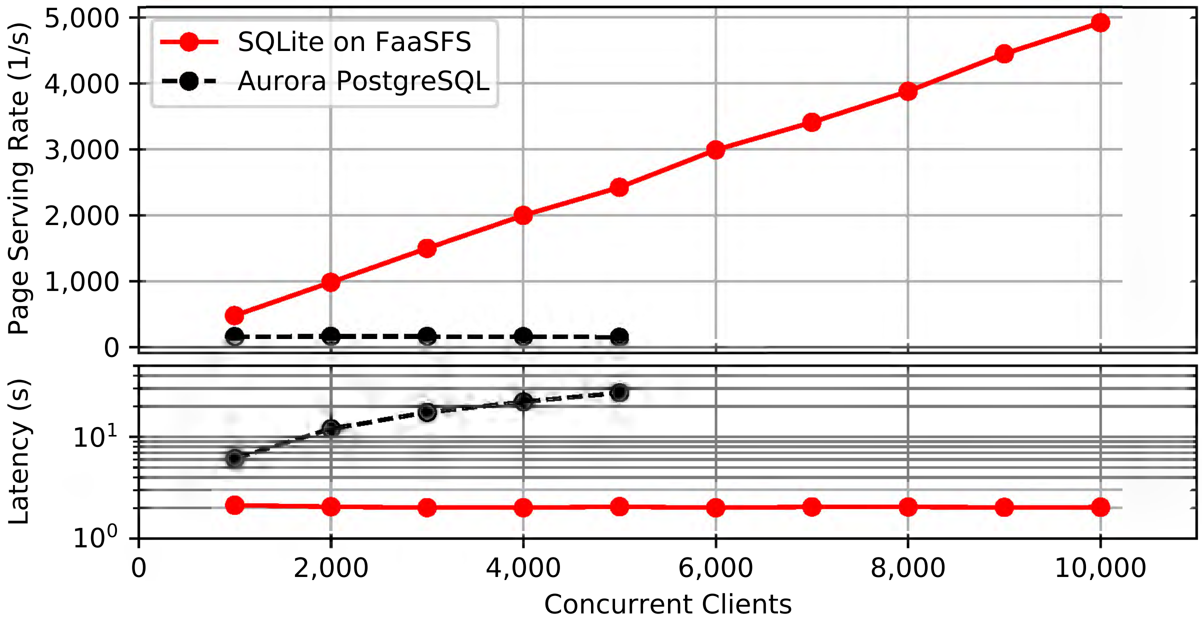


Figure 3.14: Mezzanine Blog Scaling: An open source application running on FaaSFS and SQLite scales linearly to support 10,000 concurrent clients. By comparison, when backed with a PostgreSQL database using comparable hardware, a much lower rate is achievable. This workload consists mostly of reads, with one new blog comment added every 10 s. When running EFS in place of FaaSFS this workload allows less than one page per second (not shown).

3.5.5 Full-Stack Application

To understand how FaaSFS performs in a full-stack application, we evaluated it using Mezzanine [274], a popular open source blogging platform written in Python. Mezzanine is a web application adhering to Python's WSGI standard, and we were able to deploy it to AWS Lambda using Zappa [458], an open source project that packages traditional WSGI applications as FaaS. Zappa can use AWS Elastic Load Balancer (ELB) to make cloud functions accessible over the internet. The resulting application, including libraries, is approximately 100 MB uncompressed or 30 MB compressed. We run Mezzanine in its default storage configuration, which uses a SQLite [382] database.

While SQLite does not crash when running on EFS, its ability to run the Mezzanine application is extremely limited; it achieves less than one request per second. SQLite uses locking to ensure that writers have exclusive access to the database file. This is suitable on a local file system, where communication latency is low. However, when SQLite runs on EFS, even read-only transactions incur several communication round-trips as they acquire and release locks. The problem is exacerbated because NFS clears the entire database file from cache when any part of it changes. By contrast, FaaSFS elides locks and relies instead on the underlying optimistic transaction mechanism. It also tracks changes to files at fine granularity and uses this capability when updating client caches.

We also observed that the file system locks implemented by EFS can be brittle. As with NFS, a client failing while holding a lock can block the entire system until the associated lease expires (a default time of 90 s for the Linux NFS server).

Since EFS performs so poorly for this workload, we chose to instead compare Mezzanine on SQLite and FaaSFS to a deployment backed by a database server. We used Amazon Aurora PostgreSQL, which replaces the standard file system backed storage with a distributed storage layer that replicates state to multiple AZs [419]. In this experiment, we configured FaaSFS with three nodes (`m5dn.2xlarge` instances), each in a separate AZ, and directed client traffic to a single instance node of FaaSFS while allowing the other two to serve as passive backup replicas. We believe this is the most comparable configuration, since Aurora uses only one database instance but replicates data in the storage layer.

We generated synthetic blog content to create a SQLite database file containing approximately 500 MB of data. Figure 3.14 shows a workload with varying numbers of concurrent clients that read content from the blog. A separate process updates the blog, adding one comment every 10 seconds. The impact of modifying the database is not visible in the latency numbers, though we confirmed that it has an impact on network throughput as the caches update.

With Aurora, the maximum serving rate is 167 pages/s, and the latency quickly becomes dominated by queuing delays. Using SQLite and FaaSFS, the latency remains consistent at 2.1 s, while the serving rate scales up to 4,900 pages/s as the number of concurrent clients reaches 10,000.

Key to improved scaling with FaaSFS is that the FaaS layer helps scale up the database, not only the Python application code. We are running not only thousands of Python instances but also thousands of SQLite database instances. Furthermore, since we support snapshot reads and many of the requests are read-only, speculative execution is often guaranteed to succeed.

This experiment demonstrates that FaaSFS can run unmodified real-world state-intensive applications. Whereas traditional distributed file systems fail to support any concurrency in this scenario, FaaSFS scales linearly and quickly surpasses the level of scale possible with a single database instance.

3.5.6 TPC-C Database Benchmark

We used the TPC-C [408] database benchmark to further evaluate the performance of FaaSFS on a complex workload. TPC-C simulates business activity, modeling a number of regional warehouses, each of which holds items in inventory. Transactions correspond to customer orders, deliveries, and payments as well as order status and stock level checks. Overall, it is a write-heavy workload with various points of potential contention. Partitioning is useful for TPC-C, but some transactions still involve multiple partitions; ∼90% of orders can be fulfilled from the customer's home warehouse, whereas ∼10% contain items from other warehouses as well.

We configured the benchmark with 256 warehouses. We partitioned the schema, separating each warehouse and its associated customers into a separate SQLite database. The resulting set of 256 databases amounts to 2.7 GB. Even though SQLite uses database-granularity locking, this configuration allows multiple transactions to run concurrently when they are accessing separate partitions. For those transactions that access multiple partitions, SQLite implements a two-phase commit protocol.

We configured FaaSFS using three `m5ad.xlarge` instances in separate AZs. We used `c5.large` server instances as clients, providing an environment that is somewhat more well-controlled than AWS Lambda is.

Figure 3.15 compares the performance of FaaSFS to that of EFS. FaaSFS achieves 280 txn/s with one client compared to 19 txn/s with EFS. Effective client caching is possible in both cases, but FaaSFS gains an advantage because it elides locks (see Section 3.4.4), whereas EFS does not. For 2-4 clients, the FaaSFS performance actually drops—the cache is always at the latest version with a single client, but to support multiple clients, it must ship changed blocks from one to another. For EFS, the bottleneck is waiting on lock operations (not lock contention, just network latency). This is ameliorated by added parallelism; with 4 clients, it reaches 52 txn/s. However, EFS performance saturates after that point. FaaSFS sees continued scalability up to 734 txn/s at 32 clients.

Though FaaSFS is somewhat scalable, increasing client count 32× only increases capacity 2.6×. While the capacity is still over 10× that of EFS, we conclude that this is not the best way to scale a coordination-intensive database application. Even though partitioning reduces the impact of database-granularity locking, it is still too coarse to allow good scalability in a distributed setting.

3.6 Related Work

We covered much of the background that informs the design of FaaSFS in Section 3.2. Here we add historical context and discuss various related areas.

3.6.1 Shared File Systems

Shared file systems originated with NFS [348] and have subsequently been subject to extensive research. A consistent theme in such work has been achieving both consistency and performance, with notable work including Coda [201], Sprite [287], and V [169]. Ideas from this work have found their way into contemporary protocols such as NFSv4 [185], SMB [360], and Lustre [359]. These are discussed in Section 3.2.2. Table 3.7 compares FaaSFS to a selection of other file systems.

Table 3.7: File system consistency models compared.

	Maturity	Consistency Guarantee	Comments
POSIX (local)	Deployed	Linearizable	Strong consistency and fine-grained atomicity—probably more fine-grained than programmers would prefer.
NFS/EFS [185]	Deployed	Close-to-open, linearizable file locks	Weaker consistency than POSIX. Locks allow applications to claw back linearizability.
Lustre [359]	Deployed	as POSIX	Provides for full POSIX compliance in a distributed file system.
Speculator [290]	Prototype	as NFS	NFS client predicts responses and program proceeds with speculative execution
QuickSilver [357]	Prototype	Transactional hybrid of READ_UNCOMMITTED & READ_COMMITTED	Transactions employed as distributed system integration paradigm
ScaleFS [103, 139]	Prototype	Linearizable	Non-POSIX file system designed with operations that commute better.
ECSC	Model	Linearizable-equivalent	Enforces linearizability's real-time correspondence upon external communication, otherwise maintains sequential consistency, which permits looser coupling.
FaaSFS	Prototype	Strict serializability	Provides ECSC with an expanded unit of atomicity.

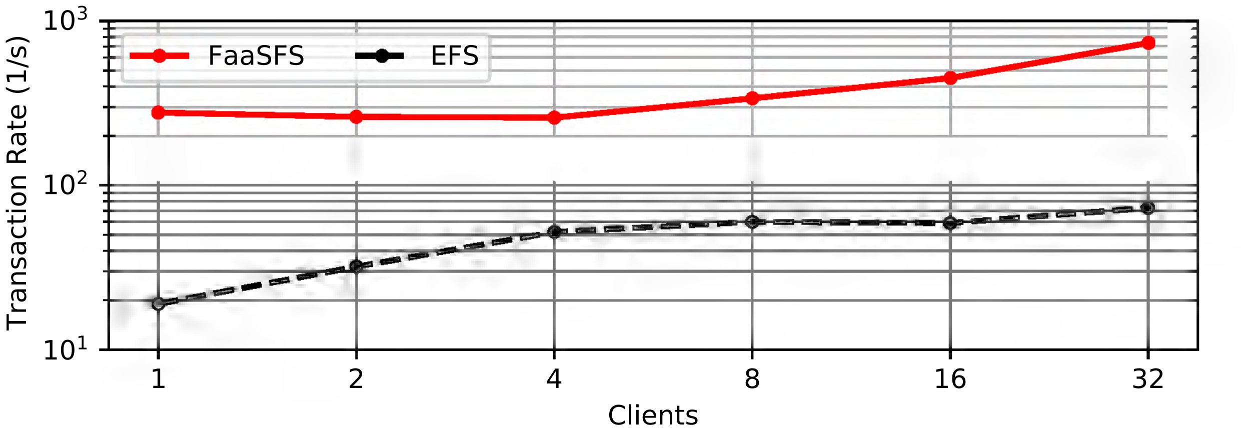


Figure 3.15: TPC-C benchmark on SQLite. See Section 3.5.6.

Another category of shared file systems includes cluster file systems like GFS [324] and OCFS2 [146]. These file systems assume that all participants have access to shared block storage, but this makes them vulnerable to misbehaving clients, and they are thus not candidates for cloud storage.

In the cloud context, shared file system research has focused on backend scalability rather than POSIX compatibility or low latency. Examples include the Google File System [164], its successor Colossus [195], and HDFS [369]. Ceph [433] is also known for enabling metadata scalability. Delta Lake [123] is a recent transactional shared file system designed specifically for analytics workloads. However, it does not offer POSIX semantics.

3.6.2 Transactional File Systems

Our implementation of ECSC is based on transactions, so it is likely to draw comparisons to transactional file systems. QuickSilver [184, 357] provides operating system support for distributed transactions. It creates a per-process *default transaction* if none was specified explicitly, which is similar to our use of FaaS functions to create transactions implicitly. QuickSilver makes weaker consistency guarantees, however. It uses various locking protocols for different types of operations, and the result is a hybrid of READ_UNCOMMITTED and READ_COMMITTED isolation levels. QuickSilver does not provide any client-side caching of remote files and does not use speculative execution or optimistic concurrency control as FaaSFS does.

The Inversion file system [298] is built on top of POSTGRES [388] and stores directory data and file block content in relational tables. It inherits the transactional isolation guarantees of the underlying database and extends the file system API to support them. Its POSIX compatibility is limited, however, because it implements only a basic subset of operations and because clients access it using a non-standard file system library. Inversion benefits from caching at the backend database server (via the buffer pool), but it does not incorporate any client-side caching.

Various other efforts have sought to combine file systems and databases. Informix patented the idea of providing a file system API atop a database backend [54], and since 2006, Microsoft has shipped Windows with TxF, a non-shared and now deprecated [410] transactional file system. Transactional file system APIs designed to make crash recovery simpler have been provided in AvdFS [420], CFS [280], TxFS [203], and TxOS [323]. However, these systems are not intended for the distributed setting.

3.6.3 Local Caching in Transaction Systems

Effective client-side caching is an important feature of FaaSFS, but it is challenging to provide. The database community has studied this topic, especially in the context of object databases. Franklin et al. [153] survey this work, comparing and categorizing various techniques. There are also middle-tier caching accelerators such as Ganymed [322], which routes read transactions to replicas, and MTCache [246], an extension to SQL Server that provides semantics equivalent to executing transactions at the database server but with the twin benefits of offloading work from the centralized system and lowering latency. MTCache relies on materialized views, which is similar to the approach used by TimesTen [240]. We have experimented with caching heuristics that correspond to some of those described in the literature, but we have not explored this area exhaustively. Future iterations of FaaSFS may benefit from some of these previous approaches.

One notable recent system is Sundial [455], which is particularly close in spirit to our implementation since it uses optimistic concurrency control and integrates this concurrency control mechanism with its caching mechanism, as we do. Sundial promises improved concurrency, but it implements serializability, rather than strict serializability, so more work is needed to determine whether its approach can be reconciled with ECSC and the consistency needs of POSIX workloads.

3.6.4 Stateful Serverless Computing

Several other systems have attempted to extend FaaS to overcome the limitations of statelessness. Cloudburst [384] provides a FaaS execution environment with integrated caching of remote storage. Cloudburst is architecturally similar to FaaSFS but uses a lattice-based eventual consistency model with a key-value store as opposed to our POSIX-compliant file system that admits only behaviors consistent with linearizability. Pocket [229] provides ephemeral storage for serverless analytics, focusing on efficient resource allocation for short time durations. Locus [326] studies the same challenge. AFT [383] provides an atomicity shim that sits between cloud functions and cloud storage, which makes it easier to achieve the idempotence that cloud functions require. Beldi [459] takes this a step further, integrating function invocations with the transaction mechanism. Kappa [463] shows how to use snapshots to provide stateful and long-lived execution within a FaaS context. We see this technique as largely complementary, since it can mitigate one of the downsides of speculative execution: the need to sometimes restart a function from the beginning.

3.7 Future Work

3.7.1 ECSC Implementation

We implemented FaaSFS using transactions, which provide stronger guarantees and greater isolation than ECSC requires. This can pose a problem for certain sorts of programs. Consider an application that polls the file system looking for some state to appear—such polling will never complete under all but the lowest levels of transactional isolation; we have implemented one of the highest ones. We emphasize that such non-termination does not produce output that a POSIX implementation would not. Our implementation violates liveness but not safety. In Section 4.5, we describe an implementation of ECSC that allows these programs to complete.

FaaSFS also imposes restrictions beyond those required by ECSC because it only supports programs that communicate at the beginning and end of the FaaS function execution (unless that communication is through the file system). This could be fixed by committing the active transaction whenever a program encounters I/O. However, recovery from speculative execution failures, or any other source of system-induced aborts, requires the ability to restart from a snapshot. This might be possible using the techniques described in Kappa [463].

3.7.2 Additional Evaluation

We compared FaaSFS to AWS EFS in Section 3.5 because that file system is both scalable and integrated with AWS Lambda. However, EFS lacks support for some NFS features, notably delegation, that might address its performance shortcomings for some of our workloads. It would also be interesting to compare FaaSFS to Lustre, which has seen significant investment in recent years and offers the standard POSIX consistency guarantees. Conducting these experiments in AWS Lambda is challenging because of the limitations of the provider-managed environment (see Section 3.2.3). Options for working around this difficulty include using server-based clients, using an open source FaaS implementation, or applying techniques for running kernel code in user space [128, 221, 328]. See Section 3.2.2.2 for a discussion of both Lustre and NFS delegation.

3.7.3 Database Techniques

FaaSFS relies on database techniques but has not exhausted the ideas that might apply to file systems. Quite a few different policies for updating local caches in distributed databases have been proposed and evaluated [153], and we imagine revisiting these. Since the success of any policy is workload-dependent, it is likely that a modern implementation would rely heavily on machine learning for policy optimization [120, 311].

We always use communication to enforce precedence relationships in FaaSFS, relying on a centralized timestamp service and the fact that a message can only be received after it was sent. One alternative that eliminates reliance on the centralized timestamp mechanism is using loosely synchronized clocks for transaction ordering [6]. In this approach, the commit mechanism guarantees serializability and external consistency. Like FaaSFS, it is an optimistic technique. Poor clock synchronization may lead to excess aborts but not consistency violations. These techniques might be applied in future implementations of FaaSFS.

The Spanner [113] database is famous for using highly accurate atomic clocks to help provide external consistency. Because timestamps are assigned carefully to all modifications, read-only transactions can run without locks, or the communication involved in acquiring them, using timestamps alone. Spanner carefully keeps track of clock uncertainty and waits when necessary to ensure external consistency. CockroachDB [106] uses a similar approach but makes do without atomic clocks [228]. These examples offer further possibilities for incorporating clocks with FaaSFS.

3.7.4 Beyond FaaS

The techniques we developed for FaaSFS are a good fit for the FaaS environment, but they might apply to other environments as well. For example, cloud applications running in container services (see Section 2.5.3) might also benefit from the scale and elasticity that FaaSFS and ECSC provide. However, FaaS provides a controlled execution environment and a built-in restart mechanism, and other non-FaaS clients would need to provide substitutes for these. One possibility is operating system transactions [291, 323], but implementing them involves cross-cutting changes which may be difficult to justify in the mainline kernel, hard to maintain in a fork, and impossible to contain to a module.

3.7.5 Production Readiness

We imagine that a production implementation of FaaSFS would be offered by a cloud provider as a serverless service. We believe that algorithms developed in our implementation can apply in a provider-managed setting, but building a secure, efficient, reliable, and metered multi-tenant service will require additional engineering. It will also require addressing concerns such as garbage collection that are not addressed in our prototype.

3.7.6 POSIX Compliance

It would be interesting to extend FaaSFS to the full range of POSIX objects. Our current implementation provides directories, regular files, and symbolic links, but it does not provide FIFOs, sockets, or device files. FIFOs and sockets are interprocess communication mechanisms that are not traditionally supported on distributed file systems. Adding them to

FaaSFS could be valuable since experience has shown that direct function-to-function communication is useful and presently not well supported [148, 189, 432]. It is also interesting because communication routed through the file system remains internal from the ECSC perspective. That means it does not induce global precedence relationships and the performance costs that can result from them.

We are also interested in running FaaSFS with a broader range of workloads. As we have explored various applications, we have sometimes discovered features of POSIX that we had failed to implement properly. These problems have usually been easy to fix once we identified them, which we generally have done by comparing system call traces.

3.8 Conclusion

FaaSFS enables some applications that maintain and share state in a file system to scale just as well as stateless FaaS applications do. This was challenging to accomplish because the POSIX standard demands a real-time correspondence, linearizability, which is subject to fundamental latency-inducing trade-offs in a distributed environment [1, 43, 77, 166]. Other distributed file systems seek to overcome such latency by providing weaker consistency guarantees, such as close-to-open consistency [224, 313], or by using techniques such as speculative execution [290] or intent locks [359] to reduce the communication overhead of ensuring linearizability. File systems designers have also experimented with non-POSIX APIs where operations are more often commutative, but these still rely on linearizability. No file system implementation that we know of breaks real-time correspondence without potentially changing application behavior. We introduce ECSC, a consistency guarantee that allows individual POSIX operations to execute out of order while ensuring that applications exhibit only those externally visible behaviors consistent with execution on an underlying POSIX file system.

Our implementation of ECSC in FaaSFS provides transactions, strengthening the model while retaining a relaxed real-time correspondence. We use optimistic concurrency control, client-side caching, and multiversion concurrency control, techniques that are well understood, though not commonly combined. Our evaluation demonstrates that FaaSFS can turn an existing full-stack database-backed application into a serverless application that is scalable to 10,000 instances. While the evaluation also makes clear that FaaSFS encounters limitations on update-intensive workloads, we believe that it can scale well on a broad class of read-mostly workloads.

While we have developed FaaSFS in the serverless context, its principles and mechanisms are not tied to FaaS or serverless computing and could be used in other contexts where strong consistency and scalability are both required. The underlying ECSC technique is also not tied to file systems and similarly could be used with key-value storage or other databases to provide performance, scalability, and consistency.

Chapter 4

Externally Consistent Sequential Consistency

4.1 Introduction

Externally consistent sequential consistency (ECSC) is an outgrowth of our efforts to improve the scalability of existing applications that rely on linearizable consistency guarantees. Its motivating use case is FaaSFS, the scalable distributed POSIX file system that we describe in Chapter 3. ECSC is guided by the observation that linearizability can impose extraneous real-time ordering constraints. These introduce a need for coordination, but can be relaxed without detriment to meaningful notions of program correctness. We refer the reader to Section 3.3 for an introduction to ECSC and additional motivating context.

In this chapter we develop a formal theory of ECSC and show how it relates to linearizability. The key result, developed in Section 4.3, tells us that ECSC always produces behavior equivalent to a behavior of the corresponding linearizable system. Showing this requires a model not just of storage, but also of computation—modeled as processes—and the environment. We choose to adopt the *I/O automaton model*, which is well established in the theory of distributed systems [257]. We also lean heavily on the theory of atomic transactions [258] to inform the structure of our model.

We review the I/O automaton formalism and set up the model in Section 4.2. Section 4.3 is devoted to showing the equivalence of ECSC and linearizability. In Section 4.4 we show how ECSC can be implemented using transactions, an approach that can take advantage of off-the-shelf databases and database techniques. In Section 4.5, we show how ECSC can be implemented using a timestamp-based cache coherence algorithm, yielding an implementation that allows some executions that the transactional implementation does not.

4.2 Preliminaries

4.2.1 I/O Automata Formalism

Input/output automaton models [257] represent distributed systems using a network of connected state machines. Such models are naturally asynchronous, meaning that their components take steps at arbitrary speeds. The *actions* of an automaton correspond to transitions between its states and fall into three categories: *input* actions, which occur in response to communication from outside, *output* actions, which generate external communication, and *internal* actions, which pertain to the automaton alone and are not visible externally. I/O automata are *input enabled*, meaning they must always accept any of their input actions, regardless of their present state.

Definition 1. An *I/O automaton* A has the following components:

- An *action signature* $S = sig(A)$ consisting of three disjoint sets of actions: input actions $in(S)$, output actions $out(S)$, and internal actions $int(S)$. The actions collectively are defined as $acts(S) = in(S) \cup out(S) \cup int(S)$, and the external actions as $ext(S) = in(S) \cup out(S)$.
- A set of states $states(A)$.
- A set of starting states $start(A) \subseteq states(A)$.
- A transition relation $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$.

An *execution* of an automaton is an alternating sequence of states and actions, $s_0\pi_1s_1\pi_2...$, where $(s_i, \pi_{i+1}, s_{i+1}) \in steps(A)$ and $s_0 \in start(A)$. A *behavior* $\beta \in behs(A)$ is the subsequence of an execution comprising its external actions, i.e., $\pi_1\pi_2...|ext(sig(A))$, where "$|$" denotes projection. An action sequence defines an order on its elements: We write $X <_\beta Y$ when X appears before Y in β. As an abbreviation, we will sometimes write $ext(A)$ in place of $ext(sig(A))$, or similarly $int(A)$ in place of $int(sig(A))$, etc.

Definition 2. *Composition* creates one automaton from a collection of automata $\{A_i\}_i$. Subject to compatibility conditions [257], a composition A has the following components:

- $sig(A) = \prod_i sig(A_i)$.
- $states(A) = \prod_i states(A_i)$.
- $start(A) = \prod_i start(A_i)$.
- $steps(A) = (s', \pi, s) \mid \forall i$, if $\pi \in acts(A_i)$ then $(s'[i], \pi, s[i]) \in steps(A_i)$, otherwise $s[i] = s'[i]$.

Here $\prod_i$ denotes the product over the collection of automata indexed by i. $s[i]$ denotes the ith component of the composite state vector. We will use composition to model collections of interacting automata.

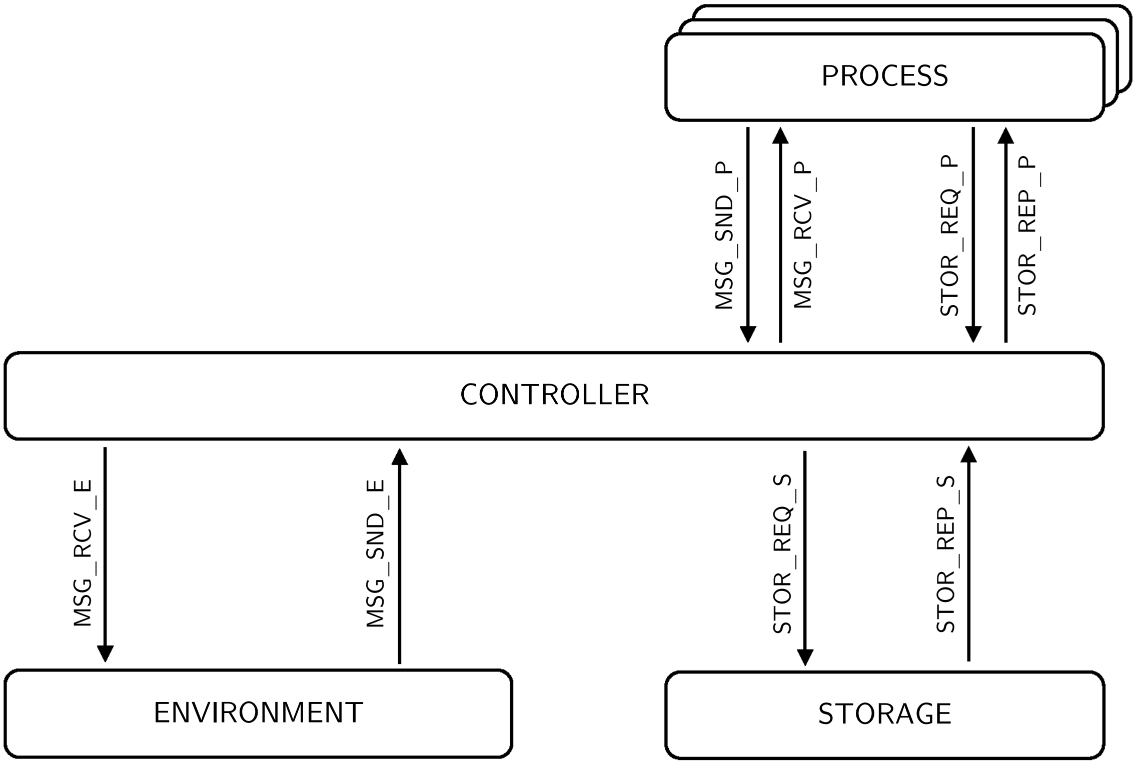


Figure 4.1: Base model for processes, storage, and the environment in the I/O automaton formalism. The controller models asynchronous transport and can interleave the actions from different operations. See the transition relation in Figure 4.2.

4.2.2 Base Model

We introduce the base model as a canonical way of representing the programs and their interactions with storage and the environment. It comprises the following automata:

- Multiple *process automata*, P_i. In the FaaS context, we use one process automaton to model each possible function invocation.
- A *storage automaton*, S, representing the storage system, e.g., a file system or key-value store.
- An *environment automaton* E representing the external environment. The environment represents everything not modeled as processes and storage; it includes all relevant aspects of the outside world.
- A *controller automaton*, C, that intermediates between the others, decoupling them and modeling asynchronous transport in a distributed system.

Figure 4.1 shows the components of the model and illustrates the external actions that connect these automata. Table 4.1 describes these actions and provides a set of abbreviations

Table 4.1: Actions of the model. Input and output denote the corresponding automaton type: (P) process, (C) controller, (E) environment, (S) storage.

Action	Abbrev.	Output	Input	Description
MSG_RCV_P (x)	M_P^{RCV}	C	P	Deliver message x
STOR_REP_P (x,v)	S_P^{REP}	C	P	Deliver response to storage request x with value v
MSG_SND_P (x)	M_P^{SND}	P	C	Initiate message x
STOR_REQ_P (x)	S_P^{REQ}	P	C	Initiate storage request x
STOR_REQ_S (x)	S_S^{REQ}	C	S	Receive storage request x
STOR_REP_S (x,v)	S_S^{REP}	S	C	Initiate reply to storage request x with value v
MSG_RCV_E (x)	M_E^{RCV}	C	E	Deliver message x to the environment
MSG_SND_E (x)	M_E^{SND}	E	C	Initiate message x from the environment

that we will often use throughout this chapter.

Definition 3. The *base model* is the composition automaton $A = E \times \prod_i P_i \times S \times C$.

The base model A encompasses our entire model: the environment, all processes, the storage, and the controller. We will write $proj_e(s)$, $proj_{p_i}(S)$, $proj_s$, and $proj_c(s)$ to denote projection to the components representing E, P_i, S, and C, respectively. We will sometimes also write $proj_{ps}(s)$ to denote projection onto the process and storage automata: $\prod_i P_i \times S$.

4.2.3 Additional Definitions

Models of messaging and storage may feature several actions that together form an *operation*; for example, a message operation involves one action to send the message and another to receive it. We denote each action by a type and one or more parameters, e.g., $M_P^{RCV}(x)$ has type M_P^{RCV} parametrized by x, whereas $S_P^{REP}(x, v)$ has type S_P^{REP} and parameters x and v. By convention, the first parameter is the *operation parameter*, and serves to identify those actions that correspond to the higher-level concept of an operation. Each operation must consist of a sequence of actions, all sharing the same operation parameter, whose types align with the operation type sequences of the model. We denote operation type sequences defined for the model as Γ, and list them in Table 4.2.

Table 4.2: The operation sequences Γ of the model are the set of the sequences below.

Operation	Action type sequence
Storage access	$(S_P^{REQ}, S_S^{REQ}, S_S^{REP}, S_P^{REP})$
Environment-to-process messaging	(M_E^{SND}, M_P^{RCV})
Process-to-environment messaging	(M_P^{SND}, M_E^{RCV})

Definition 4. An *operation* x has a corresponding sequence of actions $(\gamma_1(x, ...), ..., \gamma_n(x, ...))$ whose types, $\gamma_1, ..., \gamma_n$, conform to one of the *operation sequences* of the model Γ, i.e., $(\gamma_1, ..., \gamma_n) \in \Gamma$.

We define the *start* and *end* of each operation as the first and last actions in the operation's corresponding action sequence.

Definition 5. For operation x corresponding to $(\gamma_1(x, ...), ..., \gamma_n(x, ...))$, we define $start(x) = \gamma_1(x, ...)$ and $end(x) = \gamma_n(x, ...)$.

We define the *operation* operator to give the corresponding operation for an action.

Definition 6. For an action X, we say $x = operation(X)$ when $\exists \gamma_i$ such that $X = \gamma_i(x, ...)$ with $1 \leq i \leq n$ and $(\gamma_1(x, ...), ..., \gamma_n(x, ...)) \in \Gamma$.

We can then extend the order on the actions of a behavior $<_\beta$ to a partial order defined through the corresponding operations of those actions.

Definition 7. We define a partial order on the actions of β via the operation sequences Γ by writing $X <_\beta^{op} Y$ when $end(operation(X)) <_\beta start(operation(Y))$.

We also will sometimes use the notion of well-formedness, which describes action sequences that do not contain fragmentary or malformed operations.

Definition 8. We say that an action sequence β is *well-formed* with respect to the operation sequences Γ if whenever $\gamma_j(x, ...)$ appears in β and $\gamma = (\gamma_1, ..., \gamma_n) \in \Gamma$, $1 \leq j \leq n$, for some operation x, then

- $\forall i \in \{1, ..., n\}$, $\gamma_i(x, ...)$ appears in β exactly once
- $\gamma_1(x, ...) <_\beta ... <_\beta \gamma_n(x, ...)$

In our model, whenever an action X appears in a well-formed action sequence β, then for $x = operation(X)$ exactly one of the following must hold:

- $S_P^{REQ}(x) <_\beta S_S^{REQ}(x) <_\beta S_S^{REP}(x, v) <_\beta S_P^{REP}(x, v)$
- $M_E^{SND}(x) <_\beta M_P^{RCV}(x)$

- $M_P^{SND}(x) <_\beta M_E^{RCV}(x)$

Action sequences in which only one operation occurs at a time are sequential. Behaviors comprising such action sequences play a central role in defining correct system behavior, and we call them *sequential behaviors.*

Definition 9. An action sequence β is *sequential* with respect to the operation sequences Γ if it is well-formed with respect to Γ and if the actions of each operation appear consecutively. Whenever $\gamma_j(x, ...)$ appears in β for some operation type sequence $(\gamma_1, ..., \gamma_n) \in \Gamma$, $1 \leq j \leq n$, and operation x, then for some k, $\beta[k], ..., \beta[k+n-1] = \gamma_1(x, ...), ..., \gamma_n(x, ...)$. We say that a behavior β is a sequential behavior of A and Γ, i.e., $\beta \in \mathit{seqbehs}(A, \Gamma)$ when β is a behavior of A and is sequential with respect to the operation sequences Γ.

Finally, we define a notion of equivalence appropriate to our model—equivalence means that behaviors are indistinguishable from the perspective of any process and from the perspective of the external environment.

Definition 10. We say that behaviors $\beta_1 \sim \beta_2$ are *equivalent* if, for each process P_i, $\beta_1|P_i = \beta_2|P_i$ and also $\beta_1|E = \beta_2|E$, where E represents the environment automaton.

4.2.4 Sequential Consistency and Linearizability

We now express the established sequential consistency [241] and linearizability [192] consistency criteria using I/O automata. These are both guarantees that can be defined from the viewpoint of the process and solely in terms of the process-storage interaction. Thus, we find it useful to project onto the storage-related actions of processes, $PS = \{S_P^{REQ}, S_P^{REP}\}$.

Sequential consistency and linearizability are often formulated as properties of *histories*—these histories are similar to behaviors but lack an underlying automaton model. We choose to use a straightforward translation of such definitions [192] in our formalism, and refer to them as sequential consistency of storage (SCS) and linearizable consistency of storage (LCS) to denote that they offer guarantees specific to interactions with S.

As an aid to these definitions we introduce a mapping, $\phi_p(\pi)$, between the actions of an operation at the storage and the corresponding actions at the process: $\phi_p(S_S^{REQ}(x)) = S_P^{REQ}(x)$ and $\phi_p(S_S^{REP}(x, v)) = S_P^{REP}(x, v)$. We also use the notation $\phi_p(\beta)$ to describe mapping over the elements of an action sequence. Recalling the definition of PS, the storage-related actions of processes defined above, we now define sequential consistency in the base model.

Definition 11. We say that a behavior β satisfies sequential consistency for storage (SCS) if there exists $\beta' \in \mathit{seqbehs}(S, \Gamma|\mathit{acts}(S))$ such that $\beta|PS \sim \phi_p(\beta')$.

The notion of equivalence in Definition 10 considers the order of operations at each processor, and when restricted to PS gives us just what sequential consistency requires: "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [241].

To describe linearizability in the base model, we combine the notion of equivalence to a sequential behavior with the notion of precedence of operations $<_{\beta}^{op}$ given in Definition 5—the partial order $<_{\beta|PS}^{op}$ must also be reflected in $<_{\phi_p(\beta')}^{op}$.

Definition 12. We say that a behavior β satisfies linearizable consistency for storage (LCS) if:

- There exists $\beta' \in \mathit{seqbehs}(S, \Gamma|\mathit{acts}(S))$ such that $\beta|PS \sim \phi_p(\beta')$ (the SCS condition), and
- $<_{\beta|PS}^{op} \subseteq <_{\phi_p(\beta')}^{op}$

LCS is stricter than SCS and requires preserving the precedence relationship on operations. Note also that while the I/O automaton model is asynchronous—it has no clocks—modeling the system as a composition of state machines introduces a global sense of order. This type of model is known as a global time model [4].

