The Design of Any-scale Serverless Infrastructure with Rich Consistency Guarantees

Chenggang Wu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-204
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-204.html

December 17, 2020

The Design of Any-scale Serverless Infrastructure with Rich Consistency Guarantees

by

Chenggang Wu

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 Joseph E. Gonzalez
Professor Barna Saha

Fall 2020

The Design of Any-scale Serverless Infrastructure with Rich Consistency Guarantees

Abstract

The Design of Any-scale Serverless Infrastructure with Rich Consistency Guarantees

by

Chenggang Wu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Serverless computing has gained significant attention over the last few years. The core advantage of serverless is that it abstracts away low-level operational concerns and opens up opportunities for developers without sophisticated systems expertise to harness the power of the cloud and build scalable distributed applications. However, there are three main challenges that limit the capability and prevent the adoption of serverless infrastructure. First, serverless systems need to deliver high performance at any scale, from a single multicore machine to a geo-distributed deployment. Second, serverless systems need to dynamically respond to workload shifts and autoscale to meet performance goals while minimizing cost. Third, serverless systems need to offer robust consistency guarantees to support a wide variety of applications while maintaining high performance.

This dissertation presents a line of research that addresses these challenges. We first introduce Anna, a high-performance key-value store that employs a lattice-based coordination-free execution model. The design of Anna achieves seamless scaling while offering rich consistency guarantees. We then discuss how we extend Anna to become a serverless, tiered storage system. Anna's autoscaling mechanisms and policies enable intelligent trade-off between latency and cost under dynamic workloads. Finally, we present HydroCache, a caching layer that sits in between a function-as-a-service platform and the underlying storage system. HydroCache maintains the benefit of resource disaggregation offered by existing serverless computing platforms while delivering low-latency request handling and transactional causal consistency, the strongest consistency model that can be achieved without coordination.

To my parents

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation would not have been possible without the advice, collaboration, and friendship of a great many people. First and foremost, I am extremely blessed to have Joe Hellerstein as my advisor. When I was new to research, Joe was patient, nurturing, and hands-on; he worked closely with me throughout every stage of a research project, from idea generation to implementation. At the same time, Joe always valued my research interests and encouraged me to discover novel problems as opposed to working on existing ones. This process made me an independent researcher and gave me confidence to succeed. Another lesson learned from Joe was the importance of communication, either in the form of writing or presentation. I remember him relentlessly polishing the introductory paragraph of our papers, and I realized for the same technical content, the way we frame the problem and elaborate on the contributions can make a huge difference in how the work is received by the audience. I will take this lesson to heart and apply it throughout the rest of my career.

I am also thankful to my dissertation and qualification committee members Joseph Gonzalez, Ion Stoica, and Barna Saha; their feedback has been vital in improving this dissertation.

Vikram Sreekanti was my primary collaborator in grad school, and we partnered on almost every project in the last four years. We spent many late nights sprinting for paper deadlines together, and I really appreciate his help, advice, and friendship throughout. I am deeply impressed with his ability to search for novel yet simple solutions, his engineering discipline, and his clear communication. Having a strong collaborator like Vikram boosted my confidence when facing challenges and motivated me to constantly improve myself.

I would also like to express gratitude to other collaborators who played a key role in my PhD research: Michael Whittaker, Zongheng Yang, Johann Schleier-Smith, Jose Faleiro, Charles Lin, Alexey Tumanov, Yifan Wu, Jeff Ichnowski. In addition, a big shout-out to the staff member of RISELab: Kattt, Boban, Shane, Jon, and Jey, for their exceptional service.

Moreover, I want to thank Alekh Jindal, my mentor during the internship at Microsoft, for his guidance on conducting applied research in industry. I would also like to thank Stan Zdonik, Tim Kraska, and Ugur Cetintemel for opening the door to research for me when I was an undergrad at Brown University.

Beyond direct collaborators on research projects, many colleagues and friends have made my PhD journey a wonderful experience by celebrating paper deadlines, sympathizing over paper reviews, cheering me up during difficult times, and more. Xin, Eyal, Moustafa, Nathan, Gabe, Doris (Lee and Xin), Rishabh, Wenting, Richard, Rolando, Anurag, Devin, Jeongseok, Jenny, Hezheng, Cecilia, Danyang, Hong, Jackson, Raghav, Debbie, Yuhong, Kaushik, Mira, Jiayu, thank you for everything. Special thanks to my cat, Olie, for your company during COVID-19.

Last but not least, this dissertation is dedicated to my parents, Jialin Wu and Jie Cai, for their understanding, encouragement, support, and love.

Chapter 1

Introduction

Over the past decade, innovations in cloud computing have transformed how we build large-scale applications. Cloud providers such as Amazon Web Services [13] (AWS), Google Cloud [41], and Microsoft Azure [15], use hardware virtualization technology [21] to provide application developers with a virtual machine (VM) with access to CPUs, RAM, and disks. This has revolutionized the computing landscape, allowing organizations that previously had no access to large computing clusters to deploy and scale distributed applications. Indeed, the vast majority of today's large-scale applications, from scientific computing projects to machine learning pipelines, are run in the cloud. Moreover, Developer Operations (or "DevOps") tools are now widely used to maintain large computing clusters. Kubernetes [55], for example, is a container orchestration tool for deploying and scaling clusters. It hides the complexity of managing node membership, transparently handles node failures via heartbeats and automatic restarts, and offers state-of-the-art security and performance isolation across computing units. With Kubernetes, developers can now more easily maintain clusters with thousands of nodes.

As cloud technologies continue to evolve, serverless computing systems have emerged to bring cloud computing to the next level [44, 49, 5, 45, 77, 94]. Serverless computing is a new software design pattern with two core advantages. First, it offers a higher-level abstraction to program the cloud. Function-as-a-service (FaaS) platforms such as AWS Lambda, Google Cloud Functions, and Azure Functions allow programmers to write application code locally and upload the code to the platform without having to worry about resource configurations or the systems environment under which the applications are run. FaaS platforms then serve the application code in response to a wide variety of trigger events. AWS Lambda, for example, support triggers including events from API Gateway, S3, and DynamoDB. In contrast, in the pre-serverless world, to run an application in the cloud, a common practice was to first use containerization tools such as Docker to build an image consisting of the application as well as the appropriate versions of the operating system, author YAML configuration files specifying the resource requirements (e.g., CPUs, GPUS, RAM), and finally launch the containers with Kubernetes. Serverless computing abstracts away these complexities and lets developers focus on the application logic. This enables programmers without DevOps expertise to harness the power of the cloud and build scalable distributed applications.

The other key advantage of serverless is usage-based pricing; FaaS platforms such as AWS

Lambda charge based on compute time at the granularity of milliseconds for each function invocation. Similarly, serverless storage systems such as AWS DynamoDB bill users based on the storage consumption of the datasets. Usage-based pricing is attractive for both developers and the cloud providers. It offers developers peace of mind by eliminating the burden of manual deallocation of resources when the application is not being run. Cloud providers, on the other hand, can efficiently pack multiple customers' workloads on the same VM for resource multiplexing. The resulting fine-grained resource allocation reduces resource reservation during idle periods, leading to significant cost savings.

1.1 The Challenges

Despite the promises, today's serverless systems face three core challenges. First, modern cloud providers offer dense hardware with multiple cores and large memories, hosted in global platforms. Amazon, for example, now offers EC2 instances with up to 64 physical CPU cores and gigabytes of RAM. Conventional wisdom says that software designed for one scale point needs to be rewritten when scaling up by 10-100× [32]. However, due to the elastic scaling requirement of serverless systems, they need to scale seamlessly from a single core to multicore to the globe. Therefore, the challenge is to design a unified execution model that enables a system to deliver excellent performance at any scale, from a single multicore machine to a geo-distributed deployment.

Second, each of today's serverless systems offers a fixed trade-off of performance and cost. Taking Amazon's cloud storage systems as examples, S3 is a relatively economical option but it is not designed for low latency. On the other end of the spectrum, ElastiCache, despite achieving high performance, is very costly and cumbersome to scale. This is undesirable because application workloads dynamically shift over time. As a result, systems either over-allocate resources, leading to idle periods and increased billing, or under-allocate resources, leading to performance SLO violation. To achieve cost-efficiency, it is crucial to implement fine-grained autoscaling mechanisms and policies that optimize resource allocation to flexibly support user-defined goals in performance and cost budget.

Finally, today's serverless computing infrastructure employs a disaggregated architecture between the compute layer and the storage layer. This enables independent scaling of the two layers, leading to improved resource efficiency. However, this design comes with a cost: the compute layer needs to cross the network boundary to access the data in the remote storage. For data-intensive applications, the added network roundtrip introduces significant latency, hampering the performance of the system. Moreover, existing storage backends such as DynamoDB and ElastiCache offer limited consistency guarantees [114], leading to increased consistency anomalies observed within the application. This calls for a novel redesign that keeps the resource disaggregation benefit of existing serverless platforms while simultaneously achieving low latency and robust consistency guarantees.

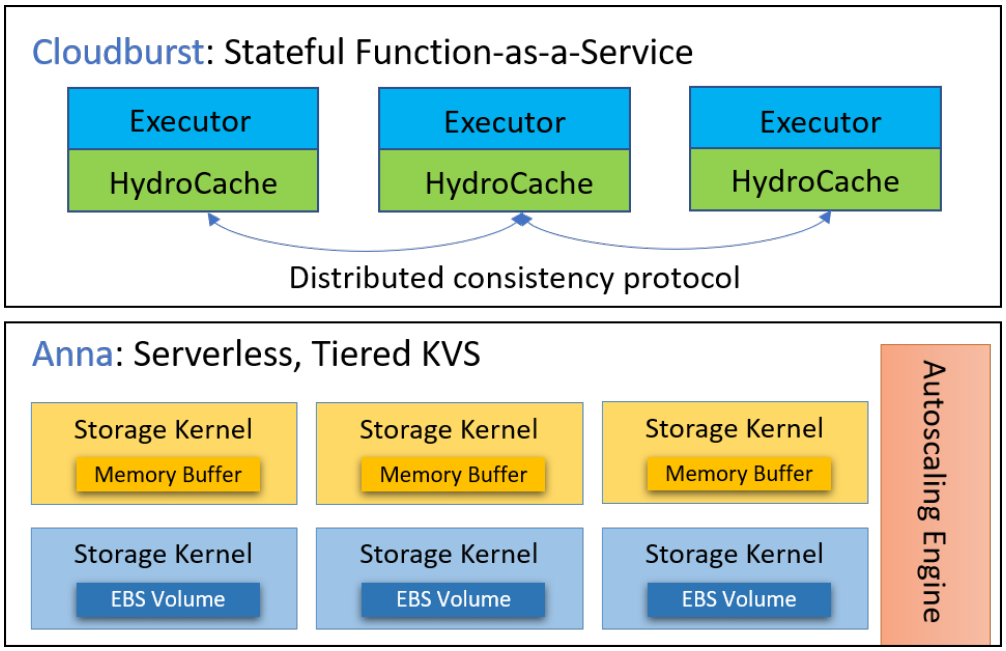

Figure 1.1: An overview of the systems discussed in this dissertation: a serverless key-value store Anna, and a stateful function-as-a-service platform Cloudburst built on top of Anna.

1.2 Towards Any-scale Serverless Infrastructure with Rich Consistency Guarantees

In this dissertation, we discuss a line of research projects that address these challenges. Figure 1.1 shows a high-level overview of the systems we built. In Chapter 2, we study the design principles of building an any-scale, high-performance system in the context of a key-value store (KVS) called Anna [114]. Anna is a partitioned, multi-mastered system that achieves high performance and elasticity via wait-free execution, which eliminates thread synchronization overhead within a single machine by letting each thread access private memory and exchange information via explicit message passing instead of via shared memory. As a result, this design efficiently exploits multicore parallelism by allowing each thread to focus on request handling instead of dealing with locking or atomic instruction overheads. Anna's multi-master replication also enables different replicas to accept updates concurrently, making the system suitable for handling both read-intensive and write intensive workloads. In addition, the wait-free execution model is extensible across different scale points; Anna employs the same execution model both within a single machine and across multiple machines, significantly reducing the system complexity.

However, although the wait-free execution model achieves excellent performance and smooth scaling, it introduces a new challenge in data consistency; because updates are propagated across threads asynchronously, the same set of updates may arrive at each thread in different orders.

To solve this challenge, Anna encapsulates system state in lattice data structures, which accept updates in a way that is associative, commutative, and idempotent. This allows data replicas to converge to the same state even under network message reordering and duplication. Moreover, Anna provides native support for a wide variety of coordination-free consistency guarantees via lattice composition, and allows programmers to register user-defined lattices to meet application-specific consistency requirements. The resulting system outperforms state-of-the-art alternatives such as Redis and DynamoDB by up to 100x. The diverse set of consistency models also allows a wide range of applications, from caching services to online shopping carts to large-scale indexing applications, to take advantage of the system.

In Chapter 3, we describe how we extended Anna into a serverless storage system for the cloud [112]. In its extended form, Anna is designed to overcome the narrow cost-performance limitations typical of current cloud storage systems. We describe three key aspects of Anna's new design. First, Anna monitors data access frequency and selectively replicates hot keys to spread the load across multiple nodes. Second, Anna employs a vertical tiering of storage layers with different cost-performance tradeoffs. Anna consists of two tiers: a memory tier where the data is maintained in an in-memory hash table, and a disk tier where the data is kept in EBS volumes. Anna promotes hot keys to the memory tier for high performance and demotes cold keys to the disk tier for cost savings. Third, Anna tracks incoming request volumes and horizontally scales the system by adding and removing storage nodes. The two tiers of Anna scale independently, achieving maximum flexibility in cost-performance trade-offs.

In addition to the new autoscaling mechanisms, Anna implements policies to balance service-level objectives (SLOs) for cost, latency and fault tolerance. Given a fault-tolerance requirement (e.g., every key must be replicated across at least 3 nodes), when a latency SLO is given, Anna's policy strives to meet the SLO with a system configuration that incurs the minimum cost. When the cost budget is specified, Anna attempts to minimize request latency while keeping the system's cost below the budget. Our experimental results explore the behavior of Anna's mechanisms and policy, exhibiting orders-of-magnitude efficiency improvements over both commodity cloud KVS services and research systems.

Finally, in Chapter 4, we turn our attention to the setting of serverless FaaS platforms, where storage services are disaggregated from the machines that support function execution. FaaS applications consist of compositions of functions, each of which may run on a separate machine and access remote storage. We address the challenge of improving I/O latency in this setting while also providing application-wide consistency. Previous work has explored providing causal consistency for individual I/Os by carefully managing the versions stored in a client-side data cache. In our setting, a single application may execute multiple functions across different nodes, and therefore issue interrelated I/Os to multiple distinct caches. This raises the challenge of Multisite Transactional Causal Consistency (MTCC): the ability to provide causal consistency for all I/Os within a given transaction even if it runs across multiple physical sites. We present distributed protocols for MTCC implemented in a system called HydroCache [113], a caching layer co-located with the function execution layer. We also discuss how different variants of the protocol can be combined to simultaneously minimize transaction aborts and coordination overheads. Our evaluation demonstrates orders-of-magnitude performance improvements due to caching, while also

protecting against consistency anomalies that otherwise arise frequently.

In summary, the contribution of this dissertation are as follows:

- Demonstrating that wait-free execution plus coordination-free consistency levels are the key design principles to unlocking systems that deliver excellent performance at any scale.
- Introducing fine-grained autoscaling mechanisms that detect and respond swiftly to workload shifts, as well as intelligent policies that perform goal-oriented optimization to achieve cost-efficiency in a serverless KVS.
- Showing that resource disaggregation, low-latency data access, and robust consistency guarantees can be achieved simultaneously in a FaaS system.

We believe this level of improved performance, scalability, and consistency will enable serverless computing platforms to support a broader spectrum of mission-critical applications. The expanded use cases will motivate new design requirements that drive future research, eventually making serverless platforms the *de facto* infrastructure for running general distributed applications.

Chapter 2

Anna: A KVS For Any Scale

As we discussed, delivering high performance at any scale is crucial for serverless systems. In this chapter, we explore the design principles that achieve this property in the context of KVS systems. High-performance KVS systems are the backbone of many large-scale applications ranging from retail shopping carts to machine learning parameter servers. Many KVS systems are designed for large-scale and even globally-distributed usage (e.g., [33, 12, 76]); others are designed for high-performance single-machine settings (e.g., [75, 86]). In recent years, these distinct hardware targets have begun to converge in the cloud. For example, Amazon now offers EC2 instances with up to 64 physical cores, while continuing to provide the ability to scale across machines, racks and the globe.

Given the convergence of dense hardware and the globe-spanning cloud, we set out to design a KVS that can run well at any scale: providing excellent performance on a single multi-core machine, while scaling up elastically to geo-distributed cloud deployment. In addition to wide-ranging architectural flexibility, we wanted to provide a wide range of consistency semantics as well, to support a variety of application needs.

In order to achieve these goals, we found that four design requirements emerged naturally. The first two are traditional aspects of global-scale data systems. To ensure data scaling, we assumed from the outset that we need to ***partition*** (shard) the key space, not only across nodes at cloud scale but also across cores for high performance. Second, to enable workload scaling, we need to employ ***multi-master replication*** to concurrently serve puts and gets against a single key from multiple threads.

The next two design requirements followed from our ambitions for performance and generality. To achieve maximum hardware utilization and performance within a multi-core machine, our third requirement was to guarantee ***wait-free execution***, meaning that each thread is always doing useful work (serving requests), and never waiting for other threads for reasons of consistency or semantics. To that end, coordination techniques such as locking, consensus protocols or even "lock-free" retries [38] need to be avoided. Finally, to support a wide range of application semantics without compromising our other goals, we require a unified implementation for a wide range of ***coordination-free consistency models*** [19].

Given these design constraints, we developed a system called Anna[1], which meets our performance goals at multiple scales and consistency levels. The architecture of Anna is based on a simple design pattern of coordination-free actors (Section 2.4), each having private memory and a thread of execution mapped to a single core. Actors communicate explicitly via messaging, be it across nodes (via a network) or within a multi-core machine (via message queues [22]). To ensure wait-free execution, Anna actors never coordinate; they only communicate with each other to lazily exchange updates, or repartition state. Finally, as discussed in Section 2.5, Anna provides replica consistency in a new way: by using lattice composition to implement recent research in coordination-free consistency. This design pattern is uniform across threads, machines, and data-centers, which leads to a system that is simple and easy to reconfigure dynamically.

We describe the design and implementation of Anna, providing a set of architectural and experimental lessons for designing across scales:

- **Coordination-free Actors:** We confirm that the coordination-free actor model provides excellent performance from individual multi-core machines up to widely distributed settings, besting state-of-the-art lock-free shared memory implementations while scaling smoothly and making repartitioning for elasticity extremely responsive.
- **Lattice-Powered, Coordination-Free Consistency:** We show that the full range of coordination-free consistency models taxonomized by Bailis, et al. [19] can be elegantly implemented in the framework of distributed lattices [95, 27], using only very simple structures in a compositional fashion. The resulting consistency code is small and modular: each of our consistency levels differs by at most 60 lines of C++ code from our baseline.
- **Cross-Scale Validation:** We perform comparison against popular KVSes designed for different scale points: Redis [86] for single-node settings, and Apache Cassandra [12] for geo-replicated settings. We see that Anna's performance is competitive at both scales while offering a wider range of consistency levels.

2.1 Related Work

Anna is differentiated from the many KVS designs in the literature in its assumptions and hence in its design. Anna was inspired by a variety of work in distributed and parallel programming, distributed and parallel databases, and distributed consistency.

2.1.1 Programming Models

The Coordination-free actor model can be viewed as an extension to distributed event-loop programming, notably Hewitt's Actor model [46], more recently popularized in Erlang and Akka. Anna follows the Actor spirit of independent local agents communicating asynchronously, but differs from Actors in its use of monotonic programming in the style of Bloom [8] and CRDTs [95],

[1]The tiny Anna's hummingbird, a native of California, is the fastest animal on earth relative to its size [26].

System	Scale	Memory Model
Masstree	M	SM
Bw-tree	M	SM
PALM	M	SM
MICA	M	SM
Redis	S	N/A
COPS, Bolt-on	D	MP
Bayou	D	MP
Dynamo	D	MP
Cassandra	D	MP
PNUTS	D	MP
CouchDB	D	MP
Voldemort	D	MP
HBase	D	MP
Riak	D	MP
DocumentDB	D	MP
Memcached	M & D	SM & MP
MongoDB	M & D	SM & MP
H-Store	M & D	MP
ScyllaDB	M & D	MP
Anna	**M & D**	**MP**

Table 2.1: Taxonomy of existing KVS systems. The scale column indicates whether a system is designed to run on a Single core (S), a multi-core machine (M), in a distributed setting (D), or a combination (M & D). The memory model column shows whether a system uses shared-memory model (SM), explicit message passing (MP), or both (SM & MP).

providing a formal foundation for reasoning about distributed consistency. Anna's actors also bear some resemblance to SEDA [110], but SEDA focuses on preemptable thread pools and message queues, whereas Anna's actors target a thread-per-core model with lattices to ensure consistency and performance.

Recent systems, such as ReactDB [93] and Orleans [23] also explore Actor-oriented programming models for distributed data. In both those cases, the Actor model is extended to provide a higher level abstraction as part of a novel programming paradigm for users. By contrast, Anna does not attempt to change user APIs or programming models; it exposes a simple key/value API to external applications. Meanwhile, those systems do not explore the use of lattice-oriented actors.