4.3 ECSC Guarantee

Our design of ECSC is based on the observation that the base model (see Section 4.2.2) has executions that cannot be distinguished from linearizable (LCS) ones from the perspective of the environment or any process. While linearizability is defined exclusively in terms of storage operations, whereas ECSC takes communication into account as well. ECSC thus benefits from a richer model, which allows it to ensure equivalent program behavior while enforcing weaker constraints on the processing of storage operations.

We begin by defining the ECSC precedence $<_{\beta}^{ECSC}$. This precedence relationship is defined when actions X and Y both occur at the same automaton, and any time X is a message send at a process that occurs before Y, a message receive at a process.

Definition 13. We say that $X <_{\beta}^{ECSC} Y$ if:

- $X <_{\beta|P_i} Y$ for some P_i
- or $X <_{\beta} Y$ and X is M_P^{SND} and Y is M_P^{RCV}

By preserving this precedence relationship, we can ensure that any dependency between X and Y that might be mediated through the environment can also be preserved. ECSC combines the sequential consistency (SCS) requirement on storage operations with the requirement to maintain the global precedence of message sends relative to message receives.

Definition 14. We say that a behavior β satisfies externally consistent sequential consistency (ECSC) when:

- There exists $\beta' \in \mathit{behs}(A)$ with $\beta'|PS \in \mathit{seqbehs}(A, \Gamma|PS)$ such that $\beta|PS \sim \beta'|PS$, and
- $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$

We say a system provides ECSC when all of its behaviors satisfy Definition 14.

In Theorem 1 we show that all ECSC behaviors are equivalent to some behavior satisfying LCS, i.e., whenever β is ECSC there exists LCS β' with $\beta \sim \beta'$. Recall from Definition 10 (for "$\sim$") that this implies that β and β' are indistinguishable from the perspective of the environment or any process.

Our path to proving Theorem 1 starts with the construction shown in Algorithm 4.1. Given a behavior β of the ECSC system, it builds an equivalent behavior β' that we can show is LCS. ECSC guarantees the existence of a behavior, named β'' in Algorithm 4.1, that has sequential storage operations and that is equivalent to β at each process. ECSC, however, says nothing about the actions of the environment in β''. Thus our challenge is to construct a behavior that has both the sequential storage operations of β'' and is also equivalent to the original behavior β at the environment. Algorithm 4.1 weaves together β and β'' to achieve this result.

In Lemma 1 we show Algorithm 4.1 produces valid executions of the model A. In Lemma 2 we show that it also produces behaviors that have sequential storage operations. These results prepare us for the proof of Theorem 1, which shows that the output of Algorithm 4.1 is both LCS and equivalent to the ECSC behavior β from the perspective of all processes as well as the environment.

Lemma 1. *When β is ECSC and finite then α' as constructed by Algorithm 4.1 is an execution of A.*

Proof sketch. A is a composition (see Definition 2) of the environment, all of the processes, the storage, and the controller. To produce α', Algorithm 4.1 weaves together α, the provided behavior, and α'' the sequential behavior derived from the ECSC condition. It takes the actions of the environment from α, the actions of the processes and the storage from α'', and synthesizes the actions of the controller. The most interesting part of the proof involves showing that messages are always sent before they are received, even though Algorithm 4.1 may take the send action of a message from α and the receive action from α'', or vice-versa. The ECSC ordering guarantee, $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$, allows us to ensure that this always the case.

Proof. We will show that s_0' is a starting state of A, and that each step of α' is a step of A. As prelude, we claim that Algorithm 4.1 terminates: every iteration of the loop of line 7 increments either i or i'', ensuring termination after $n + n''$ iterations.

We now review the satisfiability of several key assignments. Satisfiability of line 1, finding an execution α whose behavior is β, follows since β is a finite behavior of A, and all behaviors are subsequences of executions. Satisfiability of line 2 is guaranteed since β is ECSC, so a behavior having the condition specified here is guaranteed to exist along with a corresponding

Algorithm 4.1 Construction of the execution $\alpha' = s'_0\pi'_1s'_1...\pi'_{n'}s'_{n'}$ for the behavior β in context of model A and operation type sequences Γ.

```
$\alpha = s_0\pi_1s_1...\pi_ns_n \leftarrow \alpha \mid \alpha \in execs(A) \wedge beh(\alpha) = \beta$    ▷ Execution from behavior
$\beta'' \leftarrow \beta'' \in behs(A) \mid \beta''|PS \in seqbehs(A, \Gamma|PS) \wedge \beta|PS \sim \beta''|PS \wedge <^{ECSC}_{\beta} \subseteq <^{ECSC}_{\beta''}$
$\alpha'' = s''_0\pi''_1s''_1...\pi''_ns''_n \leftarrow \alpha'' \mid \alpha'' \in execs(A) \wedge beh(\alpha'') = \beta''$    ▷ Derived via ECSC definition
$i, i', i'' \leftarrow 0$
$s^c_0 \leftarrow s \mid s \in start(C)$
$s'_0 \leftarrow proj_e(s_0) \times proj_{ps}(s''_0) \times s^c_0$    ▷ Composition starting state
while $i < n \vee i'' < n''$ do
    if $i'' < n'' \wedge \neg(\pi''_{i''+1} = M^{RCV}_P(X)$
      $\wedge X \notin proj_c(s'_{i'}).msg\_sent)$ then    ▷ Use $\alpha''$
        if $\pi''_{i''+1} \in acts(sig(E))$ then
            $i'' \leftarrow i'' + 1$    ▷ Skip environment step
        else
            $\pi'_{i'+1} \leftarrow \pi''_{i''+1}$    ▷ Assign action from $\alpha''$
            if $\pi''_{i''+1} \in \bigcup_i int(sig(P_i)) \cup int(sig(S))$ then
                $s^c_{i'+1} \leftarrow proj_c(s_{i'})$    ▷ Internal step
            else
                $s^c_{i'+1} \leftarrow s \mid (proj_c(s'_{i'}), \pi''_{i''+1}, s) \in steps(C)$    ▷ Compute controller step
            end if
            $s'_{i'+1} \leftarrow proj_e(s'_{i'}) \times proj_{ps}(s''_{i''+1}) \times s^c_{i'+1}$    ▷ Assign from composition
            $i'' \leftarrow i'' + 1$, $i' \leftarrow i' + 1$
        end if
    else    ▷ Use $\alpha$
        if $\pi_{i+1} \in acts(sig(E))$ then
            $\pi'_{i'+1} \leftarrow \pi_{i+1}$    ▷ Assign action from $\alpha$
            if $\pi_{i+1} \in int(sig(E))$ then
                $s^c_{i'+1} \leftarrow proj_c(s_{i'})$    ▷ Internal step
            else
                $s^c_{i'+1} \leftarrow s \mid (proj_c(s'_{i'}), \pi_{i+1}, s) \in steps(C)$    ▷ Compute controller step
            end if
            $s'_{i'+1} \leftarrow proj_e(s_{i+1}) \times proj_{ps}(s'_{i'}) \times s^c_{i'+1}$    ▷ Assign from composition
            $i \leftarrow i + 1$, $i' \leftarrow i' + 1$
        else
            $i \leftarrow i + 1$    ▷ Skip non-environment step
        end if
    end if
end while
```

MSG_SND_P (x)
Effect:
$s.msg\_sent = s'.msg\_sent \cup x$

MSG_SND_E (x)
Effect:
$s.msg\_sent = s'.msg\_sent \cup x$

MSG_RCV_E (x)
Precondition:
$x \in s'.msg\_sent$
$destination(x) = E$
Effect:
$s.msg\_sent = s'.msg\_sent - x$

MSG_RCV_P (x)
Precondition:
$x \in s'.msg\_sent$
$destination(x) = P_i$, some P_i
Effect:
$s.msg\_sent = s'.msg\_sent - x$

STOR_REQ_P (x)
Effect:
$s.stor\_req = s'.stor\_req \cup x$

STOR_REQ_S (x)
Precondition:
$x \in s'.stor\_req$
Effect:
$s.stor\_req = s'.stor\_req - x$

STOR_REP_S (x,v)
Effect:
$s.stor\_resp = s'.stor\_resp \cup (x, v)$

STOR_REP_P (x,v)
Precondition:
$x \in s'.stor\_resp$
Effect:
$s.stor\_resp = s'.stor\_resp - (x, v)$

Figure 4.2: Transition relation for the controller describing steps $(s', \pi, s) \in steps(C)$.

execution, α'', which is given on line 3. Line 6 constructs a starting state s'_0, which is a starting state of A since it is a product of starting states of the automata for the environment, processes, and storage.

Line 16 and line 27 both advance the controller. Figure 4.2 shows the transition relation for the controller and lists preconditions that need to be met for each action. Consider first line 16, $s^c_{i'+1} \leftarrow s \mid (proj_c(s'_{i'}), \pi''_{i''+1}, s) \in steps(C)$. If $\pi''_{i''+1}$ is a messaging action then it must be M_P^{SND} or M_P^{RCV} since $\pi''_{i''+1} \notin ACT\_ENV$ on line 9 and since $\pi''_{i''+1}$ is an external action because of line 13. M_P^{SND} has no preconditions. $M_P^{RCV}(x)$ requires $x \in proj_c(s'_{i'}).msg\_sent$, and this is true on account of line 8. If $\pi''_{i''+1}$ is a storage action then it can be any one of S_P^{REQ}, S_S^{REQ}, S_P^{REP}, or S_S^{REP}. Note, however, that these actions are always assigned from β'', and they are the only actions that change $stor\_req$ or $stor\_resp$ in $states(C)$. Thus $proj_c(s'_{i'}).stor\_req = proj_c(s''_{i''}).stor\_req$ and $proj_c(s'_{i'}).stor\_resp = proj_c(s''_{i''}).stor\_resp$. Furthermore, these actions are independent of msg_sent in $states(C)$. $\pi''_{i''+1} \in ext(sig(C))$ by line 13 $(proj_c(s''_{i''}), \pi''_{i''+1}, proj_c(s''_{i''+1})) \in steps(C)$, thus $(proj_c(s'_{i'}), \pi''_{i''+1}, proj_c(s''_{i''+1})) \in steps(C)$.

Now consider $s^c_{i'+1} \leftarrow s \mid (proj_c(s'_{i'}), \pi_{i+1}, s) \in steps(C)$ on line 27. Note that if π_{i+1} is a messaging action then it must be M_E^{SND} or M_E^{RCV} since $\pi_{i+1} \in \mathit{acts}(\mathit{sig}(E))$ on line 22. M_E^{SND} has no preconditions. $M_E^{RCV}(x)$ requires $x \in \mathit{proj}_c(s'_{i'}).\mathit{msg\_sent}$. Note that from line 8, either $i'' \geq n''$ or $\pi''_{i''} = M_P^{RCV}(y)$. If $i'' \geq n''$ then Algorithm 4.1 has fully traversed α'' so $\exists j' \leq i' \mid \pi'_{j'} = M_P^{SND}(x)$. If $i'' < n''$, then $\pi''_{i''} = M_P^{RCV}(y)$. Line 2 constructs β'' in accordance with the ECSC condition $<_{\beta}^{ECSC} \subseteq <_{\beta''}^{ECSC}$ and thus $M_P^{SND}(x) <_{\beta} M_P^{RCV}(y) \implies M_P^{SND}(x) <_{\beta''} M_P^{RCV}(y)$. Since $\pi''_{i''} = M_P^{RCV}(x)$, $\exists j'' < i'' \mid \pi''_{j''} = M_P^{SND}(x)$. In either case $\exists j' \leq i' \mid \pi'_{j'} = M_P^{SND}(x)$. Since β is well formed, and since any $M_E^{RCV}(x)$ is incorporated from α (at line 23), $\nexists j \leq i' \mid \pi'_j = M_E^{RCV}(x)$. Thus $x \in \mathit{proj}_c(s'_{i'}).\mathit{msg\_sent}$.

We now show that the compositions $s'_{i'+1}$ produced by Algorithm 4.1 on line 18 and line 29 are steps of A. We first consider $\mathit{proj}_e(s'_{i'})$ and show that it is assigned in accordance with the rules for composition. On line 18, $\pi'_{i'+1} \notin \mathit{steps}(E)$ and $\mathit{proj}_e(s'_{i'+1}) = \mathit{proj}_e(s'_{i'})$. On line 29, $\pi'_{i'+1} \in \mathit{steps}(E)$ and $\mathit{proj}_e(s'_{i'+1}) = \mathit{proj}_e(s''_{i''+1})$. Furthermore, $\mathit{proj}_e(s'_{i'}) = \mathit{proj}_e(s_i)$ since only line 29 updates $\mathit{proj}_e(s')$ on successive steps of α'. Since $\pi'_{i'+1} = \pi_{i+1}$, $(\mathit{proj}_e(s'_{i'}),$ $\mathit{pi}'_{i'+1}, \mathit{proj}_e(s'_{i'+1})) = (\mathit{proj}_e(s_i), \pi_{i+1}, \mathit{proj}_e(s_{i+1})) \in \mathit{steps}(E)$, as required for composition. The explanation for $\mathit{proj}_{ps}(s'_{i'})$ is similar. On line 29, $\pi'_{i'+1} \notin \mathit{steps}(PS)$ and $\mathit{proj}_{ps}(s'_{i'+1}) = \mathit{proj}_{ps}(s'_{i'})$. On line 18, $\pi'_{i'+1} \in \mathit{steps}(PS)$ and $\mathit{proj}_{ps}(s'_{i'+1}) = \mathit{proj}_{ps}(s''_{i''+1})$. Furthermore, $\mathit{proj}_{ps}(s'_{i'}) = \mathit{proj}_{ps}(s''_{i''})$ since only line 18 updates $\mathit{proj}_{ps}(s')$ on successive steps of α'. Since $\pi'_{i'+1} = \pi''_{i''+1}$, $(\mathit{proj}_{ps}(s'_{i'}), \mathit{pi}'_{i'+1}, \mathit{proj}_{ps}(s'_{i'+1})) = (\mathit{proj}_e(s''_{i''}),$
$\mathit{pi}''_{i''+1}, \mathit{proj}_{ps}(s''_{i''+1})) \in \mathit{steps}(E)$, as required for composition. The controller projection $\mathit{proj}_c(s'_{i'+1})$ also satisfies composition by its construction. When $\pi'_{i'+1}$ is an internal step of E on line 29 or PS on line 18 and so $\pi'_{i'+1} \notin \mathit{steps}(C)$ then $\mathit{proj}_c(s'_{i'+1}) = \mathit{proj}_c(s'_{i'})$. For all other steps, we have shown the satisfiability of its construction, so that when $\pi'_{i'+1} \in \mathit{steps}(C)$ then $(\mathit{proj}_c(s'), \pi'_{i'+1}, \mathit{proj}_c(s'+1)) \in \mathit{steps}(C)$. Since α' starts with a starting state of A and every step of α' is a step of A, we conclude that α' is an execution of A.

Lemma 2. *Suppose β is ECSC and finite, α' is constructed by Algorithm 4.1, and $\beta' = \mathit{beh}(\alpha)$. Then $\beta' | PS \in \mathit{seqbehs}(A, \Gamma | PS)$.*

Proof sketch. Algorithm 4.1 constructs α' using storage operations derived from β'', the sequential behavior derived from β as a consequence of the ECSC guarantee. We inspect the construction to verify that it preserves sequential ordering for storage operations.

Proof. We note that α'' as defined in line 3 of Algorithm 4.1 satisfies $\beta'' = \mathit{beh}(\alpha'')$ and so is sequential with respect to $\Gamma | PS$. We will show that $\beta' | PS = \beta'' | PS$. It is straightforward to see that $\beta' | PS$ is a subsequence of $\beta'' | PS$. For $\pi_j \in \beta' | PS$, $\pi_j \in PS = \{S_P^{REQ}, S_P^{REP}\} \implies \pi_j \notin \mathit{acts}(\mathit{sig}(E))$, thus π_j is assigned in line 12, so π_j appears in β''. It also appears in $\beta'' | PS$ since $\pi_j \in PS$. Now suppose $\pi_j \in \beta'' | PS$. Then $\pi_j \notin \mathit{acts}(\mathit{sig}(E))$ and so Algorithm 4.1 assigns π_j to α' by line 12 and we conclude that $\beta'' | PS$ is a subsequence of $\beta' | PS$. Since $\beta' | PS$ and $\beta'' | PS$ are subsequences of one another, $\beta' | PS = \beta'' | PS$. Since $\beta'' | PS \in \mathit{seqbehs}(A, \Gamma | PS)$, $\beta' | PS$ is also sequential with respect to $\Gamma | PS$.

Theorem 1. *For all ECSC β there exists LCS $\beta' \in \mathit{behs}(A)$ with $\beta \sim \beta'$.*

Proof sketch. We previously introduced Algorithm 4.1, which constructs a behavior β' that takes the environment through the same states as β does while also taking the storage and processes through the states of a corresponding sequential behavior, β'', guaranteed to exist by ECSC. β' is trivially LCS, largely as a consequence of Lemma 2.

To show that $\beta \sim \beta'$, we review some details of, Algorithm 4.1 to show that it preserves the order of actions at each process and the order of actions at the environment.

Proof. Let β be ECSC and let β' be constructed by Algorithm 4.1. By Lemma 1, $\beta' \in \mathit{behs}(A)$. To show that β' is LCS, we begin by restating the LCS condition in Definition 12, replacing β' with θ and β with β': β' is LCS if there exists $\theta \in \mathit{seqbehs}(S, \Gamma|\mathit{acts}(S))$ such that $\beta'|PS \sim \phi_p(\theta)$, and $<^{op}_{\beta'|PS} \subseteq <^{op}_{\phi_p(\theta)}$. Let $\theta = \beta'$. By Lemma 2, $\theta|PS \in \mathit{seqbehs}(A, \Gamma|PS)$. Also, trivially, $\beta'|PS \sim \theta|PS$ and $<^{op}_{\beta|PS} \subseteq <^{op}_{\theta|PS}$. So β' as constructed by Algorithm 4.1 is LCS.

Now we show that $\beta \sim \beta'$. By Definition 10, this means that we need to show that $\forall P_i, \beta|P_i = \beta'|P_i$ and that $\beta|E = \beta'|E$. Suppose X appears in $\beta|P_i$. Thus X must be one of M_P^{RCV}, M_P^{SND}, S_P^{REQ}, or S_P^{REP}. In any case, $X \notin \mathit{acts}(\mathit{sig}(E))$, thus it must be assigned to β' from β'' in line 12. Thus $\beta'|P_i = \beta''|P_i$. From line 2, $\beta|PS = \beta''|PS$, so $\forall P_i, \beta|PS|P_i = \beta''|PS|P_i$. Since $\gamma|PS|P_i = \gamma|P_i$ for any action sequence γ, and process P_i we can say $\forall P_i, \beta|P_i = \beta''|P_i$. So $\forall P_i, \beta|P_i = \beta'|P_i$. To show that $\beta|E = \beta'|E$, we observe that if X appears in $\beta'|E$, then it must be assigned on line 23. Algorithm 4.1 steps through β in sequence, and so $\beta|E = \beta'|E$.

4.4 Implementing ECSC Using Transactions

In Chapter 3 we described FaaSFS and claimed that by using transactions it implements ECSC. In this section, we describe in more detail how to implement ECSC using transactions, an approach that makes it possible to provide ECSC using existing systems and system implementation techniques.

The basic idea is to bundle together consecutive storage operations that occur during a period when a process executes in isolation, i.e., a period during which it does not send messages and does not process any messages it may receive. We then rely upon a transaction mechanism providing strict serializability to enforce real-time correspondence around these bundles of operations—if transaction A ends before transaction B begins then every operation in A precedes all operations B. Communication occurs outside of transactions, and storage access occurs inside transactions. The resulting system satisfies ECSC, enforcing a real-time constraint when communication is involved, and a logical form of precedence otherwise.

Strict serializability provides atomic transactions which not only maintain a correspondence to real-time order, but are also indivisible. As a result, transactional implementations of ECSC are stronger than the consistency guarantee requires because they prevent the interleaving of operations from separate transactions. Still, they are generally weaker than linearizability, which is equivalent to running each storage operation as an independent transaction with strict serializability [192].

In Section 4.5 we describe a different implementation of ECSC which is weaker, and akin to providing periods of sequential consistency, which is a purely logical consistency guarantee, punctuated by real-time correspondence during times of communication.

4.4.1 Background: Atomic Transactions

4.4.1.1 The Serial System

We give a brief overview of the theory of atomic transactions [258]. This theory uses a transaction nesting model in which transactions form a tree. The approach simplifies certain proofs, allowing an "everything is a transaction" modeling approach, where the environment as well as individual operations on state are modeled as transactions. The height of the transaction tree is in principle arbitrary, but a three-level tree suffices for our purposes. The top-level transaction automaton models the environment outside of the transaction system, and is known as the *root* transaction, or T_0. At the bottom, the leaves of the tree are individual storage operations, which we refer to as *access transactions*. Between these are the *mid-level* application-defined transactions—each represented by an automaton. Reference behavior is defined by interposing a *serial scheduler* between all transactions, and by specifying a *serial object* behavior for the access transactions. This serial scheduler runs transactions according to a depth-first traversal of the transaction tree, which ensures, among other things, that we may assume that only one access transaction is active at any time.

Figure 4.3 illustrates the actions of (a) a non-access transaction automaton, (b) the serial scheduler, and (c) an access transaction. Table 4.3 provides descriptions of their actions. The model contains a transaction automaton for every possible transaction that might be executed. There can be infinitely many transaction automata, one corresponding to each possible set of parameters. In applications that retry aborted transactions, each invocation attempt is represented by a separate transaction automaton. Transaction automata do not run until they receive a CREATE action. Once running, they use the REQUEST_CREATE action to launch child transactions. They learn about the outcome of child transaction executions through the REPORT_COMMIT and REPORT_ABORT actions. Upon completion of a transaction, the automaton executes the REQUEST_COMMIT action. An access transaction (i.e., an I/O operation, the lowest level of the transaction tree), supports a subset of the transaction actions because it is a leaf node in the tree. CREATE(T) indicates the beginning of a single storage operation. REQUEST_COMMIT(T,v) marks its completion and returns the value v.

The serial system is a composition of automata:

$$\mathcal{S} = SS \times \prod_{X} S(X) \times \prod_{T} A_T \tag{4.1}$$

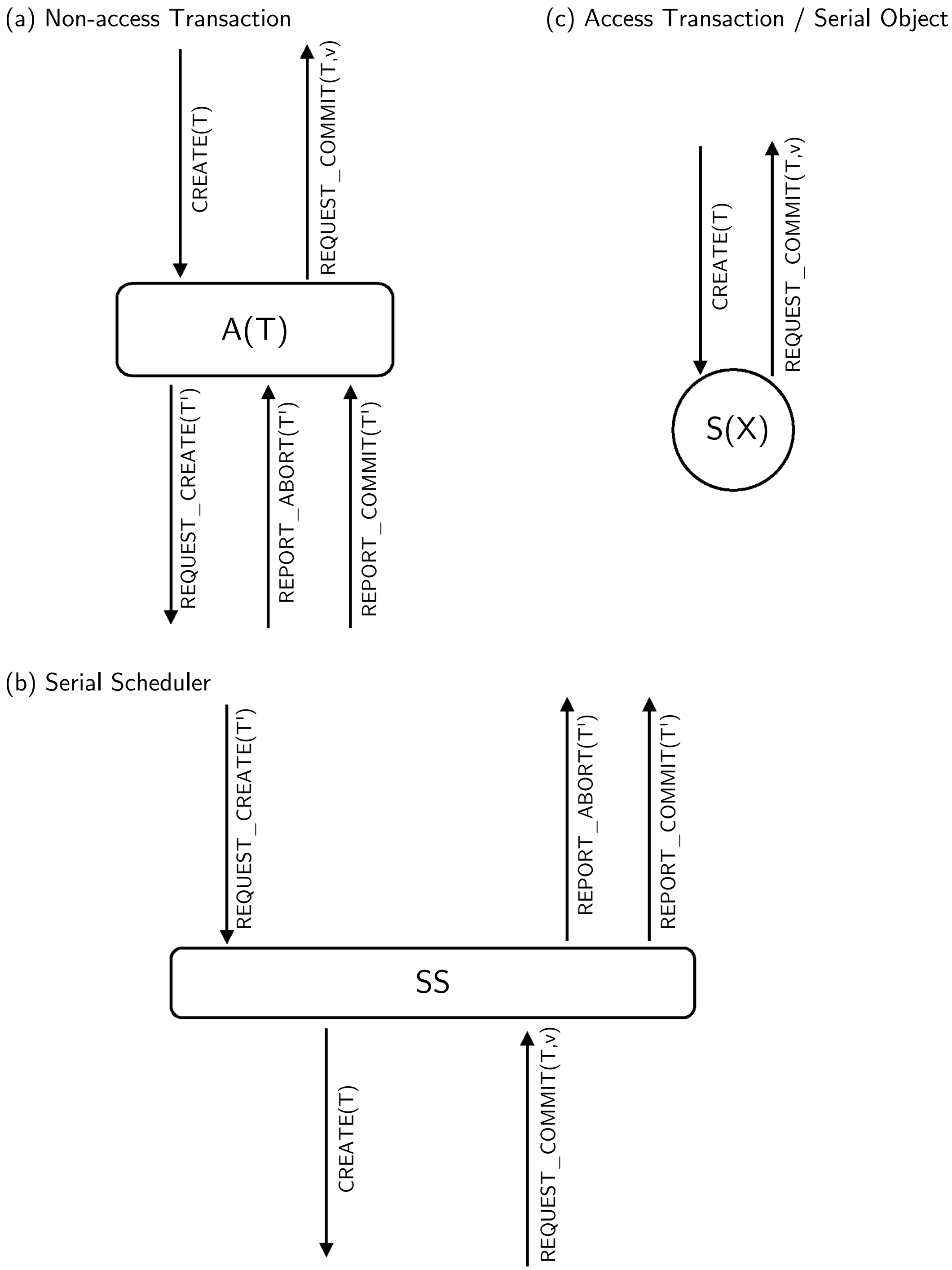


Figure 4.3: Automata of the serial system. Adopted from [258].

Here SS is the serial scheduler automaton. X ranges over all object names, and $S(X)$ is the serial object automaton corresponding to X. T ranges over all transaction names and A_T is the transaction automaton corresponding to T.

Table 4.3: Actions of transaction automata. Input and output denote the corresponding automaton type: (A_T) automaton for transaction T, (SS) serial scheduler.

Action	Output	Input	Description
CREATE(T)	SS	A_T	Begin running a transaction
REQUEST_COMMIT(T,v)	A_T	SS	Complete transaction execution
REQUEST_CREATE(T′)	A_T	SS	Initiate T′ a child of transaction T
REPORT_COMMIT(T′,v′)	SS	A_T	Report commit of T′ to parent transaction T
REPORT_ABORT(T′)	SS	A_T	Report abort of T′ to parent transaction T

4.4.1.2 Atomic Behaviors

The theory of atomic transactions defines atomic behavior in terms of behaviors of the serial system (described in Section 4.4.1.1):

Definition 15. A behavior β is *atomic* for transaction T if there exists $\gamma \in behs(\mathcal{S})$ such that $\beta|T = \gamma|T$.

When a system produces atomic behaviors, it is also called atomic:

Definition 16. A system $\mathcal{T}$ is atomic for transaction name T if all of its finite behaviors are atomic for T.

In this work we will sometimes say that a system $\mathcal{T}$ is atomic for T and $\mathcal{S}$ to be explicit about the serial system referenced. Note that Definition 16 is quite general and does not require $\mathcal{T}$ to contain T or any other transaction automaton in $\mathcal{S}$. In the work that follows we maintain this generality where possible, but will also sometimes restrict ourselves to implementations that follow the *simple system* model.

Simple systems are composed of the same transaction automata $\{A_T\}$ as the serial system $\mathcal{S}$, but they replace the serial scheduler and the serial object automata with other automata. They are a foundation for transaction system implementations ranging from locking, to timestamp order, to optimistic execution.

Proofs about simple systems are facilitated by a central theorem, the *atomicity theorem* [258], which is based around establishing a relationship between the apparent order of operations at each object X, as observed by transaction T, and a *sibling order*, a partial order among transactions. If β is a behavior of the simple system and if there is a sibling order for which $\beta|X$ is a serial object behavior for all objects X, then the atomicity theorem tells us that β is atomic for T.

Also evident from the proof of the atomicity theorem is a corollary, described by Lynch et al. [258] on page 196, at the conclusion of their proof of the atomicity theorem:

Lemma 3. *If T_0 is atomic with $\gamma|T_0 = \beta|T_0$ then $\gamma|T = \beta|T$ for all T that commit to the top level.*

Lemma 3 is an important condition because it tells us that if behavior β is shown to be atomic for T_0 using the atomicity theorem, then γ can be used to show atomicity for all transactions that commit to top level. This is different from proving atomicity for each transaction separately, which does not guarantee that all transactions can agree on the same serial system behavior. Since we will rely on this guarantee, we give it a name.

Definition 17. We call a behavior β *all-atomic* if there exists $\gamma \in \mathit{behs}(\mathcal{S})$ such that $\gamma|T = \beta|T$ for all T that commit to top-level.

4.4.1.3 Comparison to the Classical Theory

The theory of atomic transactions [258] is more general than classical serializability theory [304]. Most proofs in the classical theory focus on the properties of serialization graphs, using conflict serializability [304, 448] as the correctness criterion, even though it can be stronger than necessary. By contrast, in the theory of atomic transactions atomicity for T_0 provides view serializability [304, 448], which is a weaker condition.

The beauty of the classical theory is its simplicity, yet the theory of atomic transactions is appealing in other ways, e.g., it uses one definition of correctness for single-version, multi-version, and replicated transaction systems, whereas the classical theory must define correctness separately for each.

The theory of atomic transactions also allows for transaction nesting, storage operations other than reads and writes, and a model of transaction aborts.[1] It also implicitly includes a real-time precedence relationship, i.e., strict serializability because the automaton model is a global time model [4, 258].

Classical serializability theory also lacks a model of computation, which is indispensable in our application. By applying the theory of atomic transaction we have at our disposal a state machine formalism that allows us to model and reason about processes, the environment, and the interactions between them.

[1] Aborts occur only as a result of scheduler actions; there are no application-initiated aborts.

4.4.2 Modeling the ECSC Transactional Implementation

4.4.2.1 Augmented Automic Systems

The theory of atomic transactions uses a specific automaton signature, shown in Figure 4.3 (a). For our purposes, it is too restrictive because it allows only a limited set of external actions. To reason about ECSC, we will want to model systems that have behaviors including messaging, but we will not want to express messaging actions as part of the transactional model.

We will now show that we can turn arbitrary internal actions $int(A_T)$ of *one* transaction automaton into external actions $ext(A_{T'})$ of a related automation while maintaining a property similar to atomicity. This property is not the same as atomicity, however, because it applies to a different system. We call it *augmented atomicity.*

Definition 18. If A_T is the automaton for transaction T then $A_{T'}$ is an *augmented transaction automaton* for T if:

- $states(A_{T'}) = states(A_T)$
- $start(A_{T'}) = start(A_T)$
- $steps(A_{T'}) = steps(A_T)$
- $acts(A_{T'}) = acts(A_T)$
- $in(A_{T'}) = in(A_T)$
- $out(A_{T'}) \supset out(A_T)$
- $int(A_{T'}) \subset int(A_T)$

Recalling that $ext(A) = out(A) \cup int(A)$, we note that it also follows that $ext(A_{T'}) \supset ext(A_T)$.

Definition 19. We define the *augmented serial system* as

$$\mathcal{S}' = SS \times \prod_{X} S(X) \times \prod_{U \neq T} A_U \times A_{T'}$$

That is, $\mathcal{S}'$ is the same as $\mathcal{S}$ defined in Equation 4.1, but it substitutes $A_{T'}$ for A_T.

Definition 20. We say that a system $\mathcal{T}'$ is *augmented atomic* for T', if $\mathcal{S}'$ is an augmented serial system derived from $\mathcal{S}$ by replacing A_T with the augmented automaton $A_{T'}$ and if for any finite behavior $\beta \in behs(\mathcal{T}')$ there exists $\gamma \in behs(\mathcal{S}')$ such that $\beta|T' = \gamma|T'$.

When a transaction system $\mathcal{T}$ contains the transaction automaton that will be augmented, we can show that replacing A_T with $A_{T'}$ in $\mathcal{T}$ produces $\mathcal{T}'$, which is augmented atomic for T'.

Lemma 4. *Suppose a transaction system $\mathcal{T}$ is atomic for transaction T with the serial system $\mathcal{S}$, and suppose that $\mathcal{T}$ is a composition containing the automaton A_T. Suppose also that T' is an augmented transaction for T and that $\mathcal{S}'$ is the corresponding augmented serial system. Then $\mathcal{T}'$ is augmented atomic for T' and the augmented serial system $\mathcal{S}'$.*

Proof sketch. We proceed to construct action sequence γ and show that $\gamma \in \mathit{behs}(\mathcal{S}')$ and that $\beta|T' = \gamma|T'$. We do this by using Algorithm 4.2, which builds up γ by combining β, which is a given behavior of $\mathcal{T}'$, and γ', which is a behavior of the serial system $\mathcal{S}$ that is guaranteed to exist because $\mathcal{S}$ is atomic for T. We can think of this as replacing the external actions of A_T with the external actions of $A_{T'}$. A_T and $A_{T'}$ share a common interface with the rest of the serial system, be that $\mathcal{S}$ or $\mathcal{S}'$, i.e., $\mathit{ext}(A_T) \subseteq \mathit{ext}(A_{T'})$, which ensures that the construction is possible. Since we have required that $\mathcal{T}$ contains A_T, we can construct $\mathcal{T}'$ by replacing this transaction automaton with $A_{T'}$. It is then straightforward to see that the required conditions on γ are met.

Algorithm 4.2 Construction of $\gamma = \pi_1^\gamma \pi_2^\gamma \ldots \pi_k^\gamma$ for Lemma 4.

```
1:  $\beta \leftarrow \beta \mid \beta \in \mathit{behs}(\mathcal{T}'), \beta$ finite
2:  $\gamma' \leftarrow \gamma' \mid \gamma' \in \mathit{behs}(\mathcal{S}) \wedge \beta|T = \gamma'|T$
3:  $\pi_1^x \pi_2^x \ldots \pi_n^x \leftarrow \beta|\mathit{ext}(A_{T'})$
4:  $\pi_1^y \pi_2^y \cdot \pi_m^y \leftarrow \gamma'$
5:  $i \leftarrow j \leftarrow k \leftarrow 1$
6:  while $i \leq n \vee j \leq m$ do
7:      if $i \leq n \wedge \pi_i^x \notin \mathit{ext}(A_T)$ then
8:          $\pi_k^\gamma \leftarrow \pi_i^x$
9:          $i \leftarrow i + 1$
10:     else if $j \leq m \wedge \pi_j^y \notin \mathit{ext}(A_T)$ then
11:         $\pi_k^\gamma \leftarrow \pi_j^y$
12:         $j \leftarrow j + 1$
13:     else                                        ▷ Here $\pi_i^x = \pi_j^y$
14:         $\pi_k^\gamma \leftarrow \pi_i^x$
15:         $i \leftarrow i + 1$
16:         $j \leftarrow j + 1$
17:     end if
18:     $k \leftarrow k + 1$
19: end while
```

Proof. We begin by reviewing Algorithm 4.2, which constructs an action sequence γ. We first show that the algorithm completes, then show that γ is a behavior of $\mathcal{S}'$ and that $\beta|T' = \gamma|T'$. On line 1, we select any finite behavior $\beta \in \mathit{behs}\mathcal{T}'$. We know that $\beta|\mathit{ext}(\mathcal{T})$ is a behavior of $\mathcal{T}$ since $\mathit{acts}(\mathcal{T}') = \mathit{acts}(\mathcal{T})$—the augmented transaction system has the same actions as the unaugmented one, only some of the internal actions of the unaugmented system are external in the augmented system owing to the difference between external actions of the augmented transaction automaton $A_{T'}$ and the transaction automaton A_T. Since we have assumed that $\mathcal{T}$ is atomic for T, we know that there exists a behavior of the serial system $\gamma' \in \mathit{behs}(\mathcal{S})$ with $\beta|\mathit{ext}(\mathcal{T})|T = \gamma'|T$. Since $\mathit{ext}(A_T) \subseteq \mathit{ext}(\mathcal{T})$ we conclude that $\beta|T = \gamma'|T$, as required on line 2. We construct γ by weaving together two action sequences: $\beta|\mathit{ext}(A_{T'})$ and γ'. We note that since $\beta|T = \gamma'|T$, these sequences share the subsequence $\beta|T$. The idea is to construct γ such that we maintain this shared subsequence, i.e., $\gamma|T = \beta|T$ as well. The `while` loop beginning on line 6 constructs γ one element at a time. The `if` statements inside distinguish between three cases: on line 7 we draw actions from $\beta|ext(A_{T'})$ that are not in $ext(A_T)$, on line 10 we draw actions from γ' that are not in $ext(A_{T'})$, whereas on line 13 we encounter the overlapping elements of $\beta|ext(A_T)$ and γ'. We claim that the algorithm terminates since at each step in increments either i or j or both.