2.1.2 Key-value Stores

Table 2.1 and 2.2 show a taxonomy of existing KVS systems based on the scale at which they are designed to operate, the memory model, and the per-key as well as multi-key consistency levels

System	Per-Key Consistency	Multi-key Consistency
Masstree	Linearizable	None
Bw-tree	Linearizable	None
PALM	Linearizable	None
MICA	Linearizable	None
Redis	Linearizable	Serializable
COPS, Bolt-on	Causal	Causal
Bayou	Eventual, Monotonic Reads/Writes, Read Your Writes	Eventual
Dynamo	Linearizable, Eventual	None
Cassandra	Linearizable, Eventual	None
PNUTS	Linearizable Writes, Monotonic Reads	None
CouchDB	Eventual	None
Voldemort	Linearizable, Eventual	None
HBase	Linearizable	None
Riak	Eventual	None
DocumentDB	Eventual, Session, Bounded Staleness, Linearizability	None
Memcached	Linearizable	None
MongoDB	Linearizable	None
H-Store	Linearizable	Serializable
ScyllaDB	Linearizable, Eventual	None
Anna	**Eventual, Causal, Item Cut, Writes Follow Reads Monotonic Reads/Writes, Read Your Writes, PRAM**	**Read Committed Read Uncommitted**

Table 2.2: Consistency models offered by existing KVS systems.

supported. The remainder of this section discusses the state of the art in KVS systems in the context of the four design requirements (introduced at the beginning of this chapter) for building any-scale KVS.

Single-server storage systems

Most single-server KVS systems today are designed to efficiently exploit multi-core parallelism. These multi-core-optimized KVS systems typically guarantee that reads and writes against a single key are linearizable.

Shared memory is the architecture of choice for most single-server KVS systems. Masstree [73] and Bw-tree [63] employ a shared-memory design. Furthermore, the single-server mechanisms within distributed KVS systems, such as memcached [75] and MongoDB [76], also employ a shared-memory architecture per node. Shared-memory architectures use synchronization mechanisms such as latches or atomic instructions to protect the integrity of shared data-structures, which can significantly inhibit multi-core scalability under contention [38].

PALM [92] and MICA[66][2] each employ a partitioned architecture, assigning non-overlapping shards of key-value pairs to each system thread. KVS operations can therefore be performed without any synchronization because they are race-free by default. However, threads in partitioned systems (with ***single-master*** request handling) are prone to under-utilization if a subset of shards receive a disproportionate fraction of requests due to workload skew. To address workload skew, both PALM and MICA make selective use of shared-memory design principles. For instance, MICA processes only writes in a partitioned fashion, but allows any thread to process reads against a particular key.

Redis [86] uses a single-threaded model. Redis permits operations on multiple keys in a single request and guarantees serializability. While single-threaded execution avoids shared-memory synchronization overheads, it cannot take any advantage of multi-core parallelism.

The systems above are carefully designed to execute efficiently on a single server. Except for Redis, they all use shared-memory accesses; some directly employ the shared-memory architecture, while others employ a partitioned ("shared-nothing") architecture but selectively exploit shared memory to ameliorate skew. The designs of these systems are therefore specific to a single server, and cannot be generalized to a distributed system. Moreover, the shared-memory model is at odds with ***wait-free execution*** (Section 2.3), and therefore does not meet our performance requirement for any-scale KVS.

Moreover, as noted in Figure 2.2, prior single-node KVS systems invariably provide only a single form of consistency; typically either linearizability or serializability. Furthermore, with the exception of Redis, which is single-threaded, none of the single-node KVS systems provide any consistency guarantees for multi-key operations for groups of keys. Hence, these systems choose a different design point than we explore: they offer strong consistency at the expense of performance and scalability.

Distributed KVS

As shown in Figure 2.1, the majority of distributed KVS systems are not designed to run on a single multi-core machine, and it is unclear how they exploit multi-core parallelism (if at all). The exceptions are H-Store [51] and ScyllaDB [90]. Within a single machine, these systems partition the key-value index across threads, which communicate via explicit message-passing. However, as discussed earlier, partitioned systems with single-master request handling cannot scale well under skewed workload.

In terms of consistency, most distributed KVSes support a single, relaxed consistency level. COPS [67] and Bolt-on [17] guarantee causal consistency. MongoDB [76], HBase [43], and memcached [75] guarantee linearizable reads and writes against individual KVS objects. PNUTS [29] guarantees that writes are linearizable, and reads observe a monotonically increasing set of updates to key-value pairs.

Bayou [105] provides eventually consistent multi-key operations, and supports application-specific conflict detection and resolution mechanisms. Cassandra [12] and Dynamo [33] use

[2]Note that MICA is a key-value *cache*, and can hence evict key-value pairs from an index in order to bound its memory footprint for improved cache-locality.

quorum-based replication to provide different consistency levels. Applications can fix read and write quorum sizes to obtain either linearizable or eventually consistent single-key operations. In addition, both Cassandra and Dynamo use vector clocks to detect conflicting updates to a key, and permit application-specific conflict resolution policies. As noted in Figure 2.2, Azure DocumentDB [16] supports multiple single-key consistency levels.

We note that the majority of distributed KVS systems do not provide any multi-key guarantees for arbitrary *groups* of keys. Some systems, such as HBase, provide limited support for transactions on single shard, but do not provide arbitrary multi-key guarantees. COPS and Bolt-on provide causally consistent replication. Bayou supports arbitrary multi-key operations but requires that each server maintains a full copy of the entire KVS. H-Store supports serializability by performing two-phase commit. However, achieving this level of consistency requires coordination and waiting amongst threads and machines, leading to limited scalability.

State machine replication [89] (SMR) is the de facto standard for maintaining strong consistency in replicated systems. SMR maintains consistency by enforcing that replicas deterministically process requests according to a total order (via a consensus protocol such as Paxos [60] or Raft [78]). Totally ordered request processing requires waiting for global consensus at each step, and thus fundamentally limits the throughput of each replica-set. Anna, in contrast, uses lattice composition to maintain the consistency of replicated state. Lattices are resilient to message reordering and duplication, allowing Anna to employ asynchronous multi-master replication without need for any waiting.

2.2 Lattices

A central component of the design of Anna is its use of lattice composition for storing and asynchronously merging state. Lattices prove important to Anna for two reasons.

First, lattices are insensitive to the order in which they merge updates. This means that they can guarantee consistency across replicas even if the actors managing those replicas receive updates in different orders. Section 2.4 describes Anna's use of lattices for multi-core and wide-area scalability in detail.

Second, we will see in Section 2.5 that simple lattice building blocks can be composed to achieve a range of coordination-free consistency levels. The coordination-freedom of these levels was established in prior work [19], and while they cannot include the strongest forms of consistency such as *linearizability* or *serializability*, they include relatively strong levels including *causality* and *read-committed* transactions. Our contribution is architectural: Anna shows that these many consistency levels can all be *expressed* and implemented using a unified lattice-based foundation. Section 2.5 describes these consistency levels and their implementation in detail.

To clarify terminology, we pause to review the lattice formalisms used in settings like convergent and commutative replicated data-types (CRDTs) [95], and the Bloom^L distributed programming language [27].

A *bounded join semilattice* consists of a domain S (the set of possible states), a binary operator $\sqcup$, and a "bottom" value $\bot$. The operator $\sqcup$ is called the "least upper bound" and satisfies the

following properties:
Commutativity: $\sqcup(a, b) = \sqcup(b, a)\ \forall a, b \in S$
Associativity: $\sqcup(\sqcup(a, b), c) = \sqcup(a, \sqcup(b, c))\ \forall a, b, c \in S$
Idempotence: $\sqcup(a, a) = a\ \forall a \in S$

Together, we refer to these three properties via the acronym *ACI*. The $\sqcup$ operator induces a partial order between elements of S. For any two elements a, b in S, if $\sqcup(a, b) = b$, then we say that b's order is higher than a, i.e. $a \prec b$. The bottom value $\bot$ is defined such that $\forall a \in S$, $\sqcup(a, \bot) = a$; hence it is the smallest element in S. For brevity, we use "lattice" to refer to "bounded join semilattice" and "merge function" to refer to "least upper bound".

2.3 Distributed state model

This section describes Anna's representation and management of state across actors. Each actor maintains state using lattices, but we observe that this is not sufficient to achieve high performance. As we discuss, the potential advantages of lattices can be lost in the high cost of synchronization in shared-memory key-value store architectures. Accordingly, Anna eschews shared-memory state model for one based on asynchronous message-passing.

2.3.1 Limitations of shared-memory

The vast majority of multi-core key-value stores are implemented as shared-memory systems, in which the entirety of the system's state is shared across the threads of a server: each thread is allowed to read or write any part of the state. Conflicting accesess to this state, at the level of reads and writes to memory words, need to be synchronized for correctness. Synchronization prevents concurrent writes from corrupting state, and ensures that reads do not observe the partial effects of in-progress writes. This synchronization typically occurs in the form of locks or lock-free algorithms, and is widely acknowledged as one of the biggest limiters of multi-core scalability. Both locks and lock-free algorithms can severely limit scalability under contention due to the overhead of cache-coherence protocols, which is proportional to the number of physical cores contending on a word in memory [24, 38]. For instance, even a single word in memory incremented via an atomic `fetch-and-add` can be a scalability bottleneck in multi-version database systems that assign transactions monotonically increasing timestamps [37].

Lattices do not change the above discussion; any shared-memory lattice implementation is subject to the same synchronization overheads. On receiving update client requests, actors must update a lattice via its merge function. Although these updates commute at the abstraction of the merge function, threads must synchronize their access to a lattice's in-memory state to avoid corrupting this in-memory state due to concurrent writes. Thus, while lattices' *ACI* properties potentially allow a system to scale regardless of workload, a shared-memory architecture fundamentally limits this potential due to the its reliance on multi-core synchronization mechanisms.

2.3.2 Message-passing

In contrast to using shared memory, a message-passing architecture consists of a collection of actors, each running on a separate CPU core. Each actor maintains private state that is inaccessible to other actors, and runs a tight loop in which it continuously processes client requests and inter-core messages from an input queue. Because an actor can update only its own local state, concurrent modification of shared memory locations is eliminated, which in turn eliminates the need for synchronization.

A message-passing system has two alternatives for managing each key; single-master and multi-master replication.

In single-master replication, each key is assigned to a single actor. This prevents concurrent modifications of the key's value, which in turn guarantees that it will always remain consistent. However, this limits the rate at which the key can be updated to the maximum update rate of a single actor.

In multi-master replication, a key is replicated on multiple actors, each of which can read and update its own local copy. To update a key's value, actors can either engage in coordination to control the global order of updates, or can leave updates uncoordinated. Coordination occurs on the critical path of every request, and achieves the effect of totally-ordered broadcast. Although multiple actors can process updates, totally ordered broadcast ensures that every actor processes the same set of updates in the same order, which is semantically equivalent to single-master replication. In a coordination-free approach, on the other hand, each actor can process a request locally without introducing any inter-actor communication on the critical path. Updates are periodically communicated to other actors when a timer is triggered or when the actor experiences a reduction in request load.

Unlike synchronous multi-master and single-master replication, a coordination-free multi-master scheme could lead to inconsistencies between replicas, because replicas may observe and process messages in different orders. This is where lattices come into play. Lattices avoid inconsistency and guarantee replica convergence via their *ACI* properties, which make them resilient to message reordering and duplication. Anna combines asynchronous multi-master replication with lattice-based state management to remain scalable across both low and high conflict workloads while still guaranteeing consistency.

2.4 Anna Architecture

Figure 2.1 illustrates Anna's architecture on a single server. Each Anna server consists of a collection of independent threads, each of which runs the coordination-free actor model. Each thread is pinned to a unique CPU core, and the number of threads never exceeds the number of available CPU cores. This 1:1 correspondence between threads and cores avoids the overhead of preemption due to oversubscription of CPU cores. Anna's actors share no key-value state; they employ consistent hashing to partition the key-space, and multi-master replication with a tunable replication factor to replicate data partitions across actors. Anna actors engage in epoch-based key exchange

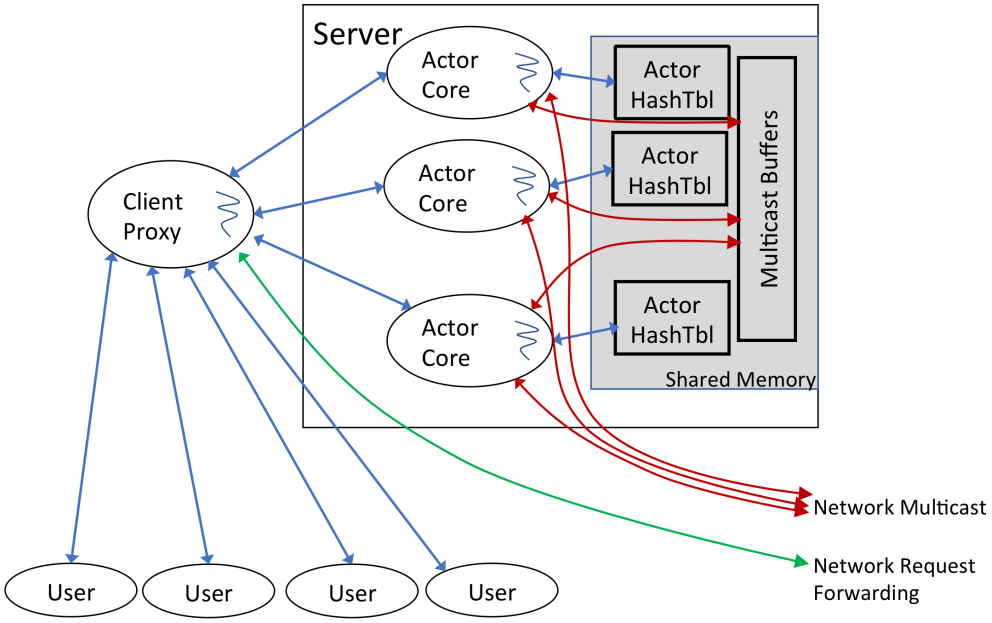

Figure 2.1: Anna's architecture on a single server. Remote users are served by client proxies that balance load across servers and cores. Anna actors run thread-per-core with private hashtable state in shared RAM. Changesets are exchanged across threads by multicasting in memory; exchange across servers is done over the network with protobufs.

to propagate key updates at a given actor to other masters in the key's replication group. Each actor's private state is maintained in a lattice-based data-structure (Section 2.5), which guarantees that an actor's state remains consistent despite message delays, re-ordering, and duplication.

2.4.1 Anna actor event loop

We now discuss Anna's actor event loop and asynchronous multicast in more detail.

Each Anna actor repeatedly checks for incoming requests for puts and gets from client proxies, serves those requests, and appends results to a local *changeset*, which tracks the key-value pair updated within a period of time (the multicast epoch).

At the end of the multicast epoch, each Anna actor multicasts key updates in its changeset to relevant masters responsible for those keys, and clears the changeset. It also checks for incoming multicast messages from other actors, and merges the key-value updates from those messages into its local state. Note that the periodic multicast does not occur on the critical path of request handling.

Anna exploits the associativity of lattices to minimize communication via a *merge-at-sender* scheme. Consider a "hot" key k that receives a sequence of updates $\{u_1, u_2, ..., u_n\}$ in epoch t.

Exchanging all these updates could be expensive in network and computation overhead. However, note that exchanging $\{u_1, u_2, ..., u_n\}$ is equivalent to exchanging just the single merged outcome of these updates, namely $\sqcup(... \sqcup (u_1, u_2), ...u_n)$. Formally, denote *s* as the state of key *k* on another replica, we have

$$\sqcup(... \sqcup (\sqcup(s, u_1), u_2), ...u_n) = \sqcup(s, \sqcup(... \sqcup (u_1, u_2), ...u_n))$$

by associativity. Hence batches of associative updates can be merged at a sending replica without affecting results; merging at the sender can dramatically reduce communication overhead for frequently-updated hot keys, and reduces the amount of computation performed on a receiving replica, which only processes the merged result of updates to a key, as opposed to every individual update.

2.5 Flexible Consistency

As discussed at the beginning of this chapter, high-performance KVSes can benefit a wide range of applications, each of which may vary in its consistency requirements. For example, Amazon's Dynamo shopping cart [33] focuses on supporting causally consistent single-key updates. On the other hand, applications that require multiple writes to succeed atomically need transactional support like *read committed* isolation [19].

Recent research has found that a wide array of consistency levels, including *causal consistency* and *read committed*, can be implemented in a coordination-free fashion [19]. A common requirement for coordination-free consistency levels is convergence: replicas of the same items should converge when they process the same set of messages, regardless of the order in which these messages arrive. This can be achieved by handling client requests and gossip in a way that is ACI (Associative, Commutative, Idempotent).

These properties are attractive, but they are far from trivial to achieve in general-purpose programs. Writing a large system to be ACI – and guaranteeing its correctness – is a difficult challenge.

In this section, we describe how Anna leverages ACI *composition* across small components to achieve a rich set of consistency guarantees—a modular software design pattern derived from the Bloom language [27]. Using ACI composition, we were able for the first time to easily build the full range of coordination-free consistency models [19] from the literature in a single KVS.

2.5.1 ACI Building Blocks

Proposals for ACI systems go back decades, to long-running transaction proposals like Sagas [40], and have recurred in the literature frequently. An ongoing question of the ACI literature was how programmers could achieve and enforce ACI properties in practice. For the Bloom language, Conway et al. proposed the composition of simple lattice-based (ACI) building blocks like counters, maps and pairs, and showed that complex distributed systems could be constructed with ACI properties checkable by induction [27]. Anna adopts Bloom's lattice composition approach. This bottom-up composition has two major advantages: First, in order to verify that a system is ACI, it

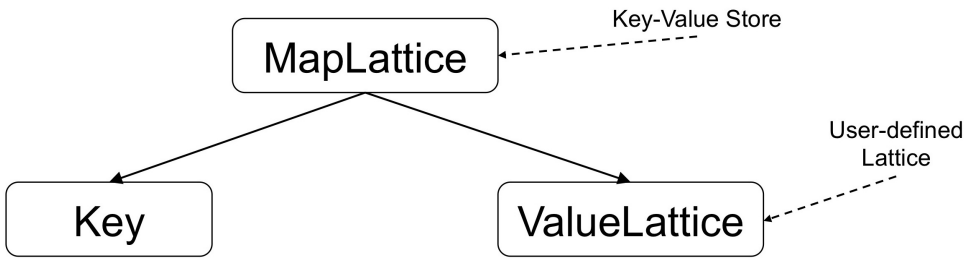

Figure 2.2: A general template for achieving coordination-free consistency

is sufficient to verify that each of its simple building blocks is a valid lattice (has ACI properties), and the composition logic is ACI—this is more reliable than directly verifying ACI for a complex data structure. Second, lattice composition results in modular system design, which allows us to easily figure out which component needs to be modified when maintaining or updating the system.

2.5.2 Anna Lattice Composition

```
1  template <typename K, typename L>
2  class Anna {
3   protected:
4    MapLattice<K, L> kvs;
5   public:
6    V get(const K& k)
7    {
8      return kvs.reveal(k);
9    }
10
11   void put(const K& k, const L& l)
12   {
13     return kvs.merge(k, l);
14   }
15 };
```

Listing 1: Anna C++ Template

Anna is built using C++ and makes use of C++'s template structures to offer a flexible hierarchy

of lattice types. As shown in Listing 1, the main data member in an `Anna` instance is represented as a C++ template of type `MapLattice`, which is a hash map parameterized by an immutable key type `K`, and a value type `L` that descends from `Lattice`. Any descendant of `Lattice` must implement a `merge` method that is ACI.

Users' GET requests are handled via the `MapLattice.reveal` method, which returns the current values associated with the requested keys. PUT requests are handled via the `MapLattice.merge` method, which merges the new key-value pairs into the `MapLattice`. If an input's key does not exist in the hash map, Anna simply stores the new key-value pair into the hash map. Otherwise, the values associated with the key are merged using the merge function of lattice type `L`.

This design allows for a wide range of ACI, coordination-free objects to be stored in Anna. The design of those object classes determine the consistency model that is provided. Figure 2.2 sketches a general template for achieving this coordination-free consistency. In the style of existing systems such as Cassandra and Bayou, programmers can embed application-specific conflict resolution logic into the merge function of an Anna ValueLattice. Anna gives the programmer the freedom to program their ValueLattices in this ad hoc style, and in these cases guarantees only replica convergence. We define this level of ad hoc consistency as *simple eventual consistency*.

2.5.3 Consistency via Lattices: Examples

One of Anna's goals is to relieve developers of the burden of ensuring that their application-specific merge functions have clean ACI semantics. To achieve this, we can compose ad hoc user-defined merge logic within simple but more principled lattices that maintain update metadata with ACI properties guaranteed by construction. In this section we demonstrate that a variety of well-known consistency levels can be achieved in this fashion. We begin by reviewing two popular consistency levels and demonstrating how Anna's modular design helps achieve their guarantees with minimal programming overhead.

Causal Consistency

Causal consistency keeps track of the causal relationship between different versions of the same object. Under *causal consistency*, if a user Alice updates a record, and the update is observed by a user Bob, then Bob's later update to the same record will overwrite Alice's update (instead of invoking the record's merge operator) since the two updates are causally related. However, if Bob updates the record without observing Alice's update, then there is no causal relationship between their updates, and the conflict will be resolved by invoking the record's merge operator.

Figure 2.3 shows Anna's lattice composition that supports causal consistency. Note that a vector clock can be implemented as a MapLattice whose keys are client proxy ids and values are version numbers associated with each proxy id. A version number can be implemented as a MaxIntLattice whose element is an integer and merge function takes the maximum between the input and its current element. Therefore, the integer associated with MaxIntLattice is always increasing, which can be used to represent the monotonically increasing version number. When the proxy performs a read-modify-write operation, it first retrieves the current vector clock, increments

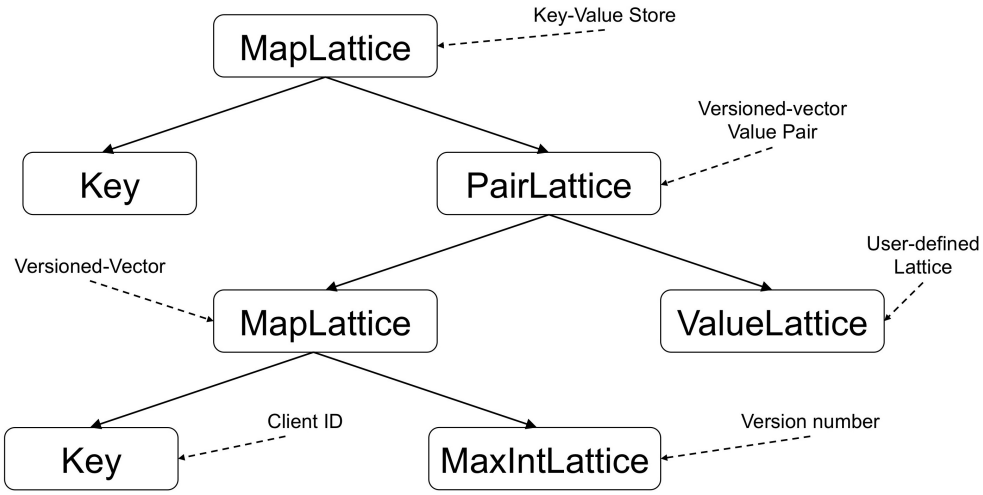

Figure 2.3: Lattice composition for achieving *causal consistency*

the version number corresponding to the proxy id, and writes the updated object together with the new vector clock to the server. The merge function of PairLattice works in lexicographic order on the pair; where the first element of the pair corresponds to a vector clock, and the second corresponds to the actual value lattice associated with a key. Given two PairLattices $P(a, b)$ and $Q(a, b)$, if $P.a \succ Q.a$, then $P(a, b)$ causally follows $Q(a, b)$, and the result is simply $P(a, b)$; the opposite is true if $Q.a \succ P.a$. However if $P.a$ and $Q.a$ are incomparable, then the two pairs correspond to concurrent writes, and the result is merged as $(P(a) \sqcup Q(a), P(b) \sqcup Q(b))$. The implementation of the merge function of PairLattice for achieving *causal consistency* in given in Listing 2.

```
template <typename T>
class CausalPairLattice {
 protected:
  VersionValuePair<T> element;
 public:
  void merge(const VersionValuePair<T> &p)
  {
    // store the previous vector clock
    // before merging
    MapLattice<int, MaxLattice<int>> prev
      = this->element.vector_clock;
    // merge the current and
    // the input vector clocks
    this->element.vector_clock
                  .merge(p.vector_clock);
    if (this->element.vector_clock == prev)
    {
      // do nothing, as the new
      // vector clock is dominated
    }
    else if (this->element.vector_clock
                  == p.vector_clock)
    {
      // overwrite the current value with
      // the new one, as its vector clock
      // is dominated
      this->element.value.assign(p.value);
    }
    else
    {
      // merge the two values, as
      // the vector clocks are not
      // comparable
      this->element.value.merge(p.value);
    }
  }
};
```

Listing 2: Implementation of the merge function of PairLattice for achieving *causal consistency*

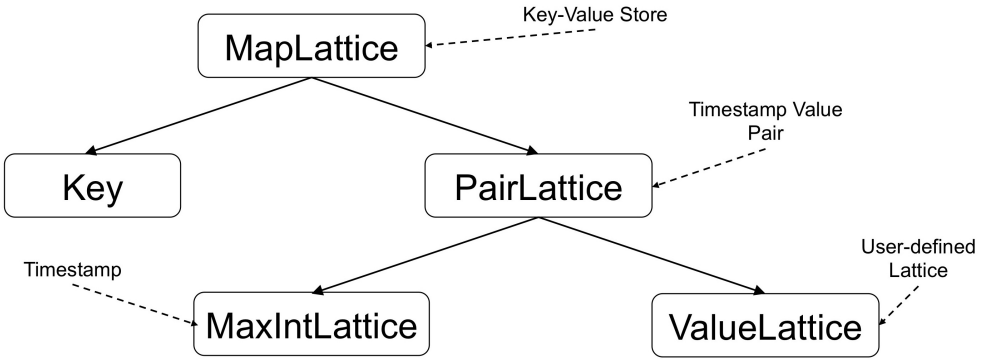

Figure 2.4: Lattice composition for achieving *read committed*

As a simple example, consider a scenario where we have two clients (x, y) performing read-modify-write operations to Anna, whose ValueLattice has *set* as the element and *set union* as the merge function. Initially, the value corresponding to key k is an empty set, with vector clock (x: 0, y: 0). Consider the following two cases. In the first case, x reads key k, retrieves the vector clock (x: 0, y: 0), and writes value $\{a\}$ with updated vector clock (x: 1, y: 0). After receiving the update, Anna determines that vector clock (x: 1, y: 0) dominates (x: 0, y: 0), and therefore overwrites the empty set with $\{a\}$. Then, y reads k, retrieves the vector clock (x: 1, y: 0), and writes value $\{b\}$ with updated vector clock (x: 1, y: 1). Anna determines that vector clock (x: 1, y: 1) dominates (x: 1, y: 0), and therefore overwrites $\{a\}$ with $\{b\}$. In the second case, x and y simultaneously read key k, retrieve the vector clock (x: 0, y: 0), and write back $\{a\}$ and $\{b\}$ with updated vector clock (x: 1, y: 0) and (x: 0, y: 1). Suppose x's update arrives first. As in the previous case, Anna updates the value of k to $\{a\}$ and sets its vector clock to (x: 1, y: 0). However, when y's update arrives, Anna determines that (x: 1, y: 0) and (x: 0, y: 1) are incomparable, and therefore invokes the merge function (*set union*) to resolve conflicts. The resulting value is then set to $\{a, b\}$, with vector clock (x: 1, y: 1).

Read Committed

Read committed is a widely used isolation level in transactional databases [18]. Anna employs the coordination-free definition of *read committed* introduced in [19]. Here, consistency is discussed at the granularity of transactions, consisting of a sequence of reads and writes to the KVS. *Read committed* prevents both dirty writes and dirty reads, and ensures atomicity of writes. In order to prevent dirty writes in a weakly consistent system, it is sufficient to ensure that writes to each key exhibit a total ordering with respect to transactions. Although different replicas may receive writes in different orders, the final state of the KVS should be equivalent to the result of a serial execution of transaction writes. This can be achieved by appending a timestamp to each transaction (and to each write within the transaction) and applying a "larger timestamp wins" conflict resolution policy

at each replica. Note that this monotonically increasing timestamp can be easily implemented using a MaxIntLattice.

To prevent dirty reads, we buffer all writes of a transaction at the client proxy until commit time, ensuring that uncommitted writes never appear in the KVS. To guarantee atomicity of writes, the client sends all writes in one batch to a single Anna actor, ensuring that either all writes reach the Anna server or none. The actor then distributes writes to other actors following the consistent hash ring. Figure 2.4 shows the lattice composition that supports *read committed* isolation level. The difference between the lattice composition for causal consistency and read committed is that we replace the MapLattice that represented growing vector clocks with a MaxIntLattice that represents transaction timestamps. The merge function of the new PairLattice compares the timestamp (MaxIntLattice) and modifies the ValueLattice to be the ValueLattice corresponding to the larger timestamp. If the timestamps are equal, then it implies that these writes are issued within the same transaction, and in this case the ValueLattice's merge logic is invoked[3]. The implementation of the merge function of PairLattice for achieving *read committed* isolation level is given in Listing 3.

Consider an example where we have two transactions T_1 and T_2, with timestamp 1 and 2 respectively. T_1 performs the following sequence of operations: $\{w_1[k_1], w_1[k_2], r_1[k_3]\}$, and T_2 performs $\{w_2[k_1], w_2[k_2], r_2[k_4]\}$. Under *read committed*, T_1 and T_2 perform reads to Anna and buffer all writes locally. Both transactions issue the buffered write requests only after receiving the responses of the read requests and determining that the transactions are safe to commit.

Buffering writes on the client proxy prevents dirty reads. For example, if T_1 failed after $w_1[k1]$, this uncommitted write is not visible to other transactions since it is buffered at the proxy.

Anna avoids dirty writes by using transaction timestamps to consistently order writes. Consider a case where one replica of k_1 receives the writes in the order $\{w_1[k_1], w_2[k_1]\}$, and another replica in the order $\{w_2[k_1], w_1[k_1]\}$. However, the value of both replica converge to $w_2[k1]$, as T_2's write has a larger timestamp, and therefore dominates T_1's write, $w_1[k1]$. Multi-key writes are also eventually consistently ordered via the above timestamp precedence mechanism.

[3]To support SQL's multiple sequential commands per transaction, we can replace these flat timestamps with a nested PairLattice of (transaction timestamp, command number), both being MaxIntLattices.

```
template <typename T>
class ReadCommittedPairLattice {
 protected:
  TimestampValuePair<T> element;
 public:
  void merge(const TimestampValuePair<T>& p)
  {
    if (p.timestamp >
        this->element.timestamp)
    {
      this->element.timestamp
                   .merge(p.timestamp);
      // overwrite the current value
      // with the new one, as its
      // timestamp is smaller
      this->element.value = p.value;
    }
    else if (p.timestamp ==
             this->element.timestamp)
    {
      // merge the two values, as
      // their timestamps are equal
      this->element.value
                   .merge(p.value);
    }
  }
};
```

Listing 3: Implementation of the merge function of PairLattice for achieving *read committed*

2.5.4 More Kinds of Consistency

Anna's modular design allows us to easily identify which component needs to be changed as we switch from simple eventual consistency to other consistency levels. This prevents the CACE (Changing Anything Changes Everything) phenomenon commonly observed in systems with monolithic design. To further demonstrate the flexibility of lattice composition, we modified Anna to support several other consistency levels including *read uncommitted*, *item-cut isolation*, and *read your writes* [19]. It turns out that the lattice composition for these consistency levels are the same as that of *read committed*.

Type of Consistency	Lattice	Server	Client Proxy
Causal Consistency	20	12	22
Read Uncommitted	17	7	4
Read Committed	17	10	9
Item Cut Isolation	17	7	10
Monotonic Reads	17	7	4
Monotonic Writes	17	7	4
Writes Follow Reads	17	7	18
Read Your Writes	17	7	4
PRAM	17	7	4

Figure 2.5: Lines of code modified per component across consistency levels.

Since *read uncommitted* does not require preventing dirty reads, we can easily achieve this level by disabling client-side buffering. Consider the same example in *read committed*, if T1 issues $w_1[k1]$ and then fails, it is possible for other transactions to observe the value v1 even if it is an uncommitted result that need to be rolled back.

Item cut isolation requires that if a transaction reads the same record more than once, it has to read the same value. To provide this guarantee, we buffer the record read at the client side, and when the transaction attempts to read the same record, it invokes the client-side cache instead of querying the server. Again, no modification to the lattice composition is required to achieve this requirement.

Read your writes is a session-based isolation level. Within a session, if a client reads a key after updating it, the read must either reflect the updated value or a value that overwrote the previously written value. Anna achieves this guarantee by attaching a unique timestamp to each client session and applying the same "larger timestamp wins" conflict resolution policy as before. The client also caches all the writes performed within the session. After it retrieves the value of a previously updated key, it merges the value with the cached value before returning the result. This way, Anna ensures that the value being read is at least as recent as the client's own update in terms of the timestamp.

Figure 2.5 shows the additional number of lines of code (loc) in C++ required on top of *simple eventual consistency* for each coordination-free consistency level. It is easy to conclude that extending Anna beyond *simple eventual consistency* incurs very little programming overhead.

2.6 Implementation

The Anna actor and client proxy are implemented entirely in C++. The codebase—including the lattice library, all the consistency levels, the server code, and client proxy code—amounts to about 2000 lines of C++ on top of commonly-used libraries including ZeroMQ and Google Protocol Buffers. In the ensuing discussion, we refer the reader back to Figure 2.1.

2.6.1 Actor

To store the private KVS replica at each actor, we use the unordered map from the C++ standard library. Inter-actor multicast is achieved via the pub-sub communication mechanism of ZeroMQ, a high-performance asynchronous messaging library. To perform well across scales, we leverage ZeroMQ in different ways depending on whether we are communicating within or across machines. When two actors communicate within a single machine, the sender first moves the message into a shared memory buffer, and then sends the address of the buffer to the receiver using ZeroMQ's *inproc* transport, which is optimized for intra-process communication. The receiver, after getting the address, reads the shared buffer, updates its local KVS replica, and garbage collects the shared buffer. When two actors communicate across different machines, the sender first serializes the message into a byte-string using Google Protocol Buffers. It then sends the byte-string using ZeroMQ's *tcp* transport, which is designed for inter-node communication. After receiving the byte-string, the receiver first de-serializes the message, and then updates its KVS replica accordingly.

Anna uses consistent hashing to partition and replicate key-value pairs across actors. Following the design of Dynamo [33], each actor has a unique id, and Anna applies a CRC32 hash on the id to assign the actor to a position on the hash ring. It applies the same hash function to a key in order to determine the actors responsible for storing the key. Each key-value pair is replicated *N-1* times on the clockwise successor actors, where *N* is the user-provided replication factor.

Anna actors support three operations: GET, PUT, and DELETE. GET retrieves the value of a key from a (single) replica. Coordination-free consistency, as discussed in Section 2.5, does not require a quorum, so GET need not merge values from more than one replica. The GET response may be stale; the staleness is bounded by the multicast period, which is an adjustable parameter to balance performance and staleness. PUT persists the merge of a new value of a key with a (single) replica using the lattice merge logic. DELETE is implemented as a special PUT request with an empty value field. Actors free the heap memory of a key/value pair only when the DELETE's timestamp dominates the key's current timestamp. To completely free the memory for a key, each actor maintains a vector clock that keeps track of the latest-heard timestamps of all actors, which is kept up-to-date during multicast. Actors free the memory for a key only when the minimum timestamp within the vector-clock becomes greater than the DELETE's timestamp. After that time, because Anna uses ordered point-to-point network channels, we can be sure no old updates to the key will arrive. This technique extends naturally to consistency levels that require per-key vector-clocks (such as causal consistency) instead of timestamp. The difference is that before an actor frees a key, it asynchronously queries other replicas for the key's vector-clock to make sure they are no less than the DELETE's vector-clock.

2.6.2 Client Proxy

Client proxies interact with actors to serve user requests. In addition to GET, PUT, and DELETE, proxies expose two special operations to the users for consistency levels that involve transactions: BEGIN_TRANSACTION and END_TRANSACTION. All operations that fall in between a pair

of special operations belong to a single transaction. Transaction ID is uniquely generated by concatenating a unique actor sequence number with a local timestamp.

Specific data structures are required at the proxy to support certain advanced consistency levels; these data structures are only accessible in a single client-proxy thread. For read committed, we need to create a message buffer that stores all PUT requests from a single transaction. For item-cut isolation, we need to cache key-value pairs that have already been queried within the same transaction. Currently, both the message buffer and the cache are implemented with the unordered map from the C++ standard library.

Client-actor communication is implemented with Linux sockets and Protocol Buffers. The client proxy uses the same consistent hashing function to determine the set of actors that maintain replicas of a given key. For load balancing, requests are routed to a randomly chosen replica. In case of failure and network delay, the client proxy times out and retries the request on other actors.

2.6.3 Actor Joining and Departure

In order to achieve steady performance under load burst, actors can be dynamically added or removed from the Anna cluster without stalling the system. Anna handles actor joining and departure in a similar fashion as Dynamo [33] and the work in [65]. Note that a new actor can be spawned from within an existing node, or from a new node. When a new actor joins the cluster, it first broadcasts its id to all existing actors. Each existing actor, after receiving the id, updates its local copy of the consistent hash ring and determines the set of key-value pairs that should be managed by the new actor. It then sends these key-value pairs to the new actor and deletes them from its local KVS replica. If the pre-existing actor receives queries involving keys that it is no longer responsible for, it redirects these requests to the new actor. After the new actor receives key-value pairs from all existing actors, it multicasts its id to all client proxies. Upon receiving the id, client proxies update the consistent hash ring so that relevant requests can be routed to the new actor.

When an actor is chosen to leave the cluster, it first determines the set of key-value pairs every other actor should be responsible for due to its departure. It then sends them to other actors along with its intention to leave the cluster. Other actors ingest the key-value pairs and remove the leaving actor from the consistent hash ring. The leaving actor then broadcasts to all client proxies to let them update the consistent hash ring and retry relevant requests to proper actors.

2.7 Evaluation

In this section, we experimentally justify Anna's design decisions on a wide variety of deployments. First we evaluate Anna's ability to exploit parallelism on a multi-core server and quantify the merit of our Coordination-free actor model. Second, we demonstrate Anna's ability to scale incrementally under load burst. We then compare Anna against state-of-the-art KVS systems on both a single multi-core machine and a large distributed deployment. Finally, we show that the consistency levels from Section 2.5 all provide high performance.

2.7.1 Coordination-free Actor Model

Exploiting Multicore Parallelism

Recall that under the Coordination-free actor model, each actor thread maintains a private copy of any shared state, and asynchronously multicasts the state to other replicas. This section demonstrates that the Coordination-free actor model can achieve orders of magnitude better performance than a conventional shared-memory architecture on a multi-core server.

To establish comparison against the shared-memory implementation, we built a minimalist multi-threaded key-value store using the concurrent hash map from the Intel Thread Building Blocks (TBB) library [48]. TBB is an open source library consisting of latch-free, concurrent data structures, and is among the most efficient libraries for writing scalable shared-memory software. We also benchmark Anna against Masstree, another shared-memory key-value store that exploits multi-core parallelism [73]. Finally, we implemented a multi-threaded key-value store using the C++ unordered map *without* any thread synchronization such as latching or atomic instructions. Note that this key-value store is not even thread-safe: torn writes could occur when multiple threads concurrently update the same key. It reflects the ideal performance one can get for any shared-memory KVS implementation like Masstree, TBB, etc.

Our experiments run on Amazon m4.16xlarge instances. Each instance is equipped with 32 CPU cores. Our experiments utilize a single table with 1M key-value pairs. Keys and values are 8 bytes and 1KB in length, respectively. Each request operates on a single key-value pair. Requests are update-only to focus on potential slowdowns from conflicts, and we use zipfian distributions with varying coefficients to generate workloads with different levels of conflict.

In our first experiment, we compare the throughput of Anna against the TBB hash map, Masstree, and the unsynchronized KVS (labeled as "Ideal") on a single multi-core machine. We measure the throughput of each system while varying the number of threads available. We pin each thread to a unique CPU core, and increase thread count up to the hardware limit of 32 CPU cores. In addition to measuring throughput, we use Linux's `perf` profiler to obtain a component-wise breakdown of CPU time. To measure the server's full capacity, requests are pre-generated based on the workload distribution at each thread. Since Anna is flexible about data placement policy, we experiment with different replication factors, from pure partitioning (like Redis Cluster or MICA) to full replication (a la Bayou) to partial replication (like ScyllaDB). As a baseline, Anna employs simple eventual consistency, and threads are set to multicast every 100 milliseconds. We use the same consistency level and multicast rate in all subsequent experiments unless otherwise stated. For each thread count, we repeat the experiment $10\times$ and plot the average throughput.

Figure 2.6a and 2.7a show the result of the high-contention experiment, with zipf coefficient set to 4. We observe that both the TBB hashmap and Masstree fail to exploit parallelism on this workload because most requests perform an update against the same key, and concurrent updates to this key have to be serialized. Furthermore, both the TBB hashmap and Masstree must employ synchronization to prevent a single key-value pair from concurrent modification by multiple threads. Synchronization overhead is proportional to the number of contending threads, which causes those systems' performance to plateau as we increase the number of threads in the system.

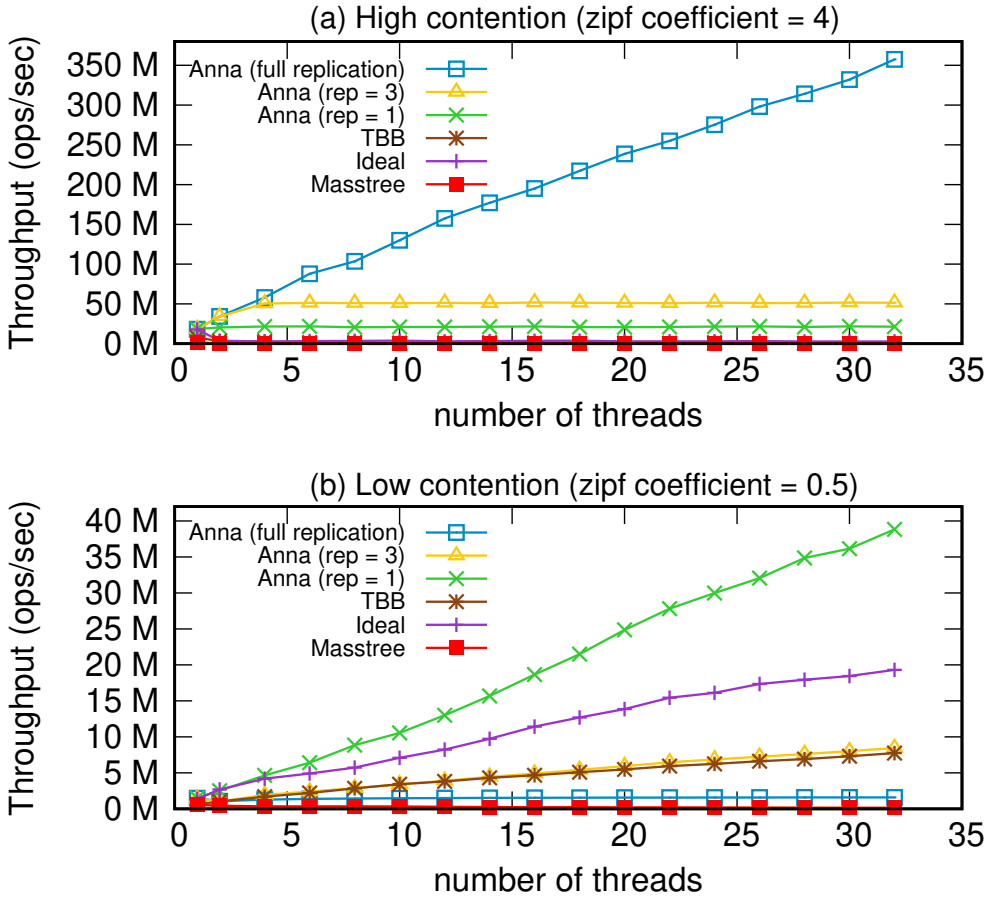

Figure 2.6: Anna's single-node throughput across thread counts.

Synchronization cost manifests as cache coherence overhead on multi-core hardware [87]. Figure 2.7a shows that TBB and Masstree spend 92% - 95% of the CPU time on atomic instructions under high contention, and only 4% - 7% of the CPU time is devoted to request handling. As a result, the TBB hash map and Masstree perform $50\times$ slower than Anna (rep = 1) and $700\times$ slower than Anna (full replication).