We construct $\mathcal{T}'$ by replacing A_T with $A_{T'}$ in $\mathcal{T}$. Thus $\beta|A_{T'} = \beta|ext(A_{T'})$, which is extracted on line 3 and is the part of β that we use in constructing γ, is a behavior of $A_{T'}$. We can see that γ is a behavior of $\mathcal{S}'$ by noting that $\mathcal{S}'$ is a composition identical to $\mathcal{S}$ with the automaton A_T replaced with $A_{T'}$, and by observing that γ is constructed from the serial behavior γ' by replacing a behavior of A_T with a behavior of $A_{T'}$. We conclude that γ is a behavior of $\mathcal{S}'$ and that $\mathcal{T}'$ is augmented atomic for T'.

4.4.3 Model for Implementing ECSC with Transactions

Figure 4.4 shows the automaton model that we use to implement ECSC using transactions. The root transaction T_0 models both the environment of the base model (see Figure 4.1) as well as parts of the processes. Whenever a process needs to operate on storage it opens a mid-level transaction to do so, running a "process fragment" within it. This mid-level transaction must terminate before the process again communicates with its environment. The messaging actions MSG_SND_P, MSG_RCV_P, MSG_SND_E, and MSG_RCV_E are all internal to T_0.

Figure 4.5 shows the transition relation for T_0, Figure 4.6 shows the transition relation for the mid-level transactions, and Figure 4.7 gives that for the access transactions, or storage. Note that we have provided redundant labels for some actions, which may be interpreted in both the context of the serial transaction system and in the context of the base model (see Section 4.2.2).

As we will show in Theorem 2, we can achieve ECSC using transaction processing implementations that are all-atomic for T_0. As we mentioned in Section 4.4.1.2, these include standard locking and timestamp-order algorithms, proofs for which are derived using the simple database and the Atomicity Theorem. The implementation of FaaSFS that we described in Chapter 3 is modeled well by optimistic hybrid atomicity (see [258], Section 10.2), which also is all-atomic for T_0.

In the proof of Theorem 2, we rely on augmented atomicity to "peek inside" of T_0, exposing behaviors that incorporate both messaging operations and storage operations. Figure 4.5 defines an augmented transaction system where T_0 is replaced by T_0', which substitutes external messaging actions for internal ones. It is these behaviors augmented behaviors that satisfy ECSC.

We also must show that our serial system model produces behaviors of the base model. This is the content of Lemma 5, which we address next.

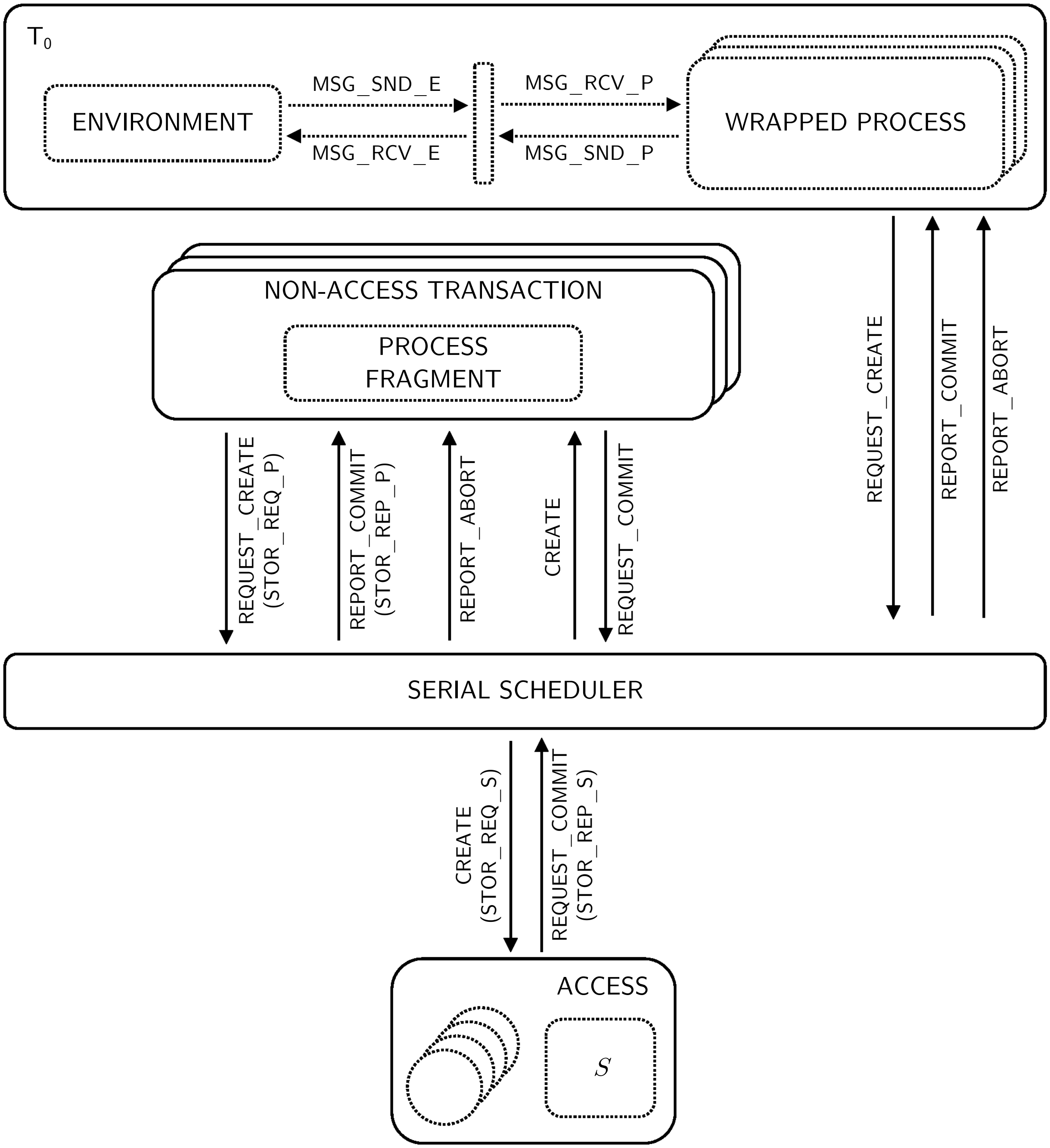


Figure 4.4: Transactional model.

Lemma 5. *Suppose β is a behavior of the augmented serial system $\mathcal{S}'$ defined in Figure 4.5, Figure 4.6, and Figure 4.7. Then $\gamma = \beta|A$ is a behavior of the base model A.*

Proof sketch. The base model A is a composition of the environment, all of the processes, and the storage. The transition relation for a composition is given in Definition 2, which says that at each step of the composition corresponds to a transition at precisely one of the component state machines. We verify this for $\mathcal{T}'$ by inspection of the transition relations in Figure 4.5, Figure 4.6, and Figure 4.7. In these figures, we have highlighted transitions for E in blue, P_j in red, and S in purple.

Proof. We first review the transitions corresponding to E. These occur in MSG_SND_E (line 4 and line 6) and MSG_RCV_E (line 27), both in Figure 4.5. In both cases the state corresponding to E is maintained within T_0 as $s.underlying[env]$.

The transitions for the processes P_j are distributed between T_0 and the various mid-level transactions. Figure 4.5 shows the transitions for T_0. There are messaging transitions MSG_RCV_P (line 13) and MSG_SND_P (line 18 and line 20). Internal process state transitions are possible (line 47 and line 49). Storage actions are possible only from within mid-level transactions, and state passes to them from T_0 in REQUEST_CREATE (line 33), then returns through REPORT_COMMIT (line 40). In a mid-level transaction, described in Figure 4.6, state first enters CREATE (line 53). Storage operations occur in REQUEST_CREATE, which maps to STOR_REQ_P, (line 67 and line 69) and REPORT_COMMIT, which maps to STOR_REP_P, (line 73). Internal transitions of process state are also possible in the mid-level transaction (line 84 and line 86). All of these transitions are an allowed transition of the state machine of one process.

The transitions for storage S are governed by the internal transitions of access transaction state. Storage requests are described by STOR_REQ_S (line 94), and responses STOR_REP_S (line 101). The storage may also make internal transitions at any time (line 111 and line 113).

The base model also has a controller, that acts as an intermediary for messaging and storage access. The potential reordering for messaging is captured in Figure 4.5 since in-flight messages are represented as sets. Storage operation ordering is restricted by the transaction scheduler, and these transitions form a subset of those possible with the base model controller.

We have identified transitions of $\mathcal{T}'$ corresponding to transitions of the base model A. Each transition corresponds to a step of one component automaton, which makes the composition valid (see Definition 2). Thus if β is a behavior of the transactional storage $\mathcal{T}'$, then $\gamma = \beta|A$ is a behavior of A, the base model.

Theorem 2. *Suppose $\mathcal{T}$ is a composition containing T_0, described in Figure 4.5, and suppose that $\mathcal{T}$ is all-atomic for T_0 and the serial system $\mathcal{S}$, as defined in Figure 4.5, Figure 4.6, and Figure 4.7. Let $\mathcal{T}'$ be the augmented system obtained by replacing T_0 with T_0', as defined in Figure 4.5. Then all behaviors $\beta \in behs(\mathcal{T}')$ satisfy ECSC, i.e., there exists $\beta' \in behs(A)$ with $\beta'|PS \in seqbehs(A, \Gamma|PS)$ such that $\beta|PS \sim \beta'|PS$ and $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$.*

Proof sketch. Lemma 4 tells us that $\mathcal{T}'$ is augmented atomic for T_0'. Thus for any $\beta \in behs(\mathcal{T}')$ there exists $\gamma \in behs(\mathcal{S}')$ such that $\beta|T_0' = \gamma|T_0'$. Furthermore, $\gamma|U = \beta|U$ for all transactions $U \neq T_0'$ since $\mathcal{T}$ is all-atomic for T_0, and because for $U \neq T_0'$ the construction of γ in Algorithm 4.2 maintains $\gamma|U = \gamma'|U$, where $\gamma' \in behs(\mathcal{S})$ is a serial behavior. This means that γ is a behavior of $\mathcal{S}'$, a sequential system. This, together with Lemma 5, tells us that $\beta' = \gamma|A$ is a behavior of A. To see that β' satisfies ECSC we proceed to show that $\beta|P_j = \beta'|P_j$ for all processes P_j. Atomicity guarantees that equivalence in a piecewise manner, but since the execution of each process is split across multiple transactions we need to reason about order across them. We now know that $\beta|PS \sim \beta'|PS$. To see that $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$ we rely on the fact that $\beta|P_j = \beta'|P_j$ and that messaging actions M_P^{SND} and M_P^{RCV} are both external actions of T_0', and so their order is preserved under augmented atomicity.

Proof. $\mathcal{T}$ contains the automaton T_0 and is atomic for T_0 since it is all-atomic for T_0, thus Lemma 4 implies that $\mathcal{T}'$, obtained by replacing A_T with $A_{T'}$ in $\mathcal{T}$ is augmented atomic for T_0'. Thus for any $\beta \in behs(\mathcal{T}')$ there exists $\gamma \in behs(\mathcal{S}')$ such that $\beta|T_0' = \gamma|T_0'$. Let $\beta' = \gamma|A$. By Lemma 5, β' is a behavior of A. Also, $\beta' \in seqbehs(A, \Gamma|PS)$ since $\gamma \in behs(\mathcal{S}')$, which employs the serial scheduler.

We now want to show that $\beta|P_j = \beta'|P_j$ for all processes P_j. Since $\mathcal{T}'$ is augmented atomic for T_0' we have $\beta|T_0' = \gamma|T_0'$, and so also $\beta|T_0'|P_j = \beta'|T_0'|P_j$. Since $\mathcal{T}$ is all-atomic for T_0 we have $\beta|U = \gamma'|U$ for any transaction $U \neq T_0$ that commits to top-level, with $\gamma' \in behs(\mathcal{S})$. Algorithm 4.2, used to construct γ in Lemma 4, maintains $\gamma'|U = \gamma|U$. Thus we have $\beta|U = \gamma|U$, and by extension $\beta|P_j|U = \beta'|P_j|U$. We now have equivalences for the various pieces of $\beta|P_j$, those that are part of the same mid-level transaction, as well as those that are part of T_0'. We also note that since every action is part of a transaction, $\beta|P_j$ and $\beta'|P_j$ have the same actions, and it remains to show that these actions occur in the same order, i.e., we want to show that $<_{\beta'|P_j}=<_{\beta|P_j}$. Suppose X and Y are actions in $\beta|P_j$ and $X <_\beta Y$. If X and Y both appear in mid-level transaction U it is clear that $X <_{\beta'} Y$. If X appears in mid-level transaction U and Y appears in mid-level transaction V then we must have $REPORT\_COMMIT(U, v) <_\beta CREATE(V)$ since T_0 and T_0' defined in Figure 4.5 run only one mid-level transaction at a time on behalf of each process. Now since $\beta|T_0' = \gamma|T_0'$, by augmented atomicity for T_0', $X <_\gamma REPORT\_COMMIT(U, v) <_\gamma CREATE(V) <_\gamma Y$, and so $X <_{\beta'} Y$. If X appears in T_0' and Y appears in a mid-level transaction V, i.e., $X <_\beta CREATE(V)$, we similarly have $X <_\gamma CREATE(V) <_\gamma Y$ and thus $X <_{\beta'} Y$. On the other hand if X appears in a mid-level transaction U and Y appears in T_0', i.e., $REPORT\_COMMIT(U, v) <_\beta Y$, then we have $X <_\gamma REPORT\_COMMIT(U, v) <_\gamma Y$ and again $X <_{\beta'} Y$. If both X and Y appear in T_0' augmented atomicity assures that their order is preserved in γ and so also β'. These cases demonstrate that $<_{\beta|P_j}\subseteq<_{\beta'|P_j}$. Next now consider cases for $X \not<_\beta Y$. If both X and Y appear in the same transaction, be it a mid-level transaction or T_0', it is clear that $X \not<_{\beta'} Y$. If X appears in mid-level transaction U and Y appears in mid-level transaction V then we have $CREATE(U) \not<_\beta REPORT\_COMMIT(V, p)$ and also $X \not<_\gamma CREATE(U) \not<_\gamma REPORT\_COMMIT(V, p) \not< \gamma Y$ and thus $X \not<_{\beta'} Y$. If X appears in T_0' and Y appears in mid-level transaction V then $X \not<_\beta REPORT_COMMIT(V, v)$ and so $X \not<_\gamma REPORT_COMMIT(V, v) \not<_\gamma Y$, thus $X \not<_{\beta'} Y$. If X appears in mid-level transaction U and Y appears T_0' then $CREATE(U) \not<_\beta Y$ and so $X \not<_\gamma CREATE(U) \not<_\gamma Y$, thus $X \not<_{\beta'} Y$. These cases demonstrate that $\not<_{\beta|P_j} \subseteq \not<_{\beta'|P_j}$, or equivalently, $<_{\beta|P_j} \supseteq <_{\beta'|P_j}$. Since we previously showed $<_{\beta|P_j} \subseteq <_{\beta'|P_j}$ we conclude that $<_{\beta'|P_j} = <_{\beta|P_j}$. Thus $\beta|P_j$ and $\beta'|P_j$ have the same actions and these actions occur in the same order, and we conclude that $\beta|P_j = \beta'|P_j$.

It follows immediately that $\beta|PS \sim \beta'|PS$. To see that $<_\beta^{ECSC} \subseteq <_{\beta'}^{ECSC}$ we need to show that $<_{\beta|P_i} \subseteq <_{\beta'|P_i}$ and that if $X <_\beta Y$ where X is M_P^{SND} and Y is M_P^{RCV}, then $X <_{\beta'} Y$. The former is clear since $<_{\beta'|P_j} = <_{\beta|P_j}$. The latter is also evident since $\beta|T_0' = \gamma|T_0'$ and M_P^{SND} and M_P^{RCV} are both external actions of T_0'.

```
**MSG_SND_E (x)** - (INTERNAL for $T_0$, EXTERNAL for $T_0'$)
    Precondition:
        $dst(x) = p$
        $(s'.underlying[env], M_E^{SND}(x), s'') \in steps(U[env])$
    Effect:
        $s.underlying[env] = s''$
        $s.msg\_in[p] = s'.msg\_in[p] \cup x$
**MSG_RCV_P (x)** - (INTERNAL for $T_0$, EXTERNAL for $T_0'$)
    Precondition:
        $s'.txn\_active[p] = false$
        $x \in s'.msg\_in[p]$
    Effect:
        $s.underlying[p] = s'' \mid (s'.underlying[p], M_P^{RCV}(x), s'') \in steps(U[p])$
        $s.msg\_in[p] = s'.msg\_in[p] - x$
**MSG_SND_P (x)** - (INTERNAL for $T_0$, EXTERNAL for $T_0'$)
    Precondition:
        $s'.txn\_active[p] = false$
        $(s'.underlying[p], M_P^{SND}(x), s'') \in steps(U[p])$
    Effect:
        $s.underlying[p] = s''$
        $s.msg\_in[env] = s'.msg\_in[env] \cup x$
**MSG_RCV_E (x)** - (INTERNAL for $T_0$, EXTERNAL for $T_0'$)
    Precondition:
        $x \in s'.msg\_in[env]$
    Effect:
        $s.msg\_in[env] = s'.msg\_in[env] - x$
        $s.underlying[env] = s'' \mid (s'.underlying[env], M_E^{RCV}(x), s'') \in steps(U[env])$
**REQUEST_CREATE((p,y,attempt_ct))**
    Precondition:
        $s'.txn\_active[p] = false$
        $\exists s'' \mid (s'.underlying[p], S_P^{REQ}(y), s'') \in steps(U[p])$
        $s'.attempts[y] + 1 = attempt\_ct$
        $y = s.uncerlying[p]'$
    Effect:
        $s.txn\_active[p] = true$
        $s.attempts[y] = attempt\_ct$
```

Figure 4.5: Transition relation for T_0 and T_0'.

```
REPORT_COMMIT((p,y,attempt_ct),v)
    Effect:
        $s.txn\_active[p] = false$
        $s.underlying[p] = \begin{cases} v.state & \text{if } v.success \\ s'.underlying[p] & \text{otherwise} \end{cases}$
REPORT_ABORT((p,y,attempt_ct))
    Effect:
        $s.txn\_active[p] = false$
INTERNAL
    Precondition:
        $s'.txn\_active[p] = false$
        $\exists s'', \pi \mid (s'.underlying[p], \pi, s'') \in steps(U[p]) \text{ and } \pi \in int(sig(U[p]))$
    Effect:
        $s.underlying[p] = s''$
```

Figure 4.5: Continued: Transition Relation for T_0 and T_0'.

```
CREATE((p,y,attempt_ct))
    Effect:
        $s.is\_created = true$
        $s.underlying = y$
REQUEST_COMMIT((p,y,attempt_ct),v)
    Precondition:
        $s'.created = true$
        $s'.finalized = false$
        $s'.access\_outstanding = \emptyset$
        $v.state = s'.underlying$
        $v.success = \neg s'.aborted$
    Effect:
        $s.finalized = true$
REQUEST_CREATE((p,x,attempt_ct)) - maps to STOR_REQ_P (x)
    Precondition:
        $s'.created = true$
        $s'.aborted = false$
        $(s'.underlying, S_P^{REQ}(x), s'') \in steps(U[p])$
    Effect:
        $s.underlying = s''$
        $s.access\_outstanding = s'.access\_outstanding \cup x$
REPORT_COMMIT((p,x,attempt_ct),v) - maps to STOR_REP_P (x)
    Effect:
        $s.underlying = s'' \mid (s'.underlying, S_P^{REP}(x), s'') \in steps(U[p])$
        $s.access\_outstanding = s'.access\_outstanding - x$
REPORT_ABORT((p,x,attempt_ct))
    Effect:
        $s.aborted = true$
        $s.access\_outstanding = s'.access\_outstanding - x$
```

Figure 4.6: Transition relation for process-fragment transactions.

```
INTERNAL ((p,y,attempt_ct))
    Precondition:
        $s'.created = true$
        $s'.aborted = false$
        $s'.finalized = false$
        $\exists s'', \pi \mid (s'.underlying, \pi, s'') \in steps(U[p]) \wedge \pi \in int(sig(U[p]))$
    Effect:
        $s.underlying = s''$
```

Figure 4.6: Continued: Transition relation for process-fragment transactions.

```
CREATE((p,x,attempt_ct)) - maps to STOR_REQ_S (x)
    Effect:
        $s.created = s.created \cup (p, x, attempt\_ct)$
INTERNAL
    Precondition:
        $(p, x, attempt\_ct) \in s'.created$
    Effect:
        $s.underlying = s'' \mid (s'.underlying, S_S^{REQ}(x), s'') \in steps(S)$
        $s.executing = s.executing \cup (p, x, attempt\_ct)$
        $s.created = s'.created - (p, x, attempt\_ct)$
INTERNAL
    Precondition:
        $(p, x, attempt\_ct) \in s'.executing$
    Effect:
        $s.underlying = s'' \mid (s'.underlying, S_S^{REP}(x, v), s'') \in steps(S)$
        $s.executing = s'.executing - (p, x, attempt\_ct)$
        $s.results = s'.results \cup ((p, x, attempt\_ct), v)$
REQUEST_COMMIT((p,x,attempt_ct),v) - maps to STOR_REP_S (x)
    Precondition:
        $((p, x, attempt\_ct), v) \in s'.results$
    Effect:
        $s'.results = s'.results - ((p, x, attempt\_ct), v)$
INTERNAL
    Precondition:
        $\exists s'' \mid (s'.underlying, \pi, s'') \in steps(S) \wedge \pi \in int(sig(S))$
    Effect:
        $s.underlying = s''$
```

Figure 4.7: Transition relation for access transactions.

4.5 Implementing ECSC Using Local Caches with Hybrid Clock Leases

In this section we describe an implementation of ECSC that allows weaker consistency than the transactional implementation described in Section 4.4. The approach, which we call *local caches with hybrid clock leases* (LCHCL), is derived from Tardis [452], a scalable cache coherence protocol for many-core microprocessors. In Tardis each core has a local cache, and the protocol achieves sequential consistency across cores using logical clocks and a timestamp reservation mechanism. In contrast to directory-based cache coherence mechanisms [191], which require $O(N)$ bits of state per cache line in a N-core system, Tardis requires only $O(\log(N))$ bits of state. This scalability makes Tardis a suitable starting point for providing consistency in a serverless environment.

In augmenting the protocol to provide ECSC, we replace logical time in Tardis with a hybrid physical-logical time representation. Like Tardis, it ticks forward using logical updates driven by access to shared state. However, it also adjusts to align with physical time when external communication is involved. We assume availability of a TrueTime API [113], which augments an underlying physical clock mechanism with bounded uncertainty.

4.5.1 Background

4.5.1.1 Cache Coherence with Tardis

We now provide a brief overview of the original Tardis protocol from which LCHCL is derived. To align with the language used in other parts of this work, we will refer to the participating compute units as processors, whereas the original work refers to these as cores.

Each processor maintains a monotonically increasing logical time, *pts*. This logical time is not a cycle counter, but rather it is updated during operations on shared state such that it maintains correspondence to the global memory order. Tardis also annotates cache lines with logical timestamps and uses these to implement its cache coherence algorithm.

Each processor has a private cache, and there is also a last-level cache that processors share. In the shared cache, each cache line has two associated timestamps: a read timestamp, *rts*, and a write timestamp, *wts*. *wts* represents the logical time at which the cache line was last written, whereas *rts* provides a lower bound on the timestamp of any future writes to the same address. *rts* thus provides a form of reservation, or logical lease.

The private cache also maintains *wts* and *rts* for each cache line. A line in the private cache is guaranteed to be valid for the interval $wts \leq pts \leq rts$. When a processor needs to access a cache line but $pts > rts$, the processor makes a request to the shared cache to refresh the cache line state. The response provides a new *rts*, extending the reservation into the future; if the state has changed it also includes updated cache line data and a new *wts*.

Several additional rules govern the progression of logical time. Whenever a processor reads from local cache, it updates its clock by setting $pts \leftarrow \max(pts, wts)$. Whenever it writes, it sets $pts \leftarrow \max(pts, rts + 1)$ and then $wts \leftarrow pts$. An interesting case arises when the shared cache responds to a read request. It increases *rts* by an interval *lease*, which is a configurable parameter that governs a trade-off between local cache freshness and the frequency of communication with the shared cache.

Logical time in Tardis is similar to a Lamport clock [241]: both increase monotonically and propagate along causal links. However, Lamport timestamps attach to messages, whereas Tardis timestamps attach to shared memory accesses (the model does not incorporate messages). There are also different update rules. Whereas a Lamport clock ticks forward every time a message is sent, logical time in Tardis moves forward at every write operation, as well as at some read operations. If we view storage as a means of communication, then Tardis timestamps might be viewed as an adaptation of Lamport clocks to this context.

For operations on different processors, a cache line sharing mechanism completes the algorithm. Each line in the last-level cache may be in *shared mode* or *exclusive mode*, which are defined as usual [191]. In shared mode, requests to extend read reservations are always granted immediately. If the cache line is in exclusive mode, a request for shared access triggers a write-back request, which must complete before reads can proceed. Before writing a cache line, a processor must obtain exclusive access. If the line is in shared mode this can again be granted immediately, but if another processor holds exclusive access the protocol must first complete a flush request.

Tardis is more scalable than other cache coherence mechanisms because its shared cache tracks only processors that have exclusive access to a cache line, and not those that have shared access. Directory protocols [191], by contrast, must track shared access on a per-processor basis. In Tardis, there is also no need to coordinate with readers to revoke shared access. Instead, the algorithm grants exclusive mode access immediately and jumps ahead in logical time to a point beyond *rts*. Revoking exclusive access requires communication, but only with the processor holding exclusive access.

4.5.1.2 TrueTime

Traditional clock sources represent time as a single number. For example, the UNIX timestamp reports nanoseconds elapsed since January 1, 1970. While some operating systems or time sources provide additional APIs that report the clock uncertainty, TrueTime makes it explicit on every read of the clock [113]. Rather than returning a single timestamp, $TT.now()$ returns a pair of timestamps that define an uncertainty interval: $[earliest, latest]$. TrueTime also provides the $TT.before(t)$ and $TT.after(t)$ calls, convenience wrappers which compare a provided interval to the current time and return *true* only after accounting for the uncertainty in both the system clock and the provided argument. As described in Spanner [113], servers synchronize their clocks at regular intervals. Uncertainty typically follows a sawtooth pattern: it drops sharply immediately following synchronization, then increases gradually due to uncertainty in clock drift.

When introducing TrueTime [113] and in the years that followed [78], Google highlighted the use of atomic clocks and other sophisticated timekeeping techniques to ensure global clock synchronization. The technology was perceived as innovative, but also somewhat esoteric—

CockroachDB, though derived from Spanner, does not use physical clocks in its consistency model [228].

Well accepted advice would encourage us to rely on physical clocks only for performance, and not for correctness [6, 124]. However, this may be changing in the cloud setting, as accurate physical time has become a common feature of cloud data centers [109, 208, 406]. Also, recent research advances such as Huygens [162] and Sundial [252] use network effects and statistical techniques to align clocks within a data center to a precision of 100 ns. 5G wireless technology brings a similar level of clock accuracy to the edge network infrastructure [249]. The algorithm that follows assumes that it may soon be practical for a broader range of systems to employ consistency algorithms that, like Spanner, rely upon physical clocks with bounded errors for correctness.

4.5.2 LCHCL Protocol

LCHCL augments Tardis, strengthening its guarantees to provide ECSC rather that sequential consistency. Whereas time in Tardis is purely logical, in LCHCL it assumes a hybrid nature.

Our model of time is a two-component vector: $t = (t_c, t_l)$. We call t_c the *clock time component* and t_l the *logical time component*. Note that clock time corresponds to a measurement of physical time, not to physical time itself. Also, while our protocol accounts for the measurement uncertainty reported by TrueTime (see Section 4.5.1.2), a scalar representation of the clock suffices for stored timestamps.

If x and t are hybrid times, we say that $x < y$ when

$$x_c < y_c \vee x_c = y_c \wedge x_l < y_l$$

LCHCL adds two rules to the Tardis protocol (described in Section 4.5.1.1):

- Before executing a MSG_SND_P action to release a message to the environment, it requires $TT.after(pts_c) = true$.
- When executing a MSG_RCV_P action to deliver a message to a process, it advances pts so it is at least $(TT.now().latest, 0)$.

LCHCL also adjusts the lease extension mechanism to support an increment of clock time, logical time, or both in combination:

$$lease\_extend(t, x) = \begin{cases} (t_c, t_l + x_l) & \text{if } x_c = 0 \\ (t_c + x_c, x_l) & \text{otherwise.} \end{cases}$$

With this change to the protocol, the timestamps in the original Tardis protocol take on a hybrid physical-logical nature. As we will show, sequential consistency is preserved, but the use of physical time leads to the enforcement of ECSC. Upon receiving a message, *pts* jumps forward. It may exceed the read reservations, *rts*, of certain cache lines, leading

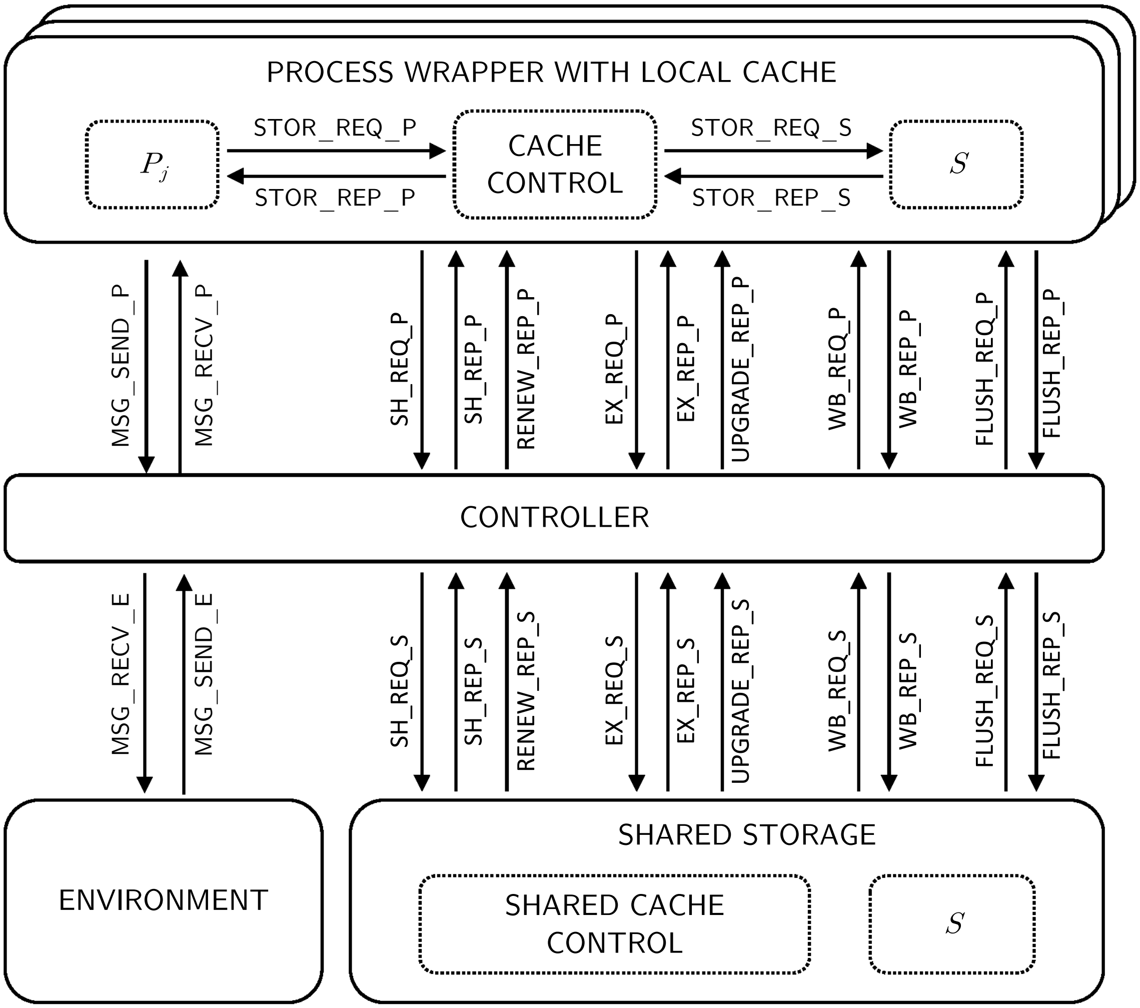


Figure 4.8: ECSC model with timestamp-based coherence protocol.

them to be checked for freshness upon use. This helps ensure that the processor will see the effect of any storage operations that may have occurred before the message it received was sent. We accommodate for clock uncertainty by introducing a delay at message send time, if necessary, in effect waiting out any uncertainty.

Figure 4.8 shows our automaton model of LCHCL. Each process is represented within an automaton that also incorporates cache control mechanisms and a local model of storage. A controller links these to the environment and to the shared storage. Cache actions include: SH_REQ requests for shared access, EX_REQ requests for exclusive access, WB_REQ requests for write-back, FLUSH_REQ requests for cache line flush, and all of their various corresponding responses. These are the same as in Tardis [452], and we refer the reader to the original work for detailed descriptions.

Figure 4.9 shows the transition relation for the process wrapper with local cache. Only two lines distinguish LCHCL from Tardis. We have highlighted these lines, which appear in MSG_RECV_P and MSG_SND_P. Figure 4.10 shows the transition relation for the storage wrapper. There are no changes here relative to Tardis.

The transition relation for the controller is straightforward and we have not included the details here. It is similar to Figure 4.2 in the base model, however, since Tardis assumes in-order commit, it also ensures ordered delivery for communication between the shared cache and each processor. This can be modeled using per-processor queues.

4.5.3 Proof of ECSC for LCHCL

In order to show that behaviors of the LCHCL system are ECSC, we first demonstrate that our changes to the Tardis protocol have not impacted its ability to provide sequential consistency for storage operations (SCS). We present this result in Lemma 6, which we prove by showing that the invariants that the Tardis proof relies upon [453] are not impacted by our changes.

We then move on to the main result of this section, given in Theorem 3. SCS ensures that there exists an action sequence with sequential storage operations that is consistent with the order of storage operations at each process, but ECSC requires reasoning about behaviors of the base model A. In order to go from a sequence of actions of the storage to a behavior of the base model, we use Algorithm 4.3 to combine the original LCHCL behavior with the sequential behavior. We show that the modifications to Tardis described in Section 4.5.2 guarantee that this is possible. We also show that the ordering constraints critical to ECSC are preserved: that we can construct a sequential behavior that preserves the order of actions at each process (i.e., M_P^{SND}, M_P^{RCV}, S_P^{REP}, and S_P^{REQ}) and that never reorders an M_P^{RCV} ahead of an M_P^{SND}.

Lemma 6. *Suppose β is a behavior of $LCHCL(E, \{P_i\}_i, S)$. Then β satisfies SCS, i.e., there exists $\gamma \in seqbehs(S, \Gamma|acts(S))$ such that $\beta|PS \sim \phi_p(\gamma)$.*

Proof. LCHCL is based on Tardis, which guarantees sequential consistency, but it incorporates slight modifications. We must show that these do not undermine the sequential consistency guarantee. As described in Section 4.5.2:

- The timestamp is replaced with a two-component vector that has a component of clock time and a component of logical time. This is a convenience but is immaterial to the Tardis proofs since in either case the timestamps define a total order, i.e., either $x < y$ or $y < x$ or $x = y$.
- We wait before sending messages, requiring that $TT.after(pts_c) = true$. This does not impact the state variables used by Tardis, and such waiting is also irrelevant in an asynchronous model, and thus is irrelevant to the Tardis proof.

- When receiving a message we advance *pts* so that it is at least $(TT.now().latest, 0)$. This does impact the state used by Tardis, but it keeps logical time moving forward. This satisfies an assumption implicit in the Tardis proof. Our changes may cause logical leases may expire, but because we make no adjustments to *wts* or *rts* of any cached item, the mechanisms Tardis uses to ensure sequential consistency are preserved.
- The lease extension mechanism is adjusted to account for the two-component timestamp. The lease extension mechanism is immaterial to the consistency guarantees.

A key assumption of Tardis is in-order processor commit. This means that if X and Y are memory operations originating at the same processor, $X <_p Y \implies X \leq_{ts} Y \wedge X <_{pt} Y$. That is, if X precedes Y in the processor's commit ordering $(<_p)$, then the timestamp of X must be less than or equal to the timestamp of Y $(\leq_{ts})$ and X must also occur before Y in physical time $(<_{pt})$. Thus an algorithm that advances *pts* monotonically is consistent with in-order processor commit, whereas one that may decrease *pts* is not.

The proof of sequential consistency in Tardis centers around a theorem with three invariants [452]. We review these and confirm that they are not impacted by our changes. In this notation $L(a)$ denotes a load at address a and $S(a)$ represents a store at address a.