The unsynchronized store performs $6\times$ faster than the TBB hashmap and Masstree but still much slower than Anna. Although it does not use any synchronization to prevent threads from concurrently modifying the same key-value pairs, it suffers from cache coherence overhead resulting from threads modifying the same memory addresses (the contended key-value pairs). This is corroborated in Figure 2.7a, which shows that although both Anna and the unsynchronized store spend the majority of the CPU time processing requests, the unsynchronized store incurs $17\times$ more cache misses than Anna.

In contrast, threads in Anna perform updates against their local state in parallel without synchronizing, and periodically exchange state via multicast. Although the performance is roughly

KVS	RH	AI	LM	M	O	CM
Anna (full)	90%	0%	4%	4%	2%	1.1
Anna (rep=3)	91%	0%	5%	2%	2%	1
Anna (rep=1)	94%	0%	5%	0%	1%	1
Ideal	97%	0%	0%	0%	3%	17
TBB	4%	95%	0%	0%	1%	19
Masstree	7%	92%	0%	0%	1%	16

(a) High Contention

KVS	RH	AI	LM	M	O	MF
Anna (full)	3%	0%	3%	92%	2%	32
Anna (rep=3)	25%	0%	4%	69%	2%	3
Anna (rep=1)	93%	0%	5%	0%	2%	1
Ideal	97%	0%	0%	0%	3%	32
TBB	70%	26%	0%	0%	4%	32
Masstree	20%	78%	0%	0%	2%	32

(b) Low Contention

Figure 2.7: Performance breakdown for different KVSes under both contention levels when using 32 threads. CPU time is split into 5 categories: Request handling (RH), Atomic instruction (AI), Lattice merge (LM), Multicast (M), and others (O). The number of L1 cache misses (CM) for the high-contention workload and the memory footprint (MF) for the low-contention workload relative to Anna (rep=1) are shown on the right-most column.

bounded by the replication factor under high contention, it is already far better than the shared-memory implementation across the majority of replication factors. Figure 2.7a indicates that Anna indeed achieves wait-free execution: the vast majority of CPU time (90%) is spent processing requests without many cache misses, while overheads of lattice merge and multicast are small. In short, Anna's Coordination-free actor model addresses the heart of the scalability limitations of multi-core KVS systems.

Figure 2.6b and 2.7b show the result of the low-contention experiment, with zipf coefficient 0.5. Unlike the high contention workload, all data are likely to be accessed with this contention level. Anna (rep=1) achieves excellent scalability due to its small memory footprint (data is partitioned across threads). However, despite the linear-scaling of Anna (rep=3), its absolute throughput is $4\times$ slower than Anna (rep=1). There are two reasons that have led to this performance degradation. First, increasing the replication factor increases the thread's memory footprint. Furthermore, under low contention, the number of distinct keys being updated within the gossip period increases significantly. Therefore, we can no longer exploit merge-at-sender to reduce the gossip overhead. Figure 2.7b shows that 69% of the CPU time is devoted to processing gossip for Anna (rep=3). Following this analysis, Anna (full replication) does not scale because any update performed at

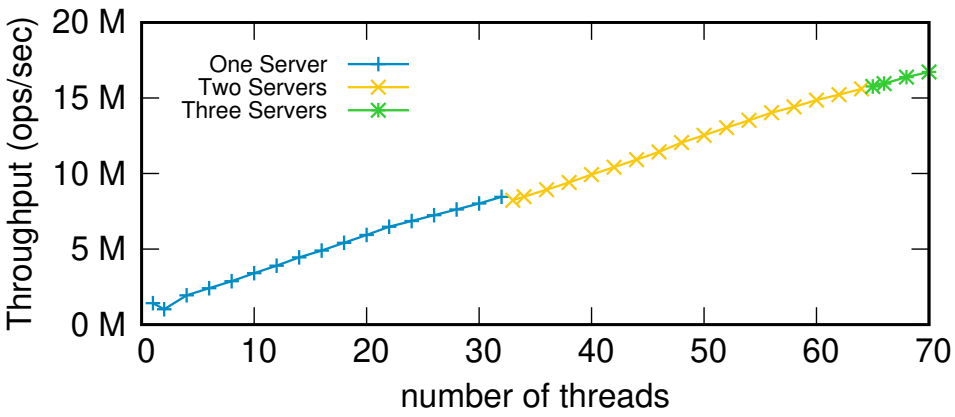

Figure 2.8: Anna's throughput while incrementally adding threads to multiple servers.

one thread will eventually be gossiped to every other thread, and therefore the performance is equivalent to serially executing requests with one thread. Although TBB and Masstree do not incur gossip overhead, they suffer from larger memory footprint and high cost of (conservative) synchronization operations as shown by our profiler measurements in Figure 2.7b. The lesson learned from this experiment is that for systems that support multi-master replication, having a high replication factor under low contention workloads can hurt performance. Instead, we want to dynamically monitor the data's contention level and selectively replicate the highly contented keys across threads. We come back to this subject in Section 2.8.

Scaling Across Scales

This section demonstrates Anna's ability to scale smoothly from a single-node deployment to a multi-node deployment. Anna's replication factor is set to 3, and we use the low contention workload from the multi-core scalability evaluation in Section 2.7.1. We measure throughput while varying the number of available threads. The first 32 threads reside on a single node. The next 32 threads reside on a second node, while any remaining threads (at thread count greater than 64) reside on a third node.

Figure 2.8 shows that Anna exhibits smooth linear scaling with increasing thread count, on both a single node (32 or fewer threads) and multiple nodes (33 or more threads). We observe a small drop in performance as we add a 33^{rd} thread because this is the first thread that resides on the second node, and therefore triggers distributed multicast across the network. We do not observe a similar drop in performance as we add threads on the third node (at 65 threads) because the overhead of distributed multicast already affects configurations with thread counts between 33 and 64. Figure 2.8 illustrates that Anna is able to achieve near-linear scalability across different scales with the Coordination-free actor model.

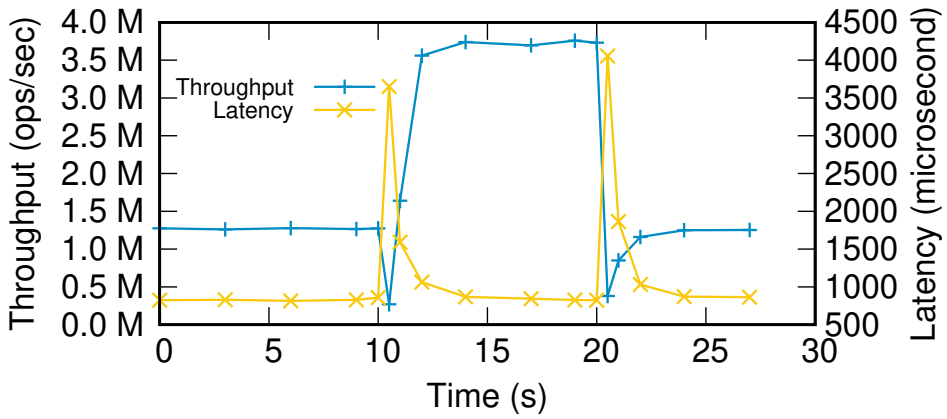

Figure 2.9: Anna's ability to elastically scale under load burst while maintaining performance.

2.7.2 Elastic Scalability

This section explores how well Anna's architecture achieves elastic scaling under load bursts. Our goal in this study is not to compare thread allocation *policies* per se, but rather to evaluate whether the Coordination-free actor model enables fine-grained elasticity. Hence we focus on Anna's reaction time, assuming an omniscient policy.

The experiment runs a read-modify-write, low contention YCSB workload [28], and uses 25 byte key, 1KB value records in a single table of 1M records. We perform our experiment on Amazon EC2 m4.x16 large instances. Anna is configured with replication factor 3. Note that the performance characteristics of experiments performed in this subsection and the next (2.7.3) differ from previous experiments. In earlier experiments, the goal was to evaluate Anna's maximum processing capacity when handling concurrent update requests; as a result, requests were update-only and pre-generated on actor threads to avoid request overhead due to network. Here, the goal is to evaluate how Anna performs in a more real-world setting, so requests are chosen to have a mix of reads and writes, and are being sent from client proxies on other nodes. Therefore in this section we expect to observe network overhead; the effects are further discussed in Section 2.7.3.

At the beginning of our experiment, we use one EC2 instance with 32 threads as the server and a sufficient number of client proxy instances to saturate the server. At the 10-second mark, we triple the load from the client proxies to create a burst. At the same time, 64 more threads from two server nodes are added to the Anna cluster. At the 20-second mark, we reduce the load back to the original and remove 64 threads from the cluster. Throughout the YCSB benchmark, we monitor Anna's throughput and the request latency.

As shown in Figure 2.9, Anna's throughput increases by $2\times$ at the 10-second mark when we add in 64 additional threads, and drops to the original throughput at the 20-second mark when we remove the same number of threads. Throughout the experiment, the request latency stays roughly the same. The brief latency spikes at the 10-second mark and the 20-second mark are due to adding and removing nodes to the cluster.

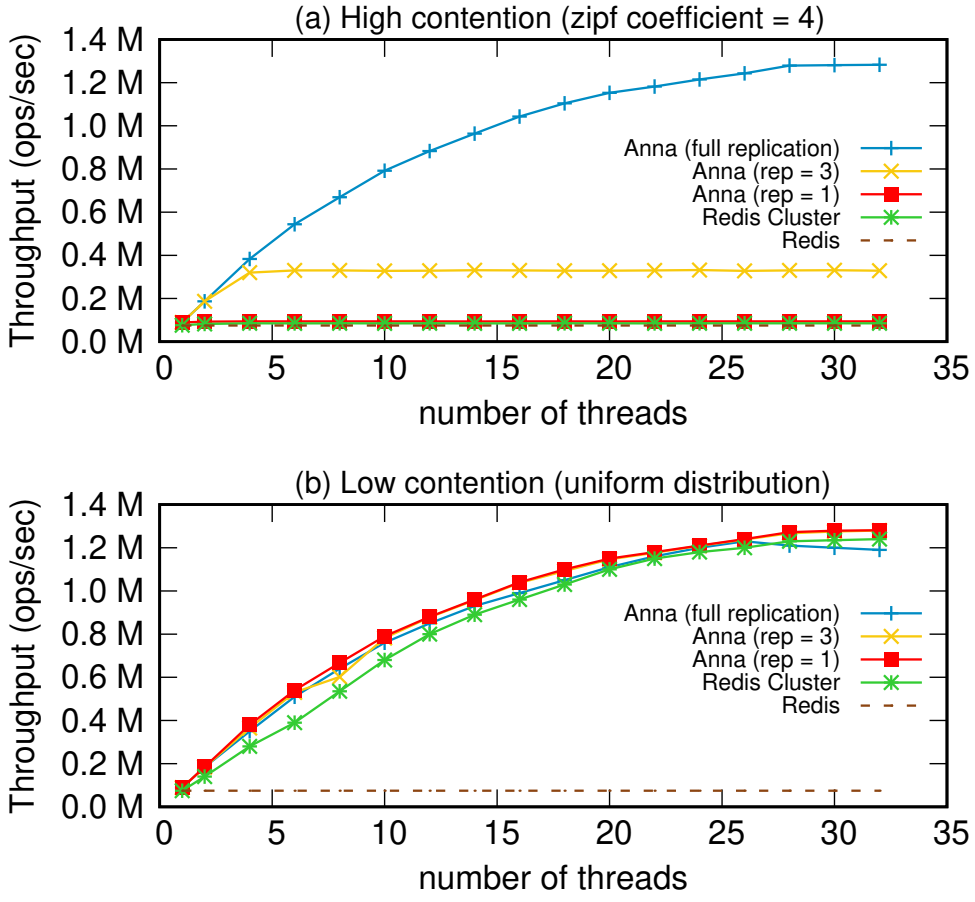

Figure 2.10: Throughput comparison between Anna and Redis on a single node.

2.7.3 Comparison with Popular Systems

This section compares Anna against widely-deployed, state-of-the-art key value stores. We perform two experiments; the first compares Anna against Redis [86] on a single node, and the second compares Anna against Cassandra [12] on a large distributed deployment. Both experiments run YCSB, and use the same configuration as in Section 2.7.2 with different contention levels.

Single node multi-core experiment

This section compares Anna with Redis on a single multi-core server. While Anna can exploit multi-core parallelism, Redis is a single-threaded KVS system, and cannot exploit any parallelism whatsoever. We therefore additionally compare Anna against Redis Cluster, which knits together multiple independent Redis instances, each of which contain a shard of the KVS.

In this experiment, we use a single EC2 instance as a server and enough client proxy instances to saturate the server. The Redis Cluster baseline runs an independent Redis instance on each

available server thread.

Figure 2.10a compares each system under high contention while varying thread count. As in our earlier high contention experiments, clients pick keys with a zipfian coefficient of 4. Under high contention, each Redis Cluster's instances are subject to skewed utilization, which limits overall throughput. In contrast, Anna can spread load for hot keys across replicas. When the replication factor is greater than 1, Anna's throughput increases until the number of threads is slightly larger than the replication factor and then plateaus. If the hot keys are fully replicated, we observe that the throughput continues to grow as we increase the number of threads.

Figure 2.10b shows the result of the low contention experiment. As expected, Redis' throughput remains constant with increasing thread count. In contrast, both Anna and Redis Cluster can exploit multi-core parallelism, and their throughputs scale with increasing thread count. Interestingly, Anna (rep=3) and Anna (full replication) scale quite nicely, and the performance penalty due to gossip is far less significant compared to the result in Section 2.7.1. The reason is that when the network is involved, the majority of overhead goes to network packet handling and message serialization and deserialization. Within a single node, gossip is performed using the shared memory buffer, and does not incur network overhead. Therefore, the overhead becomes far less significant. Experiments in this section show that Anna can significantly outperform Redis Cluster by replicating hot keys under high contention, and can match the performance of Redis Cluster under low contention.

Note that unlike the experiments in Section 2.7.1, we do not observe linear scalability for Anna and the y axis has reduced by orders of magnitude. This is in keeping with earlier studies [54, 1], which demonstrate that this is due to message overheads: at a request length of 1KB we cannot expect to generate much more than 10Gbps of bandwidth due to message overheads. We attempted to improve the performance by varying the request size and batching the requests. Although these techniques did improve the absolute throughput, the scalability trend remained the same, and we continued to be bottlenecked by the network.

Distributed experiment

In a distributed setting, we compare Anna against Cassandra, one of the most popular distributed KVS systems [12]. To ensure that Cassandra achieves the best possible performance, we configure it to use its weakest consistency level (ONE), which only requires that an update is reflected on a single node before returning success. Updates are asynchronously propagated in the background.

We deployed Cassandra and Anna across four EC2 geographical regions (Oregon, North Virginia, Ireland, and Tokyo) and measured their scalability by adjusting the number of nodes per region. The replication factor of both Cassandra and Anna are set to 3. As in the multi-core experiment, each server node is a m4.x16large instance and we use multiple client instances to saturate the server. Clients pick keys to update from a uniform distribution.

Figure 2.11 shows that both Anna and Cassandra scale near-linearly as we increase the number of nodes. However, Anna has better absolute performance due to its low-overhead single-threaded execution. Indeed, when we varied the number of threads available to Anna, we found that Anna could significantly outperform Cassandra with just four threads per node (even though Cassandra

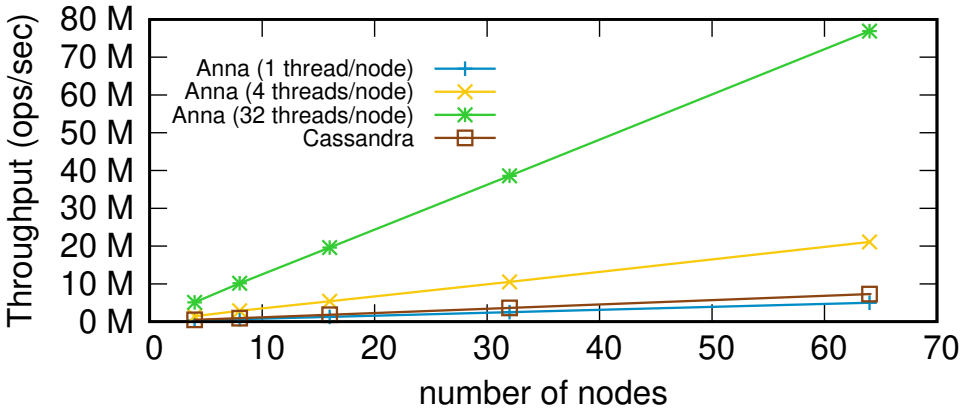

Figure 2.11: Anna vs Cassandra, distributed throughput.

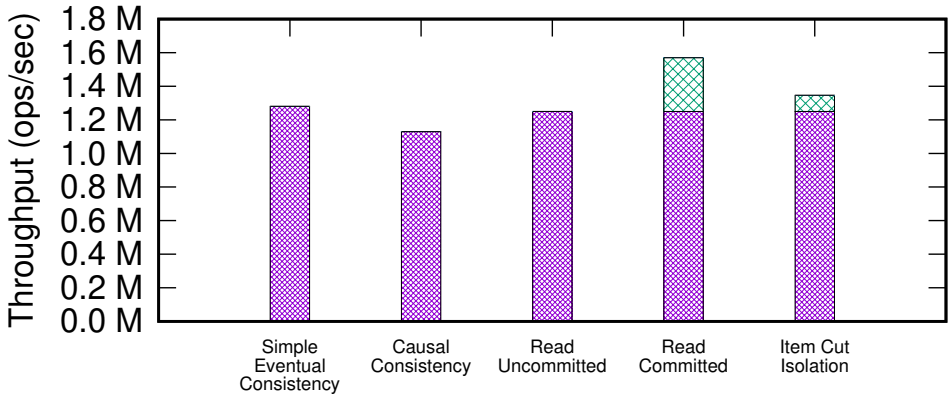

Figure 2.12: Performance Across Consistency Levels

used multi-threading). When permitted to use all 32 available cores, Anna outperformed Cassandra by $\sim 10\times$. This experiment demonstrates the importance of Anna's fast single-node mechanisms; even when a system can scale to large clusters, fast single-node mechanisms can make significantly more efficient use of available resources.

2.7.4 Performance Across Consistency Levels

Having implemented various consistency levels, we study the performance implications of the additional codepath for more advanced consistency levels. Anna is configured to use all 32 available cores, and the replication factor is set to 3. We use the low contention requests from Section 2.7.3. For transaction-based consistency levels, we group every six operations into one transaction at the YCSB client side. Figure 2.12 shows the throughput evaluation across different consistency levels.

In general, we observe that the overhead incurred by these advanced consistency levels is not significant. As explained in Section 2.5.3, client-side buffering and caching requirement sometimes lead to *higher* throughput for Anna, which we show using green bars in Figure 2.12.

For *causal consistency*, we observe a slight degradation in throughput as Anna has to maintain the vector clock associated with each key-value pair, requiring more sophisticated lattice merge logic. In addition, the size of the vector clock for a given key is proportional to the number of client proxies that access the key. Therefore, periodic garbage collection is required to reduce the size of the vector clock. Similar throughput degradation is observed for *read uncommitted* due to the management of timestamps. For *read committed*, throughput increases because the client is required to buffer all write requests and send them as a single batch at the end of a transaction, which amortizes the number of round-trips between the server and the client. For *item cut isolation*, we also observe an increase in throughput because repeated reads to the same record are handled by a client-side cache (which again saves a round-trip between the server and the client). The throughput improvement gained from client-side buffering and caching is highlighted in green. Note that although other consistency levels does not require client-side buffering or caching, it is possible to use these techniques to improve throughput.

2.8 Conclusion and Takeaways

Conventional wisdom says that software designed for one scale point needs to be rewritten when scaling up by $10 - 100\times$ [32]. In this work, we took a different approach, exploring how a system could be architected to scale across many orders of magnitude by design. That goal led us to some challenging design constraints. Interestingly, those constraints led us in the direction of simplicity rather than complexity: they caused us to choose general mechanisms (background key exchange, lattice compositions) that work well across scale points. Perhaps the primary lesson of this work is that our scalability goals led us by necessity to good software engineering discipline.

The lattice composition model at the heart of Anna was critical to both performance and expressivity. The asynchronous merging afforded by lattices enabled wait-free performance; the lattice properties provided a conceptual framework for ensuring consistency; the *composition* of simple lattices enabled a breadth of consistency levels. Scale-independence might seem to be in conflict with richly expressive consistency. The lattice composition model resolved that design conflict.

As discussed in Chapter 1, serverless systems need to not only deliver excellent performance at any scale but also react promptly to changing workload distributions for cost-efficiency. The initial architecture of Anna lacked the mechanisms to monitor and respond to usage and workloads. Another notable weakness was its need to aggressively replicate the entire database across the main memory of many machines to achieve high performance. This gave the system an unattractive cost-performance tradeoff and made its resource allocation very rigid. As a result, although a benchmark-beater, the initial design suffered from the problems highlighted above: it was expensive and inflexible for large datasets with non-uniform access distributions. We describe how we extended Anna to become an autoscaling, tiered serverless KVS in the next chapter.

Chapter 3

Autoscaling Tiered Cloud Storage in Anna

As public infrastructure cloud providers have matured in the last decade, the number of storage services they offer has soared. Popular cloud providers like AWS [13], Microsoft Azure [15], and Google Cloud Platform (GCP) [41] each have at least seven storage options. These services span the spectrum of cost-performance tradeoffs: AWS ElastiCache, for example, is an expensive, memory-speed service, while AWS Glacier is extremely high-latency and low-cost. In between, there are a variety of services such as the Elastic Block Store (EBS), the Elastic File System (EFS), and the Simple Storage Service (S3). Azure and GCP both offer a similar range of storage solutions.

Each one of these services is tuned to a unique point in that design space, making it well-suited to certain performance goals. Application developers, however, typically deal with a non-uniform *distribution* of performance requirements. For example, many applications generate a skewed access distribution, in which some data is "hot" while other data is "cold". This is why traditional storage is assembled hierarchically: hot data is kept in fast, expensive cache while cold data is kept in slow, cheap storage. These access distributions have become more complex in modern settings, because they can change dramatically over time. Realistic workloads spike by orders of magnitude, and hot sets shift and resize. These large-scale variations in workload motivate an autoscaling service design, but most cloud storage services today are unable to respond to these dynamics.