1. $\textit{Value of } L(a) = \textit{Value of } Max_{<ts} S(a) | S(a) \leq_{ts} L(a)$
2. $\forall S_1(a), S_2(a), S_1(a) \neq_{ts} S_2(a)$
3. $\forall S(a), L(a), S(a) =_{ts} L(a) \implies S(a) <_{pt} L(a)$

Invariant 1 says that the value retrieved by a load must be that which corresponds to the latest store that precedes it, as defined by the timestamp order $(<_{ts})$. This is guaranteed by the use of *rts* and *wts* timestamps and by the exclusive ownership mechanism. Invariant 2 says that no two stores to the same address have the same timestamp, unless they are the same store. This is guaranteed by the timestamp assignment mechanism for stores. Invariant 3 says that if a store and a load occur at the same timestamp then the store precedes the load in physical time. This again is a manifestation of the timestamp assignment mechanism for stores. We thus conclude that the proof of sequential consistency for Tardis is also valid for LCHCL.

Theorem 3. *Suppose β is a behavior of $LCHCL(E, \{P_i\}_i, S)$. Then β satisfies ECSC, i.e., there exists $\beta' \in behs(A)$ with $\beta'|PS \in seqbehs(A, \Gamma|PS)$ such that $\beta|PS \sim \beta'|PS$, and $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$.*

Proof sketch. We use Algorithm 4.3 to construct a behavior β' satisfying ECSC. This algorithm works by aligning and weaving together a number of action sequences: actions of the environment, actions of each process, and actions of the storage augmented by their corresponding process actions. Some key properties of this algorithm are:

- It preserves the order of actions at each process.
- It preserves the order of actions at the environment.
- It inherits the sequentially consistent order of actions at the storage, γ, from Tardis.

- It incorporates M_P^{SND} actions as soon as possible, and M_P^{RCV} actions as late as possible, which is critical to ensuring that $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$.

The SCS aspects of ECSC are inherited from Tardis and preserved through Algorithm 4.3, ensuring that $\beta'|PS \in \mathit{seqbehs}(A, \Gamma|PS)$ and $\beta|PS \sim \beta'|PS$. It remains to prove that $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$ and that Algorithm 4.3 runs successfully and terminates. To show these properties we first analyze the ordering constraints that Algorithm 4.3 places on β'. We define an incorporation dependency, $\blacktriangleleft_{\beta'}$, to reflect how the algorithm waits for one action to be added to β' before adding another.

Modeling logical time and physical time using related representations turns out to be helpful when reasoning about LCHCL. We extend the LCHCL logical time so that it applies to the environment and storage automata and not only the processes. This allows us to associate a timestamp $TS(\pi)$ with any action π in β'. We then show that $\pi_1 \blacktriangleleft_{\beta'} \pi_2 \implies TS(\pi_1) \leq TS(\pi_2)$.

To show that $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$ we need to prove that $X <_{\beta} Y \implies X <_{\beta'} Y$. The design of LCHCL, described in Section 4.5.2, creates a correspondence between physical time and logical time:

$$TS(M_P^{SND}(X)) < pt(M_P^{SND}(X))$$

and

$$TS(M_P^{RCV}(Y)) \geq pt(M_P^{RCV}(Y))$$

However, if $M_P^{RCV}(Y) <_{\beta'} M_P^{SND}(X)$, meaning $M_P^{RCV}(Y) \blacktriangleleft_{\beta'} M_P^{SND}(X)$, then this implies that $pt(M_P^{RCV}(Y)) < pt(M_P^{SND}(X))$, i.e., $M_P^{RCV}(Y) <_{\beta} M_P^{SND}(X)$, so it is not possible that $M_P^{SND}(X) <_{\beta} M_P^{RCV}(Y)$. Since Algorithm 4.3 also preserves the order of actions at each process, we conclude that $<_{\beta}^{ECSC} \subseteq <_{\beta'}^{ECSC}$.

To see that Algorithm 4.3 runs to completion without failing on the assertion at line 25 we need to show that the $\blacktriangleleft_{\beta'}$ relation does not induce any cycles on the actions of β. To see this we first show that all actions in a cycle must occur at the same logical time, i.e., if $\pi_1 \blacktriangleleft_{\beta'} \pi_2 \cdots \blacktriangleleft_{\beta'} \pi_n \blacktriangleleft_{\beta'} \pi_1$ then $TS(\pi_1) = \cdots = TS(\pi_n)$. We then note that any cycle that goes through the environment advances the logical time on M_P^{RCV}, and so the cycle must not involve the environment. The actions of the storage are sequential and consistent with the order of operations at each process, because Tardis guarantees sequential consistency, so a $\blacktriangleleft_{\beta'}$ cycle through storage only is not possible either. We conclude that the assertion of line 25 always passes.

Prerequisites. Before proceeding to the detailed proof of Theorem 3 we analyze the properties of Algorithm 4.3 and define some helpful machinery.

We can infer a number of ordering constraints, or incorporation dependencies, imposed

Algorithm 4.3 Construction of $\beta' = \pi_1^{\beta'} \pi_2^{\beta'} \ldots \pi_n^{\beta'}$ for Theorem 3.

```
$\beta \leftarrow \beta \mid \beta \in behs(LCHCL(E, \{P_i\}_i, S))$
$\pi_1^{\gamma}\pi_2^{\gamma} \ldots \pi_{n_\gamma}^{\gamma} \leftarrow \gamma \mid \gamma \in seqbehs(S, \Gamma|acts(S))$ such that $\beta|PS \sim \phi_p(\gamma)$
$\pi_1^{e}\pi_2^{e} \ldots \pi_{n_e}^{e} \leftarrow \beta|ext(E)$    $\triangleright\ \alpha_e$
$\forall P_j : \pi_1^{p_j}\pi_2^{p_j} \ldots \pi_{n_{p_j}}^{p_j} \leftarrow \beta|ext(P_j)$    $\triangleright\ \alpha_{p_j}$
$\pi_1^{s}\pi_1^{s} \ldots \pi_{n_s}^{s} \leftarrow \phi_p(\pi_1^{\gamma})\pi_1^{\gamma}\pi_2^{\gamma}\phi_p(\pi_2^{\gamma})\phi_p(\pi_3^{\gamma})\pi_3^{\gamma}\pi_4^{\gamma}\phi_p(\pi_4^{\gamma}) \ldots \phi_p(\pi_{n_\gamma - 1}^{\gamma})\pi_{n_\gamma - 1}^{\gamma}\pi_{n_\gamma}^{\gamma}\phi_p(\pi_{n_\gamma}^{\gamma})$    $\triangleright\ \alpha_s$
$i_e \leftarrow 1$, $i_s \leftarrow 1$, $i_{p_j} \leftarrow 1\ \forall p_j$, $k \leftarrow 1$
while $i_e \leq n_e \vee i_s \leq n_s \vee \exists p_j \mid i_{p_j} \leq n_{p_j}$ do
    if $\exists p_j \mid i_{p_j} \leq n_{p_j} \wedge \pi_{i_{p_j}}^{p_j} = M_P^{SND}(X)$ then
        $p_l \leftarrow \underset{p_j}{\mathrm{argmin}_{<_\beta}}\ \pi_{i_{p_j}}^{p_j} \mid i_{p_j} \leq n_{p_j} \wedge \pi_{i_{p_j}}^{p_j} = M_P^{SND}(X)$
        $msg\_p2e\_ready \leftarrow msg\_p2e\_ready \cup \{X \mid \pi_{i_{p_l}}^{p_l} = M_P^{SND}(X)\}$
        $\pi_k^{\beta'} \leftarrow \pi_{i_{p_l}}^{p_l}$
        $i_{p_l} \leftarrow i_{p_l} + 1$
    else if $i_e \leq n_e \wedge \pi_{i_e}^{e} = M_E^{RCV}(X) \wedge X \in msg\_p2e\_ready$ then
        $\pi_k^{\beta'} \leftarrow \pi_{i_e}^{e}$
        $i_e \leftarrow i_e + 1$
    else if $i_s \leq n_s \wedge \exists p_j \mid i_{p_j} \leq n_{p_j} \wedge \pi_{i_{p_j}}^{p_j} = \pi_{i_s}^{s}$ then
        $\pi_k^{\beta'} \leftarrow \pi_{i_s}^{s}$
        $i_{p_j} \leftarrow i_{p_j} + 1$
        $i_s \leftarrow i_s + 1$
    else if $i_e \leq n_e \wedge \pi_{i_e}^{e} = M_E^{SND}(X)$ then
        $msg\_e2p\_ready \leftarrow msg\_e2p\_ready \cup \{X \mid \pi_{i_e}^{e} = M_E^{SND}(X)\}$
        $\pi_k^{\beta'} \leftarrow \pi_{i_e}^{e}$
        $i_e \leftarrow i_e + 1$
    else
        $Assert(\exists p_j \mid i_{p_j} \leq n_{p_j} \wedge \pi_{i_{p_j}}^{p_j} = M_P^{RCV}(X) \wedge X \in msg\_e2p\_ready)$
        $p_l \leftarrow \underset{p_j}{\mathrm{argmin}_{<_\beta}}\ \pi_{i_{p_j}}^{p_j} \mid i_{p_j} \leq n_{p_j} \wedge \pi_{i_{p_j}}^{p_j} = M_P^{RCV}(X) \wedge X \in msg\_e2p\_ready$
        $\pi_k^{\beta'} \leftarrow \pi_{i_{p_l}}^{p_l}$
        $i_{p_l} \leftarrow i_{p_l} + 1$
    end if
    $k \leftarrow k + 1$
end while
```

on β' by Algorithm 4.3. The algorithm enforces $\pi_i^\mu <_{\beta'} \pi_j^\nu$, which we write as, $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$, if:

$$\begin{gathered}
i < j \land \exists A \in \{E\} \cup \{P_i\}_i \cup \{S\} \mid \{\pi_i^\mu, \pi_j^\nu\} \subseteq ext(A) \\
\text{or} \\
\pi_j^\nu = M_E^{RCV}(X) \land \pi_i^\mu = M_P^{SND}(X) \\
\text{or} \\
\pi_j^\nu = M_P^{RCV}(X) \land \pi_i^\mu = M_E^{SND}(X) \\
\text{or} \\
\pi_j^\nu = S_P^{REQ}(X) \land \exists k > i \mid \pi_k^\mu = S_P^{REQ}(X) \\
\text{or} \\
\pi_j^\nu = S_P^{REP}(X, v) \land \exists k > i \mid \pi_k^\mu = S_P^{REP}(X, v) \\
\text{or} \\
\pi_i^\mu = \pi_h^\lambda \land \pi_h^\lambda \blacktriangleleft_{\beta'} \pi_j^\nu \\
\text{or} \\
\pi_j^\nu = \pi_h^\lambda \land \pi_i^\mu \blacktriangleleft_{\beta'} \pi_h^\lambda
\end{gathered} \tag{4.2}$$

The first condition simply says that Algorithm 4.3 must draw in order from the actions of each automaton of the base model. This is actually slightly weaker than what the algorithm does, which is to draw in order from the action sequences α_e, α_s, and $\{\alpha_{p_i}\}_{p_i}$. The second condition describes the case where the environment requires a message from a process to progress, and the third condition describes the case where a process requires a message from the environment to progress. The fourth and fifth describe either the case where a process cannot progress until the storage progresses, or where the storage cannot progress until a process does. The last two conditions apply to those actions that appear in two action sequences: S_P^{REQ} and S_P^{REP}, and reflect the fact that Algorithm 4.3 incorporates them both at the same time.

In Table 4.4 we define a timestamp operator, $TS(\pi)$, that extends the notion of logical time encoded in *pts*. For those actions that take place at a process P_i, $TS(\pi)$ simply reflects the process *pts* after action π. For actions at storage, $TS(S_S^{REQ}(X))$ and $TS(S_S^{REP}(X))$ are both defined based on the logical timestamp of the reply. We will sometimes write $\pi_i^\mu <_{TS} \pi_j^\nu$ when $TS(\pi_i^\mu) < TS(\pi_j^\nu)$.

Lemma 7. *If $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$ then $\pi_i^\mu \leq_{TS} \pi_j^\nu$.*

Proof. At each process *pts* increases monotonically, so it is clear that if π_i^μ and π_j^ν are both process actions then $\pi_i^\mu \leq_{TS} \pi_j^\nu$ whenever $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$. For the environment, the definition of TS in Table 4.4 describes *ets*, a logical timestamp that is updated such that it increases monotonically along α_e, lagging behind physical time and never exceeding it. *ets* reflects the highest *pts* at the time a message was sent across all messages received by the environment.

Table 4.4: Timestamp operator definition.

Action: π	$TS(\pi)$	Update rule
$M_P^{SND}(X)$	pts at P_i after π	
$M_E^{RCV}(X)$	ets after π	$ets = max(e.ts, TS(M_P^{SND}(X)))$
$M_E^{SND}(X)$	ets	
$M_P^{RCV}(X)$	pts at P_i after π	
$S_P^{REQ}(X)$	pts at P_i after π	
$S_S^{REQ}(X)$	pts contained in v in $S_S^{REP}(X, v)$	
$S_S^{REP}(X, v)$	pts contained in v	
$S_P^{REP}(X, v)$	pts at P_i after π	

Monotonicity of ets ensures that whenever $\pi_i^e \blacktriangleleft_{\beta'} \pi_j^e$ then also $TS(\pi_i^e) \leq TS(\pi_j^e)$. For operations of the storage we rely on the fact that Tardis uses logical timestamps to help enforce sequential consistency. In Tardis, the global memory order $X <_m Y$ is defined as $X <_{ts} Y \vee (X =_{ts} Y \wedge X <_{pt} Y)$ [452], where $<_{ts}$ is logical-time order and $<_{pt}$ is physical-time order.[2] Since γ reflects the global memory order, we know that when π_i^μ and π_j^ν are both actions of S, then also $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$ implies $\pi_i^\mu \leq_{TS} \pi_j^\nu$.

We next turn our attention to the incorporation dependencies that cross between processes, the environment, and storage. In the case of $M_P^{SND}(X) \blacktriangleleft_{\beta'} M_E^{RCV}(X)$, $S_P^{REQ}(X) \blacktriangleleft_{\beta'} S_S^{REQ}(X)$, and $S_P^{REP}(X) \blacktriangleleft_{\beta'} S_S^{REP}(X)$, it is clear that $TS(\pi)$ as defined in Table 4.4 ensures that $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$ implies $\pi_i^\mu \leq_{TS} \pi_j^\nu$. In the case of $M_E^{SND}(X) \blacktriangleleft_{\beta'} M_P^{RCV}(X)$, we note that $TS(M_E^{SND}(X))$ is equal to the maximum of all $M_P^{SND}(Y)$ previously received at E, and that LCHCL ensures that $M_P^{SND}(Y)$ only occurs once $TS(M_P^{SND}(Y))$ is in the past. However LCHCL also ensures that $TS(M_P^{RCV}(X))$ is greater than or equal to the present time. Thus again when $\pi_i^\mu = M_E^{SND}(X)$ and $\pi_j^\nu = M_P^{RCV}(X)$, $\pi_i^\mu \blacktriangleleft_{\beta'} \pi_j^\nu$ implies $\pi_i^\mu \leq_{TS} \pi_j^\nu$.

Proof of Theorem 3. Lemma 6 tells us that β satisfies sequential consistency, so there exists $\gamma \in seqbehs(S, \Gamma | acts(S))$ such that $\beta | PS \sim \phi_p(\gamma)$. We construct β' using Algorithm 4.3. This algorithm works by weaving together three types of action sequences:

- External actions of the environment, M_E^{SND} and M_E^{RCV}, drawn from β. See line 3.
- External actions of each process, M_P^{SND}, M_P^{RCV}, S_P^{REQ}, and S_P^{REP}, drawn from $\beta | P_j$ for process P_j. See line 4.
- External actions of the storage, S_S^{REQ} and S_S^{REP}, drawn from γ and augmented with their corresponding process-mapped equivalents: S_P^{REQ} and S_P^{REP}. See line 5 and also Section 4.2.4.

Algorithm 4.3 ensures that the resulting action sequence β' is a behavior of the composition

[2] Our definition of $<_{TS}$ is consistent with the original Tardis definition of $<_{ts}$, and extends it to describe progress of the environment and storage automata.

A by incorporating an action from one of its constituent automata at each iteration of the loop beginning on line 7. We can also see that the resulting action sequence β' is well formed:

- We ensure that $M_E^{SND}(X)$ precedes $M_P^{RCV}(X)$ using *msg_e2p_ready* (line 21 and line 26). This condition is guaranteed structurally by the algorithm, but we have added the assertion on line 25 for clarity.
- We ensure that $M_P^{SND}(X)$ precedes $M_E^{RCV}(X)$ using *msg_p2e_ready* (line 9 and line 13).
- We ensure that $S_P^{REQ}(X)$ precedes $S_S^{REQ}(X)$ and that $S_S^{REQ}(X)$ precedes $S_S^{REP}(X, v)$ and that $S_S^{REP}(X, v)$ precedes $S_P^{REP}(X, v)$ using the construction of α_s on line 5. The algorithm aligns α_s with the process actions of α_{p_j} on line 16.

It is clear that $\beta'|PS \in \mathit{seqbehs}(A, \Gamma|PS)$ since $\beta'|PS = \alpha_s|PS$ and since α_s is derived from γ, which is sequential. Similarly, $\beta|PS \sim \beta'|PS$. First, $\beta'|E = \alpha_e = \beta|$E. Second, Algorithm 4.3 builds β' by incorporating actions of each process in sequence, i.e., $\beta'|P_i = \alpha_{p_i}|P_i = \beta|P_i$, so we also know for each P_i that $\beta|PS|P_i = \beta'|PS|P_i$.

We next show that $<_\beta^{ECSC} \subseteq <_{\beta'}^{ECSC}$. Let μ and ν be action sequences used in Algorithm 4.3, i.e.,

$$\mu, \nu \in \{\pi_1^{p_j} \dots \pi_{n_{p_j}}^{p_j}\}_{p_j} \cup \{\pi_1^e \dots \pi_{n_e}^e\} \cup \{\pi_1^s \dots \pi_{n_s}^s\}$$

Recall from Definition 14 that $\pi_i^\mu <_\beta^{ECSC} \pi_j^\nu$ when $\pi_i^\mu <_{\beta|P_i} \pi_j^\nu$ for some P_i, or when $\pi_i^\mu <_\beta \pi_j^\nu$ and π_i^μ is $M_P^{SND}(X)$ and π_j^ν is $M_P^{RCV}(Y)$. Since $\beta|P_i = \beta'|P_i$, it remains to show that $M_P^{SND}(X) <_\beta M_P^{RCV}(Y)$ implies $M_P^{SND}(X) <_{\beta'} M_P^{RCV}(Y)$.

Suppose now that Algorithm 4.3 incorporates $M_P^{RCV}(Y)$ before $M_P^{SND}(X)$. As $M_P^{SND}(Y)$ is incorporated in the case beginning on line 8 whereas $M_P^{SND}(X)$ is incorporated in the case beginning on line 24, there must exist a dependency, possibly indirectly, between $M_P^{RCV}(Y)$ and $M_P^{SND}(X)$. That is,

$$M_P^{RCV}(Y) \blacktriangleleft_{\beta'} \cdots \blacktriangleleft_{\beta'} M_P^{SND}(X)$$

By Lemma 7, $M_P^{RCV}(Y) \leq_{TS} \cdots \leq_{TS} M_P^{SND}(X)$, and so $TS(M_P^{RCV}(Y)) \leq TS(M_P^{SND}(X))$. As described in Section 4.5.2, LCHCL enforces relationships between physical time, represented here as $pt(\pi)$ and logical time: $TS(M_P^{SND}(X)) < pt(M_P^{SND}(X))$ and $TS(M_P^{RCV}(Y)) \geq pt(M_P^{RCV}(Y))$. Thus we have

$$pt(M_P^{RCV}(Y)) \leq TS(M_P^{RCV}(Y)) \leq TS(M_P^{SND}(X)) < pt(M_P^{SND}(X))$$

This is a contradiction, however, for $M_P^{SND}(X) <_\beta M_P^{RCV}(Y)$. We thus conclude that $M_P^{SND}(X) <_\beta M_P^{RCV}(Y)$ implies $M_P^{SND}(X) <_{\beta'} M_P^{RCV}(Y)$ and that as a result $<_\beta^{ECSC} \subseteq <_{\beta'}^{ECSC}$.

We next verify that Algorithm 4.3 completes successfully: that the assertion on line 25 always passes and that the algorithm terminates. If the assertion fails then either all $M_P^{RCV}(X)$ have been incorporated, or all those not yet incorporated have a dependency on some other action. In either case, this implies a cyclic dependency of the form:

$$\pi_1 \blacktriangleleft_{\beta'} \pi_2 \cdots \blacktriangleleft_{\beta'} \pi_n \blacktriangleleft_{\beta'} \pi_1$$

By Lemma 7, this implies $\pi_1 \leq_{TS} \pi_2 \leq_{TS} \cdots \leq_{TS} \pi_1$, and so all actions in the cycle occur at the same logical time: $TS(\pi_1) = \cdots = TS(\pi_n)$. Receiving a message advances *pts*, so a cyclic dependency through messaging is not possible. SCS ensures that storage actions occur sequentially in γ, and that this order is consistent with the order of actions at each process. Thus a cycle is not possible through storage alone. We conclude that the assertion on line 25 of Algorithm 4.3 always succeeds. In closing, we note that at least one of the index variables i_e, i_s, i_{p_j} is incremented on every iteration of the loop beginning on line 7. This ensures that Algorithm 4.3 terminates.

4.5.4 Future Work

Additional work is required to make LCHCL practical in a real system. Tardis assumes a single failure domain, as is reasonable for a processor cache coherence protocol. This is not an appropriate assumption in distributed systems, but we have not provided for fault tolerance in this work. We believe that part of the solution could come from techniques developed for shared file systems, such as using leases [169] to ensure that delegated access can always be reclaimed within a bounded time interval, even in the presence of failures. Leases added for fault tolerance should not be confused with the logical leases already used by Tardis and LCHCL to decouple readers from writers. They would fit naturally with LCHCL, however, since it already relies upon physical clocks.

Another area for future work involves tuning the logical lease mechanism that we have already described. Increasing the lease extension interval can reduce cache coherence traffic, but also can push *pts* further into the future, which in LCHCL may introduce delays when sending messages. There are also a number of related optimizations to explore: buffering outgoing messages so that they do not hold up the sending process, preemptively requesting lease extensions to avoid read delays, and dynamically choosing the lease extension interval. It may also be interesting to explore the consequences of extending leases using clock time, logical time, or both in combination.

One limitation of LCHCL is that it is restricted to read and write operations, whereas ECSC can describe a more general class of stateful objects, e.g., queues. LCHCL inherits this limitation from Tardis, and the opportunity remains to develop implementations of ECSC for other classes of storage operations.

```
MSG_RCV_P (x)
    Precondition:
        $\exists s'' \mid (s'.underlying, M_P^{RCV}(x), s'') \in steps(P_j)$
    Effect:
        $s.underlying = s''$
        $s.pts = max(s'.pts, (TT.now().latest, 0))$                ▷ LCHCL extension

MSG_SND_P (x)
    Precondition:
        $\exists s'' \mid (s'.underlying, M_P^{SND}(x), s'') \in steps(P_j)$
        $s'.working\_on = \textsc{None}$
        $TT.after(s'.pts_c)$                                        ▷ LCHCL extension
    Effect:
        $s.underlying = s''$

STOR_REQ_P (x) (INTERNAL)
    Precondition:
        $\exists s'' \mid (s'.underlying, S_P^{REQ}(x), s'') \in steps(P_j)$
        $s'.working = false$
    Effect:
        $s.underlying = s''$
        $s.stor\_req\_p = x$
        $s.working = true$

STOR_REP_P (x,v) (INTERNAL)
    Precondition:
        $s'.stor\_rep\_p = (x, v)$
        $\exists s'' \mid (s'.underlying, S_P^{REP}(x), s'') \in steps(P_j)$
    Effect:
        $s.underlying = s''$
        $s.stor\_rep\_p = \textsc{None}$
        $s.working = false$

SH_REQ_P(x,wts,pts) - cache miss on read, shared mode
    Precondition:
        $s'.stor\_req\_p = x$
        $kind(x) = \textsc{Read}$
        $(wts, pts) = ((0,0), s'.pts) \wedge s'.cc[x].mode = \text{INV}$
            $\vee (wts, pts) = ((0,0), s'.cc[x].wts) \wedge s'.pts > s'.cc[x].rts$
    Effect:
        $s.stor\_req\_p = \textsc{None}$
```

Figure 4.9: Transition relation for process wrapper with local cache.

INTERNAL - cache hit on read, shared mode

```
Precondition:
    $s'.stor\_req\_p = x$
    $kind(x) = \textsc{Read}$
    $s'.cc[x].mode = \text{SH} \wedge s'.pts \leq s'.cc[x].rts$
Effect:
    $s.pts = max(s'.pts, s'.cc[x].wts)$
    $s.stor\_req\_p = \textsc{None}$
    $s.stor\_req\_s = x$
```

INTERNAL - cache hit on read, exclusive mode

```
Precondition:
    $s'.stor\_requests = x \cdot rest$
    $kind(x) = \textsc{Read}$
    $s'.cc[x].mode = \text{EX}$
Effect:
    $s.pts = max(s'.pts, s'.cc[x].wts)$
    $s.cc[x].rts = max(s'.pts, s'.cc[x].rts)$
    $s.stor\_req\_p = \textsc{None}$
    $s.stor\_req\_s = x$
```

INTERNAL - write, in exclusive mode

```
Precondition:
    $s'.stor\_req\_p = x$
    $kind(x) = \textsc{Write}$
    $s.cc[x].mode = \text{EX}$
Effect:
    $s.pts = s.cc[x].rts = s.cc[x].rts = max(s'.pts, s'.cc[x].wts + (0, 1))$
    $s.stor\_req\_p = \textsc{None}$
    $s.stor\_req\_s = x$
```

Figure 4.9: Transition relation for process wrapper with local cache.

EX_REQ_P(x,wts) - write, in shared mode

Precondition:

$s'.stor\_req\_p = x$

$kind(x) = \textsc{Write}$

$wts = 0 \wedge s'.cc[x].mode = \text{INV}$

$s'.working\_on = \textsc{None}$

$\vee wts = s'.cc[x].wts \wedge s'.pts > s'.cc[x].rts$

Effect:

$s.stor\_req\_p = \textsc{None}$

STOR_REQ_S (x) (INTERNAL)

Precondition:

$s'.stor\_req\_s = x$

$\exists s'' \mid (s'.stor, S_S^{REQ}(x), s'') \in steps(S)$

Effect:

$s'.stor = s''$

$s'.stor\_req\_s = none$

$s'.stor\_rep\_s = (x, v)$

STOR_REP_S (x,v) (INTERNAL)

Precondition:

$s'.stor\_rep\_s = (x, v)$

Effect:

$s'.stor = s''$

$s'.stor\_rep\_s = none$

$s'.stor\_rep\_p = (x, v)$

RENEW_REP_P(x,rts)

Effect:

$s.cc[x].rts = rts$

$s.stor\_req\_s = x$

SH_REP_P(x,wts,rts,value)

Effect:

$s.cc[x].mode = \text{SH}$

$s.cc[x].rts = rts$

$s.cc[x].wts = wts$

$s.stor = s''' \mid \{(s.stor, S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$

$s.stor\_req\_s = x$

Figure 4.9: Transition relation for process wrapper with local cache.

```
EX_REP_P(x,p,wts,value)
  Effect:
    $s.cc[x].mode = \mathrm{EX}$
    $s.cc[x].rts = rts$
    $s.cc[x].wts = wts$
    $s.stor = s''' \mid \{(s.stor, S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$
    $s.stor\_req\_s = x$

UPGRADE_REP_P(x,rts)
  Effect:
    $s.cc[x].mode = \mathrm{EX}$
    $s.cc[x].rts = rts$
    $s.stor\_req\_s = x$

FLUSH_REQ_P(x,p)
  Effect:
    $s.cc[x].mode = \mathrm{INV}$
    $s.flush\_resp = s'.flush\_resp \cdot (x, s'.cc[x].wts, s'.cc[x].rts, value)$
      where $\{(s.stor, S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$

FLUSH_REP_P(x,wts,rts,value)
  Precondition:
    $s'.flush\_resp = (x, wts, rts, value) \cdot rest$
  Effect:
    $s.flush\_resp = rest$

WB_REQ_P(x,p,rts)
  Effect:
    $s.cc[x].mode = \mathrm{SH}$
    $s.cc[x].rts = max(s'.cc[x].rts, lease\_extend(s'.cc[x].wts + lease), rts)$
    $s.wb\_resp = s'.wb\_resp \cdot (x, s'.cc[x].wts, s'.cc[x].rts, value)$
      where $\{(s.stor, S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$

WB_REP_P(x,wts,rts,value)
  Precondition:
    $s'.wb\_resp = y \cdot rest$
  Effect:
    $s.wb\_resp = rest$
```

Figure 4.9: Transition relation for process wrapper with local cache.

SH_REQ_S(x,pts,wts)

Effect:

$s.sh\_req[x] = s'.sh\_req[x] \cdot (x, pts, wts)$

INTERNAL - shared request, can be granted immediately

Precondition:

$s'.cc[x].mode = \mathrm{SH}$

$s'.sh\_req[x] = (x, pts, wts) \cdot rest$

$s.working\_on[x] = false$

Effect:

$s.sh\_req[x] = rest$

$s.cc[x].rts = max(s'.cc[x].rts, lease\_extend(max(pts, s'.cc[x].wts), lease))$

$$s.renew\_rep = \begin{cases} s'.renew\_rep \cdot (x, s.cc[x].rts) & \text{if } wts = s'.cc[x].wts \\ s'.renew\_rep & \text{otherwise} \end{cases}$$

$$s.sh\_rep = \begin{cases} s'.sh\_rep & \text{if } wts = s'.cc[x].wts \\ s'.sh\_rep \cdot (x, s.cc[x].wts, s.cc[x].wts, value) & \text{otherwise} \end{cases}$$

where $\{(s.stor[x], S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$

INTERNAL - shared request, not granted immediately

Precondition:

$s'.cc[x].mode = \mathrm{EX}$

$s'.sh\_req[x] = (x, pts, wts) \cdot rest$

Effect:

$s.sh\_req[x] = rest$

$s.wb\_req[x] = (x, cc[x].owner, lease\_extend(pts, lease))$

$s.sh\_rep\_wait[x] = s'.sh\_rep\_wait[x] \cdot x$

$s.working\_on[x] = true$

Figure 4.10: Transition relation for storage wrapper.

EX_REQ_S(x,wts)

Effect:

$s.ex\_req[x] = s'.ex\_req[x] \cdot (x, wts)$

(INTERNAL) - exclusive request, can be granted immediately

Precondition:

$s'.cc[x].mode = \text{SH}$

$s'.ex\_req[x] = (x, wts) \cdot rest$

$s.working\_on[x] = false$

Effect:

$s.ex\_req[x] = rest$

$s.cc[x].mode = \text{EX}$

$$s.upgrade\_rep = \begin{cases} s'.upgrade\_rep \cdot (x, rts) & \text{if } wts = s'.cc[x].wts \\ s'.upgrade\_rep & \text{otherwise} \end{cases}$$

$$s.ex\_rep = \begin{cases} s'.ex\_rep & \text{if } wts = s'.cc[x].wts \\ s'.ex\_rep \cdot (x, s'.cc[x].wts, s'.cc[x].rts, value) & \text{otherwise} \end{cases}$$

where $\{(s.stor[x], S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, value), s''')\} \subseteq steps(S)$

(INTERNAL) - exclusive request, not granted immediately

Precondition:

$s'.cc[x].mode = \text{EX}$

$s'.ex\_req[x] = (x, wts) \cdot rest$

$s.working\_on[x] = false$

Effect:

$s.ex\_req[x] = rest$

$s.cc[x].mode = \text{EX}$

$s.flush\_req[x] = s'.flush\_req \cdot (x, s'.cc[x].owner)$

$s.working\_on[x] = true$

Figure 4.10: Transition relation for storage wrapper.

```
WB_REQ_S(x,p,rts)
    Precondition:
        $s'.\mathit{wb\_req}[x] = (x, p, \mathit{rts})$
    Effect:
        $s.\mathit{wb\_req} = \textsc{None}$

WB_REP_S(x,wts,rts,value)
    Effect:
        $s.\mathit{cc}[x].\mathit{mode} = \mathrm{SH}$
        $s.\mathit{cc}[x].\mathit{wts} = \mathit{wts}$
        $s.\mathit{cc}[x].\mathit{rts} = \mathit{rts}$
        $s.\mathit{stor}[x] = s''' \mid \{(s.\mathit{stor}[x], S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, \mathit{value}), s''')\} \subseteq \mathit{steps}(S)$
        $s.\mathit{sh\_rep} = s'.\mathit{sh\_rep} \cdot (\mathit{rts}, \mathit{wts}, s'.\mathit{sh\_rep\_wait}[x][0]) \cdot \ldots$
            $\cdot (\mathit{rts}, \mathit{wts}, s'.\mathit{sh\_rep\_wait}[x][n])$
        $s.\mathit{sh\_rep\_wait}[x] = []$
        $s.\mathit{working\_on}[x] = \mathit{false}$

FLUSH_REQ_S(x,p)
    Precondition:
        $s'.\mathit{flush\_req}[x] = (x, p)$
    Effect:
        $s.\mathit{flush\_req} = \textsc{None}$

FLUSH_REP_S(x,wts,rts,value)
    Effect:
        $s.\mathit{cc}[x].\mathit{wts} = \mathit{wts}$
        $s.\mathit{cc}[x].\mathit{rts} = \mathit{rts}$
        $s.\mathit{stor}[x] = s''' \mid \{(s.\mathit{stor}[x], S_S^{REQ}(x), s''), (s'', S_S^{REP}(x, \mathit{value}), s''')\} \subseteq \mathit{steps}(S)$
        $s.\mathit{ex\_rep} = (x, \mathit{rts}, \mathit{wts}, \mathit{value})$
        $s.\mathit{working\_on} = \mathit{false}$
```

Figure 4.10: Transition relation for storage wrapper.

SH_REP_S(x,wts,rts,value)
  Precondition:
    $s'.sh\_rep = (x, wts, rts, value) \cdot rest$
  Effect:
    $s.sh\_rep = rest$

RENEW_REP_S(x,rts)
  Precondition:
    $s'.renew\_rep = (x, rts) \cdot rest$
  Effect:
    $s.renew\_rep = rest$

EX_REP_S(x,wts,rts,value)
  Precondition:
    $s'.ex\_rep = (x, wts, rts, value)$
  Effect:
    $s.ex\_rep = \textsc{None}$

UPGRADE_REP_S(x,rts)
  Precondition:
    $s'.upgrade\_rep = (x, rts)$
    $y.rts = rts$
  Effect:
    $s.upgrade\_rep = \textsc{None}$

Figure 4.10: Transition relation for storage wrapper.

Chapter 5

Servers Are Here to Stay

5.1 Introduction

Servers are plentiful and easily accessible due to the rise of cloud computing, but their proliferation causes problems. In Chapter 1 and Chapter 2, we explained how developing programs for the cloud can be much more complicated than developing programs that run on a single computer. In Section 2.8.5, we explained why servers embody this problem: Reasoning about them creates "accidental complexity" that has nothing to do with the problem the programmer is trying to solve.

By hiding servers behind abstractions, serverless computing can simplify cloud programming. Yet while serverless computing is about abstracting away servers, other work has sought to reimagine how data centers are physically built. Servers are tightly coupled units containing CPUs, memory, and storage. Hardware disaggregation explores how these might be replaced with loosely coupled designs, say by connecting each of a server's major internal components directly to a next-generation data center network. This would offer the flexibility to assemble resources in whatever configuration an application needs rather than being limited to a configuration fixed by server design. This might speed up data sharing and enable higher utilization.

Could the serverless computing movement and the hardware disaggregation movement, taken together, spell the end of server-based data centers? While we believe that there are important synergies between these areas [315], we show in this chapter that server hardware has important characteristics that guarantee it an ongoing role. Even if programming abstractions are fully serverless, server hardware, of some form, is irreplaceable.

Instead of defining server hardware based on the extent of a sheet metal enclosure, we focus on a defining functional aspect: a domain of low-latency communication. Preserving low latency when scaling an interconnect to encompass more endpoint nodes leads to quadratically increasing costs, so building large servers, defined in this way, becomes expensive. Coupling many such servers using an interconnect with more gradual cost scaling lowers the cost of adding resources to a workload but necessarily leads to higher communication latency.

Low-latency communication is not important for all workloads, but it is critical for some. For these workloads, a large server, despite its high cost per processor, always gets the job done more cheaply than a collection of smaller servers. We show this using Amdahl's law, which was first used to make the case that parallel processing alone could not meet growing demands for computing power; Amdahl suggested that faster sequential processing would be essential as well. In our model, we incorporate communication into Amdahl's law alongside computation. We distinguish between *sequential communication*, that which lies on a workload's critical path, and the remaining *parallelizable communication*. We then seek to understand what sorts of workloads are likely to run best on large servers and which will run well on collections of small servers. Those workloads that make heavy use of coordination protocols are most likely to produce sequential communication and require large servers to scale, whereas those that rely on coordination-free protocols can take advantage of many smaller servers, or disaggregated hardware.