The narrow performance goals of cloud storage services result in poor cost-performance tradeoffs for applications. To improve performance, developers often take matters into their own hands by addressing storage limitations in custom application logic. This introduces significant complexity and increases the likelihood of application-level errors. Developers are inhibited by two key types of barriers when building applications with non-uniform workload distributions:

Cost-Performance Barriers

Each of the systems discussed above—ElastiCache, EBS, S3, etc.—offers a different, *fixed* tradeoff of cost, capacity, latency, and bandwidth. These tradeoffs echo traditional memory hierarchies built from RAM, flash, and magnetic disk arrays. To balance performance and cost, data should

ideally move adaptively across storage tiers, matching workload skew and shifting hotspots. However, current cloud services are largely unaware of each other, so software developers and DevOps engineers must cobble together ad hoc memory hierarchies. Applications must explicitly move and track data and requests across storage systems in their business logic. This task is further complicated by the heterogeneity of storage services in terms of deployment, APIs, and consistency guarantees. For example, single-replica systems like ElastiCache are linearizable, while replicated systems like DynamoDB are eventually consistent.

Static Deployment Barriers

Cloud providers offer very few truly autoscaling storage services; most such systems have hard constraints on the number and type of nodes deployed. In AWS for example, high-performance tiers like ElastiCache are surprisingly inelastic, requiring system administrators to allocate and deallocate instances manually. Two of the lower storage tiers—S3 and DynamoDB—are autoscaling, but are insufficient for many needs. S3 autoscales to match data volume but ignores workload; it is designed for "cold" storage, offering good bandwidth but high latency. DynamoDB offers workload-based autoscaling but is prohibitively expensive to scale to a memory-speed service. This motivates the use of ElastiCache over DynamoDB, which again requires an administrator to monitor load and usage statistics, and manually adjust resource allocation.

In Chapter 2, we presented the initial architecture of the Anna KVS and described a design with excellent performance across orders of magnitude in scale. Here, we extend the initial version of Anna to perform serverless autoscaling, allowing the system to span the cost-performance design space more flexibly and adapt dynamically to workload variation in a cloud-native setting. The architecture presented here removes the cost-performance and static deployment barriers by adding three key mechanisms: (1) horizontal elasticity to adaptively scale deployments; (2) vertical data movement in a storage hierarchy to reduce cost by demoting cold data to cheap storage; and (3) multi-master selective replication of hot keys across nodes and cores to efficiently scale request handling for non-uniform access patterns. The architecture we present here is simplified by deploying the same storage kernel across many tiers, by entirely avoiding coordination, and by keeping most components stateless through reuse of the storage engine. The additions to Anna described in this chapter enable system operators to specify high-level goals such as fault tolerance and cost-performance objectives, without needing to manually configure the number of nodes and the replication factors of keys. A new policy engine automatically responds to workload shifts using the mechanisms mentioned above to meet these SLOs. While Chapter 2's evaluation focused on raw performance, here we also emphasize *efficiency*: the ratio of performance to cost. For various cost points, Anna beats in-memory systems (e.g., AWS ElastiCache, Masstree [73]) by up to an order of magnitude in performance. Anna also outperforms DynamoDB, an elastic database, by more than two orders of magnitude in efficiency.

The rest of this chapter proceeds as follows. In Section 3.1, we describe the mechanisms that Anna uses to respond to mixed and changing workloads. Section 3.2 introduces the architecture of Anna including the implementation of these mechanisms, and Section 3.3 describes Anna's policy

engine. Section 3.4 introduces Anna's API. In Section 3.5, we present an evaluation of Anna's mechanisms and policies, and we describe how they fare in comparison to the state of the art. Section 3.6 discusses related work, and we conclude with future work in Section 3.7.

We use AWS as the public cloud provider underlying Anna. The design principles and lessons learned here are naturally transferable to other cloud providers with similar offerings.

3.1 Distributions and Mechanisms

In this section, we first classify and describe common workload distributions across data and time. We then discuss the mechanisms that Anna uses to respond to the workload properties and changes. We believe that an ideal cloud storage service should gracefully adapt to three aspects of workload distributions and their dynamics in time:

A. Volume. As overall workload grows, the aggregate throughput of the system must grow. During growth periods, the system should automatically increase resource allocation and thereby cost. When workload decreases, resource usage and cost should decrease correspondingly as well.

B. Skewness. Even at a fixed volume, skewness of access distributions can affect performance dramatically. A highly skewed workload will make many requests to a small subset of keys. A uniform workload of similar volume will make a few requests to each key. Different skews warrant different responses, to ensure that the resources devoted to serving each key are proportional to its popularity.

C. Shifting Hotspots. Workloads that are static in both skew and volume can still exhibit changes in distribution over time: hot data may become cold and vice versa. The system must be able to not only handle skew, but also changes in the specific keys associated with the skew (hotspots) and respond accordingly by prioritizing data in the new hot set and demoting data in the old one.

We address these three workload variations with three mechanisms in Anna, which we describe next.

Horizontal Elasticity. In order to adapt to variation in workload volume, each storage tier in Anna must scale elastically and independently, both in terms of storage and request handling. Anna needs the storage capacity of many nodes to store large amounts of data, and it needs the compute and networking capacity of many nodes to serve large numbers of requests. This is accomplished by partitioning (sharding) data across all the nodes in a given tier. When workload volume increases, Anna can respond by automatically adding nodes and repartitioning a subset of data. When the volume decreases, Anna can remove nodes and repartition data among the remainders.

Multi-Master Selective Replication. When workloads are highly skewed, simply adding shards to the system will not alleviate pressure. The small hot set will be concentrated on a few nodes that will be receiving a large majority of the requests, while the remaining nodes lie idle. The only solution is to replicate the hot set onto many machines. However, we do not want to repeat the mistakes of our first iteration of Anna's design, replicating cold keys as well—this simply wastes space and increases overhead. Instead, replication must be selective, with hot keys replicated more

Table 3.1: The mechanisms used by Anna to deal with various aspects of workload distributions.

Workload Dynamics	Relevant Mechanisms
Volume	Elasticity
Skew	Replication, Tiering
Hotspot	Replication, Tiering

than cold keys. Thus, Anna must accurately track which data is hot and which is cold, and the replication factors and current replica locations for each key. Note that this aids both in handling skew in general and also changes in hotspots with fixed skew.

Vertical Tiering. As in a traditional memory hierarchy, hot data should reside in a fast, memory-speed tier for efficient access; significant cost savings are available by demoting data that is *not* frequently accessed to cold storage. Again, Anna must correctly classify hot and cold data in order to promote or demote appropriately to handle skew and hotspots. While the previous two mechanisms are aimed at improving performance, this one primarily attempts to minimize cost without compromising performance.

3.1.1 Summary

Table 3.1 shows which mechanisms respond to which properties of workload distributions. There is a direct mapping between an increase (or decrease) in volume—with other factors held constant—and a requirement to automatically add (or remove) nodes. Changes in workload skew require a response to the new hot set size via promotion or demotion, as well as appropriate selective replication. Similarly, a change in hotspot location requires correct promotion and demotion across tiers, in addition to shifts in per-key replication factors. We describe how Anna implements each one of these mechanisms in Sections 3.2 and 3.3. In Section 3.5, we evaluate how well Anna responds to these dynamics.

3.2 Systems Architecture

In this section, we introduce Anna's (extended) architecture and illustrate how the mechanisms discussed in Section 3.1 are implemented. We present an overview of the core subsystems and then discuss each component in turn. As mentioned at the beginning of this chapter, Anna is built on AWS components. In our initial implementation and evaluation, we validate this architecture over two storage tiers: one providing RAM cost-performance and another providing flash disk cost-performance. Anna's memory tier stores data in RAM attached to AWS EC2 nodes. The flash tier leverages the Elastic Block Store (EBS), a fault-tolerant block storage service that masquerades as a mounted disk volume on an EC2 node. There is nothing intrinsic in our choice of layers. We could easily add a third layer (e.g., S3) and a fourth (e.g., Glacier), but demoting data to cold storage in these tiers operates on much longer timescales that are beyond the scope of this work.

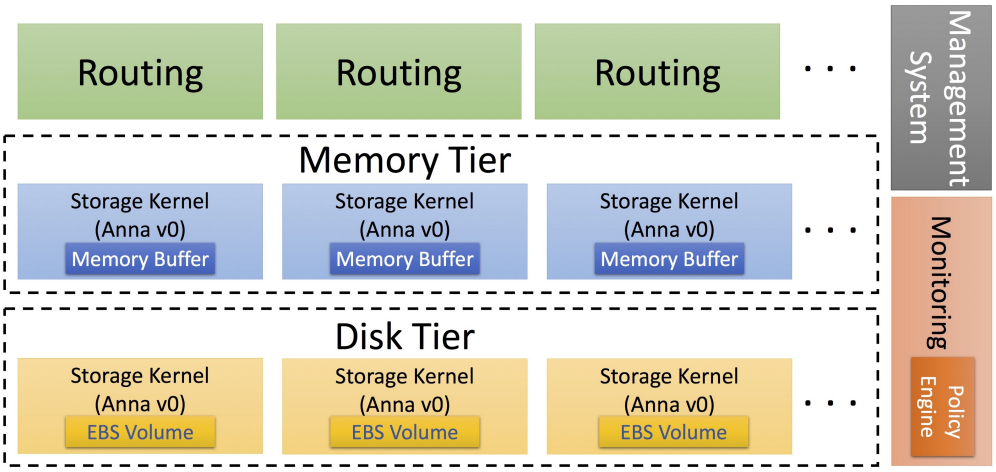

Figure 3.1: The Anna architecture.

3.2.1 Overview

Figure 3.1 presents an overview of Anna, with each rectangle representing a node. In Chapter 2, we described an extremely performant, coordination-free key-value store with a rich variety of consistency levels. We demonstrated how a KVS could scale from multicore to distributed settings while gracefully tolerating the natural messaging delays that arise in distributed systems. To enable the mechanisms described in Section 3.1, we first extended the storage kernel (labeled as Anna v0 in Figure 3.1) to support multiple storage media and then designed three new subsystems: a monitoring system/policy engine, a routing service, and a cluster management system. Each subsystem is bootstrapped on top of the key-value storage component in Anna, storing and modifying its metadata in the system.

The monitoring system and policy engine are the internal services responsible for responding to workload dynamics and meeting SLOs. Importantly, these services are stateless and thus are not concerned with fault tolerance and scaling; they rely on the storage service for these features.

The routing service is a stateless client-facing API that provides a stable abstraction above the internal dynamics of the system. The resource allocation of each tier may be in flux—and whole tiers may be added or removed at workload extremes—but clients are isolated from these changes. The routing service consistently returns a correct endpoint that will answer client requests. Finally, the cluster management system is another stateless service that modifies resource allocation based on decisions reached by the policy engine.

3.2.2 Storage System

Figure 3.2 shows the architecture of Anna's storage kernel. As discussed in Chapter 2, the kernel contains many worker threads, and each thread interacts with a thread-local storage medium (a

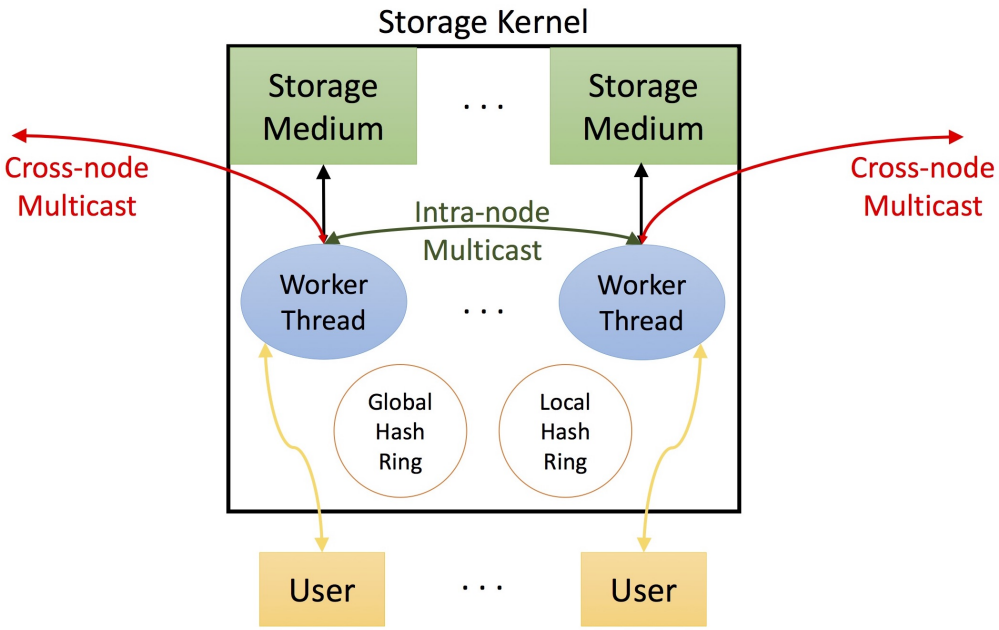

Figure 3.2: The architecture of storage kernel.

memory buffer or disk volume), processes client requests, and sends & receives multicasts to & from other Anna workers.

The Anna storage kernel is deployed across many storage tiers. The only difference between tiers is the procedure for translating data for persistence (serialization/deserialization, a.k.a. "serde"). Memory-tier nodes read from and write to local memory buffers, while disk-tier nodes serialize data into files that are stored on EBS volumes. Anna's uniformity across storage tiers makes adding additional tiers very simple: we set the serde mode and adjust the number of worker threads based on the underlying hardware. For instance, the total number of threads for memory nodes matches the number of CPU cores to fully utilize computing resources and to avoid costly preemption of threads. However, in other storage tiers where the performance bottleneck lies in serializing the key-value pairs to and from persistent storage, the optimal strategy for resource allocation is different. Our EBS tier allocates 4× as many threads per node (4) as we have physical CPU core (1).

Anna uses consistent hashing [52] to partition and replicate keys. For performance and fault tolerance (discussed further in Sections 3.3 and 3.5), each key may be replicated onto many nodes in each tier and multiple threads in each node. Following the model of early distributed hash tables, we use virtual nodes [83] in our consistent hashing algorithm. Each physical node (or thread) handles traffic for many virtual nodes (or threads) on the hash ring to ensure an even distribution. Virtual nodes also enable us to add heterogeneous nodes in the future by allocating more virtual nodes to more powerful physical machines.

3.2.3 Metadata Management

Anna requires maintaining certain metadata to efficiently support mechanisms discussed in Section 3.1 and help the policy engine adapt to changing workloads. In this section, we introduce the types of metadata managed by Anna and how they are stored and used by various system components.

Types of Metadata

Anna manages three distinct kinds of metadata. First, every storage tier has two *hash rings*. A global hash ring, G, determines which nodes in a tier are responsible for storing each key. A local hash ring, L, determines the set of worker threads *within* a single node that are responsible for a key.

Second, each individual key K has a *replication vector* of the form $[< R_1, ...R_n >, < T_1, ...T_n >]$. R_i represents the number of nodes in tier i storing key K, and T_i represents the number of threads *per node* in tier i storing key K. In our current implementation, i is either M (memory tier) or E (EBS tier). During request handling and multicast, both hash rings and key K's replication vector are used to determine the threads responsible for the key. For every tier, i, that maintains a replica of K, we first hash K against G_i, tier i's global hash ring to determine which nodes are responsible for K. We then look at L_i, tier i's local hash ring to determine which threads are responsible for the key.

Lastly, Anna tracks monitoring statistics, such as the access frequency of each key and the storage consumption of each node. This information is analyzed by the policy engine to trigger actions in response to variations in workload. Currently, we store 16 bytes of metadata per key and about 10 KB of metadata per worker thread.

Metadata Storage

Clearly, the availability and consistency of metadata is as important as that of regular data—otherwise, Anna would be unable to determine a key's location (under changing node membership and keys' replication vectors) or get an accurate estimate of workload characteristics and resource usage. In many systems [97, 103, 57, 109], metadata is enmeshed in the implementation of "master nodes" or stateful services like ZooKeeper [47]. Anna simply stores metadata in the storage system. Our metadata automatically derives all the benefits of our storage system, including performance guarantees, fault tolerance, and consistency. Anna employs last-writer-wins consistency to resolve conflicts among metadata replicas. Due to the eventual consistency model, worker threads may have stale views of hash rings and replication vectors. This can cause threads to disagree on the location of a key and can potentially cause multiple rounds of request redirection. However, since the metadata will eventually converge, threads will agree on the key's location, and requests will reach the correct destination. Note that multicast is performed every few seconds, while cluster state changes on the order of minutes, so cluster state metadata is guaranteed to converge before it undergoes further changes.

Enabling Mechanisms

Interestingly, manipulating two of these types of metadata (hash rings and replication vectors) is the key to enabling the mechanisms described earlier in Section 3.1. In this section, we discuss *only* the implementation of each mechanism. When and why each action is executed is a matter of policy and will differ based on system configuration and workload characteristics—we save this discussion for Section 3.3.

Elasticity. Anna manages cluster churn similarly to previous storage systems [33, 12] that use consistent hashing and distributed hash tables. When a new node joins a storage tier, it queries the storage system to retrieve the hash ring, updates the ring to include itself, and broadcasts its presence to all nodes in the system—storage, monitoring, and routing. Each existing node updates its copy of the hash ring, determines if it stores any keys that the new node is now responsible for, and gossips those keys to the new node. Similarly, when a node departs, it removes itself from the hash ring and broadcasts its departure to all nodes. It determines which nodes are now responsible for its data and gossips its keys to those nodes. Once all data has been broadcast, the node goes offline and its resources are deallocated.

Key migration overheads can be significant (see Section 3.5.4). To address this challenge, Anna interleaves key migration with client request handling to prevent system downtime. This is possible due to Anna's support for coordination-free consistency: The client may retrieve stale data during the key migration phase, but it can maintain a client-side cache and merge future retrieved results with the cached value. Anna's lattice-based conflict resolution guarantees that the state of the cached data is monotonically growing.

Selective Replication & Cross-Tier Data Movement. Both these mechanisms are implemented via updates to replication vectors. Each key in our two-tier implementation has a default replication vector of the form $[< 1, k >, < 1, 1 >]$, meaning that it has one memory tier replica and k EBS-tier replicas. Here, k is the number of replica faults per key the administrator is willing to tolerate (discussed further in Section 3.2.7 and 3.3). By default, keys are not replicated across threads within a single node. Anna induces cross-tier data movement by simply manipulating metadata. It increments the replication factor of one tier and decrements that of the other tier; as a result, gossip migrates data across tiers. Similarly, selective replication is achieved by adjusting the replication factor in each tier, under the fault tolerance constraint. After updating the replication vector, Anna updates metadata across replicas via asynchronous multicast.

3.2.4 Monitoring System & Policy Engine

In this section, we discuss the design of the monitoring system and the policy engine. As shown in Figure 3.3, each monitoring node has a monitoring thread, a statistics buffer, and a policy engine. The monitoring thread is stateless and periodically retrieves the stored statistics from the storage engine and triggers the policy engine. Note that per-key statistics such as key access frequency are reported by each worker thread that maintains the key replica. The monitoring thread aggregates the per-key statistics across all replicas before sending them to the policy engine.

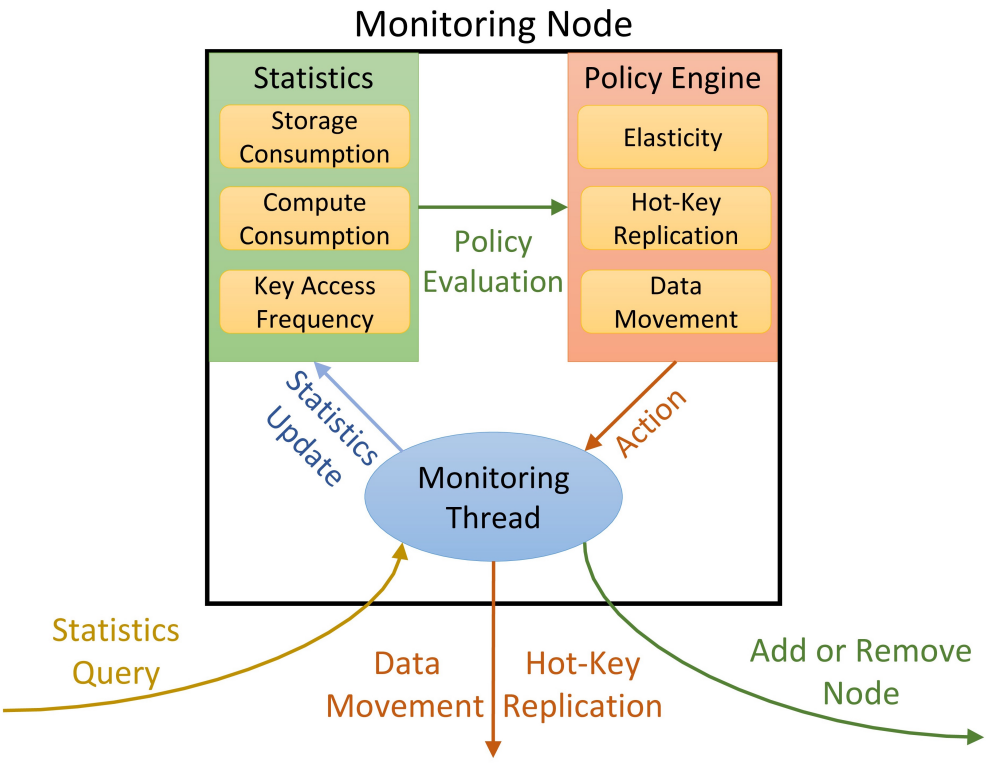

Figure 3.3: Monitoring node architecture.

The policy engine analyzes these statistics and issues actions to meet its SLOs. Anna currently supports three types of actions: *elasticity change*, *hot-key replication*, and *cross-tier data movement*. The implementation of these actions is covered above in Section 3.2.3. We discuss when each of these actions is triggered and describe the end-to-end policy algorithm in Section 3.3.