To evaluate our model in the context of a workload, we applied a dataflow graph analysis to the YCSB [112] cloud database benchmark. Coordination protocols are usually used to enforce strong consistency, whereas coordination-free protocols enable weak consistency models, and in this experiment we evaluated both. Our analysis confirms that strong consistency is more likely to benefit from large servers, whereas weak consistency is better able to take advantage of low-cost small servers. This result aligns with previous work [51, 81], but our model is based on simple fundamental relations, including Amdahl's law and those governing interconnect cost scaling. It also arises in a deterministic and failure-free setting, which demonstrates that neither non-determinism nor faults are necessary to give this advantage to weak consistency.

It is tempting to predict the demise of physical servers as a consequence of the rise of serverless computing. However, while we expect serverless computing to accelerate the adoption of new data center hardware [315, 356], we also expect server hardware to maintain an ongoing role in the data center even as serverless abstractions remove servers from the programming model.

5.2 Background

5.2.1 Disaggregated Hardware

Almost as soon as the concept of warehouse-scale computers [60] became widely recognized, there arose a movement to tear them apart and reimagine them. This included hardware vendors, academic researchers, and some owners of data centers. The proposals included Firebox [40] from UC Berkeley, The Machine from HP [202], Facebook's Disaggregated Rack [141], Huawei Data Center 3.0 [250], Intel's Rack Scale Architecture [206], and dReD-Box[15, 223], a consortium project.

The case for disaggregated hardware shares some motivations with serverless computing: resource utilization in the data center can be very low due to overprovisioning and inflexible allocations [334], and the optimal hardware mix must evolve over time to meet changing workloads [102, 151, 186].

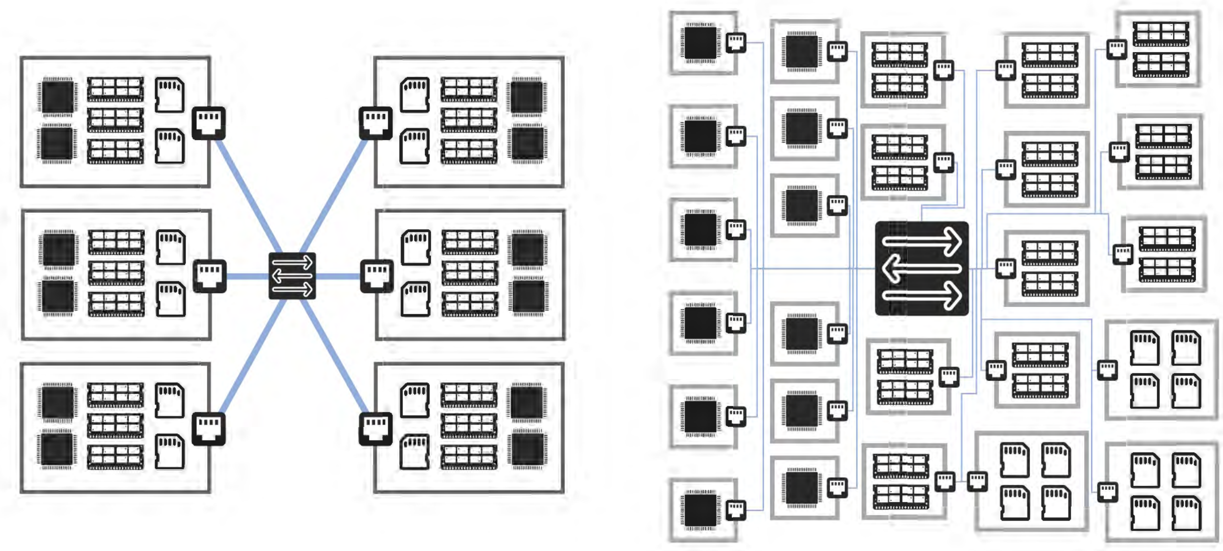
The hardware disaggregation movement has been emboldened by rapid advancements in network technologies such as InfiniBand [174, 318]. For example, 400 Gb/s InfiniBand is commercially available today, and the InfiniBand roadmap outlines a progression up to rates of 4.8 Tb/s [205]. Additional encouragement comes from new memory technologies [97] and new memory interfaces [253, 314]. For example, Gen-Z [161] provides load/store memory semantics over a fabric, and similar technologies like CCIX [93] and CAPI [438] focus on connecting CPUs and accelerators using a memory interface.

Somewhat paradoxically, another trend that encourages disaggregation is the tighter integration of the components that would otherwise comprise a server. One form of this is known as system-on-chip (SoC) design [344, 440]. SoC designs can combine a CPU, memory controller, network interface, accelerators, and other elements on a single piece of silicon. Memory chips might be sliced from wafers and glued directly to the SoC, connected to it by through-silicon vias, thus yielding a single chip-like package containing all of the core elements of a server [310].

Google has indicated that SoCs factor prominently in its cloud roadmap [414]. SoC techniques are used extensively in mobile phones and thus benefit from massive economies of scale. However SoC-based server designs are not yet as powerful as traditional high-end servers, so it takes more of them to assemble an equivalent amount of resources. The result is a collection of smaller servers connected by a data center network, which must be even bigger as a result.

Data center designers could also use SoCs to create modules containing a network interface and just one other type of resource. This could allow them to deploy compute, memory, and storage as independent network-addressable resources. SoCs are well suited to such a network-centric approach because they allow the network interface to be integrated closely with other circuitry. In the work that follows we do not distinguish between disaggregated hardware built from small servers and that built from single-resource network-attached nodes. In either case, the implication is similar: disaggregation turns communication within a server into communication over a data center network.

Research advances in silicon photonics [379, 405] could encourage a shift toward disaggregated hardware. One exciting example is a single-chip processor that communicates using light [390]. Light can carry signals faithfully and efficiently over longer distances than metal wires can, and silicon photonics promises the integration of optical communication components on logic and memory chips. The problem with doing this today is that the semiconductor materials used to make light sources and detectors are incompatible with the manufacturing techniques used to make silicon circuits. This means that one set of industrial processes is used to make CPUs, DRAM, SSDs, and most of the other chips that computers are made of, whereas another set of processes is used to make optical signaling chips. The ability to build optical connectivity directly into standard silicon chips could be



(a)

(b)

Figure 5.1: Illustration of (a) server-based data center organization and (b) disaggregated hardware.

a breakthrough that spurs hardware disaggregation.

One of the open questions facing the hardware disaggregation movement is its choice of programming model. The problem is that we currently have many applications that are optimized to run on collections of servers. At several levels of the stack, a great deal of effort has been invested in making these applications run well on today's data center hardware. When ported to a disaggregated environment an application's performance may suffer simply because it was optimized for a different hardware target. Serverless computing removes servers from the programming abstraction; as a result, serverless applications are more likely to run well on disaggregated hardware because they are unlikely to be optimized for today's server-based data center. Serverless computing thus may facilitate the use of disaggregated resources in the data center, but as we will see, a need for server hardware exists independent of the programming model.

5.2.2 Data Center Networks

Data center networks have experienced tremendous advances in recent years. In describing its Jupiter network technology, Google claims a 100× increase in bandwidth over 10 years, an advance achieved using new network designs relying on commodity components, centralized control protocols, and scalable multi-level topologies [370]. Some aspects of this approach were anticipated by the earlier academic work of Al-Fares et al. [144]. Facebook also describes the evolution of its data center networks and the workloads they serve [145, 207, 340]. This involved moving away from hierarchical topologies limited by the size and performance of high-radix switches. The "fabric" architecture that replaced them uses smaller switches with a lower cost per port. It also employs a multi-level but non-hierarchical interconnect topology. Figure 5.2 illustrates this evolution. Like Google, Facebook also adopted a centralized, or top-down, approach to network management.

Despite these advances, data center networks still face fundamental limitations and trade-offs: increasing network scale leads to increasing latency as well as increasing per-node costs. While the speed of light poses an insignificant limitation in today's large-scale data center networks, its role is non-trivial for the fastest technologies, and one can expect it to be an increasing consideration in the future. For example, low-latency networking technologies such as InfiniBand can achieve round trip latencies of 2 μs [158], whereas light needs roughly 1 μs to make a 100 m round trip in a glass fiber. Google has suggested that such technologies will become mainstream in future data centers [61]. Scale and increasing physical separation increase latency simply because signals must travel further.

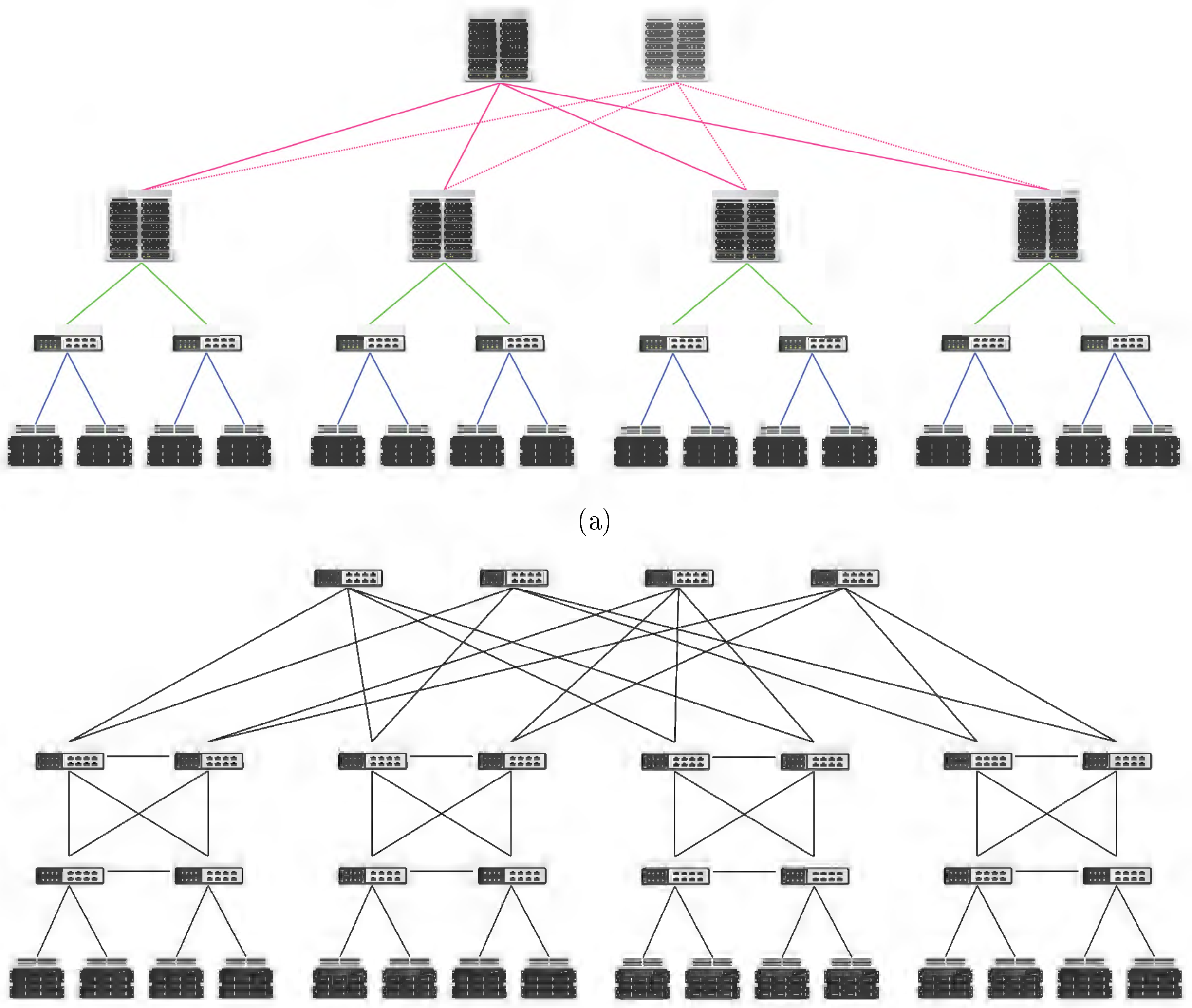
Data center networks that support more devices also require an increasing number of levels. Each additional level introduces a network hop that can add switching and buffering delays. We explore the trade-off between cost, latency, and scale in Section 5.2.3.

5.2.3 Interconnection Networks

There are scaling rules and fundamental tradeoffs that apply to all network interconnects. They apply inside a commodity server, where they govern the design of connections between CPUs and between CPUs and memory. They also apply between servers, to connections in the data center, and to HPC "supercomputers" that have cluster interconnects.

As networks grow, their cost and complexity increase more quickly than the number of endpoints nodes does [191]. In a fully-connected network, where each node has a direct connection to every other node, the total number of links is $N(N-1)$. Such networks can make sense for a small number of nodes, but the wiring quickly becomes intractable with increasing scale. A practical alternative is a crossbar switch, which requires only $O(N)$ wires, though the amount of circuitry still scales as $O(N^2)$.

Crossbar switches are attractive because, like fully connected networks, the delay they introduce is constant (independent of the number of nodes) or close to it. However, quadratic cost scaling eventually limits the number of nodes that can be connected with such low latency. Multistage interconnects combine several crossbar switches to link more nodes than a single switch can. The result is that the number of switches, the number of wires, and cost all scale as $O(N\log(N))$. Figure 5.3 illustrates the difference between single-stage and multistage networks schematically. Multistage interconnects can achieve throughput similar to that of a crossbar switch, but latency increases in proportion to the number of stages. In addition, buffering is usually used to ensure high throughput, and this also introduces delays.



(a)

(b)

Figure 5.2: (a) Hierarchical networks topologies use larger and higher-capacity switches at each level. Even though there are multiple paths at uppermost "core" level, only one path is active at any time while the other provides redundancy in case of failure. Link capacity increases at the higher levels, but significant oversubscription is common. (b) The fat-tree or Clos network architecture uses the same size of switch and link speed at each level and load balances traffic over all paths. Adapted from [144].

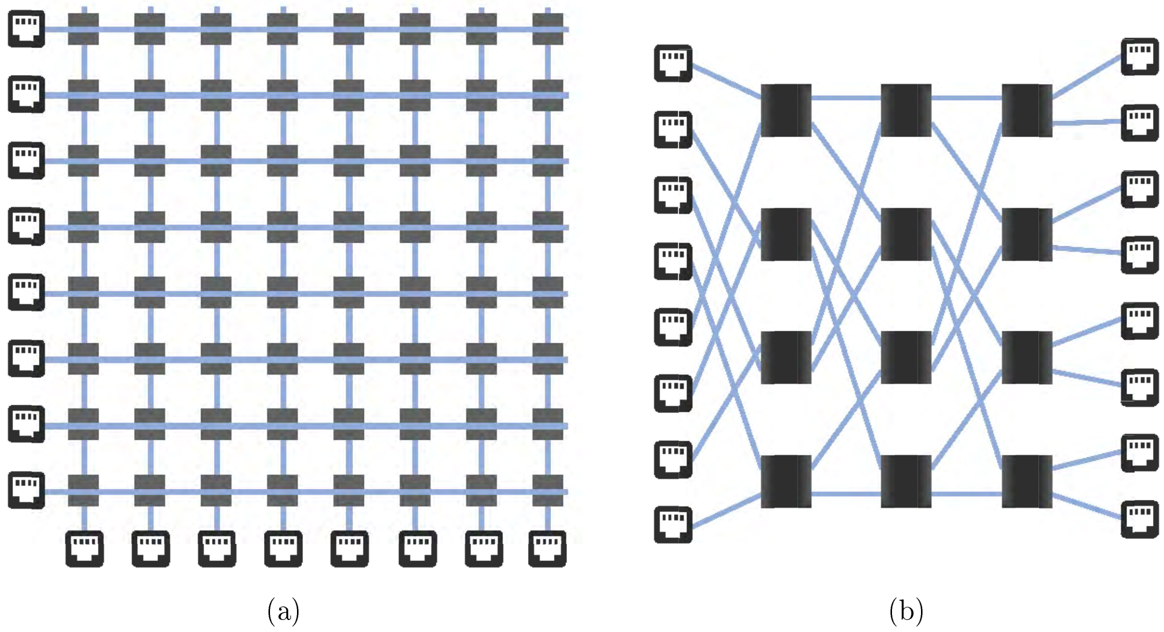


Figure 5.3: Single-stage vs. Multistage Interconnect Topology. (a) Single-stage crossbar switch. (b) A example multistage interconnect, the Omega network [247].

Table 5.1: Interconnect scaling

	Cost	Latency	Throughput
Single Stage (Crossbar or Fully-Connected)	$O(N^2)$	$O(1)$	$O(1)$
Multistage	$O(N\log(N))$	$O(\log(N))$	$O(1)$

There are many types of multistage interconnects with differing properties, most notably with regard to blocking. In a non-blocking switch, traffic between a pair of nodes is never impacted by traffic between other nodes, but in a blocking switch there can be interference [191]. Crossbar switches also have various properties depending on their construction. For example, if the fabrication technology allows only one layer of wires, so wires cannot cross, latency increases as the switch grows larger.

For our purposes, the important distinction is between the two classes of interconnects, rather than the differences within each class. As shown in Table 5.1, single-stage switches have more desirable latency scaling, whereas multistage switches have more desirable cost scaling.

5.2.4 Making the Most of Density

The COST metric [270] measures how large a distributed system needs to be to outperform an implementation that uses a single thread. Sometimes this gives astonishing results. For example, some systems require hundreds of cores to outperform a single thread. Others have unbounded COST, meaning that regardless of size, these distributed systems never outperform a single thread. Our work aligns with these findings, though we compare distributed systems to large servers rather than to single threads.

Researchers have also studied how to implement databases so they use large servers effectively [56, 454]. This work highlights the significant engineering challenges to scaling concurrency control algorithms to make good use of machines with more than 1,000 cores, but also indicates that it is generally possible to do so. This result suggests that databases, along with their established consistency guarantees, can be made to scale to ever-larger servers.

5.2.5 Cloud Server Selection

Cloud providers offer a large selection of virtual server types, which can turn choosing the optimal ones into a bewildering challenge for people or an expensive optimization task for automated tools. A number of research systems, such as PARIS [445], Ernest [418], and CherryPick [17], have tackled this challenge. They focus on analytic workloads and combine experimentation with predictive models to avoid brute-force exploration. Google Autopilot [342], a commercial technology, is used by the company internally to adjust provisioned compute and memory resources. Its aim is to reduce costs while maintaining application performance.

Public cloud providers also offer a number of tools to help customers better select instances. These include Google Cloud Recommender [333], Azure Cloud Cost Management [47], and AWS Compute Optimizer [45]. Third-party products for cloud cost management are also available [104, 105]. Such tools can identify patterns across many cloud customers and incorporate them into provisioning recommendations.

Clearly, instance sizing is important for traditional server-based cloud applications. It is also relevant to serverless computing since an abstraction that hides servers must assume responsibility for selecting them.

There are, however, many open questions regarding instance selection for serverless computing. For example, do serverless platforms benefit from the availability of many different instance types, or could a data center designed for serverless make do with fewer types—maybe just one or two? Also, could serverless computing reduce the need for the most expensive and powerful instances, and instead distribute the workload across many smaller instances? The rest of this chapter speaks to these questions.

5.3 Measuring Interconnects

Before building theoretical models of workload performance, we wanted to gain a quantitative understanding of the performance in contemporary data centers. To do so, we conducted a few basic experiments to measure the message latency between the processors of a single server and between servers joined by various network types.

5.3.1 Intra-Server Interconnects

To get a sense for the diversity of performance characteristics, we tested a variety of architectures, including Intel, AMD, and ARM. Our test programs runs two processes, each of which we pin to a specific processor. One uses an atomic compare-and-swap instruction to toggle a 64-bit integer from 0 and 1 while the other toggles it from 1 to 0. The actions of these processes are alternating and coordinated, and the total number of value changes in a period of time gives the one-way communication latency between the two cores. Figures 5.4(a)-(c) show measurements for three processor types. In each case, we show the latency between core 0 and every other core in the system, plotting a bar for each that represents the minimum of five measurements.

In Figure 5.4(a), we show the performance of a server with 2.5 GHz Intel Xeon Platinum 8259CL CPUs. This is a two-socket design, and a division between two levels of latency presumably represents the difference in latency between communicating between cores in different sockets and cores within the same socket; it appears to be a two-stage interconnect. A finer periodic structure is also visible that seems to reflect some other hierarchy of connectivity within the socket. These cores are actually hardware threads [133, 411], meaning that pairs of logical cores share many of the same underlying microarchitectural resources. In this case, we believe that logical cores 0 and 48 are actually the same physical core. Our measured latencies are 3.5 ns between hardware threads on the same core, 31 ns between separate cores on the same socket, and 72 ns between separate sockets in the same server.

In Figure 5.4(b), the AMD EPYC 7571 that we tested runs at 2.55 GHz and has 24 cores, each of which has two hardware threads. Like the Intel chip, it implements the x86 instruction set, but it displays different latency characteristics. We identify four levels of latency here: a same-core latency of 10.4 ns and three cross-core latency tiers: 6 processors (3 cores) at 15.5 ns, 32 processors (16 cores) at 168 ns, and 24 processors (12 cores) at 186 ns. While we are unsure of the details, we conclude that groups of 8 processors (4 cores) are closely coupled and the server interconnect has three stages in all. Interestingly, the farther latency tiers do not appear to correspond to socket boundaries.

The AWS Graviton2 processor (Figure 5.4(c)) runs at a 2.5 GHz clock rate, which is similar to the speed of the other systems. It does not show any evidence of hardware threads, and the one-way latency appears to be quite uniform at an average of 26.5 ns, meaning that this is a single-stage interconnect. This processor provides a third point of reference, and represents both a different architecture and a different instruction set.

Table 5.2: Latency of MPI communication corresponding to Figure 5.4

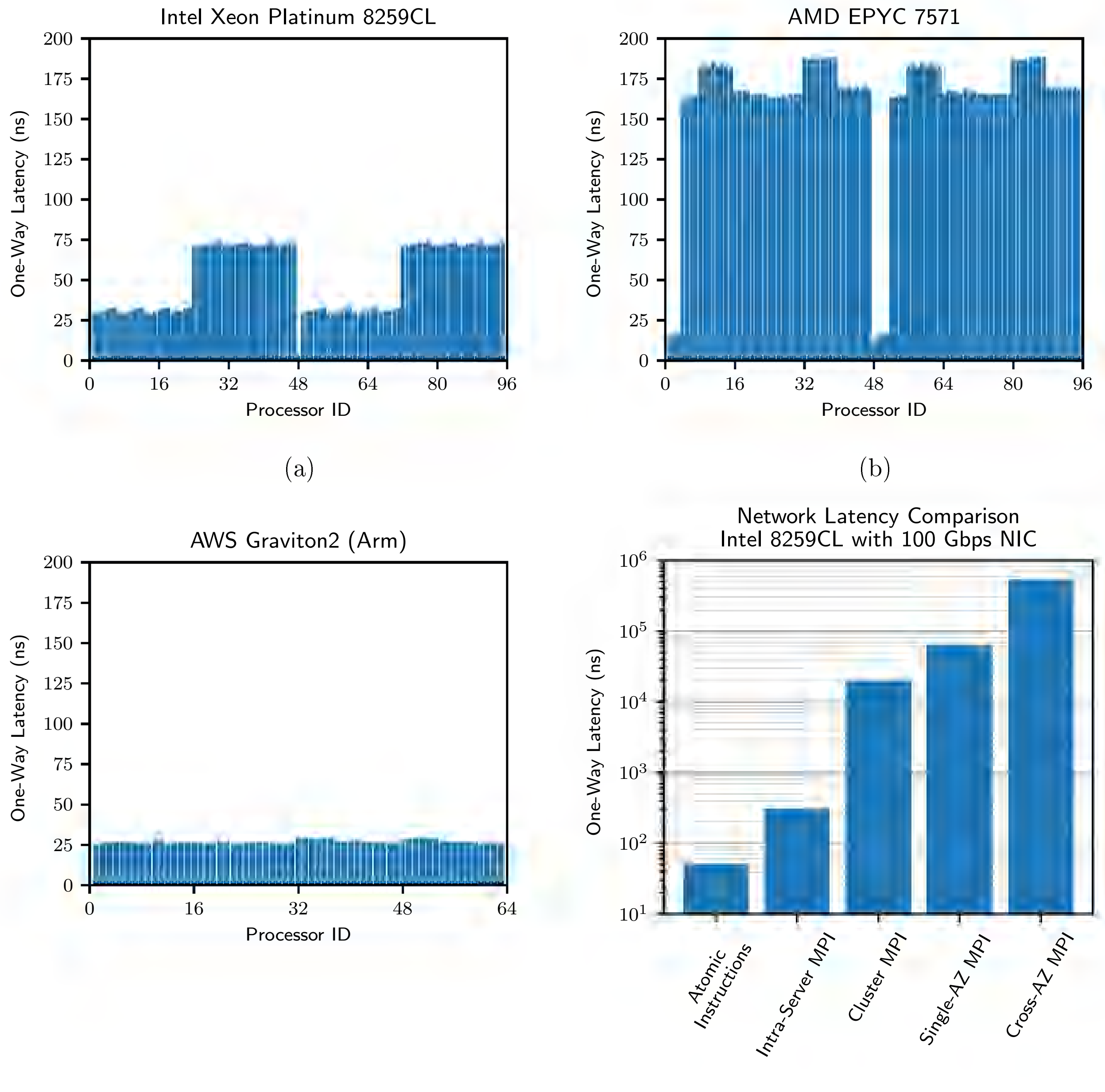
	One-way Latency (ns)
Atomic Instructions	51
Intra-Server MPI	308
Cluster MPI	19,817
Single-AZ MPI	63,914
Cross-AZ MPI	530,190

We are not privy to the design considerations and trade-offs used to create each of these chips, but they lead to interesting consequences. The Intel chip has lower same-core latency than AMD, 3.5 ns vs. 10.4 ns, but the AMD can connect separate cores with lower latency than either the Intel chip or the ARM-based AWS chip: 15.5 ns vs. 31 ns or 25 ns, respectively. However, it takes longer to access the more remote cores in the AMD system, between 168 ns and 186 ns vs. 72 ns for Intel. More important than the specific numbers is the overall principle: there are different ways to link up the cores inside a server, and this creates meaningful differences in performance characteristics. When message latency is what matters, then for programs using 8 or fewer processes AMD offers the tightest integration, and likely the best performance. AWS on ARM is best between 9 and 64 processes, and the advantage goes to Intel between 65 and 96 processes.

5.3.2 Inter-server Interconnects

Our next set of experiments looks at the latency of communication between servers. We adapted a simple MPI [172] “ping-pong” demo program that passes a counter back and forth between two processes, incrementing it on each handoff [225]. We tested four different scenarios: communication between two processes on the same server; communication between two servers linked by the AWS Elastic Fabric Adapter (EFA) [137], a high-performance “cluster” interconnect; and communication between servers linked by the ordinary AWS data center network, both within the same AZ and across AZs. We used AWS `m5n.24xlarge` instances, which are equipped with 100 Gbps NICs and the 2.5 GHz Intel Plantinum 8259CL processors studied in Section 5.3.1. We ran each experiment 10 times, and we report the minimum time achieved.

The results of these experiments are shown in Figure 5.4 (d) and Table 5.2. For MPI communication within a server, the latency is 308 ns, roughly 6× the average inter-core latency that we reported for this server in Section 5.3.1. MPI is a more general mechanism, and we believe that this generality is reflected in the additional ∼250 ns delay. Within a data center, latency is about 20 μs using EFA and about 64 μs using the standard in-AZ network. Across availability zones within the region, the latency is significantly higher, coming in at about 530 μs.



(a) (b)

(c) (d)

Figure 5.4: One-way message latency between logical processors within a server for (a) 2.5 GHz Intel Platinum 8259CL (AWS `m5n.24xlarge`), (b) 2.55 GHz AMD EPYC 7571 (AWS `m5a.24xlarge`, and (c) 2.5 GHz AWS Graviton2 (ARM architecture, AWS `m6g`). (d) shows the latency of an MPI operation between pairs of processes linked by different sorts of interconnects, and for comparison includes the mean of (a) labeled as "Atomic Instructions."

In these experiments, MPI communication is 64× faster within a server than is possible between servers, even in the best of circumstances. We know of some technologies that offer somewhat faster communication between servers than what we measured here, however we did not experiment with them, and we know that they too run into limitations. RDMA technologies provide one-way latencies at or even below 1 μs [158], and the speed of light in fiber optics puts a limit of about 490 ns on how quickly a signal can travel 100 m, which is roughly the size of a data center. Since electronic delays also apply at both ends, we estimate that this latency will always be at least 10× as great as the delays of messaging between cores on a modern server such as the ARM-based AWS Graviton, and it is possible that it could remain much higher, as it is today.

These experiments show just how much slower communication is between servers than within servers. We have focused this analysis on latency rather than bandwidth because bandwidth is not subject to the cost-performance trade-offs discussed in Section 5.2.3.

5.4 Amdahl's Law and Communication

In this section, we apply classic scaling rules for multiprocessing to modern distributed computing. We can think of sending and receiving messages as processor operations like any others, different only because they may incur substantial latency. Our approach puts communication on equal footing with computation and extends Amdahl's law to workloads where waiting on communication is the bottleneck. Algorithms requiring coordination are prime examples of such workloads, as we explain in Section 5.5. We begin by presenting the model, then explore its implications.

5.4.1 Background

Amdahl's Law is classic advice for those who design parallel algorithms and the machines that run them. It follows from Gene Amdahl's observation [28] that even when much of a workload can be divided among multiple processors and processed in parallel, there is usually some portion that must be performed sequentially. We denote these portions as W_p and W_s, respectively, and express the total amount of work as $W = W_s + W_p$. The completion time T for a task can then written as the sum of T_s, the time needed for the sequential portion, and T_p, the time needed for the parallel portion, where we assume that these portions are non-overlapping and do not run concurrently.

If M processes are available, the completion time becomes

$$T = T_s + T_p = \frac{W_s}{R_s} + \frac{W_p}{MR_p} \tag{5.1}$$

where the sequential and parallel components are processed at rates R_s and R_p. For simplicity, from here forward, we assume that serial and parallel portions are processed at the same rate, and let $R = R_s = R_p$.

We define as f the fraction of the workload that is parallelizable,

$$f = \frac{W_p}{W_s + W_p} \tag{5.2}$$

We can now express the parallel speedup with m processes as

$$s(m) = \frac{T|_{M=1}}{T|_{M=m}} = \frac{1}{(1-f) + \frac{f}{m}} \tag{5.3}$$

Note that the speedup asymptotically approaches a limiting value as the number of processors increases. The maximum possible speedup is

$$\lim_{m\to\infty} s(m) = \frac{1}{1-f} \tag{5.4}$$

If the workload represents interactive processing—say, all the requests that arrive at an API during some fixed interval—then there is also a *capacity limit.* We can compute this capacity limit as

$$\lim_{m\to\infty} \frac{W}{T|_{M=m}} = \frac{W_s + W_p}{\frac{W_s}{R}} = \frac{R}{1-f} \tag{5.5}$$

In other words, if a portion $1-f$ of the workload is necessarily sequential, then it will be impossible to keep up with a workload arrival rate greater than $\frac{R}{1-f}$, no matter how many processesors are provided.

Amdahl's law has provoked substantial controversy over the years, even though this was not Amdahl's original intent [27]. The most important reinterpretation was given by Gustafson [179], who observed that the number of processors desired and the amount of parallelizable work can be correlated. In the context of HPC and scientific computing, it is often increasing problem scale that drives the need for more processors, but increasing problem scale often results in more parallelizable work but not more sequential work.

Let W_p^0 be the amount of parallelizable work provided at the one-process scale. Gustafson observes that for M processes, the amount of parallelizable work is often $W_p = MW_p^0$, which suggests the scaling

$$s'(m) = \frac{T|_{M=1}^{W_p=MW_p^0}}{T|_{M=m}^{W_p=MW_p^0}} = (1-f) + fM \tag{5.6}$$

Under these assumptions, the maximum speedup is limited only by the problem size.

We note that both Amdahl and Gustafson adopt simplified perspectives and ignore many details. Shi shows that the two are not in conflict [367] and suggests how to determine which formulation is most appropriate for a given problem.

While Amdahl's law has, in the past, been of interest principally to the HPC community, more recent work by Hill and Marty has explored how it applies to multi-core processor designs [196]. Given any resource budget, say chip area or power dissipation, designers can choose to build a smaller number of cores, each having faster sequential performance, or a larger number of cores, each having slower sequential performance but having greater parallel performance in aggregate. Hill and Marty show that in some cases it may be best to build asymmetric processors that have a mix of slower and faster cores.

5.4.2 Incorporating Communication

We now propose a use of Amdahl's Law for workloads that are rich in communication, like modern cloud workloads. For concreteness, consider W to be a batch of input, perhaps a collection of transactions, subject to some consistency guarantees. Moreover, consider the case where the specification or implementation of these guarantees imposes dependencies between their processing, e.g., those originating from a dataflow graph representation of the computation. We treat the communication and computation along the longest path through this graph, its critical path, as the sequential portion of the workload, and the rest as the parallelizable portion. We show in Section 5.6 how such graphs can be constructed.

We now split the overall time to process W into four components, which we assume to be non-overlapping:

$$T = T_s^\Sigma + T_p^\Sigma + T_s^\Phi + T_p^\Phi \tag{5.7}$$

Here T_s^Σ and T_p^Σ are the compute time spent on the sequential and parallelizable components of the workload, while T_s^Φ and T_p^Φ are the corresponding times spent waiting on communication.

We let M_s and M_p represent the number of messages waited for in the sequential and parallelizable portions of the workload, respectively, and we let $\bar{\tau}$ be the mean message latency. Then we can write

$$T = \frac{W_s}{R_s} + \frac{W_p}{mR_p} + M_s\bar{\tau}_s + \frac{M_p\bar{\tau}_p}{m} \tag{5.8}$$

The message latency $\bar{\tau}$ depends on the underlying transport mechanism. For messages sent from a process to itself, we take this latency to be zero. Sending messages from one process to another incurs nonzero latency. While there is a hierarchy of connectivity both within a server and among servers in a data center (see measurements in Section 5.3), to keep this model simple, we use just two latencies: one for intra-server messaging and another for inter-server messaging.

$$\bar{\tau} = \begin{cases} 0 & \text{within a process} \\ \tau_s & \text{for intra-server messaging} \\ \tau_x & \text{for inter-server messaging} \end{cases} \tag{5.9}$$

We also introduce parameters that describe the rate of messaging associated with both the sequential and parallelizable portions, letting $\rho_s = \frac{M_s R}{W_S}$ and $\rho_p = \frac{M_p R}{W_p}$.

We can now express the speedup as

$$s(m) = \frac{1}{(1-f) + \frac{f}{m} + (1-f)\rho_s\bar{\tau} + \frac{f\rho_p\bar{\tau}}{m}} \tag{5.10}$$

We estimate $\bar{\tau}$ assuming that messages are distributed uniformly across all source and destination processors, and let

$$\bar{\tau} = \begin{cases} 0 & \text{if } m = 1 \\ \frac{m-1}{m}\tau_s & \text{if } 1 < m \leq N \\ \frac{N-1}{m}\tau_s + \frac{m-N}{m}\tau_x & \text{if } m > N \end{cases} \tag{5.11}$$

In the sections that follow, we develop a cost model to accompany the performance model of Equation 5.10 and use it to compare between using large servers and using many small servers. These choices correspond to, respectively, using powerful but expensive servers and using disaggregated hardware.

5.4.3 Cost Model

We assume that a certain high-volume commodity server has the lowest cost per processor. We denote the number of processes supported by this "basic" server as N_0 and the cost of this processor as C_0. Up to some level of scale, manufacturing and packaging efficiencies outweigh quadratic growth in integration costs, so any number of processors $m \leq N_0$ are most cost-effectively provisioned as a slice of the basic server.

Let C_N be the cost of a server supporting N processes. We assume that some $(1-\beta)C_0$ of the basic server cost is the cost of the interconnect, whereas a part βC_0 is the cost of everything else (cores, memory, power supply, etc.). We assume that the interconnect cost scales quadratically (see Section 5.2.3 and Table 5.1). Then, for $N > N_0$, we may write the cost of the server as

$$C_N = \left(\beta\frac{N}{N_0} + (1-\beta)\left(\frac{N}{N_0}\right)^2\right)C_0 \tag{5.12}$$

We then write the cost of m processes provided using servers of size N as

$$C(m,N) = \begin{cases} \frac{mC_0}{N_0} & \text{if } N \leq N_0 \\ \frac{mC_0}{N_0}(\beta + (1-\beta)\frac{N}{N_0}) & \text{if } N > N_0 \end{cases} \tag{5.13}$$

Here we assume serverless pay-as-you-go to allow purchasing fractional server resources, but we also assume that these resources are provisioned for the entire duration T required to execute the workload. This assumption is appropriate when processor idle periods are similar to the scheduling quantum or less than it, so the overheads of resource multiplexing outweigh its benefits. This is the case when sequential and parallelizable portions of the workload are finely interleaved and when communication latencies are relatively small. We also assume that the total capacity of the data center is much greater than that allocated to the workload, so the cost per node of the data center network does not depend on the number of servers used.

5.4.4 Large Servers vs. Disaggregation

In each of Figure 5.5 through Figure 5.9, we plot (a) speedup as a function of the number of processes; (b) the relative cost and relative capacity for the full range of processes in (a); and (c) a "zoomed" view of the cost-capacity relationship that focuses on the trade-off in the range of processors near the size of the largest server considered (1024 cores). We also show how time spent is divided between sequential computation (Σ_s), parallelizable computation (Σ_p), sequential communication (Φ_s), and parallelizable communication (Φ_p), for servers having processor counts of (d) 1,024, (e) 64, and (f) 8. We fix the workload parameter $f = 0.9999$ and vary ρ_s, ρ_p, as well N, as the number of cores in a server. We inserted parameters based on commercially available technology, choosing $N_0 = 64$, representing the AWS Graviton2 processor. We set $R = 1$, $\tau_s = 50$, and $\tau_x = 4,000$, using a dimensionless time unit that corresponds roughly to CPU clock cycles.

There are two situations in which large servers are preferable to disaggregated collections of smaller servers. In some cases, they achieve performance that is not possible with basic servers, regardless of how many of them there are, as seen in Figure 5.8 and Figure 5.9. In Figure 5.7, large servers are cheaper for certain ranges of capacity, whereas the smaller servers are cheaper in other ranges. In other cases, such as those of Figure 5.6, small servers are cheaper across a broad range of capacity levels, but large servers still offer higher maximum capacity. In models with any communication, i.e., $\rho_s \neq 0$ or $\rho_p \neq 0$, there is always a capacity level that is achievable only with a single large server—the collection of smaller servers has a maximum capacity

$$\lim_{m\to\infty} \frac{W}{T|_{M=m}} = \frac{R}{(1-f)(1+\rho_s\tau_x)} \tag{5.14}$$

whereas for large servers it is

$$\lim_{m\to\infty} \frac{W}{T|_{M=m,N=m}} = \frac{R}{(1-f)(1+\rho_s\tau_s)} \tag{5.15}$$

Which is larger since $\tau_x > \tau_s$. We thus ask, what are the workload parameters under which it can make sense to use a disaggregated system, splitting the work across many smaller servers to achieve lower cost than we might with one or a few large servers? To find these configurations, we look for those points where the cost-capacity curve of a single server intersects with that of a distributed system built from the basic server. To do this, we fix a cost budget C and express it in terms of a scaling parameter α, writing

$$C = \alpha^2 C_0 \tag{5.16}$$

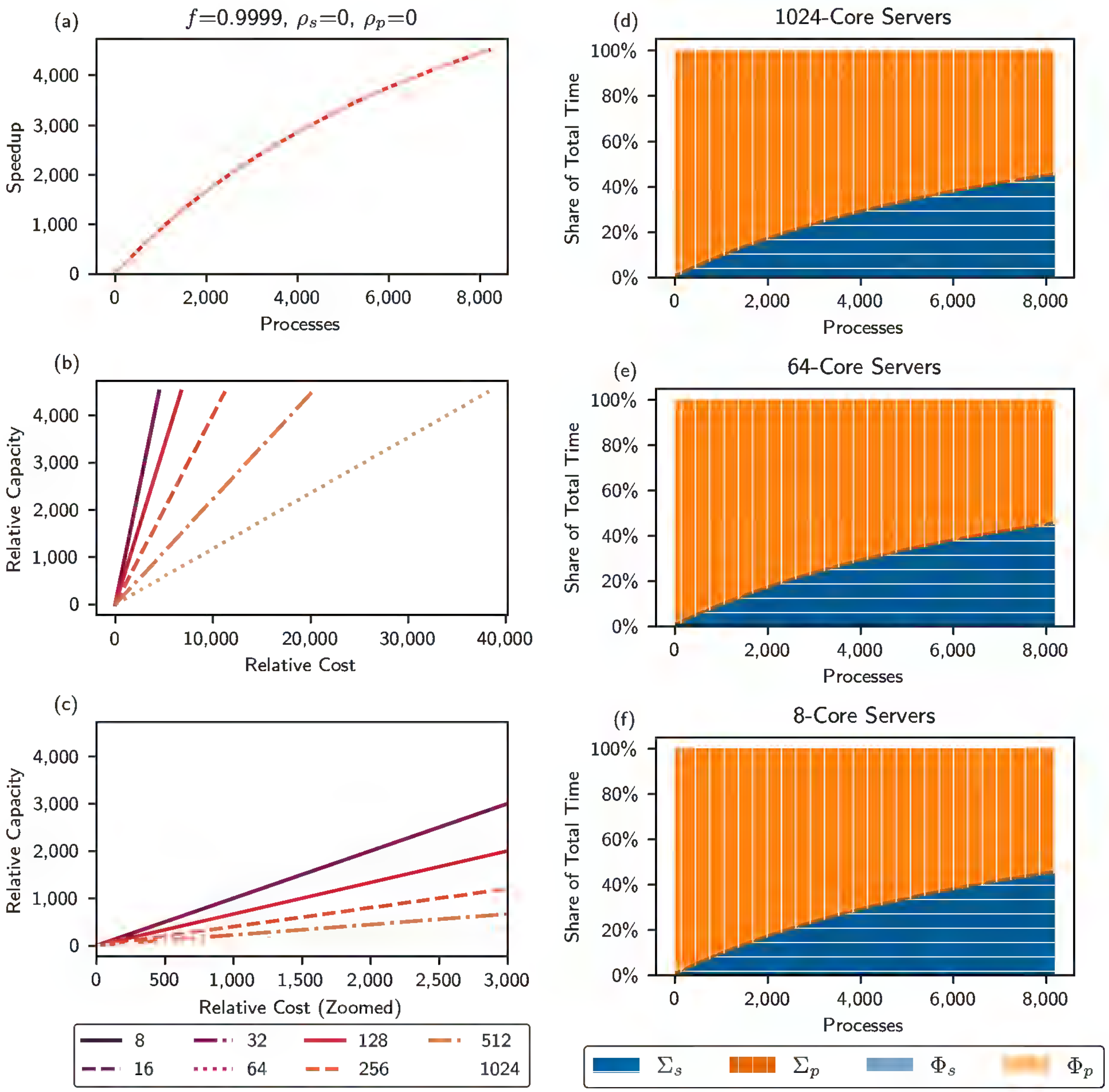


Figure 5.5: $f = 0.9999$, $\rho_s = 0$, $\rho_p = 0$. In this case there is no communication and so our model reduces to Amdahl's law—the speedup saturates as it approaches $1/f$. Using a larger server increases cost, but does not alter performance.

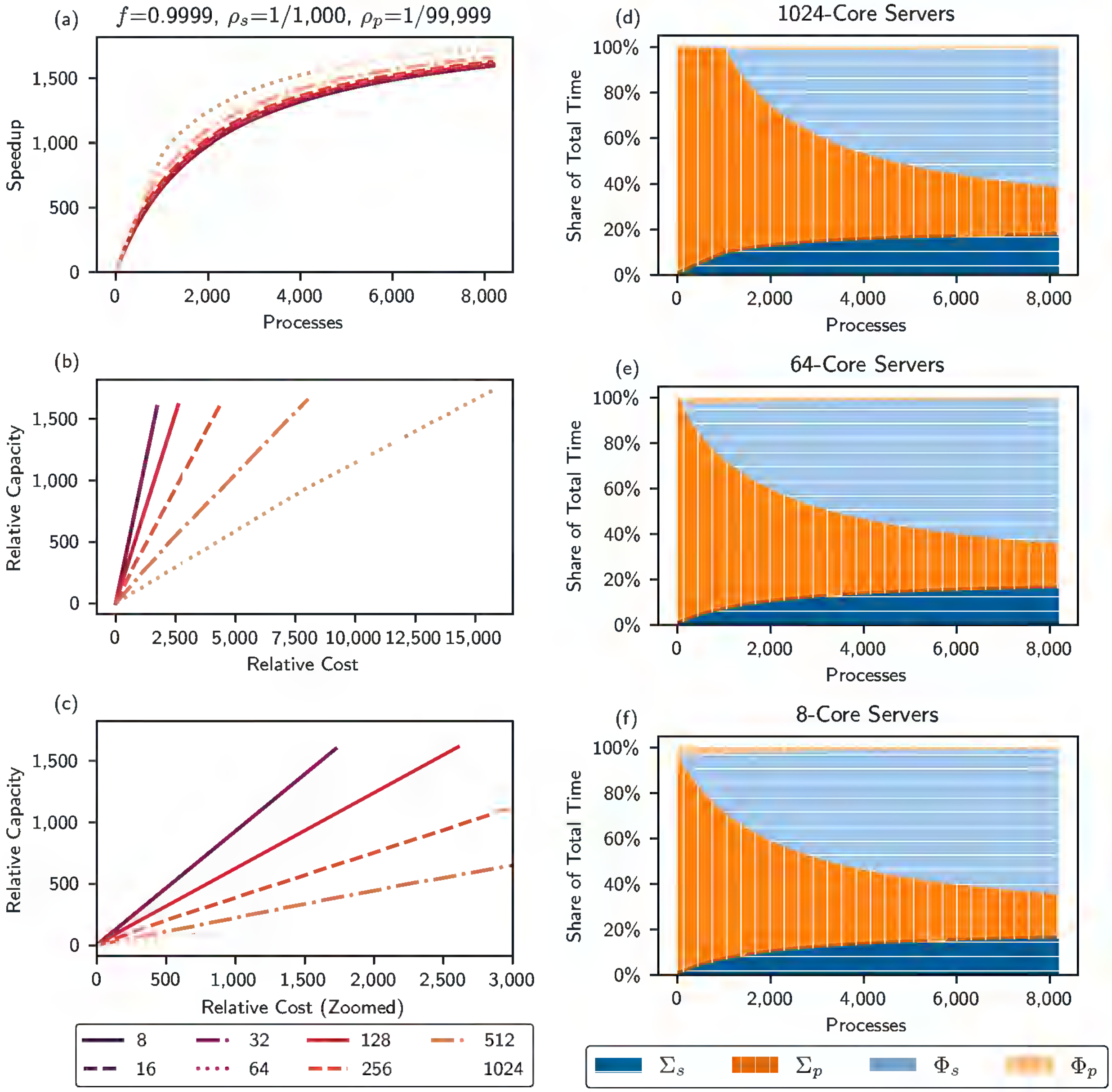


Figure 5.6: $f = 0.9999$, $\rho_s = 1/1,000$, $\rho_p = 1/100,000$. At high process counts sequential communication accounts for the greatest fraction of time spent. For any given number of processors, putting them in a larger server boosts performance, but the same performance can usually be achieved by using a larger number of smaller servers, and when possible this results in lower cost.

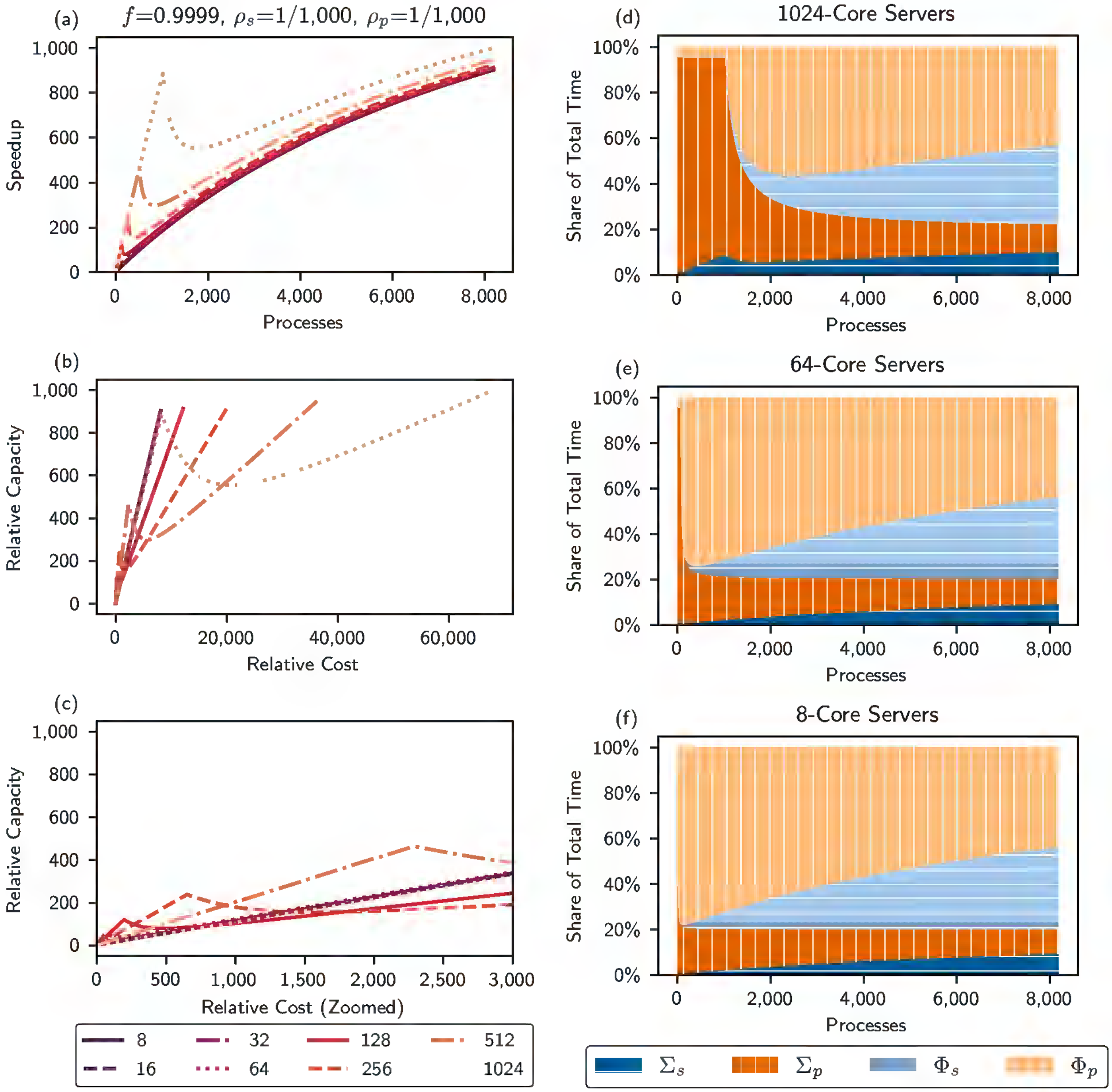


Figure 5.7: $f = 0.9999$, $\rho_s = 1/1,000$, $\rho_p = 1/1,000$. A large server can handle this workload with significantly fewer processor than smaller servers can, however the greater per-process cost of larger servers can lead to a more expensive solution. In this case the 128- and 256-core severs can be the cheapest way to meet certain capacity needs, but the 1024-core server always costs more.

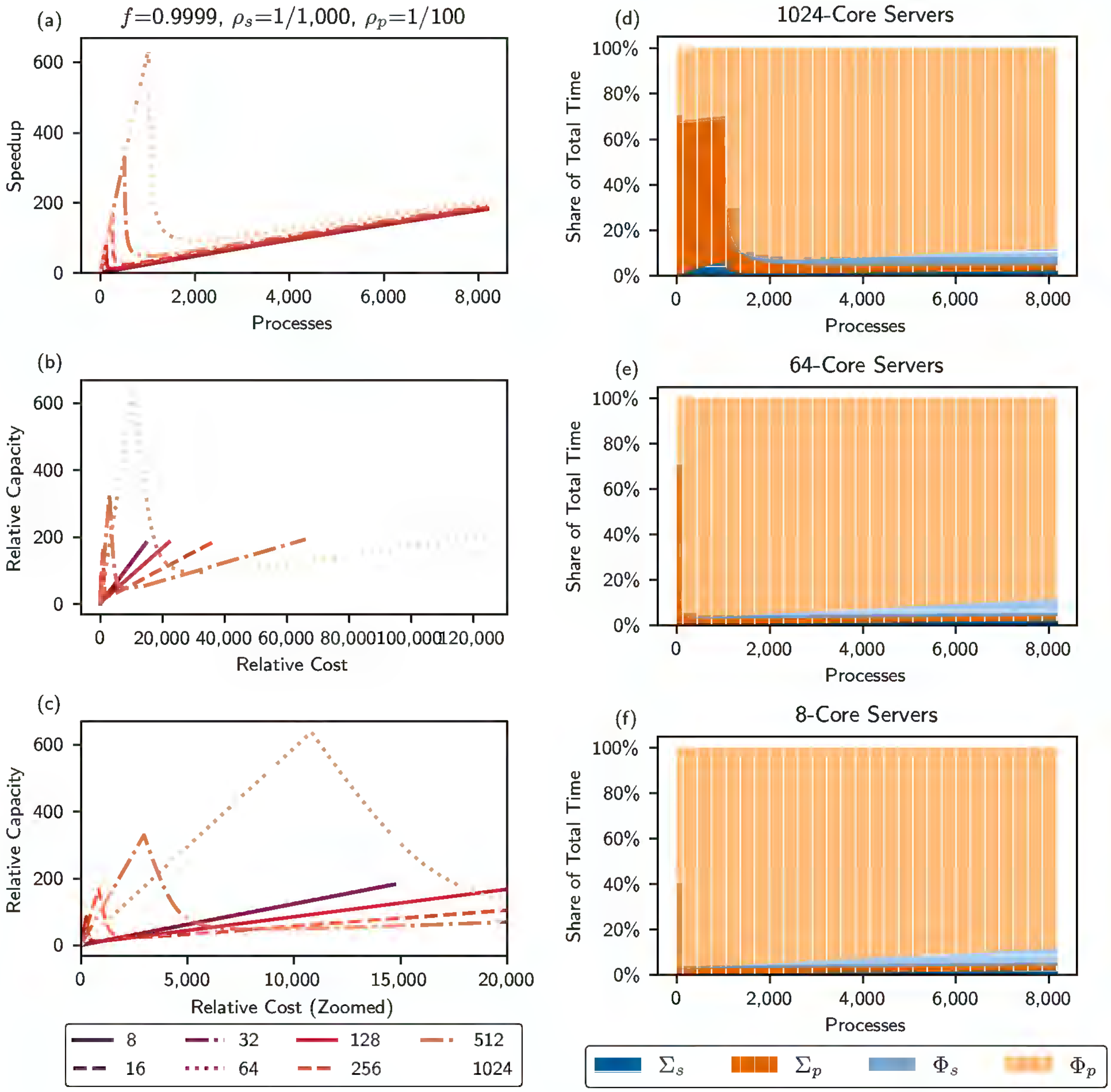


Figure 5.8: $f = 0.9999$, $\rho_s = 1/1,000$, $\rho_p = 1/100$. An increasing amount of parallelizable communication relative to Figure 5.6 creates additional benefits for larger servers. In this case, 128-core and larger servers offer the lowest-cost way to meet certain capacity demands, but they all cease to be cost effective once the workload scales beyond one server.

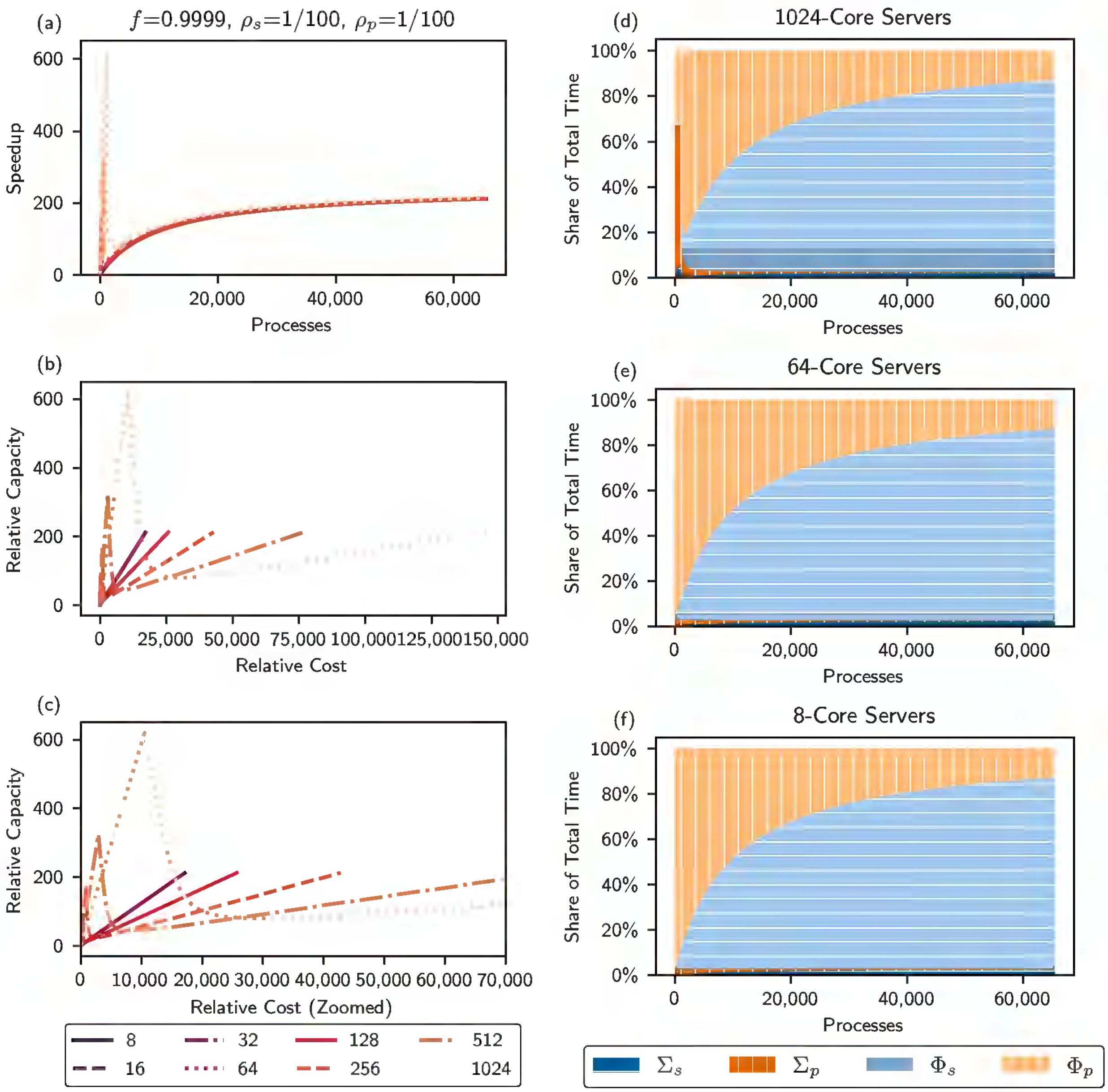


Figure 5.9: $f = 0.9999$, $\rho_s = 1/100$, $\rho_p = 1/100$. In this case sequential communication dominates and capacity is determined by the size of the server, as in Figure 5.7. Here using more than one server again offers no benefit to either cost or performance. The optimal solution is choose the server size so that a single server provides the needed capacity.

By Equation 5.13, if we use basic servers to build the system, this cost budget constrains the number of processes as

$$C(m_{dis}, N_0) = \frac{m_{dis}C_0}{N_0} \tag{5.17}$$

Which gives the number of processes as

$$m_{dis} = N_0\alpha^2 \tag{5.18}$$

If, instead, we put the budget toward a single large server, then Equation 5.13 gives

$$C(m_{big}, m_{big}) = m_{big}\frac{C_0}{N_0}\left(\beta + (1-\beta)\frac{m_{big}}{N_0}\right) \tag{5.19}$$

And solving for m_{big} yields:

$$m_{big} = \frac{2\alpha^2}{\beta + \sqrt{\beta^2 + 4(1-\beta)\alpha^2}}N_0 \tag{5.20}$$

We now set $\beta = 0$ to allow a simpler analytical derivation, yielding

$$m_{big} = \alpha N_0 \tag{5.21}$$

Later, we show that this makes little difference to the overall interpretation of the results (see Figure 5.10).

The collection of small servers gives a greater speedup than a single large server when $T_{big} > T_{dis}$, i.e., when

$$(1-f) + \frac{f}{m_{big}} + (1-f)\bar{\tau}_{big}\rho_s + f\frac{\bar{\tau}_{big}\rho_p}{m_{big}} > (1-f) + \frac{f}{m_{dis}} + (1-f)\bar{\tau}_{dis}\rho_s + f\frac{\bar{\tau}_{dis}\rho_p}{m_{dis}} \tag{5.22}$$

We approximate $\bar{\tau}_{big}$ and $\bar{\tau}_{dis}$ from Equation 5.11 by assuming $m \gg N \gg 1$, so that $\bar{\tau}_{big} \approx \tau_s$ and $\bar{\tau}_{dis} \approx \tau_x$. We also assume that f is close to 1, i.e., the workload is highly parallelizable. We then substitute $m_{big} = \alpha N_0$ and $m_{dis} = \alpha^2 N_0$ (Equation 5.18 and Equation 5.21) to write

$$\frac{1}{\alpha N_0} + (1-f)\tau_s\rho_s + f\frac{\tau_s\rho_p}{\alpha N_0} > \frac{1}{\alpha^2 N_0} + (1-f)\tau_x\rho_s + \frac{\tau_x\rho_p}{\alpha^2 N_0} \tag{5.23}$$

This may be rearranged as

$$\left(N_0(\tau_x - \tau_s)(1-\rho_s)f\right)\alpha^2 - (\tau_x\rho_p + 1)\alpha + (1+\tau_x) < 0 \tag{5.24}$$

We require all variables to be real and positive, and assume $\tau_x > \tau_s$ and $\rho_s < 1$. Under these conditions, Equation 5.24 has a solution when

$$(\tau_s\rho_p + 1)^2 > 4N_0(\tau_x - \tau_s)(1 + \tau_x\rho_p)(1-f)\rho_s \tag{5.25}$$

Or, equivalently, when

$$(1-f)\rho_s < \frac{(1+\tau_s\rho_p)^2}{4N_0(\tau_x-\tau_s)(1+\tau_x\rho_p)} \tag{5.26}$$

Figure 5.10 plots this relationship in terms of the workload parameters ρ_p and $(1-f)\rho_s$. For those combinations below the threshold curve (in red), a single server is always preferred. For those above the threshold, a network of servers will be preferred for some capacity amounts. Depending on the workload, routine capacity needs could be greater than or less than the maximum capacity supported by the disaggregated system (see Equation 5.14).

We note that the line representing Equation 5.26, as shown in Figure 5.10, is confined to a horizontal band. In the limit of $\rho_p \to 0$, we obtain the relationship

$$(1-f)\rho_s < \frac{1}{4N_0(\tau_x-\tau_s)} \tag{5.27}$$

This is the disaggregation threshold for workloads that require little communication in their parallelizable portions.

The right-hand side of Equation 5.26 reaches its maximum when $\rho_p = (\tau_x - 2\tau_s)/(\tau_x\tau_s)$, so disaggregation is cost-effective at some capacity level whenever

$$(1-f)\rho_s < \frac{\tau_s}{N_0\tau_x^2} \tag{5.28}$$

The quantity $(1-f)\rho_s$ appearing on the left-hand side of Equation 5.26 represents the rate of sequential communication in the context of the overall workload. In Equation 5.27 and Equation 5.28, the dependence on ρ_p drops out, giving constant thresholds involving only the amount of sequential communication. It is thus a workload's necessarily sequential communication, which we can also think of as its critical-path communication, that determines whether it can run on a disaggregated system as opposed to a single large server.

We briefly return to the $\beta = 0$ assumption used to derive Equation 5.21 from Equation 5.20. This choice implies that server cost is driven primarily by interconnect costs, even at the scale of a basic server. Quadratic scaling of interconnect costs means that this assumption becomes true eventually, i.e., the cost of the largest servers is dominated by the cost of the interconnect, but this is probably not a reasonable assumption when N is near to N_0. Dropping the $\beta = 0$ condition leads to an alternative formulation of the relationship in Equation 5.26

$$\begin{gathered}\left((1+\tau_x\rho_p)(2-\beta)(\tau_x-\tau_s)(1-f)\rho_s N_0 - (1+\tau_s\rho_p)^2(1-\beta)\right)^2 > \\ \left((\tau_x-\tau_s)(1-f)\rho_s N_0\right)^2(1+\tau_x\rho_p)^2(1-\beta/2)^2 - (1+\tau_s\rho_p)^2\beta^2\end{gathered} \tag{5.29}$$

In Figure 5.10, we illustrate the difference between $\beta = 0$ (red line) and $\beta = 0.5$ (background shading). The curves are similar, though shifted slightly, which suggests that the essential features of the model are captured by the analysis for $\beta = 0$.

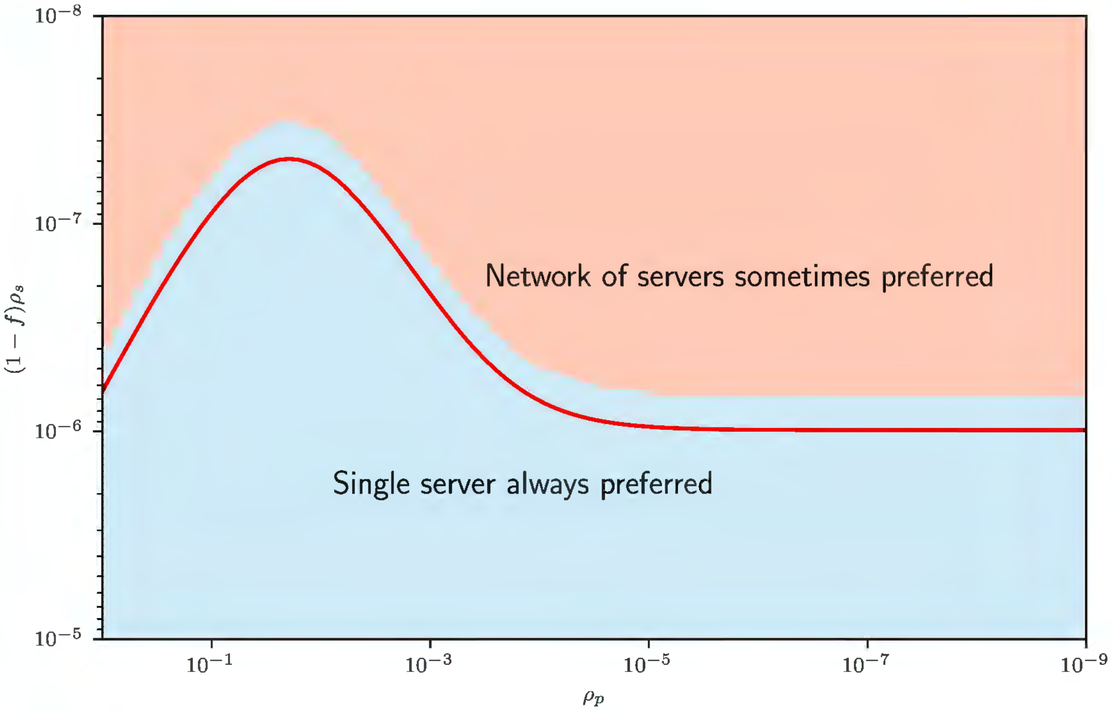


Figure 5.10: Under certain workload parameters, a single large server dominates a distributed system—any collection of smaller servers that meets the capacity requirement will cost more than a large server providing equal capacity. The red line indicates the $\beta = 0$ threshold as given by Equation 5.26. The background shading shows the numerical solution to Equation 5.29 for $\beta = 0.5$.

5.5 Consistency and Communication

Our formulation of Amdahl's law in Section 5.4 describes workloads using several parameters: f, the fraction of the work that is parallelizable; ρ_s, the rate communication for the sequential work; and ρ_p, the rate of communication for the parallelizable work. We observed that, for some workload parameters, a disaggregated system comprising many small servers is a good solution, whereas, for other workload parameters, such a collection of servers costs more than a single large server, even though the large server may have a much higher cost per processor. However, how can we determine the parameters that characterize any given workload?

Section 5.6, which follows, describes how to compute the parameters from workload traces. However, it is also possible to use intuition to understand its implications for a workload: Equations 5.26-5.29 define thresholds on $(1-f)\rho_s$, the rate of sequential communication in the workload. Equation 5.27, which disregards parallelizable communication (the case $\rho_p = 0$), gives a particularly simple formulation.

This fits neatly with the CALM theorem [187], which tells us that *coordination protocols*, those that require waiting on communication, have limited scalability. On the other hand, *coordination-free* implementations can scale. We refer the reader to Section 2.8.4.1 for additional context.

When an application engages in coordination, e.g., by using Paxos [242, 243], two-phase commit [170], bulk synchronous processing with barriers [415], or simply locks [188, 244], it may be putting communication on its critical path. This may not present a problem if done occasionally, but if the frequency of sequential communication exceeds a certain threshold, the workload becomes unsuitable for running on a distributed system.

Coordination protocols are generally used to provide strong consistency guarantees such as linearizability [192] or serializability [305]. Coordination-free protocols provide consistency guarantees such as causal consistency [52, 255, 260], bounded staleness [53], or eventual consistency [398]. Application designers need to be careful when requesting strong consistency, because putting too much of it on the critical path will produce an application that requires expensive large servers to scale. Opting for weak consistency frees them to use coordination-free protocols and guarantees scalability using small, low-cost servers.

5.6 Simulation Experiments

In this section, we describe how to derive model parameters from actual workloads. Using the YCSB [112] benchmark as an example, we begin by building a dataflow graph describing how state must flow between transactions in order to produce the expected result. Each edge in this graph represents potential communication, and in a system with many processors, it represents likely communication—unless the workload partitions cleanly, these edges will most often cross from one processor to another. The dataflow graph is directed and acyclic, so it has a well-defined topological order. This allows us to compute a critical path, which represents the necessarily sequential part of the workload.

The YCSB workload also allows us to compare strong consistency to weak consistency—we show that applications that enforce strong consistency are more likely to best be implemented on large servers, whereas applications that enforce weak consistency may be a better fit in a disaggregated environment. Specifically, we compare sequential consistency [241] with eventual consistency [398], varying the rate of convergence for eventual consistency to explore a continuum of consistency levels [53]. In our YCSB-based example, weak consistency produces shorter critical path lengths and a lower graph degree (ratio of edges to nodes). These factors contribute to shifting model parameters toward disaggregation and away from the parameter regime where large servers are universally preferred.

5.6.1 YCSB Benchmark

YCSB [112] was developed to support the benchmarking of cloud database systems. Many of these offer non-relational models and weaker consistency guarantees than previous high-performance systems, meaning that traditional benchmarks such as TPC-C [408] are not suitable for measuring them. In the words of its authors, YCSB is a "cloud OLTP" benchmark.

YCSB includes a number of standard workloads, two of which we used in our experiments. `Workload A`, described as "update heavy," consists of 50% reads and 50% writes. `Workload B`, described as "read heavy," is similar but consists of 95% reads and 5% writes. The data model for YCSB is key-indexed rows, each of which consists of several fields. Read and write transactions each operate on a single row that is accessed by key. For these experiments, we used 1,000 clients and configured YCSB to distribute accesses over 1,000 keys by drawing from a Zipfian distribution.

5.6.2 Dataflow Graph Construction

We ran the YCSB load-generating tool in logging mode, without attaching it to a database, then used the access traces to construct dataflow graphs. For strong consistency, we use the order of transactions in the log as the global total order. For each `READ`, we construct an edge from the most recent preceding `UPDATE` on the same key. Similarly, each `UPDATE` receives an edge from the `UPDATE` that precedes it on the same key. In addition, we construct an edge from the last transaction executed by the client; Algorithm 5.1 describes this construction.

For weak consistency, we again process transactions in log order, but we construct dependency edges in a different way. Rather than tracking a single latest version of the row at a key, we maintain a set of transactions, $liveVersions[key]$, any of which might be accessed at different parts of the system. For each `UPDATE`, we add to $liveVersions[key]$, but we do not create any edges to previous transactions, in contrast to our algorithm for strong consistency. Instead, we merge transactions in $liveVersions[key]$ on a pairwise basis at future points in time. The rate of merging transactions, λ_m, is a configurable parameter. For higher merge rates, the behavior approaches strong consistency, whereas a low merge rate can lead to significant state divergence. For each `READ`, we draw an edge from one of $liveVersions[key]$ selected at random. We also simulate a client-side read cache with a FIFO retention policy. Transactions satisfied from the read cache effectively have zero-weighted edges and do not incur any communication latencies. In this case, the zero-weighting also applies to the edge linking a transaction to the one that the client executed previously, which we construct in the same way as for strong consistency. Algorithm 5.2 describes dataflow graph construction for weak consistency.

We note that the edges in our dataflow graphs put a lower bound on the dependencies and potential communication. However, some implementations could have more communication or dependencies. For example, enforcing real-time correspondence (linearizability rather than sequential consistency) requires read-before-write dependencies to be respected, and this can involve additional communication. Providing fault tolerance also introduces additional communication, but we have not modeled this here.