3.2.5 Routing Service

The routing service isolates clients from the underlying storage system: A client asks where to find a key and is returned the set of all valid addresses for that key. Anna's routing service only maintains soft state. Each routing node caches the storage tiers' hash rings and key replication vector metadata to respond to the clients' key address requests. If a key has any memory-tier replicas, the routing service only returns memory-tier addresses to maximize performance. The client caches these addresses locally to reduce request latency and load on the routing service.

When a client's cached address set becomes invalid because of a change in cluster configuration, a storage server receiving an invalid request will give the client the correct set of addresses. These will again be cached until they are invalidated, and the routing service will also refresh its cached cluster state.

3.2.6 Cluster Management

Anna uses Kubernetes [55] as a cluster management tool. Kubernetes is responsible for allocating and deallocating nodes, ensuring that nodes are alive, and rebooting failed nodes. An Anna deployment has four kinds of nodes: storage nodes, routing nodes, monitoring nodes, and a single, stateless "cluster management" node described below.

A "pod" is the atomic unit of a Kubernetes application and is a collection of one or more Docker [34] containers. Each node in Anna is instantiated in a separate Kubernetes pod, and each pod contains only one instance of a Anna node. Storage system and routing service pods are pinned on separate EC2 instances for resource isolation purposes. The monitoring system is less resource intensive and can tolerate preemption, so it is not isolated. Finally, Anna maintains a singleton cluster management pod, whose role is to issue requests to add or remove nodes to the Kubernetes cluster. A simple, stateless Python server in this pod receives REST requests from the policy engine and uses bash scripts to add or remove nodes.

3.2.7 Fault Tolerance

Anna guarantes k-fault tolerance by ensuring $k + 1$ replicas are live at all times. The choice of k determines a trade-off between resilience and cost. The $k + 1$ replicas of each key can be spread across tiers arbitrarily, according to hotness.

When a storage node fails, other nodes detect the failure via a timeout and remove the node from the hash ring. When such a timeout happens, Anna automatically repartitions data using the updated hash ring. The cluster management pod then issues a request to spawn a new node, which enters the join protocol discussed in Section 3.2.3.

Anna does not rely on the persistence of EBS volumes for fault tolerance in the disk tier. Similar to nodes in the memory tier, these nodes lose their state when they crash—this is desirable because it allows all tiers to be symmetric, regardless of the durability of the underlying storage medium.

Both routing nodes and monitoring nodes only store soft state and do not require any recovery mechanisms. If a routing node fails, it queries other routing nodes for up-to-date cluster information, and if a monitoring node fails, it retrieves system statistics from the storage service.

When the cluster management pod fails, Kubernetes automatically revives it. No recovery is necessary as it does not manage any state. The state of the cluster will not change while the pod is down since it is the actor responsible for modifying resource allocation. As a result, the policy engine will re-detect any issue requiring an elasticity change before the crash and re-issue the request upon revival.

In summary, Anna consists of a stateful storage kernel that is partitioned and selectively replicated for performance and fault tolerance with multi-master updates. Every other component is either stateless and optionally caches soft state that is easily recreated. As a result, the only single point of failure in Anna is the Kubernetes master. Kubernetes offers high-availability features to mitigate this problem [56]. We also note that Kubernetes is not integral to the design of Anna; we

rely on it primarily to bootstrap the system and reduce the engineering burden of mundane tasks such as receiving heartbeats, allocating VMs, and deploying containers.

3.3 Policy Engine

Anna supports three kinds of SLOs: an average request latency (L_{obj}) in milliseconds, a cost budget (B) in dollars/hour, and a fault tolerance (k) in number of replicas. The fault tolerance indicates the allowed number of replica failures, k. The latency objective, L_{obj}, is the average expected request latency. The budget, B, is the maximum cost per hour that will be spent on Anna.

As discussed in Section 3.2.7, Anna ensures there will never be fewer than $k + 1$ replicas of each key to achieve the fault tolerance goal. The latency objective and cost budget goals, however, are conflicting. The cheapest configuration of Anna is to have $k+1$ EBS nodes and 1 memory node (for metadata). Clearly, this configuration will not be very performant. If we increase performance by adding memory nodes to the system, we might exceed our budget. Conversely, if we strictly enforce the budget, we might not be able to achieve the latency objective.

Anna administrators only specify *one* of the two goals. If a latency SLO is specified, Anna minimizes cost while meeting the latency goal. If the budget is specified, Anna uses no more than $\$B$ per hour while maximizing performance.

In Sections 3.3.1, 3.3.2, and 3.3.3, we describe heuristics to trigger each policy action—data movement, hot key replication, and elasticity. In Section 3.3.4, we present Anna's complete policy algorithm, which combines these heuristics to achieve the SLO. Throughout this section, we represent each key's replication vector as $[< R_M, R_E >, < T_M, T_E >]$ (a general form is defined in Section 3.2.3) since our initial prototype only uses two tiers—M for memory and E for EBS.

3.3.1 Cross-Tier Data Movement

Anna's policy engine uses its monitoring statistics to calculate how frequently each key was accessed in the past T seconds, where T is an internal parameter. If a key's access frequency exceeds a configurable threshold, P, and all replicas currently reside in the EBS tier, Anna promotes a single replica to the memory tier. If the key's access frequency falls below a separate internal threshold, D, and the key has one or more memory replicas, all replicas are demoted to the EBS tier. The EBS replication factor is set to $k + 1$, and the local replication factors are restored to 1. Note that in Anna, all metadata is stored in the memory tier, is never demoted, and has a constant replication factor. If the aggregate storage capacity of a tier is full, Anna adds nodes (Section 3.3.3) to increase capacity before performing data movement. If the budget does not allow for more nodes, Anna employs a least-recently used caching policy to demote keys.

3.3.2 Hot-Key Replication

When the access frequency of a key stored in the memory tier increases, hot-key replication increases the number of memory-tier replicas of that key. In our initial implementation, we configure

only the memory tier to replicate hot keys. Because the EBS tier is not intended to be as performant, a hot key in that tier will first be promoted to the memory tier before being replicated. This policy will likely vary for a different storage hierarchy.

The policy engine classifies a key as "hot" if its access frequency exceeds an internal threshold, H, which is s standard deviations above the mean access frequency. Because Anna is a shared-nothing system, we can replicate hot keys both across cores in a single node and across nodes. Replicating across nodes seems preferable, because network ports are a typical bottleneck in distributed system, so replicating across nodes multiplies the aggregate network bandwidth to the key. However, replicating across cores within a node can also be beneficial, as we will see in Section 3.5.2. Therefore, hot keys are *first* replicated across more nodes before being replicated across threads within a node.

The policy engine computes the target replication factor, R_{M_ideal}, using the ratio between the observed latency for the key and the latency objective. Cross-node replication is only possible if the current number of memory replicas, R_M, is less than the number of memory-tier nodes in the cluster, N_M. If so, we increment the key's memory replication factor to $min(R_{M\_ideal}, N_M)$. Otherwise, we increment the key's *local* replication factor on memory-tier machines up to the maximum number of worker threads (N_{T_memory}) using the same ratio. Finally, if the access frequency of a previously-hot key drops below a threshold, L, its replication vector is restored to the default: R_M, T_M, and T_E are all set to 1 and R_E is set to k.

3.3.3 Elasticity

Node Addition. Anna adds nodes when there is insufficient storage or compute capacity. When a tier has insufficient storage capacity, the policy engine computes the number of nodes required based on data size, subject to cost constraints, and instructs the cluster management service to allocate new nodes to that tier.

Node addition due to insufficient compute capacity only happens in the memory tier because the EBS tier is not designed for performance. Compute pressure on the EBS tier is alleviated by promoting data to the memory tier since a memory node can support 15× the requests at 4× the cost. The policy engine uses the ratio between the observed latency and the latency objective to compute the number of memory nodes to add. This ratio is bounded by a system parameter, c, to avoid overly aggressive allocation.

Node Removal. Anna requires a minimum of one memory node (for system metadata) and $k + 1$ EBS nodes (to meet the k-fault SLO when all data is demoted). The policy engine respects these lower bounds. We first check if any key's replication factor will exceed the total number of storage nodes in any tier after node removal. Those keys' replication factors are decremented to match the number of nodes at each tier before the nodes are removed. Anna currently only scales down the memory tier based on compute consumption and not based on storage consumption. This is because selective replication can significantly increase compute consumption without increasing storage consumption. Nonetheless, this may lead to wasteful spending under adversarial workloads; we elaborate in the next section.

Grace Periods. When resource allocation is modified, data is redistributed across each tier, briefly increasing request latency (see Section 3.5.4). Due to this increase, as well as data location changes, key access frequency decreases. To prevent over-correction during key redistribution, we apply a grace period to allow the system to stabilize. Key demotion, hot-key replication, and elasticity actions are all delayed till after the grace period.

3.3.4 End-to-End Policy

In this section, we discuss how Anna's policy engine combines the above heuristics to meet its SLOs. If the average storage consumption of *all* nodes in a particular tier has violated configurable upper or lower thresholds (S_{upper} and S_{lower}), nodes are added or removed respectively. We then invoke the data movement heuristic from Section 3.3.1 to promote and demote data across tiers. Next, the policy engine checks the average latency reported by clients. If the latency exceeds a fraction, f_{upper} (defaulting to 0.75), of the latency SLO and the memory tier's compute consumption exceeds a threshold, C_{upper}, nodes are added to the memory tier. However, if not all nodes are occupied, hot keys are replicated in the memory tier, as per Section 3.3.2. Finally, if the observed latency is a fraction, f_{lower} (defaulting to 0.5), below the objective and the compute occupancy is below C_{lower}, we invoke the node removal heuristic to check if nodes can be removed to save cost.

The compute threshold, C_{upper}, is set to 0.20. Consistent with Chapter 2, each storage node saturates its network bandwidth well before its compute capacity. Compute occupancy is a proxy for the saturation of the underlying network connection. This threshold varies significantly based on the hardware configuration; we found that 20% was optimal for our experimental setup (see Section 3.5).

Discussion

Storage Node Saturation. There are two possible causes for saturation. If all nodes are busy processing client requests, Anna must add more nodes to alleviate the load. Performing hot-key replication is not productive: Since all nodes are busy, replicating hot keys to a busy node will, in fact, decrease performance due to additional gossip overhead. The other cause is a skewed access distribution in which most client requests are sent to a small set of nodes serving the hot keys while most nodes are free. The optimal solution is to replicate the hot keys onto unsaturated nodes. If we add nodes to the cluster, the hot keys' replication factors will not change, and clients will continue to query the few nodes storing those keys. Meanwhile, the newly added nodes will idle. As discussed in Section 3.3.4, Anna's policy engine is able to differentiate the two causes for node saturation and take the appropriate action.

Policy Limitations. There are cases in which our policy engine fails to meet the latency objective and/or wastes money. Due to current cloud infrastructure limitations, for example, it takes about five minutes to allocate a new node. An adversary could easily abuse this limitation. A short workload spike to trigger elasticity, followed by an immediate decrease would lead Anna to allocate unnecessary nodes. These nodes will be under-utilized, but will only be removed if the observed

Table 3.2: A summary of all variables mentioned in Section 3.3.

Variable Name	Meaning	Default Value	Type
L_{obj}	Latency Objective	2.5ms	SLO Spec
B	Cost Budget	N/A (user-specified)	SLO Spec
k	Fault Tolerance	2	SLO Spec
T	Monitoring report period	15 seconds	Policy Knob
H	Key hotness threshold	3 standard deviations above the mean key access frequency	Policy Knob
L	Key coldness threshold	The mean key access frequency	Policy Knob
P	Key promotion threshold	2 accesses in 60 seconds	Policy Knob
$[S_{lower}, S_{upper}]$	Storage consumption thresholds	Memory: [0.3, 0.6] EBS: [0.5, 0.75]	Policy Knob
$[f_{lower}, f_{upper}]$	Latency thresholds	[0.5, 0.75]	Policy Knob
$[C_{lower}, C_{upper}]$	Compute occupancy thresholds	[0.05, 0.20]	Policy Knob
c	Upper bound for latency ratio	1.5	Policy Knob

latency drops below $f_{lower} * L_{obj}$. Unfortunately, removing this constraint would make Anna susceptible to reducing resource allocation during network outages, which is also undesirable. We discuss potential solutions to these issues in future work.

Knobs. There are a small number of configuration variables mentioned in this section, which are summarized in Table 3.2. We distinguish variables that are part of the external SLO Spec from the internal parameters of our current policy. In our evaluation, our parameters were tuned by hand to match the characteristics of the AWS services we use. There has been interesting work recently on autotuning database system configuration knobs [107]; our setting has many fewer knobs than those systems. As an alternative to auto-tuning our current knobs, we are exploring the idea of replacing the current threshold-based policy entirely with a dynamic Reinforcement Learning policy that maps directly and dynamically from performance metrics to decisions about system configuration changes. These changes to the policy engine are easy to implement, but tuning the policy is beyond the scope of this work: It involves extensive empirical work on multiple deployment configurations.

3.3.5 Algorithm Pseudocode

We include pseudocode for the algorithms described in Section 3.3 here. Note that some algorithms included here rely on a latency objective, which may or may not be specified. When no latency objective is specified, Anna aspires to its unsaturated request latency (2.5ms) to provide the best possible performance but caps spending at the specified budget.

Algorithm 1 DataMovement

Input: Key, $[< R_M, R_E >< T_M, T_E >]$
1: **if** access(Key, T)$> P$ & $R_M = 0$ **then**
2: adjust(Key, $R_M + 1$, $R_E - 1$, T_M, T_E)
3: **else if** access(Key, T)$< D$ & $R_M > 0$ **then**
4: adjust(Key, 0, $k + 1$, 1, 1)

3.4 Anna API

As shown in Table 3.3, Anna exposes seven APIs to the application: `Get`, `Put`, `Delete`, `GetAll`, `PutAll`, `GetDelta`, and `Subscribe`. The first three APIs are straightforward and discussed in detail in Chapter 2. Here, we introduce four new APIs that we added.

GetAll. Unlike the `Get` API, which queries a single replica of a key, `GetAll` queries all replicas of a key, merges them on the client side, and returns the merged result to the user. This allows the user to observe the most up-to-date state of a key in Anna. As an extension, the Anna client can query a subset of the replicas, achieving a trade-off between performance and data freshness.

Algorithm 2 HotKeyReplication

Input: Key, $[< R_M, R_E >< T_M, T_E >]$

if access(Key, T)$> H$ & $R_M < N_M$ **then**
    SET $R_{M\_ideal} = R_M * L_{obs}/L_{obj}$
    SET $R'_M = min(R_{M\_ideal}, N_M)$
    adjust(Key, R'_M, R_E, T_M, T_E)
else if access(Key, T)$> H$ & $R_M = N_M$ **then**
    SET $T_{M\_ideal} = T_M * L_{obs}/L_{obj}$
    SET $T'_M = min(T_{M\_ideal}, N_{T\_memory})$
    adjust(Key, R_M, R_E, T'_M, T_E)
else if access(Key, T)$< L$ & $(R_M > 1 \parallel T_M > 1)$ **then**
    adjust(Key, 1, k, 1, 1)

Algorithm 3 NodeAddition

Input: $tier$, $mode$

if $mode = storage$ **then**
    SET N_{target} = required_storage($tier$)
    if $Cost_{target} > Budget$ **then**
        SET N_{target} = adjust()
    add_node(tier, $N_{target} - N_{tier\_current}$)
else if $mode = compute$ & $tier = M$ **then**
    SET $N_{target} = N_{M\_current}$*min($L_{obs}/L_{obj}$, c)
    if $Cost_{target} > Budget$ **then**
        SET N_{target} = adjust()
    add_node(M, $N_{target} - N_{M\_current}$)

Algorithm 4 NodeRemoval

Input: $tier$, $mode$

if $mode = storage$ & $tier = E$ **then**
    SET N_{target} = max(required_storage(E), $k + 1$)
    reduce_replication()
    remove_node(E, $N_{E\_current} - N_{target}$)
else if $mode = compute$ & $tier = M$ **then**
    if $N_{M\_current} > 1$ **then**
        reduce_replication()
        remove_node(M, 1)

Algorithm 5 AnnaPolicy

Input: $tiers = \{M, E\}$, $keys$
for $tier$ in $tiers$ **do**
    if storage($tier$)> S_{upper} **then**
        NodeAddition($tier$, $storage$)
    else if storage($tier$)< S_{lower} **then**
        NodeRemoval($tier$, $storage$)
for $key \in keys$ **do**
    DataMovement(key)
if $L_{obs} > f_{upper} * L_{obj}$ & compute(M)> C_{upper} **then**
    NodeAddition(M, $compute$)
else if $L_{obs} > f_{upper} * L_{obj}$ & compute(M)<= C_{upper} **then**
    for $key \in keys_{memory}$ **do**
        HotKeyReplication(key)
else if $L_{obs} < f_{lower} * L_{obj}$ & compute(M)< C_{lower} **then**
    NodeRemoval(M, $compute$)

Table 3.3: Summary of Anna's APIs.

API	Description
`Get(key)->value`	Retrieves the value of `key` from a single replica.
`Put(key, value)`	Performs an update to a single replica of `key`.
`Delete(key)`	Deletes `key`.
`GetAll(key)->value`	Retrieves the value of `key` from all replicas and returns the merged result.
`PutAll(key, value)`	Performs an update to all replicas of `key`.
`GetDelta(key, id)`	Retrieves the value of `key` only when it has changed from the previously queried version, identified by `id`.
`Subscribe(key, address)`	Subscribes to `key` and receives the updated value at `address`.

When `GetAll` is used in place of `Get` on an 8-byte key with a replication factor of 3, we observe no performance change for the median latency (0.58ms) and a 6% performance degradation (from 0.68ms to 0.72ms) for the 99-th percentile latency. This is because the client needs to wait for responses from every node that stores a replica of the key before responding to the user, which impacts the tail latency. The node configuration of this micro-benchmark is the same as other experiments in Section 3.5.

PutAll. Anna's regular `Put` API updates a single replica of a key and relies on asynchronous gossip for the update to propagate to other replicas. However, if the node accepting the client update crashes before gossiping, the update will be lost. To avoid potential data loss, `PutAll` sends the updates to all replicas of a key and returns only when all replicas accept the update. As an extension, the Anna client can send the update to a subset of the replicas, achieving a trade-off

between performance and fault-tolerance.

When `PutAll` is used in place of `Put` on an 8-byte key with a replication factor of 3, we observe similar performance changes as in the comparison between `GetAll` and `Get`; the median latency increases from 0.58ms to 0.59ms and the 99-th percentile latency increases from 0.66ms to 0.69ms.

GetDelta. A client interacting with Anna may query the same key multiple times. `GetDelta` returns the value of the key only when its payload has changed since the previous query. This API takes two arguments: a key and a version identifier of the key from the previous query. `GetDelta` is currently supported under last-writer-wins consistency and causal consistency. When enabled, the client sends the version metadata (timestamp and vector clock, respectively) as part of the request for Anna to check if the payload has changed. In case the payload has not changed, Anna responds with just an acknowledgement. Since the payload is not shipped as part of the response, this optimization significantly reduces network overhead for workloads that involves large payload.

For keys with small payload, however, `GetDelta` may be less efficient than `Get`, as the overhead of shipping the version metadata becomes pronounced. Accordingly, the Anna client makes `GetDelta` request only when the payload is larger than the version identifier. Otherwise, it falls back to the regular `Get` protocol.

To support the `GetDelta` API, the Anna client maintains the following metadata for each previously queried key: the size of the payload and the key's version identifier. The size information is stored as an 8-byte integer. For last-writer-wins consistency, each key's timestamp is 8 bytes. For causal consistency, each key's vector clock typically ranges from 10 bytes to 1KB. The Anna client stores these identifiers in a buffer and uses LRU policy for eviction. When the previously queried key's version identifier is unavailable, Anna falls back to the regular `Get` protocol.

Subscribe. A client can subscribe to a key and get notified when the value of the key changes. `Subscribe` takes two arguments: a key to subscribe and a notification IP address. When the value of the key changes, Anna pushes the new value to the notification address. This API is suitable for applications that operate in a passive, event-driven mode. To support this API, for each key under subscription, Anna maintains a list of IPs that listen for the key updates.

3.5 Evaluation

In this section, we present an evaluation of Anna. We first explore the optimal instance type for Anna's memory-tier storage node that balances the CPU, memory, and network bandwidth (Section 3.5.1). This instance type is used throughout the experiments. We then explore the advantage of different replica placement strategies in Section 3.5.2. Next, we show the benefit of selective replication in Section 3.5.3. We demonstrate Anna's ability to detect and adapt to variation in workload volume, skew, and hotspots in Sections 3.5.4 and 3.5.5. Section 3.5.6 covers Anna's ability to respond to unexpected failures. Finally, Section 3.5.7 evaluates Anna's ability to trade off performance and cost according to its SLO.

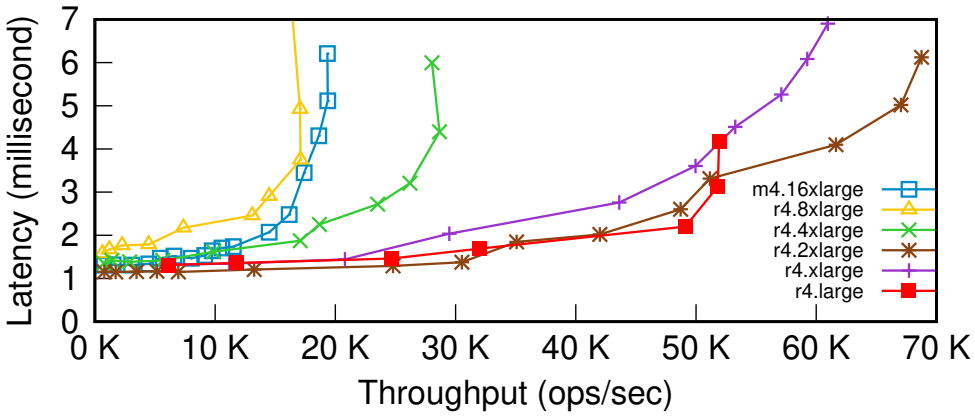

Figure 3.4: Performance comparison across different node types. The cost across all node types is set to $6.384 per hour. This corresponds to 3 `r4.8xlarge` nodes, 6 `r4.4xlarge` nodes, 12 `r4.2xlarge` nodes, 24 `r4.xlarge` nodes, and 48 `r4.large` nodes. The Zipfian coefficient of the workload is set to 0.5.