Figure 5.11 illustrates the difference between strong consistency and weak consistency for `UPDATE` transactions. Under strong consistency, these updates form a chain, resulting in a critical path length of 4. Under weak consistency, writes are merged in a tree fashion, leading to a critical path length of 3.

Figure 5.12 illustrates the difference between strong consistency and weak consistency for `READ` transactions. When $C1$ executes `READ` against the cache, it takes a zero-weighted edge dependency to the previous `UPDATE`. The resulting critical path length is 1 with weak consistency, compared to 2 with strong consistency.

Algorithm 5.1 Add edges for strong consistency.

```
procedure LinkNode($txn$)
    switch $txn.type$ do
        case UPDATE
            MakeEdge(lastUpdate[$txn.key$], $txn$)
            lastUpdate[$txn.key$] $\leftarrow$ $txn$
        end case
        case READ
            MakeEdge(lastUpdate[$txn.key$], $txn$)
        end case
    end switch
    AddClientEdge($txn$)
end procedure
```

5.6.3 Comparison

We analyzed YCSB `Workload A` and `Workload B` using the dataflow graph construction of Section 5.6.2, and studied both strong consistency and weak consistency. For weak consistency, we used a read cache size of 500 entries per client and let the merge rate λ_m vary. Table 5.3 shows the results of these computations.

We computed the fraction of the work that is parallelizable, i.e. off the critical path, as

$$f = 1 - \frac{\texttt{critical nodes}}{\texttt{total nodes}} \tag{5.30}$$

The rate of sequential communication is

$$\rho_s = \frac{\texttt{critical edges}}{\texttt{unit work} \times \texttt{critical nodes}} \tag{5.31}$$

And the rate of parallelizable communication is given by

$$\rho_p = \frac{\texttt{total edges} - \texttt{critical edges}}{\texttt{unit work} \times (\texttt{total nodes} - \texttt{critical nodes})} \tag{5.32}$$

We used $\texttt{unit work} = 2,000$, a number that represents the average number of CPU cycles between communication operations (see Section 5.4.4).

Algorithm 5.2 Add edges for eventual consistency. λ_m is the merge rate.

```
procedure LinkNode($txn$)
    switch $txn.type$ do
        case UPDATE                                  ▷ Create a new version, merge later
            $liveVersions[txn.key] \leftarrow liveVersions[txn.key] + txn$
            AddClientEdge($txn$)
            $clientCaches[txn.clientId]$.Insert($txn$)
        end case
        case READ                                    ▷ No edges for client cache hit
            if $\neg$ $clientCaches[txn.clientId]$.Find($txn.key$) then
                NewEdge(RandomChoice($liveVersions[txn.key]$), $txn$)
                AddClientEdge($txn$)
                $clientCaches[txn.clientId]$.Insert($txn$)
            end if
        end case
    end switch
    for $key, txns \leftarrow$ Range(Entries($liveVersions$)) do       ▷ Merge write versions
        for $i \leftarrow 1 \ldots$ Poisson($\lambda_m \times$ Len($txns$)) do
            $txns \leftarrow$ MergeTxns($txns[n-2], txns[n-1]$) $+ txns[0 : n-2]$
        end for
        $liveVersions[key] \leftarrow txns$
    end for
end procedure
```

Algorithm 5.3 Subroutine for drawing client dependencies.

```
procedure AddClientEdge($txn$)                       ▷ Add the client dependencies
    MakeEdge($lastTxn[txn.clientId]$)
    $lastTxn[txn.clientId] \leftarrow txn$
end procedure
```

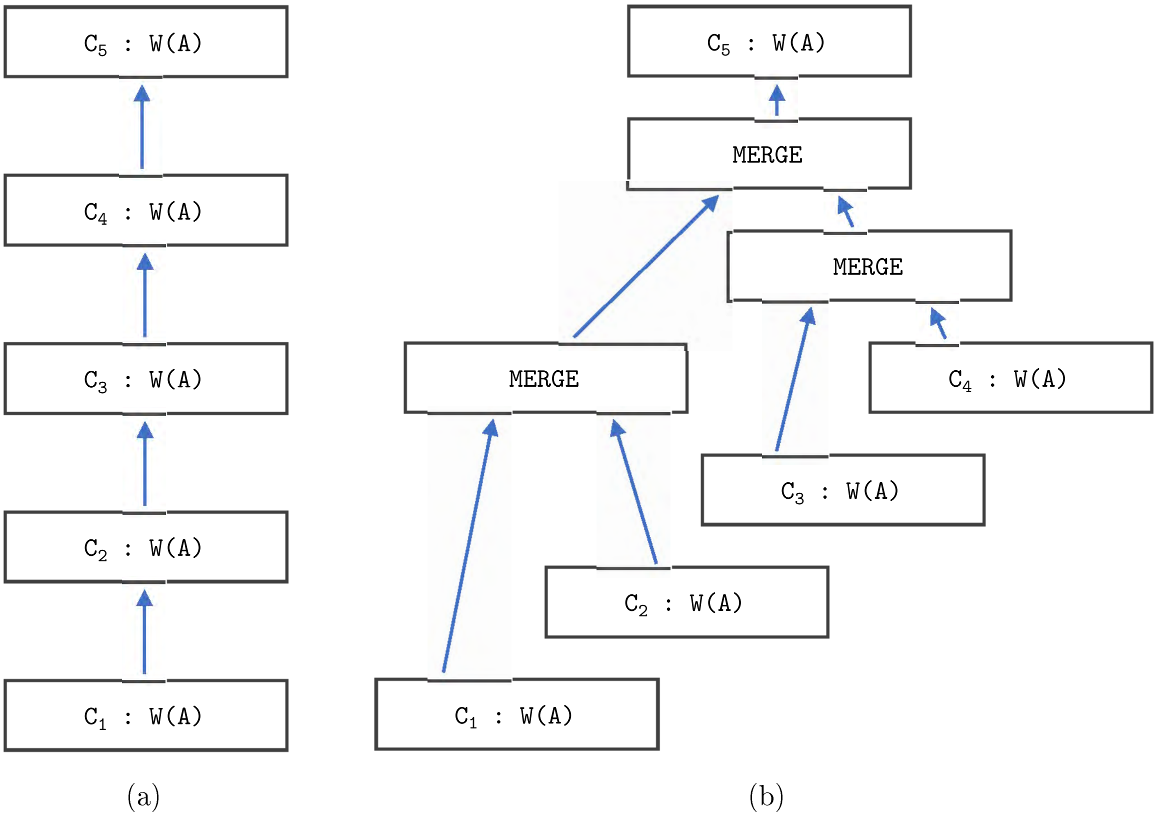


Figure 5.11: Write dataflow example for (a) strong and (b) weak consistency.

Figure 5.13 shows how the scenarios we have considered map to Equation 5.26, which indicates whether a disaggregated system would be more cost-effective than a single large server. With strong consistency, both `Workload A` and `Workload B` always prefer a large server to a disaggregated system comprised of basic servers. As we relax consistency, both workloads ultimately cross the threshold that allows cost-effective disaggregated implementation.

Our model thus demonstrates an important advantage of weak consistency: It can run on low-cost hardware. We want to emphasize that this advantage arises even though we consider only deterministic and failure-free execution. This is in contrast to the arguments in favor of weak consistency that emphasize its benefits in the face of potentially unreliable or oversubscribed underlying resources [51, 81].

5.7 Future Work

Our model has emphasized simplicity, which we hoped would bring greater understanding, but there are many ways that it can be extended. For example, one could study a workload

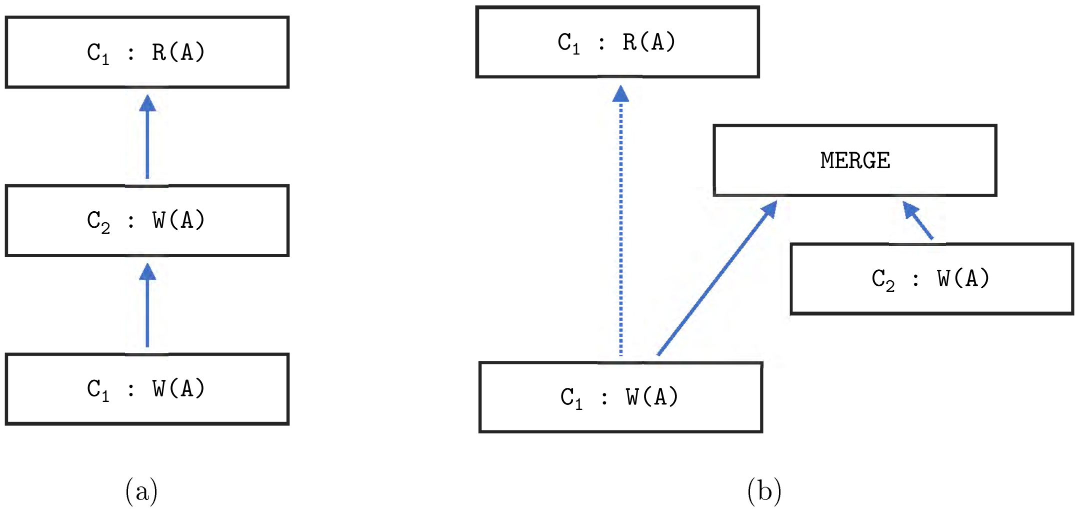


Figure 5.12: Read dataflow example for (a) strong and (b) weak consistency.

Table 5.3: Simulating YCSB. λ_m is the update rate. $\overline{\Delta V}$ is the average staleness measured in versions. Distributes is given by Equation 5.26.

Benchmark	Consistency	λ_m	f	ρ_s	ρ_p	$\overline{\Delta V}$	Distributes
YCSB-A	Strong	–	0.98044	0.00050	0.00101	0	No
YCSB-A	Weak	0.001	0.99904	0.00050	0.00064	3.11	No
YCSB-A	Weak	0.0001	0.99954	0.00050	0.00064	16.8	Yes
YCSB-A	Weak	0.00001	0.99990	0.00050	0.00063	153	Yes
YCSB-B	Strong	–	0.99814	0.00050	0.00100	0	No
YCSB-B	Weak	0.001	0.99904	0.00050	0.00045	0.157	No
YCSB-B	Weak	0.0001	0.99972	0.00050	0.00045	1.17	Yes
YCSB-B	Weak	0.00001	0.99990	0.00050	0.00045	12.8	Yes

running on a mixture of servers of different sizes. In the same way that Hill and Marty [196] showed that workloads might benefit from a mixture of processor cores with differing sequential performance, we expect that an optimal interconnect mix might include some large servers and some small servers. Whereas the present work suggests that a data center running a mixture of workloads should have a mixture of server types, the same may be true for a single workload. However, showing this probably requires a more detailed workload model.

A more detailed model might also distinguish between resource types, perhaps separately modeling CPU, memory, and storage. It could account for failures and non-determinism,

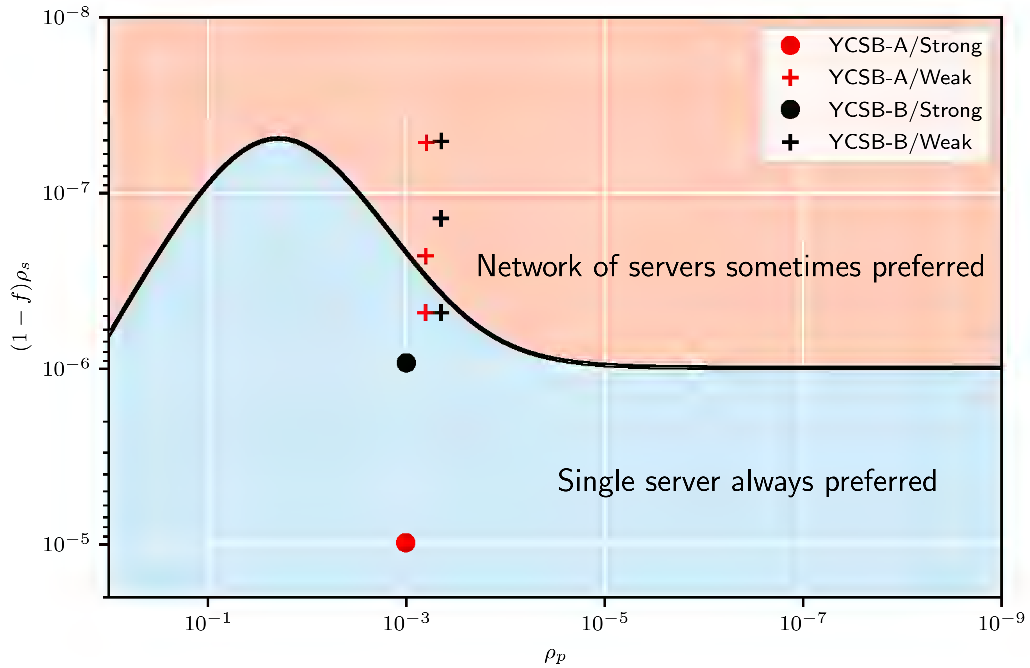


Figure 5.13: Workload parameters for YCSB extracted through dataflow analysis. We compare strong consistency to various levels of eventual consistency for both the update-heavy YCSB-A workload and the read-heavy YCSB-B workload. Equation 5.26 describes the regimes in which a single server offers the cost-optimal solution regardless of scale.

both of which we have excluded from the present model. State replication, partitioning, pipelined execution, and detailed modeling of various consistency guarantees are all possible as well.

We caution that any of these extensions should be approached with discretion. The benefit of simple models, like the one that underpins Amdahl's law, is that they can produce quick approximate results and lead to insights in a broad variety of circumstances. More detailed models may be less general, harder to apply, and could even be less accurate than a simple model [74].

5.8 Conclusion

We use Amdahl's law together with a simple model of interconnect costs to show that large server hardware provides a more cost-effective way to run some workloads than any collection of smaller servers. This is true even though large servers have a well-justified reputation for being expensive. Their defining characteristic is a low-latency interconnect, and the cost of such an interconnect, grows as the square of the number of nodes it links. Still, for communication-intensive workloads, the benefits of large server hardware can outweigh its costs.

These results have implications for future data center designs. Hardware disaggregation places individual resource types, such as CPU, storage, and accelerators, directly on the data center network, an approach that can be modeled as breaking up a server into many smaller pieces. Such disaggregation is appropriate for some workloads but not for those with a large amount of communication relative to computation, especially when that communication lies on the workload's critical path, which must be executed sequentially.

We have modeled communication, along with computation, in the context of Amdahl's law. In its original framing, Amdahl's law was used to make the case that parallel processing alone could not meet the growing demands for more computing power; that faster sequential processing would be essential as well. By modeling sequential and parallelizable communication within Amdahl's law, we extended the original result to suggest that low-latency integration, the sort that servers provide, will remain important for some workloads. We then showed how to construct the parameters of our model from a dataflow analysis of workload traces. We used the popular YCSB benchmark and compared both strong and weak consistency in this context.

The workloads most sensitive to latency are those that have order-sensitive strong consistency requirements, which, in practice, are enforced by coordination protocols. These workloads may do best with server hardware under all circumstances, even if such hardware has a high per-processor cost. In contrast, coordination-free workloads, which generally provide weak consistency, are more readily hosted on small servers or with disaggregated resources.

Today's serverless computing provides a programming abstraction over an underlying collection of servers. For suitable workloads, it offers the illusion of a single large computer. It is tempting to imagine disaggregated or "serverless" hardware that breaks down the underlying server units and instead provides resource integration at data center scale. We have shown some limitations of this approach. Even if disaggregated hardware makes its way into data centers, some workloads will continue to benefit from, or even require, the closely coupled resources that servers provide. We believe that serverless computing, as an abstraction, is the future of cloud computing, but also conclude that server hardware is here to stay.

Bibliography

[1] Daniel Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story". In: *Computer* 45.2 (2012), pp. 37–42.

[2] Daniel J. Abadi and Jose M. Faleiro. "An Overview of Deterministic Database Systems". In: *Communications of the ACM* 61.9 (2018), pp. 78–88.

[3] *About AWS / Global Infrastructure / Regions and Availability Zones.* `https://aws.amazon.com/about-aws/global-infrastructure/regions_az/`.

[4] Uri Abraham, Shai Ben-David, and Menachem Magidor. "On Global-Time and Inter-Process Communication". In: *Semantics for Concurrency.* Springer, 1990, pp. 311–323.

[5] Paarijaat Aditya et al. "Will Serverless Computing Revolutionize NFV?" In: *Proceedings of the IEEE* 107.4 (2019), pp. 667–678.

[6] Atul Adya et al. "Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks". In: *ACM SIGMOD Record* 24.2 (1995), pp. 23–34.

[7] Atul Adya et al. "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 1–14.

[8] Gojko Adzic and Robert Chatley. "Serverless Computing: Economic and Architectural Impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 2017, pp. 884–889.

[9] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX symposium on networked systems design and implementation (NSDI 20).* 2020, pp. 419–434.

[10] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Tech. rep. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[11] Nabeel Akhtar et al. "COSE: Configuring Serverless Functions Using Statistical Learning". In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications.* IEEE. 2020, pp. 129–138.

[12] *Akka - Build Concurrent, Distributed, and Resilient Message-Driven Applications for Java and Scala.* `https://akka.io`.

[13] *Akka Serverless - Stateful Serverless Architectue.* `https://www.lightbend.com/akka-serverless`.

[14] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 923–935.

[15] Nikolaos Alachiotis et al. “dReDBox: A Disaggregated Architectural Perspective for Data Centers”. In: *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 35–56.

[16] Zaid Al-Ali et al. “Making Serverless Computing More Serverless”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 456–459.

[17] Omid Alipourfard et al. “Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 469–482.

[18] Kalev Alpernas et al. “Cloud-Scale Runtime Verification of Serverless Applications”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 92–107.

[19] Kalev Alpernas et al. “Secure Serverless Computing Using Dynamic Information Flow Control”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–26.

[20] Peter Alvaro et al. “Consistency Analysis in Bloom: a CALM and Collected Approach.” In: *CIDR*. 2011, pp. 249–260.

[21] Peter Alvaro et al. “Dedalus: Datalog in Time and Space”. In: *International Datalog 2.0 Workshop*. Springer. 2010, pp. 262–281.

[22] *Amazon Compute Service Level Agreement.* `https://aws.amazon.com/compute/sla/`.

[23] *Amazon EFS Performance.* `https://docs.aws.amazon.com/efs/latest/ug/performance.html`.

[24] *Amazon Elastic File System.* `https://aws.amazon.com/efs/`.

[25] *Amazon States Language.* `https://states-language.net/spec.html`. 2016.

[26] *Amazon Web Services Launches.* `https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-launches-amazon-s3-simple-storage-service`.

[27] Gene M. Amdahl. “Computer Architecture and Amdahl’s Law”. In: *Computer* 46.12 (2013), pp. 38–46.

[28] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[29] Raghav Anand et al. “Serverless Multi-Query Motion Planning for Fog Robotics”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 7457–7463.

[30] Thomas E. Anderson, David E. Culler, and David Patterson. “A Case for NOW (Networks of Workstations)”. In: *IEEE Micro* 15.1 (1995), pp. 54–64.

[31] Thomas E. Anderson et al. “Serverless Network File Systems”. In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 1995, pp. 109–126.

[32] Anjali, Tyler Caraza-Harter, and Michael M. Swift. “Blending Containers and Virtual Machines: A Study of Firecracker and gVisor”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 101–113.

[33] *Announcing Databricks Serverless SQL*. `https://databricks.com/blog/2021/08/30/announcing-databricks-serverless-sql.html`.

[34] *Apache OpenWhisk - Open Source Serverless Cloud Platform*. `https://www.openfaas.com/`.

[35] Avinash Arjavalingam and Aditya Parameswaran. “HASTE: Serverless DAG Execution Optimizer”. In: (2021).

[36] Konstantine Arkoudas et al. “Verifying a File System Implementation”. In: *International Conference on Formal Engineering Methods*. Springer. 2004, pp. 373–390.

[37] Michael Armbrust et al. “A View of Cloud Computing”. In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[38] Michael Armbrust et al. “Lakehouse: A New Generation of Open Platforms That Unify Data Warehousing and Advanced Analytics”. In: CIDR. 2021.

[39] Joe Armstrong et al. “Concurrent Programming in ERLANG”. In: (1996).

[40] Krste Asanović. “Firebox: A Hardware Building Block for 2020 Warehouse-Scale Computers”. In: (2014).

[41] Austin Aske and Xinghui Zhao. “Supporting Multi-Provider Serverless Computing on the Edge”. In: *Proceedings of the 47th International Conference on Parallel Processing Companion*. 2018, pp. 1–6.

[42] Mohammad S. Aslanpour et al. “Serverless Edge Computing: Vision and Challenges”. In: *2021 Australasian Computer Science Week Multiconference*. 2021, pp. 1–10.

[43] Hagit Attiya and Jennifer L. Welch. “Sequential Consistency Versus Linearizability”. In: *ACM Transactions on Computer Systems (TOCS)* 12.2 (1994), pp. 91–122.

[44] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A Survey”. In: *Computer Networks* 54.15 (2010), pp. 2787–2805.

[45] *AWS Compute Optimizer*. `https://aws.amazon.com/compute-optimizer/`.

[46] *AWS Lambda Announces Provisioned Concurrency.* https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/. 2019.

[47] *Azure Cloud Cost Management.* https://azure.microsoft.com/en-us/services/cost-management/.

[48] *Azure Functions.* https://docs.microsoft.com/en-us/azure/azure-functions/.

[49] *Azure Storage Redundancy.* https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy.

[50] David F. Bacon et al. "Spanner: Becoming a SQL System". In: *Proceedings of the 2017 ACM International Conference on Management of Data.* 2017, pp. 331–343.

[51] Peter Bailis and Ali Ghodsi. "Eventual consistency today: Limitations, extensions, and beyond". In: *Queue* 11.3 (2013), p. 20.

[52] Peter Bailis et al. "Bolt-on Causal Consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 2013, pp. 761–772.

[53] Peter Bailis et al. "Quantifying Eventual Consistency With PBS". In: *The VLDB Journal* 23.2 (2014), pp. 279–302.

[54] Igor V. Balabine, Ramiah Kandasamy, and John A Skier. *File System Interface to a Database.* US Patent 5,937,406. 1999.

[55] Ioana Baldini et al. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing.* Springer, 2017, pp. 1–20.

[56] Tiemo Bang et al. "The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware". In: *Proceedings of the 16th International Workshop on Data Management on New Hardware.* 2020, pp. 1–9.

[57] Paul Baran. "The Future Computer Utility". In: *The Public Interest* 8 (1967), p. 75.

[58] Jeff Barber et al. "Bladerunner: Stream Processing at Scale for a Live View of Backend Data Mutations at the Edge". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 2021, pp. 708–723.

[59] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. "Web search for a planet: The Google cluster architecture". In: *IEEE Micro* 23.2 (2003), pp. 22–28.

[60] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The Datacenter as a Computer: Designing Warehouse-Scale machines". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189.

[61] Luiz Barroso et al. "Attack of the Killer Microseconds". In: *Communications of the ACM* 60.4 (2017), pp. 48–54.

[62] Sujit Bebortta et al. "Geospatial Serverless Computing: Architectures, Tools and Future Directions". In: *ISPRS International Journal of Geo-Information* 9.5 (2020), p. 311.

[63] Benjamin Berg et al. “The CacheLib Caching Engine: Design and Experiences at Scale”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 753–768.

[64] Philip A. Bernstein and Nathan Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Computing Surveys (CSUR)* 13.2 (1981), pp. 185–221.

[65] Philip A. Bernstein et al. “Orleans: Distributed virtual actors for programmability and scalability”. In: *MSR-TR-2014-41* (2014).

[66] Dominic Betts et al. *Exploring CQRS and Event Sourcing: A Journey Into High Scalability, Availability, and Maintainability With Windows Azure*. 2013.

[67] Anirban Bhattacharjee et al. “Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-Based Data Analytics Tasks”. In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. 2019, pp. 59–61.

[68] Anupam Bhide and Spencer Shepler. “A Highly Available Lock Manager for HA-NFS”. In: *USENIX Summer 1992 Technical Conference (USENIX Summer 1992 Technical Conference)*. 1992.

[69] Alessandro Bocci et al. “Secure FaaS Orchestration in the Fog: How Far Are We?” In: *Computing* 103.5 (2021), pp. 1025–1056.

[70] William J. Bolosky, John R. Douceur, and Jon Howell. “The Farsite Project: A Retrospective”. In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 17–26.

[71] William J. Bolosky et al. “Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs”. In: *ACM SIGMETRICS Performance Evaluation Review* 28.1 (2000), pp. 34–43.

[72] Maria C. Borges, Sebastian Werner, and Ahmet Kilic. “Faaster Troubleshooting-Evaluating Distributed Tracing Approaches for Serverless Applications”. In: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2021, pp. 83–90.

[73] James Bornholt et al. “Specifying and Checking File System Crash-Consistency Models”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 83–98.

[74] George E.P. Box. “Science and Statistics”. In: *Journal of the American Statistical Association* 71.356 (1976), pp. 791–799.

[75] Peter Braam. “The Lustre Storage Architecture”. In: *arXiv preprint arXiv:1903.01955* (2019).

[76] Lukas Brand and Markus Mock. “SFL: A Compiler for Generating Stateful AWS Lambda Serverless Applications”. In: *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. 2021, pp. 29–35.

[77] Eric Brewer. “CAP Twelve Years Later: How the “Rules” Have Changed”. In: *Computer* 45.2 (2012), pp. 23–29.

[78] Eric Brewer. “Spanner, TrueTime and the CAP Theorem”. In: (2017).

[79] Frederick P. Brooks. “No silver bullet: essence and accidents of software engineering”. In: *Information Processing*. IEEE. 1986.

[80] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings. “SOSIMPLE: A Serverless, Standards-Based, P2P SIP Communication System”. In: *First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA'05)*. IEEE. 2005, pp. 42–49.

[81] Sebastian Burckhardt. “Principles of Eventual Consistency”. In: (2014).

[82] Sebastian Burckhardt et al. “Durable Functions: Semantics for Stateful Serverless”. In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–27.

[83] Brendan Burns et al. “Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade”. In: *Queue* 14.1 (2016), pp. 70–93.

[84] James Cadden et al. “SEUSS: Skip Redundant Paths to Make Serverless Fast”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[85] Mengchu Cai et al. “Integrated Querying of SQL Database Data and S3 Data in Amazon Redshift”. In: *IEEE Data Eng. Bull.* 41.2 (2018), pp. 82–90.

[86] Brad Calder et al. “Windows Azure Storage: A Highly Available Cloud Storage Service With Strong Consistency”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 143–157.

[87] George Candea and Armando Fox. “Crash-Only Software.” In: *HotOS*. Vol. 3. 2003, pp. 67–72.

[88] Claudio Canella et al. “Fallout: Leaking Data on Meltdown-Resistant CPUs”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2019.

[89] Joao Carreira et al. “Cirrus: A Serverless Framework for End-to-End ML Workflows”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 13–24.

[90] Benjamin Carver et al. “In Search of a Fast and Efficient Serverless Dag Engine”. In: *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE. 2019, pp. 1–10.

[91] Benjamin Carver et al. “Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 1–15.

[92] Paul Castro et al. “The Rise of Serverless Computing”. In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[93] *Cache Coherent Interconnect for Accelerators*. `https://www.ccixconsortium.com`. Accessed: 2019-04-12. 2017.

[94] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. “Distributed Deadlock Detection”. In: *ACM Transactions on Computer Systems (TOCS)* 1.2 (1983), pp. 144–156.

[95] Ryan Chard et al. “FuncX: A Federated Function Serving Fabric for Science”. In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 2020, pp. 65–76.

[96] Saqib Rasool Chaudhry et al. “Improved QoS at the Edge Using Serverless Computing to Deploy Virtual Network Functions”. In: *IEEE Internet of Things Journal* 7.10 (2020), pp. 10673–10683.

[97] An Chen. “A Review of Emerging Non-Volatile Memory (NVM) Technologies and Applications”. In: *Solid-State Electronics* 125 (2016), pp. 25–38.

[98] Haogang Chen et al. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015, pp. 18–37.

[99] Huan Chen and Liang-Jie Zhang. “Fbaas: Functional Blockchain as a Service”. In: *International Conference on Blockchain*. Springer. 2018, pp. 243–250.

[100] Alvin Cheung et al. “New Directions in Cloud Programming”. In: *CIDR* (2021).

[101] Jaeghang Choi and Kyungyong Lee. “Evaluation of Network File System as a Shared Data Storage in Serverless Computing”. In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 2020, pp. 25–30.

[102] Eric Chung et al. “Serving DNNs in Real Time at Datacenter Scale With Project Brainwave”. In: *IEEE Micro* 38.2 (2018), pp. 8–20.

[103] Austin T. Clements et al. “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”. In: *ACM Transactions on Computer Systems (TOCS)* 32.4 (2015), pp. 1–47.

[104] *Cloud Cost Management - Apptio*. `https://www.apptio.com/solutions/cloud-cost-management/`.

[105] *Cloud Cost Management - Harness*. `https://harness.io/products/cloud-cost/`.

[106] *CockroachDB - Architecture Overview*. `https://www.cockroachlabs.com/docs/stable/architecture/overview.html`.

[107] *Common Internet File System (CIFS) Protocol*. `https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-CIFS/[MS-CIFS].pdf`. v20201001. 2021.

[108] *Compute Engine Service Level Agreement (SLA)*. `https://cloud.google.com/compute/sla`.

[109] *Configure NTP on a VM*. `https://cloud.google.com/compute/docs/instances/configure-ntp`.

[110] Neil Conway et al. “Logic and Lattices for Distributed Programming”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 1.

[111] Blaine Cook. *Scaling Twitter*. `https://www.slideshare.net/Blaine/scaling-twitter` and `https://www.youtube.com/watch?v=AEef0ZPvgKs`. Presentation at SDForum Silicon Valley. Apr. 2007.

[112] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems With YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[113] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[114] *Custom AWS Lambda Runtimes*. `https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html`.

[115] Benoit Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[116] *Data protection in Amazon S3*. `https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html`.

[117] *Dataprep by Trifacta*. `https://cloud.google.com/dataprep`.

[118] Pubali Datta et al. “Valve: Securing Function Workflows on Serverless Computing Platforms”. In: *Proceedings of The Web Conference 2020*. 2020, pp. 939–950.

[119] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. “Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments”. In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 356–370.

[120] Jeff Dean. “Machine learning for systems and systems for machine learning”. In: *Presentation at 2017 Conference on Neural Information Processing Systems*. 2017.

[121] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[122] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[123] *Delta Lake*. `https://delta.io/`.

[124] Murat Demirbas. “The Advent of Tightly Synchronized Clocks in Distributed Systems”. In: (2018).

[125] Peter J. Denning. “The Locality Principle”. In: *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*. World Scientific, 2006, pp. 43–67.

[126] Peter J. Denning. “The Working Set Model for Program Behavior”. In: *Communications of the ACM* 11.5 (1968), pp. 323–333.

[127] Akon Dey et al. “YCSB+T: Benchmarking Web-Scale Transactional Databases”. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE. 2014, pp. 223–230.

[128] Jeff Dike. “A User-Mode Port of the Linux Kernel”. In: *Annual Linux Showcase & Conference*. Vol. 10. 1268379.1268386. 2000.

[129] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. “Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems”. In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[130] *Dragonboat - A Multi-Group Raft Library in Go*. https://github.com/lni/dragonboat.

[131] Dong Du et al. “Catalyzer: Sub-Millisecond Startup for Serverless Computing With Initialization-Less Booting”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 467–481.

[132] *DynamoDB*. https://aws.amazon.com/dynamodb/.

[133] Susan J. Eggers et al. “Simultaneous Multithreading: A Platform for Next-Generation Processors”. In: *IEEE Micro* 17.5 (1997), pp. 12–19.

[134] Simon Eismann et al. “Sizeless: Predicting the Optimal Size of Serverless Functions”. In: *Proceedings of the 22nd International Middleware Conference*. 2021, pp. 248–259.

[135] Simon Eismann et al. “The State of Serverless Applications: Collection, Characterization, and Community Consensus”. In: *IEEE Transactions on Software Engineering* (2021).

[136] Adam Eivy and Joe Weinman. “Be Wary of the Economics of “Serverless” Cloud Computing”. In: *IEEE Cloud Computing* 4.2 (2017), pp. 6–12.

[137] *Elastic Fabric Adapter - Amazon Web Services*. https://aws.amazon.com/hpc/efa/.

[138] Tarek Elgamal. “Costless: Optimizing Cost of Serverless Computing Through Function Fusion and Placement”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 300–312.

[139] Rasha Eqbal. “ScaleFS: A Multicore-Scalable File System”. MA thesis. Massachusetts Institute of Technology, 2014.

[140] Nafise Eskandani and Guido Salvaneschi. “The Wonderless Dataset for Serverless Computing”. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 565–569.

[141] Facebook. *Disaggregated Rack*. https://web.archive.org/web/20160421092139/http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf. 2013.

[142] Jose M. Faleiro and Daniel J. Abadi. "Rethinking Serializable Multiversion Concurrency Control". In: *Proceedings of the VLDB Endowment* 8.11 (2015).

[143] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. "High Performance Transactions via Early Write Visibility". In: *Proceedings of the VLDB Endowment* 10.5 (2017).

[144] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008), pp. 63–74.

[145] Nathan Farrington and Alexey Andreyev. "Facebook's Data Center Network Architecture". In: *2013 Optical Interconnects Conference*. Citeseer. 2013, pp. 49–50.

[146] Mark Fasheh. "OCFS2: The Oracle Clustered File System, Version 2". In: *Proceedings of the 2006 Linux Symposium*. Vol. 1. Citeseer, 2006, pp. 289–302.

[147] *Filesystem in Userspace*. `https://github.com/libfuse/libfuse`.

[148] Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In: *NSDI*. 2017, pp. 363–376.

[149] Sadjad Fouladi et al. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 475–488.

[150] Armando Fox et al. "Cluster-Based Scalable Network Services". In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. 1997, pp. 78–91.

[151] E. Frachtenberg. "Holistic Datacenter Design in the Open Compute Project". In: *Computer* 45.7 (July 2012), pp. 83–85. ISSN: 0018-9162.

[152] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. "Concurrency Control for High Contention Environments". In: *ACM Transactions on Database Systems (TODS)* 17.2 (1992), pp. 304–345.

[153] Michael J. Franklin, Michael J. Carey, and Miron Livny. "Transactional client-server cache consistency: Alternatives and performance". In: *ACM Transactions on Database Systems (TODS)* 22.3 (1997), pp. 315–363.

[154] Justin Franz et al. "Reunifying Families after a Disaster via Serverless Computing and Raspberry Pis". In: *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE. 2018, pp. 131–132.

[155] *Friendster Lost Lead Because of a Failure to Scale*. `http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html`. Nov. 2007.

[156] Ken Fromm. *Why the Future of Software and Apps Is Serverless*. `https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/`. 2012.

[157] Maurizio Gabbrielli et al. "No More, No Less - A Formal Model for Serverless Computing". In: *International Conference on Coordination Languages and Models*. Springer. 2019, pp. 148–157.

[158] Carsten Binnig Andrew Crotty Alex Galakatos and Tim Kraska Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *Proceedings of the VLDB Endowment* 9.7 (2016).

[159] Yu Gan et al. "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*. 2019, pp. 3–18.

[160] Fabian Gand et al. "Serverless Container Cluster Management for Lightweight Edge Clouds." In: *CLOSER*. 2020, pp. 302–311.

[161] Gen-Z Consortium. *Gen-Z Overview*. Tech. rep. Gen-Z Consortium, 2018. URL: `https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview-V1.pdf`.

[162] Yilong Geng et al. "Exploiting a Natural Network Effect for Scalable, Fine-Grained Clock Synchronization". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 81–94.

[163] Sara Ghaemi, Hamzeh Khazaei, and Petr Musilek. "ChainFaaS: An Open Blockchain-Based Serverless Platform". In: *IEEE Access* 8 (2020), pp. 131760–131778.

[164] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: (2003).

[165] David K. Gifford. "Information Storage in a Decentralized Computer System". PhD thesis. Stanford University, 1981.

[166] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[167] David Goltzsche et al. "Acctee: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting". In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 123–135.

[168] *GraphQL*. `http://spec.graphql.org/June2018/`. 2018.

[169] Cary Gray and David Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency". In: *ACM SIGOPS Operating Systems Review* 23.5 (1989), pp. 202–210.

[170] James N. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems*. Springer, 1978, pp. 393–481.

[171] Jim Gray and Franco Putzolu. "The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time". In: *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 1987, pp. 395–398.

[172] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. 3rd ed. MIT press, 2014.

[173] *gRPC: A High Performance, Open Source Universal RPC Framework*. `https://www.grpc.io/`.

[174] Paul Grun. "Introduction to InfiniBand for End Users". In: *White Paper, InfiniBand Trade Association* 55 (2010).

[175] Jashwant Raj Gunasekaran et al. "Spock: Exploiting Serverless Functions for Slo and Cost Aware Resource Procurement in Public Cloud". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 199–208.