Anna uses `r4.2xlarge` instances for memory-tier nodes and `r4.large` instances for EBS-tier nodes. Each node has 4 worker threads; at peak capacity they can handle a workload that saturates the network link of the node. `r4.2xlarge` memory nodes have 61GB of memory, which is equally divided among all worker threads. Each thread in a EBS node has access to its own 64GB EBS volume. In our experiments, Anna uses two `m4.large` instances for the routing nodes and one `m4.large` instance for the monitoring node. We include these nodes in all cost calculation below. Unless otherwise specified, all experiments are run on a database with 1 million key-value pairs. Keys and values are 8 bytes and 256KB long, respectively. We set the k-fault tolerance goal to $k = 2$; there are 3 total replicas of each key. This leads to a total dataset size of about 750GB: $1M\ keys \times 3\ replicas \times 256KB\ values$.

Our workload is a YCSB-style read-modify-write of a single key chosen from a Zipfian distribution. We adjust the Zipfian coefficient to create different contention levels—a higher coefficient means a more skewed workload. We use the regular `Get` and `Put` APIs in our experiments. The clients were run on `r4.16xlarge` machines, with 8 threads each. Client machines ran in the same AWS region (us-east-1) as the server machines. Unless stated otherwise, experiments used 40 client machines for a total of 320 concurrent, single-threaded clients.

3.5.1 Node Configuration

Our first experiment explores the most cost-effective node configuration offered by the cloud vendor. Specifically, we explore which node types deliver the best performance relative to their cost on AWS for Anna's memory-tier. We mainly focus on the `r4` instance family, as nodes from this instance family are well-suited for high-performance in-memory databases.

For each instance type, we first adjust the number of nodes to match a fixed cost budget of $6.384 per hour. This corresponds to 3 `r4.8xlarge` nodes, 6 `r4.4xlarge` nodes, 12 `r4.2xlarge` nodes, 24 `r4.xlarge` nodes, and 48 `r4.large` nodes. We then record system's latency and throughput as we increase the workload volume (number of concurrent client threads) until the system is saturated. In this experiment, we set the Zipfian coefficient to 0.5. We see in Figure 3.4 that performance characteristics vary significantly across node types. Interestingly, although larger instance types such as `m4.16xlarge`, `r4.8xlarge`, and `r4.4xlarge` have abundant CPU cores and memory, their peak throughput is relatively poor (less than 30K operations per second). According to Anna's monitoring system, their CPU utilization at peak throughput are all below 15% and performance is bottlenecked by the network bandwidth (10Gbps - 25Gbps), consistent with the observations made in Chapter 2. On the other hand, small instances such as `r4.large` and `r4.xlarge` deliver better peak throughput (60K operations per second). However, these instances are bottlenecked by the CPU, as we observe that their CPU utilization are above 95% at peak throughput.

In comparison, `r4.2xlarge` instance achieves the best peak throughput (70K operations per second) with a CPU utilization of around 80%. Therefore, it has the best balance of CPU, memory, and network resources for our workload and we pick this instance for Anna's memory-tier for the rest of the experiments.

3.5.2 Replica Placement

In this section, we compare the benefits of intra-node vs. cross-node replication; for brevity, no charts are shown for this topic. On 12 memory-tier nodes, we run a highly skewed workload with the Zipfian coefficient set to 2. With a *single* replica per key, we observe a maximum throughput of just above 2,000 operations per second (ops). In the case of cross-node replication, four *nodes* each have one thread responsible for each replicated key; in the intra-node case, we have only one node with four *threads* responsible for each key. Cross-node replication improves performance by a factor of four to 8,000 ops, while intra-node replication only improves performance by a factor of two to 4,000 ops. This is because the four threads on a single node all compete for the same network bandwidth, while the single threads on four separate nodes have access to four times the aggregate bandwidth. Hence, as discussed in Section 3.3.2, we prioritize cross-node replication over intra-node replication whenever possible but *also* take advantage of intra-node replication.

3.5.3 Selective Replication

A key weakness of our initial work in Chapter 2 (referred to as Anna v0) is that all keys are assigned a uniform replication factor. A poor choice of replication factor can lead to significant performance degradation. Increasing the replication factor boosts performance for skewed workloads, as requests to hot keys can be processed in parallel on different replicas. However, a uniform replication factor means that cold keys are *also* replicated, which increases gossip overhead (slowing down the system) and storage utilization (making the system more expensive). By contrast,

Anna selectively replicates hot keys to achieve high performance, without paying a storage cost for replicating cold keys.

This experiment explores the benefits of selective replication by comparing Anna's memory-tier against Anna v0, AWS ElastiCache (using managed Memcached), and a leading research system, Masstree [73], at various cost points. We hand-tune Anna v0's single replication factor to the optimal value for each Zipfian setting and each cost point. This experiment uses a database of 100,000 keys across all cost points; we use a smaller database since the data must fit on one node, corresponding to the minimum cost point. We configure keys in Anna to have a default replication factor of 1 since neither ElastiCache nor Masstree supports replication of any kind. To measure the performance for a fixed price, we also disabled Anna's elasticity mechanism.

Figure 3.5(a) shows that Anna consistently outperforms both Masstree and ElastiCache under low contention. As discussed in our previous work, this is because Anna's thread-per-core coordination-free execution model efficiently exploits multi-core parallelism, while other systems suffer from thread synchronization overhead through the use of locks or atomic instructions. Neither Anna nor Anna v0 replicates data in this experiment, so they deliver identical performance.

Under high contention (Figure 3.5(b)), Anna's throughput increases linearly with cost, while both ElastiCache and Masstree plateau. Anna selectively replicates hot keys across nodes and threads to spread the load, enabling this linear scaling; the other two systems do not have this capability. Anna v0 replicates the *entire* database across all nodes. While Anna v0's performance scales, the absolute throughput is worse than Anna's because naively replicating the entire database increases multicast overhead for cold keys. Furthermore, Anna v0's storage consumption is significantly higher: At \$7.80/hour (14 memory nodes), Anna v0's constant replication generates 13$\times$ the original data size, while Anna incurs $<$1% extra storage overhead.

3.5.4 Dynamic Workload Skew & Volume

We now combine selective replication and elasticity to react to changes in workload skew and volume. In this experiment, we start with 12 memory-tier nodes and a latency objective of 3.3ms—about 33% above our unsaturated latency. All servers serve a light load at time 0. At minute 3, we start a high contention workload with a Zipfian coefficient of 2. We see in Figure 3.6(a) that after a brief spike in latency, Anna replicates the highly contended keys and meets the latency SLO (the dashed red line). At minute 13, we reduce the Zipfian coefficient to 0.5, switching to a low contention workload. Simultaneously, we increase the load volume by a factor of 4. Detecting these changes, the policy engine reduces the replication factors of the previously-hot keys. It finds that all nodes are occupied with client requests and triggers addition of four new nodes to the cluster. We see a corresponding increase in the system cost in Figure 3.6(b).

It takes 5 minutes for the new nodes to join the cluster. Throughput increases to the saturation point of all nodes (the first plateau in Figure 3.6(b)), and the latency spikes to the SLO maximum from minutes 13 to 18. At minute 18, the new nodes come online and trigger a round of data repartitioning, seen by the brief latency spike and throughput dip. Anna then further increases throughput and meets the latency SLO. At the 28-minute point, we reduce the load, and Anna removes nodes to save cost.

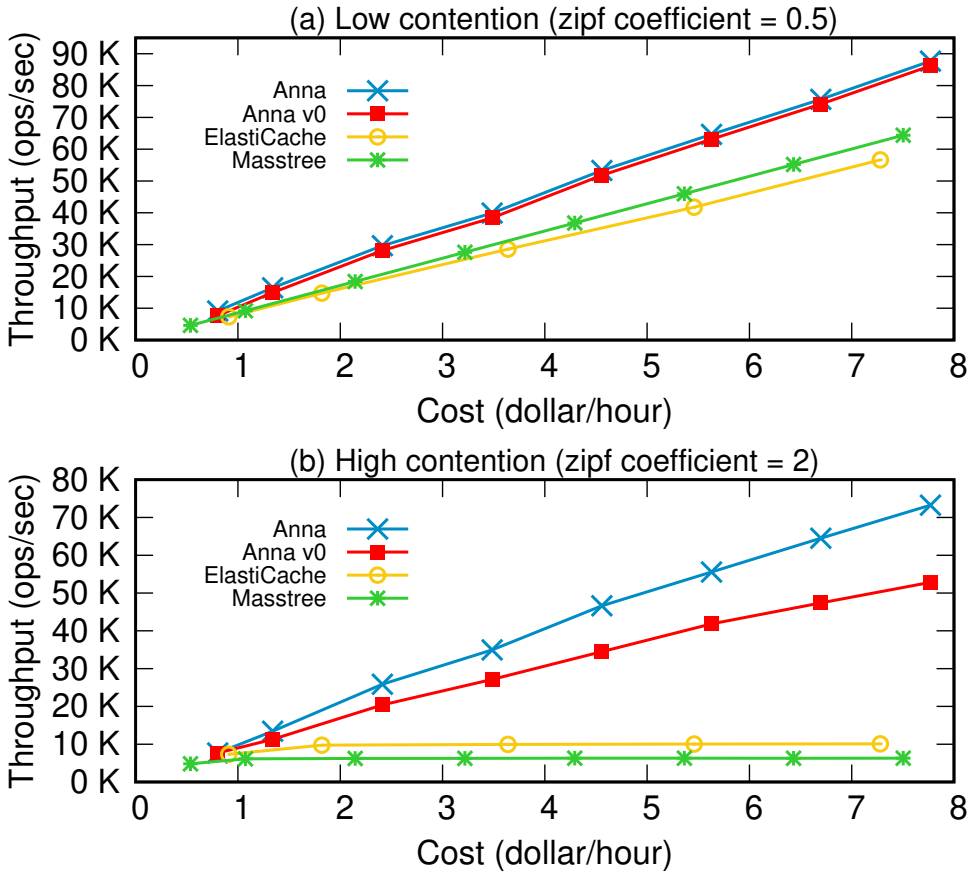

Figure 3.5: Cost-effectiveness comparison between Anna, Anna v0, ElastiCache, and Masstree.

Throughout the 32-minute experiment, the latency SLO is satisfied 97% of the time. We first violate the SLO during hot-key replication by 4× for 15 seconds. Moreover, the latency spikes to 7× the SLO during redistribution for about 30 seconds. Data redistribution causes multicast overhead on the storage servers and address cache invalidation on the clients. The latency effects are actually not terrible. As a point of comparison, TCP link latencies in data centers are documented tolerating link delays of up to 40× [6].

From minutes 13 to 18, we meet our SLO of 3.3ms exactly. With a larger load spike or lower initial resource allocation, Anna could have easily violated its SLO during that period, putting SLO satisfaction at 83%—a much less impressive figure. Under any reactive policy, large workload variations can cause significant SLO violations. As a result, cloud providers commonly develop client-specific service level agreements (SLAs) that reflect access patterns and latency expectations. In practice, these SLAs allow for significantly more leeway than a service's internal SLO [42].

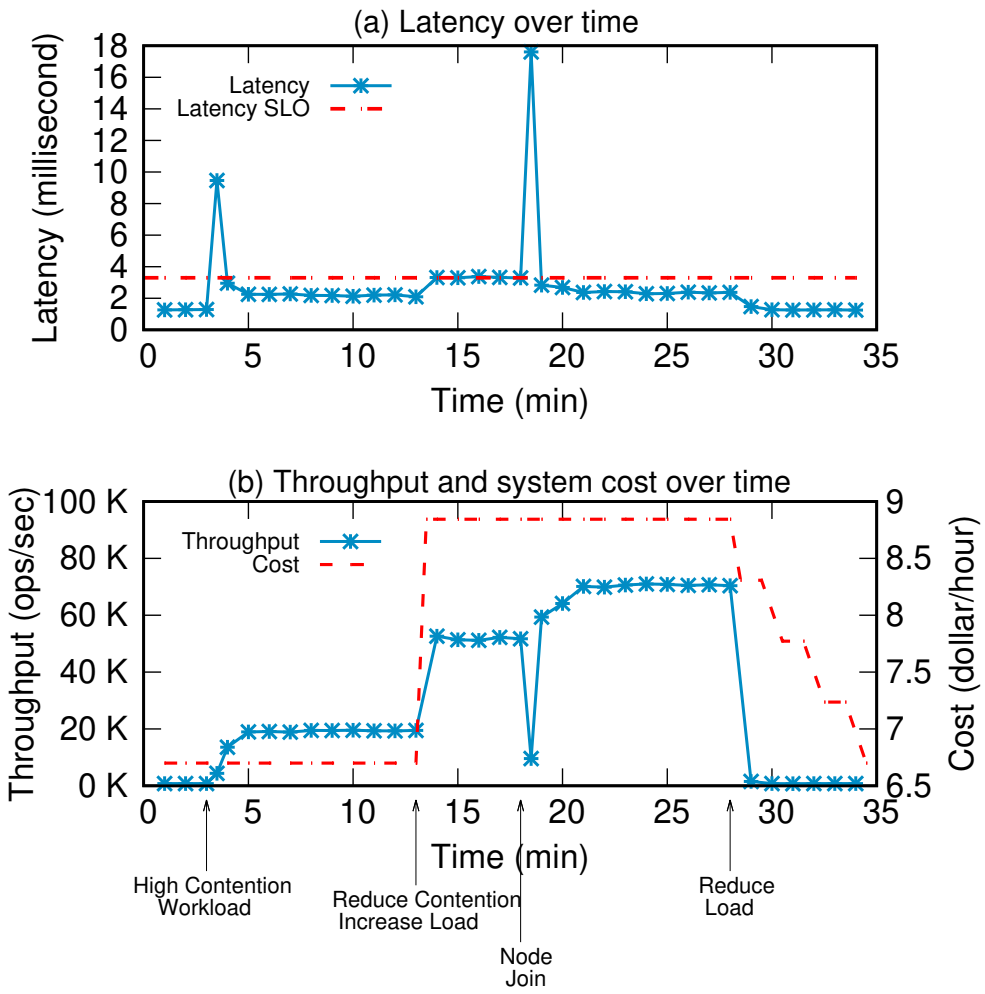

Figure 3.6: Anna's response to changing workload.

3.5.5 Varying Hotspot

Next, we introduce multiple tiers and run a controlled experiment to demonstrate the effectiveness of cross-tier promotion and demotion. The goal is to evaluate Anna's ability to detect and react to changes in workload hotspots. We do not consider a latency objective and disable autoscaling; we narrow our focus to how quickly Anna identifies hot data.

We fix total data size while varying the number of keys and the length of the values. This stress-tests selective replication, for which the amount of metadata (i.e., a per-key replication vector) increases linearly with the number of keys. Increasing the number of keys helps us evaluate how robust Anna's performance is under higher metadata overheads.

We allocate 3 memory nodes (insufficient to store all data) and 15 EBS-tier nodes. At time 0, most data is in the EBS tier. The blue curve in Figure 3.7 shows a moderately skewed workload,

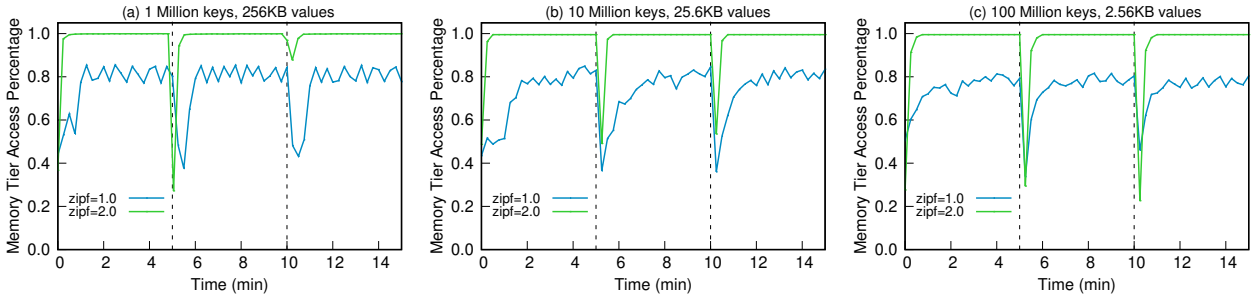

Figure 3.7: Adapting to changing hotspots in workload.

and the green curve shows a highly skewed workload. At minute 0, we begin a workload centered around one hotspot. At minute 5, we switch to a different, largely non-overlapping hotspot, and at minute 10, we switch to a third, unique hotspot. The y-axis measures what percent of queries are served by the memory tier—the "cache hit" rate.

With 1 million keys and 256KB values (Figure 3.7(a)), we see that Anna is able to react almost immediately and achieve a perfect hit rate under a highly skewed workload (the green curve). The hot set is very small—on the order of a few thousand keys—and all hot keys are promoted in about ten seconds. The moderately skewed workload shows more variation. We see the same dip in performance after the hotspot changes; however, we do not see the same stabilization. Because the working set is much larger, it takes longer for hot keys to be promoted, and there is a probabilistic "fringe" of keys that are in cold storage at time of access, leading to hit-rate variance. Nonetheless, Anna is still able to achieve an average of 81% hit rate less than a minute after the change.

Increasing the number of keys (Figures 3.7(b, c)) increases the time to stabilization. Achieving a hit-rate of 99.5% under the highly skewed workload (the green curves) takes around 15 and 25 seconds for 10 million and 100 million keys, respectively. Under the moderately skewed workload (the blue curves), the hit-rate in both settings takes around 90 seconds to stabilize. We observe a slightly reduced average hit-rate (79% and 77%, respectively) due to a larger probabilistic fringe of cold keys. Overall, despite orders of magnitude more keys, Anna still adapts and achieves a high memory tier hit-rate. In Section 3.7, we discuss opportunities to improve time to stabilization further via policy tuning.

3.5.6 Recovery

We evaluate Anna's ability to recover from node failure and compare against Redis on AWS ElastiCache. We choose Redis because it is the only KVS in our experiments with recovery features. Both systems were run on 42 memory-tier nodes and maintain three replicas per key. The results are shown in Figure 3.8. Note that we report *normalized* throughput here to compare against each system's baseline.

Both systems are run at steady state before a random node is terminated non-gracefully at minute 4, marked with a red line in Figure 3.8. Anna (the blue curve) experiences a 1.5-minute dip

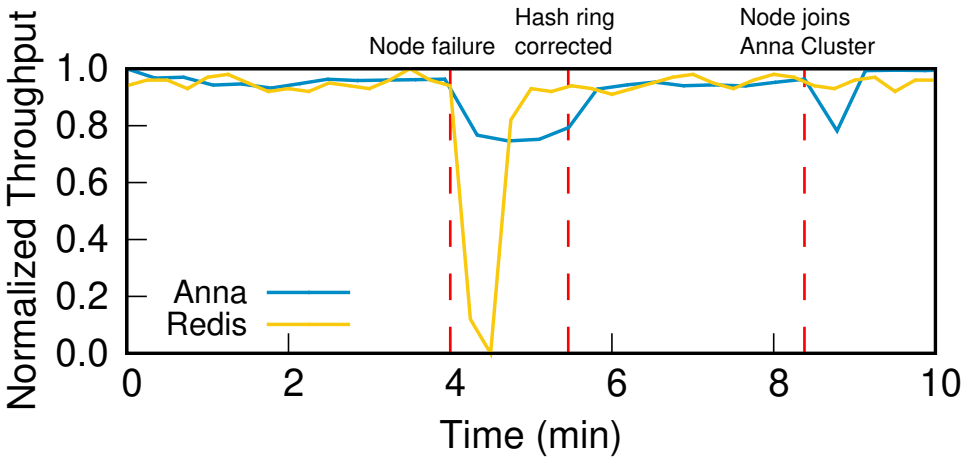

Figure 3.8: Impact of node failure and recovery for Anna and Redis (on AWS ElastiCache).

in performance while requests to the now-terminated node timeout. The performance change is not immediately apparent as the node continues serving requests for a few seconds before all processes are terminated. Nonetheless, Anna maintains above 80% of peak throughput because replication is multi-mastered; the remaining replicas still serve requests. After a minute, the system detects a node has departed and updates its hash ring. There is slightly diminished performance (about 90% of peak) from minutes 5.5 to 8.5 while the system operates normally but with 1 fewer node. At minute 8.5, we see another dip in throughput as a new node joins and the system repartitions data[1]. By minute 9, the system returns to peak performance.

User requests are set to time out after 500ms. We observe a steady state latency of about 18ms. After node failure, roughly $\frac{1}{42}$ of requests query the failed node and wait until timeout to retry elsewhere. This increases latency for those requests, but reduces load on live nodes; as a result, other requests observe latencies drop to about 10ms. Hence with one failed node, we expect to see an average latency of $510 \times \frac{1}{42} + 10 \times \frac{41}{42} = 21.90$ms. This implies throughput at roughly 82% of peak and matches the performance in Figure 3.8. A larger cluster would further mitigate the performance dip.

Redis maintains 14 shards, each with one primary and two read replicas. We terminate one of the primary replicas. The yellow curve in Figure 3.8 shows that throughput immediately drops to 0 as Redis stops serving requests and elects a new leader for the replica group with the failed node. A new node is allocated and data is repartitioned by minute 6, after which Redis returns to peak performance. As a single-master system that provides linearizability, it is not designed to run in an environment where faults are likely.

In summary, Anna is able to efficiently respond to node failure while maintaining over 80% peak throughput, whereas Redis pauses the system during leader election. Anna's high availability

[1]Note that repartitioning overhead is not as high as in Section 3.5.4 because here we are using more machines and only add one new node, as opposed to four in that experiment.