[176] Anurag Gupta et al. "Amazon Redshift and the Case for Simpler Data Warehouses". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1917–1923.

[177] Vipul Gupta et al. "Serverless Straggler Mitigation Using Error-Correcting Codes". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2020, pp. 135–145.

[178] Vipul Gupta et al. "Utility-Based Resource Allocation and Pricing for Serverless Computing". In: *arXiv preprint arXiv:2008.07793* (2020).

[179] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[180] *gVisor*. `https://gvisor.dev/`.

[181] Andreas Haas et al. "Bringing the Web Up to Speed With WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

[182] Adam Hall and Umakishore Ramachandran. "An Execution Model for Serverless Functions at the Edge". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. 2019, pp. 225–236.

[183] Paul Harrison. *Serverless Computing: Terrible Name but Brilliant Service*. `https://medium.com/@smoothml/serverless-computing-terrible-name-but-brilliant-service-bcf072b9c279`.

[184] Rober Haskin, Yoni Malachi, and Gregory Chan. "Recovery Management in QuickSilver". In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 82–108.

[185] T. Haynes and D. Noveck. *Network File System (NFS) Version 4 Protocol*. RFC 7530, March 2015, `https://tools.ietf.org/html/rfc7530`.

[186] K. Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 620–629.

[187] Joseph M. Hellerstein and Peter Alvaro. "Keeping CALM: When Distributed Consistency Is Easy". In: *Communications of the ACM* 63.9 (2020), pp. 72–81.

[188] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System*. Now Publishers Inc, 2007.

[189] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR* (2019).

[190] Scott Hendrickson et al. "Serverless Computation With OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[191] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2017.

[192] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[193] Martijn de Heus et al. "Distributed Transactions on Serverless Stateful Functions". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 2021, pp. 31–42.

[194] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[195] Dean Hildebrand and Denis Serenyi. *Colossus Under the Hood: A Peek Into Google's Scalable Storage System*. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`. 2019.

[196] Mark D. Hill and Michael R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (2008), pp. 33–38.

[197] Todd Hoff. *The Instagram Architecture Facebook Bought for a Cool Billion Dollars*. `http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html`. Apr. 2012.

[198] Sanghyun Hong et al. "Go Serverless: Securing Cloud via Serverless Design Patterns". In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. 2018.

[199] *Host, n.2*. In: *OED Online*. Oxford University Press. URL: `https://www.oed.com/view/Entry/88744?result=2#eid`.

[200] *How Aurora Serverless V2 (Preview) Works*. `https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-2.how-it-works.html`.

[201] John H. Howard et al. “Scale and Performance in a Distributed File System”. In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 51–81.

[202] HP Labs. *The Machine*. `https://www.hpl.hp.com/research/systems-research/themachine/`. 2017.

[203] Yige Hu et al. “TxFS: Leveraging File-system Crash Consistency to Provide ACID Transactions”. In: *ACM Trans. Storage* 15.2 (May 2019), pp. 1–20.

[204] Razin Farhan Hussain, Mohsen Amini Salehi, and Omid Semiari. “Serverless Edge Computing for Green Oil and Gas Industry”. In: *2019 IEEE Green Technologies Conference (GreenTech)*. IEEE. 2019, pp. 1–4.

[205] *InfiniBand Roadmap*. `https://www.infinibandta.org/infiniband-roadmap/`.

[206] Intel. *Intel Rack Scale Design Architecture*. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf`. 2018.

[207] *Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network*. `https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/`. 2014.

[208] *Introducing the Amazon Time Sync Service*. `https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/`. 2017.

[209] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. “Serving Deep Learning Models in a Serverless Platform”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 257–262.

[210] Vitalii Ivanov and Kari Smolander. “Implementation of a DevOps Pipeline for Serverless Applications”. In: *International conference on product-focused software process improvement*. Springer. 2018, pp. 48–64.

[211] Abhinav Jangda et al. “Formal Foundations of Serverless Computing”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–26.

[212] Zhipeng Jia and Emmett Witchel. “Boki: Stateful Serverless Computing With Shared Logs”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 691–707.

[213] Jiawei Jiang et al. “Towards Demystifying Serverless Machine Learning Training”. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 857–871.

[214] Paul Johnston. *“Serverless” Is Just a Name. We Could Have Called It “Jeff”*. `https://serverless.zone/serverless-is-just-a-name-we-could-have-called-it-jeff-1958dd4c63d7`.

[215] *Joins in Azure Cosmos DB*. `https://docs.microsoft.com/en-us/azure/cosmos-db/sql/sql-query-join`.

[216] Eric Jonas et al. *Cloud Programming Simplified: a Berkeley View on Serverless Computing.* UC Berkeley Technical Report No. UCB/EECS-2019-3. 2019.

[217] Eric Jonas et al. "Occupy the Cloud: Distributed Computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing.* ACM. 2017, pp. 445–451.

[218] Andrew Josey, Eric Blake, Geoff Clare, et al. *The Open Group Base Specifications Issue 7.* `https://pubs.opengroup.org/onlinepubs/9699919799/`. 2018.

[219] Norman P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture.* 2017, pp. 1–12.

[220] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. "Centralized Core-Granular Scheduling for Serverless Functions". In: *Proceedings of the ACM Symposium on Cloud Computing.* 2019, pp. 158–164.

[221] Antti Kantee. "Rump File Systems: Kernel Code Reborn". In: *USENIX Annual Technical Conference.* 2009, pp. 15–15.

[222] *Kata Containers - The Speed of Containers, the Security of VMs.* `https://katacontainers.io/`.

[223] Kostas Katrinis et al. "Rack-Scale Disaggregated Cloud Data Centers: The dReD-Box Project Vision". In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2016, pp. 690–695.

[224] Michael Leon Kazar et al. *Synchronization and Caching Issues in the Andrew File System.* Carnegie Mellon University, Information Technology Center, 1988.

[225] Wes Kendall. *MPI Tutorial - MPI Send and Receive.* `https://mpitutorial.com/tutorials/mpi-send-and-receive/`.

[226] Jeongchul Kim and Kyungyong Lee. "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* IEEE. 2019, pp. 502–504.

[227] Youngbin Kim and Jimmy Lin. "Serverless Data Analytics With Flint". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* IEEE. 2018, pp. 451–455.

[228] Spencer Kimball and Irfan Sharif. *Living Without Atomic Clocks.* `https://www.cockroachlabs.com/blog/living-without-atomic-clocks/`. 2021.

[229] Ana Klimovic et al. "Pocket: Elastic ephemeral storage for serverless analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* 2018, pp. 427–444.

[230] Ana Klimovic et al. "Understanding Ephemeral Storage for Serverless Analytics". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 2018, pp. 789–794.

[231] *Knative - Enterprise-Grade Serverless on Your Own Terms.* `https://knative.dev/`.

[232] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[233] Eddie Kohler et al. "The Click Modular Router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[234] Ricardo Koller and Dan Williams. "Will serverless end the dominance of Linux in the cloud?" In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 2017, pp. 169–173.

[235] Brian Krebs. *What We Can Learn From the Capital One Hack*. 2019.

[236] *Kubeless - The Kubernetes Native Servlerss Framework*. `https://kubeless.io/`.

[237] Jörn Kuhlenkamp et al. "Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 1576–1585.

[238] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 239–250.

[239] Jacek Kuśnierz et al. "Distributed Parallel Analysis Engine for High Energy Physics Using AWS Lambda". In: *Proceedings of the 1st Workshop on High Performance Serverless Computing*. 2020, pp. 13–16.

[240] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 6–13.

[241] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* 9 (1979), pp. 690–691.

[242] Leslie Lamport. "The Part-Time Parliament". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[243] Leslie Lamport et al. "Paxos Made Simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[244] Butler W. Lampson and David D. Redell. "Experience With Processes and Monitors in Mesa". In: *Communications of the ACM* 23.2 (1980), pp. 105–117.

[245] James Larisch, James Mickens, and Eddie Kohler. "Alto: Lightweight VMs Using Virtualization-Aware Managed Runtimes". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 2018, pp. 1–7.

[246] P-A Larson, Jonathan Goldstein, and Jingren Zhou. "MTCache: Transparent mid-tier database caching in SQL Server". In: *Proceedings. 20th International Conference on Data Engineering*. IEEE. 2004, pp. 177–188.

[247] Duncan H. Lawrie. "Access and Alignment of Data in an Array Processor". In: *IEEE Transactions on Computers* 100.12 (1975), pp. 1145–1155.

[248] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. "Evaluation of Production Serverless Computing Environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 442–450.

[249] Han Li et al. "Analysis of the Synchronization Requirements of 5G and Corresponding Solutions". In: *IEEE Communications Standards Magazine* 1.1 (2017), pp. 52–58.

[250] Jian Li. *High Throughput Computing Data Center Architecture - Thinking of Data Center 3.0*. `http://acs.ict.ac.cn/asbd2014/slides/ASBD_InvitedTalk_Li.pdf`. 2014.

[251] Xing Li, Xue Leng, and Yan Chen. "Securing Serverless Computing: Challenges, Solutions, and Opportunities". In: *arXiv preprint arXiv:2105.12581* (2021).

[252] Yuliang Li et al. "Sundial: Fault-Tolerant Clock Synchronization for Datacenters". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 1171–1186.

[253] Kevin Lim et al. "System-Level Implications of Disaggregated Memory". In: *IEEE International Symposium on High-Performance Comp Architecture*. IEEE. 2012, pp. 1–12.

[254] Xiayue Charles Lin, Joseph E. Gonzalez, and Joseph M. Hellerstein. "Serverless Boom or Bust? An Analysis of Economic Incentives". In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[255] Wyatt Lloyd et al. "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 401–416.

[256] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. "FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing". In: *Proceedings of the 22nd International Middleware Conference*. 2021, pp. 159–171.

[257] Nancy A. Lynch. *Distributed Algorithms*. Elsevier, 1996.

[258] Nancy A. Lynch et al. *Atomic Transactions: in Concurrent and Distributed Systems*. Morgan Kaufmann Publishers Inc., 1993.

[259] Theo Lynn et al. "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pp. 162–169.

[260] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. "Consistency, Availability, and Convergence". In: *University of Texas at Austin Tech Report* 11 (2011), p. 158.

[261] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. "The Serverless Shell". In: *Proceedings of the 22nd International Middleware Conference: Industrial Track*. 2021, pp. 9–15.

[262] Ashraf Mahgoub et al. "SONIC: Application-Aware Data Passing for Chained Serverless Applications". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 285–301.

[263] Nima Mahmoudi et al. "Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm". In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 2019, pp. 203–213.

[264] Pascal Maissen et al. "FaaSdom: A Benchmark Suite for Serverless Computing". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 73–84.

[265] Filipe Manco et al. "My VM Is Lighter (And Safer) Than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 218–233.

[266] Johannes Manner, Stefan Kolb, and Guido Wirtz. "Troubleshooting Serverless Functions: A Combined Monitoring and Debugging Approach". In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (2019), pp. 99–104.

[267] Horácio Martins, Filipe Araujo, and Paulo Rupino da Cunha. "Benchmarking Serverless Computing Platforms". In: *Journal of Grid Computing* 18.4 (2020), pp. 691–709.

[268] Garrett McGrath and Paul R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[269] Ross Mcilroy et al. "Spectre is here to stay: An analysis of side-channels and speculative execution". In: *arXiv preprint arXiv:1902.05178* (2019).

[270] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at What COST?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[271] Dominik Meissner et al. "Retro-λ: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 2018, pp. 76–87.

[272] Alexander Mejía et al. "Serverless Based Control and Monitoring for Search and Rescue Robots". In: *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE. 2020, pp. 1–6.

[273] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[274] *Mezzanine: An open source content management platform built using the Django framework*. `http://mezzanine.jupo.org/`.

[275] Brenda M. Michelson. "Event-Driven Architecture Overview". In: *Patricia Seybold Group* 2.12 (2006), pp. 10–1571.

[276] Samuel P. Midkiff. "Automatic Parallelization: An Overview of Fundamental Compiler Techniques". In: *Synthesis Lectures on Computer Architecture* 7.1 (2012), pp. 1–169.

[277] Rashid Mijumbi et al. “Network Function Virtualization: State-of-the-Art and Research Challenges”. In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 236–262.

[278] Mae Milano and Andrew C. Myers. “MixT: A Language for Mixing Consistency in Geodistributed Transactions”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2018. Philadelphia, PA, USA: ACM, June 2018, pp. 226–241. ISBN: 978-1-4503-5698-5. DOI: `10.1145/3192366.3192375`.

[279] Dejan S. Milojicic et al. *Peer-to-Peer Computing.* 2002.

[280] Changwoo Min et al. “Lightweight Application-Level Crash Consistency on Transactional Flash Storage”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15).* 2015, pp. 221–234.

[281] Anup Mohan et al. “Agile Cold Starts for Scalable Serverless”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19).* 2019.

[282] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. “An Evaluation of Open Source Serverless Computing Frameworks.” In: *CloudCom.* 2018, pp. 115–120.

[283] Carroll Morgan and Bernard Sufrin. “Specification of the UNIX Filing System”. In: *IEEE Transactions on Software Engineering* 2 (1984), pp. 128–142.

[284] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* 2018, pp. 561–577.

[285] Ingo Müller, Renato Marroquín, and Gustavo Alonso. “Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2020, pp. 115–130.

[286] Irakli Nadareishvili. *Serverless Is a Wrong Name. Fight It With FIRE.* `https://www.freshblurbs.com/blog/2018/01/10/Stop-Serverless-Call-It-Fire.html`.

[287] Michael N. Nelson, Brent B. Welch, and John K Ousterhout. “Caching in the Sprite Network File System”. In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 134–154.

[288] Sam Newman. *Building Microservices.* " O’Reilly Media, Inc.", 2021.

[289] Hai Duc Nguyen, Zhifei Yang, and Andrew A. Chien. “Motivating High Performance Serverless Workloads”. In: *Proceedings of the 1st Workshop on High Performance Serverless Computing.* 2020, pp. 25–32.

[290] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. “Speculative Execution in a Distributed File System”. In: *ACM Transactions on Computer Systems (TOCS)* 24.4 (2006), pp. 361–392.

[291] Edmund B. Nightingale et al. “Rethink the Sync”. In: *ACM Transactions on Computer Systems (TOCS)* 26.3 (2008), p. 6.

[292] Bill Nitzberg and Virginia Lo. “Distributed shared memory: A survey of issues and algorithms”. In: *Computer* 24.8 (1991), pp. 52–60.

[293] Geoffrey Noer and David P. Moulton. *How Cloud Storage Delivers 11 Nines of Durability—And How You Can Help.* `https://cloud.google.com/blog/products/storage-data-transfer/understanding-cloud-storage-11-9s-durability-target`. 2021.

[294] Gian Ntzik. “Reasoning About POSIX File Systems”. PhD thesis. Imperial College London, 2016.

[295] Gian Ntzik et al. “A Concurrent Specification of POSIX File Systems”. In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2018.

[296] Edward Oakes et al. “SOCK: Rapid task provisioning with serverless-optimized containers”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 2018, pp. 57–70.

[297] Matthew Obetz et al. “Formalizing Event-Driven Behavior of Serverless Applications”. In: *European Conference on Service-Oriented and Cloud Computing.* Springer. 2020, pp. 19–29.

[298] Michael A. Olson. “The Design and Implementation of the Inversion File System.” In: *USENIX Winter.* 1993, pp. 205–218.

[299] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 2014, pp. 305–319.

[300] *OpenFaas - Serverless Functions, Made Simple.* `https://www.openfaas.com/`.

[301] *Orleans at Microsoft.* `https://www.youtube.com/watch?v=KhgYlvGLv9c`.

[302] Ippokratis Pandis. “The Evolution of Amazon Redshift”. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3162–3174.

[303] Ruoming Pang et al. “Zanzibar: Google’s Consistent, Global Authorization System”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19).* 2019, pp. 33–46.

[304] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Inc., 1986.

[305] Christos H. Papadimitriou. “The Serializability of Concurrent Database Updates”. In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 631–653.

[306] Mark S. Papamarcos and Janak H. Patel. “A Low-Overhead Coherence Solution for Multiprocessors With Private Cache Memories”. In: *Proceedings of the 11th annual international symposium on Computer architecture.* 1984, pp. 348–354.

[307] Douglas F. Parkhill. "Challenge of the Computer Utility". In: (1966).

[308] David L. Parnas. "On the Criteria to Be Used in Decomposing Systems Into Modules". In: *Pioneers and Their Contributions to Software Engineering*. Springer, 1972, pp. 479–498.

[309] Andrea Passwater. *2018 Serverless Community Survey: Huge growth in serverless usage*. `https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/`. Accessed: 2019-01-23. 2018.

[310] Robert S. Patti. "Three-Dimensional Integrated Circuits and the Future of System-on-Chip Designs". In: *Proceedings of the IEEE* 94.6 (2006), pp. 1214–1224.

[311] Andrew Pavlo et al. "Self-Driving Database Management Systems." In: *CIDR*. Vol. 4. 2017, p. 1.

[312] Andrew Pawloski et al. "Improving Information and Communications in a Disaster Scenario With AWS Snowball Edge". In: *AGU Fall Meeting Abstracts*. Vol. 2019. 2019, IN23B–09.

[313] Brian Pawlowski et al. "NFS Version 3: Design and Implementation." In: *USENIX Summer*. Boston, MA. 1994, pp. 137–152.

[314] Nathan Pemberton. "Exploring the Disaggregated Memory Interface Design Space". In: *Workshop on Resource Disaggregation (WORD)*. 2019.

[315] Nathan Pemberton and Johann Schleier-Smith. "The Serverless Data Center: Hardware Disaggregation Meets Serverless Computing". In: *The First Workshop on Resource Disaggregation*. Vol. 4. 2019.

[316] Nathan Pemberton, Johann Schleier-Smith, and Joseph E. Gonzalez. "The RESTless Cloud". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 49–57.

[317] Matthew Perron et al. "Starling: A Scalable Query Engine on Cloud Functions". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 131–141.

[318] Gregory F. Pfister. "An Introduction to the InfiniBand Architecture". In: *High Performance Mass Storage and Parallel I/O* 42.617-632 (2001), p. 10.

[319] Chuck Pheatt. "Intel® Threading Building Blocks". In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298.

[320] Thanumalayan Sankaranarayana Pillai et al. "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 433–448.

[321] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. "Dynamic Allocation of Serverless Functions in IoT Environments". In: *2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC)*. IEEE. 2018, pp. 1–8.

[322] Christian Plattner and Gustavo Alonso. "Ganymed: Scalable replication for transactional web applications". In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag. 2004, pp. 155–174.

[323] Donald E. Porter et al. "Operating System Transactions". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, Oct. 2009, pp. 161–176.

[324] Kenneth W. Preslan et al. "A 64-Bit, Shared Disk File System for Linux". In: *16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (Cat. No. 99CB37098)*. IEEE. 1999, pp. 22–41.

[325] *Presto - Distributed SQL Query Engine for Big Data*. `https://prestodb.io/`.

[326] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 193–206.

[327] Kevin Pulo. "Fun With LD_PRELOAD". In: *linux. conf. au*. Vol. 153. 2009, p. 103.

[328] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. "LKL: The Linux Kernel Library". In: *9th RoEduNet IEEE International Conference*. IEEE. 2010, pp. 328–333.

[329] Weizhong Qiang, Zezhao Dong, and Hai Jin. "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2018, pp. 451–470.

[330] Paul Rad et al. "ZeroVM: Secure Distributed Processing for Big Data Analytics". In: *2014 World Automation Congress (WAC)*. IEEE. 2014, pp. 1–6.

[331] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[332] Ravi Rajwar and James R. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution". In: *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE. 2001, pp. 294–305.

[333] *Recommender - Google Cloud*. `https://cloud.google.com/recommender/docs/overview`.

[334] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the third ACM symposium on cloud computing*. 2012, pp. 1–13.

[335] Kun Ren, Dennis Li, and Daniel J. Abadi. "Slog: Serializable, Low-Latency, Geo-Replicated Transactions". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1747–1761.

[336] Mike Roberts. *Defining Serverless—Part 1*. https://blog.symphonia.io/posts/2017-06-22_defining-serverless-part-1. 2017.

[337] Mike Roberts. *Serverelss Architectures*. https://martinfowler.com/articles/serverless.html#origin. See ‘Origin of Serverless’ sidebar. 2018.

[338] Francisco Romero. “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications”. In: (2021).

[339] Francisco Romero et al. “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications”. In: *arXiv preprint arXiv:2104.13869* (2021).

[340] Arjun Roy et al. “Inside the Social Network’s (Datacenter) Network”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015, pp. 123–137.

[341] Jeff Rulifson. *RFC 5: Decode Encode Language (DEL)*. https://datatracker.ietf.org/doc/html/rfc5. 1969.

[342] Krzysztof Rzadca et al. “Autopilot: Workload Autoscaling at Google”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[343] Andrei Sabelfeld and Andrew C. Myers. “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[344] Resve Saleh et al. “System-on-Chip: Reuse and Integration”. In: *Proceedings of the IEEE* 94.6 (2006), pp. 1050–1069.

[345] Jerome H. Saltzer, David P. Reed, and David D. Clark. “End-to-End Arguments in System Design”. In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

[346] Josep Sampé et al. “Serverless Data Analytics in the IBM Cloud”. In: *Proceedings of the 19th International Middleware Conference Industry*. 2018, pp. 1–8.

[347] Marc Sánchez-Artigas and Pablo Gimeno Sarroca. “Experience Paper: Towards Enhancing Cost Efficiency in Serverless Machine Learning Training”. In: *Proceedings of the 22nd International Middleware Conference*. 2021, pp. 210–222.

[348] Russel Sandberg et al. “Design and implementation of the Sun network filesystem”. In: *Proceedings of the Summer USENIX conference*. 1985, pp. 119–130.

[349] Arnav Sankaran, Pubali Datta, and Adam Bates. “Workflow Integration Alleviates Identity and Access Management in Serverless Computing”. In: *Annual Computer Security Applications Conference*. 2020, pp. 496–509.

[350] Klaus Satzke et al. “Efficient GPU Sharing for Serverless Workflows”. In: *Proceedings of the 1st Workshop on High Performance Serverless Computing*. 2020, pp. 17–24.

[351] Steve Scargall. “Introducing the Persistent Memory Development Kit”. In: *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berkeley, CA: Apress, 2020, pp. 63–72. DOI: 10.1007/978-1-4842-4932-1_5.

[352] Stephan van Schaik et al. “RIDL: Rogue In-Flight Data Load”. In: *S&P (May 2019)* (2019).

[353] *Schema Reference Guide for the Workflow Definition Language in Azure Logic Apps.* `https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-workflow-definition-language`.

[354] Joel Scheuner and Philipp Leitner. “Function-as-a-Service Performance Evaluation: A Multivocal Literature Review”. In: *Journal of Systems and Software* 170 (2020), p. 110708.

[355] Johann Schleier-Smith. “Serverless Foundations for Elastic Database Systems”. In: *CIDR* (2019).

[356] Johann Schleier-Smith et al. “What Serverless Computing Is and Should Become: The Next Phase of Cloud Computing”. In: *Communications of the ACM* 64.5 (2021), pp. 55–63.

[357] Frank Schmuck and Jim Wylie. “Experience With Transactions in QuickSilver”. In: *ACM SIGOPS Operating Systems Review*. Vol. 25. 5. ACM. 1991, pp. 239–253.

[358] Erick Schonfeld. *Twitter Downtime on the Upswing.* `https://techcrunch.com/2007/12/20/twitter-downtime-on-the-upswing/`. Dec. 2007.

[359] Philip Schwan et al. “Lustre: Building a File System for 1000-Node Clusters”. In: *Proceedings of the 2003 Linux symposium*. Vol. 2003. 2003, pp. 380–386.

[360] *Server Message Block SMB Protocol.* `https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SMB/[MS-SMB].pdf`. v20180912. 2018.

[361] *Server, n.* In: *OED Online*. Oxford University Press. URL: `https://www.oed.com/view/Entry/176669#eid`.

[362] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *2020 USENIX Annual Technical Conference (USENIX 20)*. 2020, pp. 205–218.

[363] Yizhou Shan et al. “LegoOS: A disseminated, distributed OS for hardware resource disaggregation”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 69–87.

[364] Vaishaal Shankar et al. “Serverless Linear Algebra”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 281–295.

[365] Marc Shapiro et al. “Conflict-Free Replicated Data Types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[366] Weisong Shi et al. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[367] Yuan Shi. “Reevaluating Amdahl’s Law and Gustafson’s Law”. In: *Computer Sciences Department, Temple University (MS: 38-24)* (1996).

[368] Simon Shillaker and Peter Pietzuch. “Faasm: lightweight isolation for efficient stateful serverless computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 419–433.

[369] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Vol. 10. 2010, pp. 1–10.

[370] Arjun Singh et al. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 183–197.

[371] Arjun Singhvi et al. “Atoll: A Scalable Low-Latency Serverless Platform”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 138–152.

[372] Arjun Singhvi et al. “SNF: Serverless Network Functions”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 296–310.

[373] Alok Sinha. “Client-Server Computing”. In: *Communications of the ACM* 35.7 (1992), pp. 77–98.

[374] Swaminathan Sivasubramanian. “Amazon dynamoDB: a seamlessly scalable non-relational database service”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 729–730.

[375] *SLA for Virtual Machines*. `https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_9/`.

[376] Fedor Smirnov, Behnaz Pourmohseni, and Thomas Fahringer. “Apollo: Modular and Distributed Runtime System for Serverless Function Compositions on Cloud, Edge, and Iot Resources”. In: *Proceedings of the 1st Workshop on High Performance Serverless Computing*. 2020, pp. 5–8.

[377] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[378] *Snowflake: Understanding Billing for Serverless Features*. `https://docs.snowflake.com/en/user-guide/admin-serverless-billing.html`.

[379] Richard Soref. “The Past, Present, and Future of Silicon Photonics”. In: *IEEE Journal of Selected Topics in Quantum Electronics* 12.6 (2006), pp. 1678–1687.

[380] Michael Specter and J. Alex Halderman. “Security Analysis of the Democracy Live Online Voting System”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

[381] Josef Spillner, Cristian Mateos, and David A. Monge. “Faaster, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC”. In: *Latin American High Performance Computing Conference*. Springer. 2017, pp. 154–168.

[382] *SQLite*. https://sqlite.org/.

[383] Vikram Sreekanti et al. “A Fault-Tolerance Shim for Serverless Computing”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[384] Vikram Sreekanti et al. “Cloudburst: Stateful Functions-as-a-Service”. In: *VLDB* 13.11 (2020), pp. 2438–2452.

[385] Nate Stewart. *CockroachDB Serverless: Build What You Dream, Never Worry About Your Database Again*. https://www.cockroachlabs.com/blog/announcing-cockroachdb-serverless/. 2021.

[386] Ion Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[387] Michael Stonebraker. “SQL Databases v. NoSQL Databases”. In: *Communications of the ACM* 53.4 (2010), pp. 10–11.

[388] Michael Stonebraker and Lawrence A. Rowe. *The Design of POSTGRES*. Vol. 15. 2. ACM, 1986.

[389] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. “NoSQL Databases”. In: *Lecture Notes, Stuttgart Media University* 20 (2011), p. 24.

[390] Chen Sun et al. “Single-Chip Microprocessor That Communicates Directly Using Light”. In: *Nature* 528.7583 (Dec. 2015), pp. 534–538. ISSN: 0028-0836. URL: http://dx.doi.org/10.1038/nature16454%20http://www.nature.com/nature/journal/v528/n7583/abs/nature16454.html%7B%5C#%7Dsupplementary-information.

[391] Amoghvarsha Suresh and Anshul Gandhi. “FnSched: An Efficient Scheduler for Serverless Functions”. In: *Proceedings of the 5th International Workshop on Serverless Computing*. 2019, pp. 19–24.

[392] *Syntax Overview - Workflows - Google Cloud*. https://cloud.google.com/workflows/docs/reference/syntax.

[393] Chiu C. Tan, Bo Sheng, and Qun Li. “Secure and Serverless RFID Authentication and Search Protocols”. In: *IEEE Transactions on Wireless Communications* 7.4 (2008), pp. 1400–1407.

[394] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015.

[395] Vasily Tarasov, Erez Zadok, and Spencer Shepler. “Filebench: A flexible framework for file system benchmarking”. In: *Login: The USENIX Magazine* 41.1 (2016), pp. 6–12.

[396] Ali Tariq et al. “Sequoia: Enabling Quality-of-Service in Serverless Computing”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 311–327.

[397] Daniel Terdiman. *Jack Dorsey: Twitter Was Built in Two Weeks.* `https://www.cnet.com/news/jack-dorsey-twitter-was-built-in-two-weeks/`. Nov. 2012.

[398] Douglas B. Terry et al. “Managing update conflicts in Bayou, a weakly connected replicated storage system”. In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 172–182.

[399] *The System Call Intercepting Library.* `https://github.com/pmem/syscall_intercept`.

[400] Thomas N. Theis and H-S Philip Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50.

[401] Robert H. Thomas. “A Resource Sharing Executive for the ARPANET”. In: *Proceedings of the June 4-8, 1973, national computer conference and exposition.* 1973, pp. 155–163.

[402] Shelby Thomas et al. “Particle: Ephemeral Endpoints for Serverless Networking”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing.* 2020, pp. 16–29.

[403] Alexander Thomson and Daniel J. Abadi. “CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems”. In: *13th USENIX Conference on File and Storage Technologies (FAST 15).* 2015, pp. 1–14.

[404] Alexander Thomson et al. “Calvin: Fast Distributed Transactions for Partitioned Database Systems”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 2012, pp. 1–12.

[405] David Thomson et al. “Roadmap on Silicon Photonics”. In: *Journal of Optics* 18.7 (2016), p. 073003.

[406] *Time sync for Linux VMs in Azure.* `https://docs.microsoft.com/en-us/azure/virtual-machines/linux/time-sync`. 2022.

[407] Ankit Toshniwal et al. “Storm @Twitter”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* 2014, pp. 147–156.

[408] *TPC-C.* `http://www.tpc.org/tpcc/`.

[409] Bohdan Trach et al. “Clemmys: Towards Secure Remote Execution in FaaS”. In: *Proceedings of the 12th ACM International Conference on Systems and Storage.* 2019, pp. 44–54.

[410] *Transactional NTFS (TxF).* `https://docs.microsoft.com/en-us/windows/win32/fileio/transactional-ntfs-portal`.

[411] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. “Simultaneous Multithreading: Maximizing On-Chip Parallelism”. In: *Proceedings of the 22nd annual international symposium on Computer architecture.* 1995, pp. 392–403.

[412] *Twitter Form S-1.* `https://www.sec.gov/Archives/edgar/data/1418091/000119312513390321/d564001ds1.htm`. Oct. 2013.

[413] *Twitter Open Source.* `https://twitter.github.io/projects/`.

[414] Amin Vahdat. *The Past, Present and Future of Custom Compute at Google.* `https://cloud.google.com/blog/topics/systems/the-past-present-and-future-of-custom-compute-at-google`.

[415] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[416] Erwin Van Eyk et al. "Serverless Is More: From PaaS to Present Cloud Computing". In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17.

[417] Erwin Van Eyk et al. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18.

[418] Shivaram Venkataraman et al. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16).* 2016, pp. 363–378.

[419] Alexandre Verbitski et al. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *Proceedings of the 2017 ACM International Conference on Management of Data.* 2017, pp. 1041–1052.

[420] Rajat Verma et al. "Failure-atomic updates of application data in a Linux file system". In: *13th USENIX Conference on File and Storage Technologies (FAST 15).* 2015, pp. 203–211.

[421] Werner Vogels. *Announcing AWS Lambda.* `https://www.youtube.com/watch?v=9eHoyUVo-yg`. 2014.

[422] Marco Von Arb et al. "Veneta: Serverless Friend-of-Friend Detection in Mobile Social Networking". In: *2008 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications.* IEEE. 2008, pp. 184–189.

[423] Tim A. Wagner. *Serverless Networking is the next step in the evolution of serverless.* `https://read.acloud.guru/https-medium-com-timawagner-serverless-networking-the-next-step-in-serverless-evolution-95bc8adaa904`. 2019.

[424] Timothy A. Wagner. *Getting Started With AWS Lambda.* `https://www.youtube.com/watch?v=UFj27laTWQA`. 2014.

[425] Jim Waldo et al. "A Note on Distributed Computing". In: *International Workshop on Mobile Object Systems.* Springer. 1996, pp. 49–64.

[426] Ao Wang et al. "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache". In: *18th USENIX Conference on File and Storage Technologies (FAST 20).* 2020, pp. 267–281.

[427] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. “LaSS: Running Latency Sensitive Serverless Computations at the Edge”. In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 2020, pp. 239–251.

[428] Feiyi Wang et al. “Understanding Lustre Filesystem Internals”. In: *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* (2009).

[429] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 133–146.

[430] Randolph Y. Wang and Thomas E. Anderson. “xFS: A Wide Area Mass Storage File System”. In: *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*. IEEE. 1993, pp. 71–78.

[431] Stephanie Wang et al. “Lineage Stash: Fault Tolerance Off the Critical Path”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 338–352.

[432] Mike Wawrzoniak et al. “Boxer: Data Analytics on Network-enabled Serverless Platforms”. In: *11th Annual Conference on Innovative Data Systems Research (CIDR’21)*. 2021.

[433] Sage A. Weil et al. “Ceph: A scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[434] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An architecture for well-conditioned, scalable internet services”. In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 230–243.

[435] Sebastian Werner, Richard Girke, and Jörn Kuhlenkamp. “An Evaluation of Serverless Data Processing Frameworks”. In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 2020, pp. 19–24.

[436] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. “Automated Empirical Optimizations of Software and the ATLAS Project”. In: *Parallel Computing* 27.1-2 (2001), pp. 3–35.

[437] Michael Whittaker et al. “Scaling Replicated State Machines With Compartmentalization”. In: *Proceedings of the VLDB Endowment* 14.11 (2021), pp. 2203–2215.

[438] Bruce Wile. *Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems*. Tech. rep. IBM Systems and Technology Group, Sept. 2014.

[439] Stefan Winzinger and Guido Wirtz. “Model-Based Analysis of Serverless Applications”. In: *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE. 2019, pp. 82–88.

[440] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. "Multiprocessor System-on-Chip (MPSoC) Technology". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1701–1713.

[441] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna". In: *Proceedings of the VLDB Endowment* 12 (2019).

[442] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Transactional Causal Consistency for Serverless Computing". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 83–97.

[443] Chenggang Wu et al. "Anna: A KVS for Any Scale". In: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[444] Junjie Xiong et al. "Warmonger: Inflicting Denial-of-Service via Serverless Functions in the Cloud". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 955–969.

[445] Neeraja J. Yadwadkar et al. "Selecting the Best VM Across Multiple Public Clouds: A Data-Driven Performance Modeling Approach". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 452–465.

[446] Mengting Yan et al. "Building a Chatbot With Serverless Computing". In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. 2016, pp. 1–4.

[447] Ryan Yang et al. *PyPlover: A System for GPU-Enabled Serverless Instances*. Tech. rep. Technical report, University of California, Berkeley, 2020.

[448] Mihalis Yannakakis. "Serializability by Locking". In: *Journal of the ACM (JACM)* 31.2 (1984), pp. 227–244.

[449] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.

[450] Minchen Yu et al. "Gillis: Serving Large Neural Networks in Serverless Functions With Automatic Model Partitioning". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 138–148.

[451] Tianyi Yu et al. "Characterizing Serverless Platforms With Serverlessbench". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 30–44.

[452] Xiangyao Yu and Srinivas Devadas. "Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 227–240.

[453] Xiangyao Yu, Muralidaran Vijayaraghavan, and Srinivas Devadas. "A Proof of Correctness for the Tardis Cache Coherence Protocol". In: *arXiv preprint arXiv:1505.06459* (2015).

[454] Xiangyao Yu et al. "Staring Into the Abyss: An Evaluation of Concurrency Control With One Thousand Cores". In: (2014).

[455] Xiangyao Yu et al. "Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1289–1302.

[456] Vladimir Yussupov et al. "Serverless Parachutes: Preparing Chosen Functionalities for Exceptional Workloads". In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE. 2019, pp. 226–235.

[457] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[458] *Zappa: Serverless Python*. `https://github.com/Miserlou/Zappa`.

[459] Haoran Zhang et al. "Fault-tolerant and transactional stateful serverless workflows". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 1187–1204.

[460] Hong Zhang et al. "Caerus: NIMBLE Task Scheduling for Serverless Analytics." In: *NSDI*. 2021, pp. 653–669.

[461] Lu Zhang et al. "Tapping Into NFV Environment for Opportunistic Serverless Edge Function Deployment". In: *IEEE Transactions on Computers* (2021).

[462] Tian Zhang et al. "Narrowing the Gap Between Serverless and Its State With Storage Functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 1–12.

[463] Wen Zhang et al. "Kappa: A Programming Framework for Serverless Computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 328–343.

[464] Yanqi Zhang et al. "Faster and Cheaper Serverless Computing on Harvested Resources". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 724–739.

[465] Siyuan Zhuang et al. "Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 641–656.