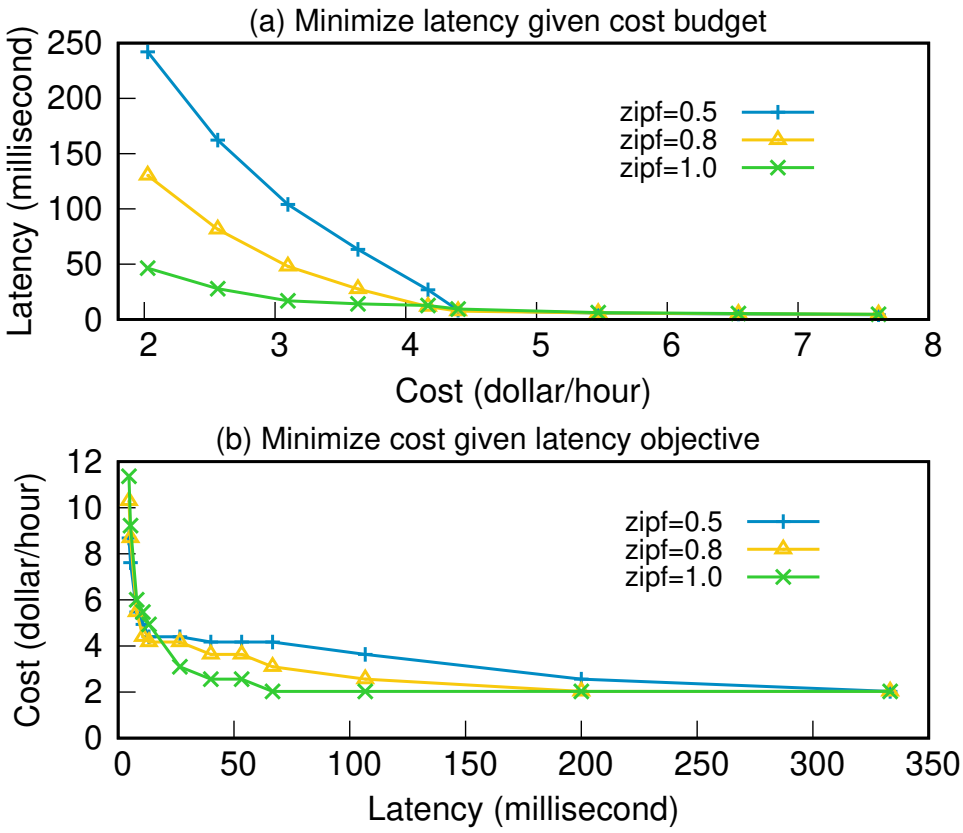

Figure 3.9: Varying contention, we measure (a) Anna latency per cost budget; (b) Anna cost per latency objective.

makes it a much better fit for cloud deployments. On the other hand, Redis's performance normalizes after a minute as its node spin-up time is much lower than ours (about 4 minutes)—we return to this point in Section 3.7.

3.5.7 Cost-Performance Tradeoffs

Finally, we assess how well Anna is able to meet its SLOs. We study the Pareto efficiency of our policy: How well does it find a frontier of cost-performance tradeoffs? We sweep the SLO parameter on one of the two axes of cost and latency and observe the outcome on the other. Anna uses both storage tiers and enable all policy actions. We evaluate three contention levels—Zipfian coefficients of 0.5 (about uniform), 0.8, and 1.0 (moderately skewed). For a database of 1M keys with a three replicas per key, Anna needs four EBS nodes to store all data and one memory node for metadata; this is a minimum deployment cost of $2.06 per hour.

At each point, we wait for Anna to achieve steady state, meaning that nodes are not being added

Table 3.4: Throughput comparison between Anna and DynamoDB at different cost budgets.

Cost	Anna	DynamoDB
$2.50/hour	1271 ops/s	35 ops/s
$3.60/hour	3352 ops/s	55 ops/s
$4.40/hour	23017 ops/s	71 ops/s
$5.50/hour	33548 ops/s	90 ops/s
$6.50/hour	38790 ops/s	108 ops/s
$7.60/hour	43354 ops/s	122 ops/s

or removed and latency is stable. In Figure 3.9(a), we plot Anna's steady state latency for a fixed cost SLO. We measure average request latency over 30 seconds. At $2.10/hour (4 EBS nodes and 1 memory node), only a small fraction of hot data is stored in the memory tier due to limited storage capacity. The observed latency ranges from 50ms to 250ms across contention levels. Requests under the high contention workload are more likely to hit the small set of hot data in the memory tier. As we increase the budget, latency improves for all contention levels: more memory nodes are added and a larger fraction of the data is memory-resident. At $4.40/hour, Anna can promote at least one replica of all keys to the memory tier. From here on, latency is under 10ms across all contention levels. Performance differences between the contention levels are negligible thanks to hot-key replication.

We also compare the throughput between Anna and DynamoDB at each cost budget. Similar to Anna, DynamoDB is a serverless KVS. System deployment details, such as node instance type, node count, and resource utilization, are hidden from the users. DynamoDB allows users to define high-level objectives such as read/write capacity units which translate to maximum throughput, and cost budget. The system autoscales based on the workload to meet these goals. Note that in this experiment, DynamoDB is configured to provide the same eventual consistency guarantees and fault tolerance metric ($k = 2$) as Anna. As shown in Table 3.4, Anna outperforms DynamoDB by 36× under a low-cost regime and by as much as 355× at higher costs. Our observed DynamoDB performance is actually somewhat better than AWS's advertised performance [9], which gives us confidence that this result is a reasonable assessment of DynamoDB's efficiency.

Lastly, we set Anna to minimize cost for a stated latency objective (Figure 3.9(b)). Once more, when the system reaches steady state, we measure its resource cost. To achieve sub-5ms latency—the left side of Figure 3.9(b)—Anna requires $9-11 per hour depending on the contention level. This latency requires at least one replica of all keys to be in the memory tier. Between 5 and 200ms, higher contention workloads are cheaper, as hot data can be concentrated on a few memory nodes. For the same latency range, lower contention workloads require more memory and are thus more expensive. Above 200ms, most data resides on the EBS tier, and Anna meets the latency objective at about $2 an hour.

3.6 Related Work

As a KVS, Anna builds on prior work, both from the databases and distributed systems literature. Nonetheless, it is differentiated in how it leverages and combines these ideas to achieve new levels of efficiency and automation.

Autoscaling Cloud Storage. A small number of cloud-based file systems have considered workload responsiveness and autoscaling. Sierra [106] and Rabbit [10] are single-master systems that handle the problem of read and write offloading: when a node is inactive or overloaded, requests to blocks at that node need to be offloaded to alternative nodes. This is particularly important for writes to blocks mastered at the inactive node. SpringFS [115] optimizes this work by finding a minimum number of machines needed for offloading. By contrast, Anna supports multi-master updates and selective key replication. When nodes go down or get slow in Anna, writes are simply retried at any existing replica, and new replicas are spawned as needed by the policy.

ElastMan [96] is a "bolt-on" elasticity manager for cloud KVSes that responds to changing workload volume. Anna, on the other hand, manages the dynamics of skew and hotspots in addition to volume. ElastMan's proactive policy is an interesting feature that anticipates workload changes like diurnal patterns; we return to this in Section 3.7.

Consistent hashing and distributed hash tables [52, 101, 84] are widely used in many storage systems [33, 12] to facilitate dynamic node arrival and departure. Anna allows request handling and key migration to be interleaved, eliminating downtime during node membership change while ensuring consistency, thanks to its lattice-based conflict resolution.

Key-Value Stores. There has been a wide range of work on key-value stores for both multicore and distributed systems—more than we have room to survey. Chapter 2 offers a recent snapshot overview of that domain. Here, our focus is not on the KVS kernel, but on mechanisms to adapt to workload distributions and trade-offs in performance and cost.

Selective Key Replication. Selective replication of data for performance has a long history, dating back to the Bubba database system [30]. More recently, the ecStore [108], Scarlett [11], E2FS [25], and SWORD [81] systems perform single-master selective replication, which creates *read-only* replicas of hot data to speed up read performance. Content delivery network (CDN) providers such as Google Cloud CDN [41], Swarmify [104], and Akamai [3] use similar techniques to replicate content close to the edge to speed up delivery. In comparison, Anna's multi-master selective replication improves *both* read and write performance, achieving general workload scaling. Conflicting writes to different replicas are resolved asynchronously using our lattices' merge logic [114].

Selective replication requires maintaining metadata to track hot keys. ecStore uses histograms to reduce hot-key metadata, while Anna currently maintains access frequencies for the full key set. We are exploring two traditional optimizations to reduce overhead: heavy hitter sketches rather than histograms [62] and the use of distributed aggregation for computing sketches in parallel with minimal bandwidth [72].

Another effort to address workload skew is Blowfish [53], which combines the idea of replication and compression to trade-off storage and performance under time-varying workloads. Adding compression to Anna to achieve fine-grained performance cost trade-off is an interesting future

direction.

Tiered Storage. Beyond textbook caching, there are many interesting multi-tier storage systems in the literature. A classic example in the file systems domain is the HP AutoRaid system [111]. Databases also considered tertiary storage during the era of WORM devices and storage robots [102, 69]. Broadcast Disks envisioned using multiple broadcast frequencies to construct arbitrary hierarchies of virtual storage [2]. More recently, there has been interest in filesystem caching for analytics workloads. OctopusFS [50] is a tiered file system in this vein. Tachyon [64] is another recent system that serves as a memory cache for analytics working sets, backing a file system interface. Our considerations are rather different than prior work: The size of each tier in Anna can change due to elasticity, and the volume of data to be stored overall can change due to dynamic replication.

3.7 Conclusion and Takeaways

Anna provides a simple, unified API to efficient key-value storage in the cloud. Unlike popular storage systems today, it supports a non-trivial *distribution* of access patterns by eliminating common static deployment and cost-performance barriers. Developers declare their desired tradeoffs, instead of managing a custom mix of heterogenous services.

Behind this API, three core mechanisms are the keys for Anna to meet performance SLOs. *Horizontal elasticity* right-sizes the service by adding and removing nodes, while *vertical data movement across tiers* and *multi-master selective replication* scale request handling at a fine granularity. Integration of these features makes Anna an efficient, autoscaling system that represents a new design point for cloud storage. These features are enabled by a policy engine which monitors workloads and responds by taking the appropriate actions.

Our evaluation shows that Anna is extremely efficient. In many cases, Anna is orders of magnitude more cost-effective than popular cloud storage services and prior research systems. Anna is also unique in its ability to automatically adapt to variable workloads.

Up to now, we have focused on studying how to build scalable, performant serverless storage systems with rich consistency guarantees. In the next chapter, we turn our attention to the compute layer, specifically FaaS infrastructure, and discuss how the design principles of Anna inspired innovations in distributed caching and consistency protocols that simultaneously achieve smooth autoscaling, low latency request handling, and robust consistency models for a serverless computing platform.

Chapter 4

Low Latency Transactional Causal Consistency for Serverless Computing

As we discussed in Chapter 1, Serverless computing has gained significant attention recently, with a focus on Function-as-a-Service (FaaS) systems [44, 49, 5, 45, 77, 94]. These systems—e.g., AWS Lambda, Google Cloud Functions—allow programmers to upload arbitrary functions and execute them in the cloud without having to provision or maintain servers.

These platforms enable developers to construct applications as compositions of multiple functions [59]. For example, a social network generating a news feed might have three functions: authenticate a user, load posts in that user's timeline, and generate an HTML page. Functions in the workflow are executed independently, and different functions may not run on the same physical machine due to load balancing, fault tolerance, and varying resource requirements.

For developers, the key benefit of FaaS is that it transparently autoscales in response to workload shifts; more resources are provisioned when there is a burst in the request rate, and resources are de-allocated as the request rate drops. As a result, cloud providers can offer developers attractive consumption-based pricing. Providers also benefit from improved resource utilization, which comes from dynamically packing the current workload into servers.

FaaS platforms achieve flexible autoscaling by *disaggregating* the compute and storage layers, so they can scale independently. For example, FaaS applications built on AWS Lambda typically use AWS S3 or DynamoDB as the autoscaling storage layer [14]. This design, however, comes at the cost of high-latency I/O—often orders of magnitude higher than attached storage [44]. This makes FaaS ill-suited for low-latency services that would naturally benefit from autoscaling—e.g. webservers managing user sessions, discussion forums managing threads, or ad servers managing ML models. These services all dynamically manipulate data based on request parameters and are therefore sensitive to I/O latency.

A natural solution is to attach caches to FaaS compute nodes to eliminate the I/O latency for data that is frequently accessed from remote storage. However, this raises challenges around maintaining consistency of the cached data—particularly in the context of multi-I/O applications that may run across different physical machines with different caches [99, 100].

Returning to the social network setting, consider a scenario where Alice updates her photo

access permissions to ban Bob from viewing her pictures and then posts a picture that makes fun of him. When Bob views his timeline, the cache his request visits for Alice's permissions may not yet reflect her recent update, but the cache visited to fetch posts may include the picture that she posted. The authentication process mistakenly will allow Bob to view the picture, creating a consistency anomaly [79, 67, 29].

The source of this inconsistency is that reads and writes fail to respect *causality*: Bob first observes a stale set of permissions, then observes a photo whose write was influenced by a newer permission set. Such anomalies can be prevented by transactional causal consistency (TCC), the strongest consistency model that can be achieved without expensive consensus protocols [19, 71, 67] required in stricter consistency models such as serializable transactions.

However, providing TCC for low-latency applications creates unprecedented challenges in a serverless environment, where cluster membership rapidly changes over time due to autoscaling infrastructure and user requests span multiple compute nodes. Recent systems such as Cure [4] and Occult [74] enforce TCC at the storage layer, so FaaS-layer caches would need to access storage for each causally consistent I/O, which reintroduces network roundtrips and violates our low-latency goal. Moreover, the consistency mechanisms in prior work rely on fixed node membership, which we cannot assume of an autoscaling system.

An alternative approach is Bolt-on Causal Consistency [17] (BCC). BCC enforces consistency in a cache layer similar to the one proposed here and does not rely on fixed-size clusters. However, BCC only guarantees Causal+ Consistency [67], which is weaker than TCC and inadequate to prevent the anomaly described above. BCC also does not guarantee consistency across multiple caches.

To solve these challenges, we present HydroCache, a distributed caching layer attached to each node in a FaaS system. HydroCache simultaneously provides low-latency data access and introduces *multisite transactional causal consistency* (MTCC) protocols to guarantee TCC for requests that execute on multiple nodes. Our MTCC protocols do not rely on the membership of the system, and HydroCache does not interfere with a FaaS layer's crucial autoscaling capabilities. In summary, this chapter's contributions are:

1. The design of HydroCache, which provides low latencies while also guaranteeing TCC for individual functions executed at a single node (Section 4.2).
2. Efficient MTCC protocols to guarantee TCC for compositions of functions, whose execution spans multiple nodes (Section 4.3).
3. An evaluation that shows TCC offers an attractive trade-off between performance and consistency in a serverless setting and HydroCache achieves a 10× performance improvement over FaaS architectures without a caching layer while simultaneously offering stronger consistency (Section 4.4).

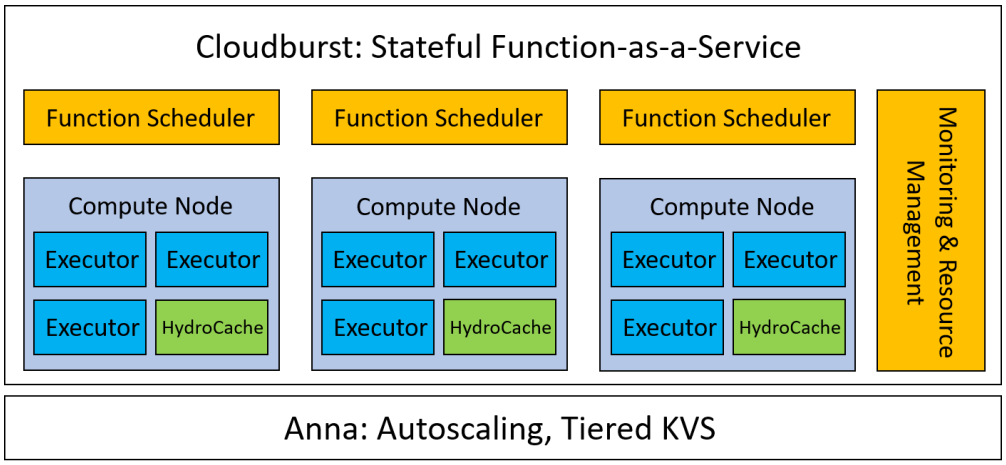

Figure 4.1: Cloudburst architecture.

4.1 Background

In this section, we briefly introduce the system architecture within which we implement HydroCache. We also define causal consistency and its extensions, which are essential for understanding the material in the rest of this chapter.

4.1.1 System Architecture

Figure 4.1 shows an overview of our system architecture, which consists of a high-performance key-value store (KVS) Anna [114, 112] and a function execution layer Cloudburst [100]. We chose Anna as the storage engine as it offers low latencies and flexible autoscaling, a good fit for serverless. Anna also supports custom conflict resolution policies to resolve concurrent updates. As we discuss in Section 4.1.2, this provides a necessary foundation to support causal consistency.

Cloudburst deploys KVS-aware caches on the same nodes as the compute workers, allowing for low-latency data access. We build our own function execution layer as there is no way to integrate our cache into existing FaaS systems.

In Cloudburst, all requests are received by a scheduler and routed to worker threads based on compute utilization and data locality heuristics. Each compute node has three function executor threads, each of which has a unique ID. As we discuss in Section 4.1.2, these IDs are used to capture causal relationships. The compute threads on a single machine interact with one HydroCache instance, which retrieves data for the function executors as necessary. The cache also transparently writes updates back to the KVS.

Cloudburst users write functions in vanilla Python, and register them with the system for execution. The system enables low-latency function chaining by allowing users to register function compositions forming a *DAG* of functions. DAG execution is optimized by automatically passing results from one function executor to the next. Each DAG has a single *sink* function with no downstream functions, the results of which are either returned to the user or written to Anna.

4.1.2 Causal Consistency

Causal Consistency (CC). Under CC, reads and writes respect Lamport's "happens-before" relation [61]. If a read of key a_i (i denotes a version of key a) influences a write of key b_j, then a_i *happens before* b_j, or b_j depends on a_i; we denote this as $a_i \rightarrow b_j$. *happens before* is transitive: if $a_i \rightarrow b_j \wedge b_j \rightarrow c_k$, then $a_i \rightarrow c_k$. In our system, dependencies are explicitly generated during function execution (known as explicit causality [20, 58]). A write causally depends on keys that the function previously read from storage.

A key k_i has four components $[k, VC_{k_i}, deps, payload]$; k is the key's identifier, VC_{k_i} is a *vector clock* [85, 98] that identifies its version, $deps$ is its dependency set, and $payload$ is the value. VC_{k_i} consists of a set of $\langle id, clock\rangle$ pairs where the id is the ID of a function executor thread that updated k_i, and the $clock$ represents that thread's monotonically growing logical clock. $deps$ is a set of $\langle dep\_key, VC\rangle$ pairs representing key versions that k_i depends on.

During writes, the VC and dependency set are modified as follows. Let thread e_1 write a_i. Thread e_2 then writes b_j with $a_i \rightarrow b_j$. a_i will have $VC_{a_i} = \langle\langle e_1, 1\rangle\rangle$ and an empty dependency set, and b_j will have $VC_{b_j} = \langle\langle e_2, 2\rangle\rangle$ and a dependency set $\langle\langle a_i, VC_{a_i}\rangle\rangle$. If another thread e_3 writes b_k such that $b_j \rightarrow b_k$, then b_k will have $VC_{b_k} = \langle\langle e_2, 2\rangle, \langle e_3, 3\rangle\rangle$ and a dependency set $\langle\langle a_i, VC_{a_i}\rangle\rangle$. In this example, 1, 2 and 3 are the values of logical clocks of e_1, e_2 and e_3 during the writes. Dependencies between versions of the same key are captured in the key's VC, and dependencies across keys are captured in the dependency set.

Given a_i and a_j, $a_i \rightarrow a_j \iff VC_{a_i} \rightarrow VC_{a_j}$. Let E be a set that contains all executor threads in our system. We define $VC_i \rightarrow VC_j$ as $\forall e \in E \mid e \notin VC_i \vee VC_i(e) \leq VC_j(e)$ and $\exists e' \in E \mid (e' \notin VC_i \wedge e' \in VC_j) \vee (VC_i(e') < VC_j(e'))$. In other words, VC_j "dominates" VC_i if and only if all $\langle id, clock\rangle$ pairs of VC_j are no less than the matching pairs in VC_i and at least one of them dominates. If $a_i \nrightarrow a_j \wedge a_j \nrightarrow a_i$, then a_i is *concurrent* with a_j, denoted as $a_i \sim a_j$.

CC requires that if a function reads b_j, which depends on a_i ($a_i \rightarrow b_j$), then the function can subsequently only read $a_k \mid a_k \nrightarrow a_i$—i.e. $a_k == a_i$, $a_i \rightarrow a_k$, or $a_k \sim a_i$.

Causal+ Consistency (CC+) [67] is an extension to CC that—in addition to guaranteeing causality—ensures that replicas of the same key eventually converge to the same value. To ensure converence, we register a conflict resolution policy in Anna by implementing the following definitions:

Definition 1 (Concurrent Version Merge)**.** *Given two concurrent versions a_i and a_j, let a_k be a merge of a_i and a_j (denoted as $a_k = a_i \cup a_j$). Then $VC_{a_k} = VC_{a_i} \cup VC_{a_j} = \langle\langle e, c\rangle | \langle e, c_i\rangle \in VC_{a_i} \wedge \langle e, c_j\rangle \in VC_{a_j} \wedge c = \max(c_i, c_j)\rangle$.*

The merged VC is the key-wise maximum of the input VCs. Note that the above definition has a slight abuse of notation: e might not exist in one of the two VCs. If $\langle e, c_i\rangle \notin VC_{a_i}$, we simply set $\langle e, c\rangle = \langle e, c_j\rangle \in VC_{a_j}$ and vice versa.

In addition to merging the VCs, we also merge the dependency sets using the same mechanism above. Finally, we merge the payloads by taking a set union ($a_k.payload = \langle a_i.payload, a_j.payload\rangle$). When a function requests a key that has a set of payloads, applica-

tions can specify which payload to return; by default we return the first element in the set to avoid type error. If a_i and a_j are not concurrent (say $a_i \rightarrow a_j$), then a_j overwrites a_i during the merge.

Transactional Causal Consistency (TCC) [4, 74] is a further extension of CC+ that guarantees the consistency of reads and writes for a *set* of keys. Specifically, given a read set R, TCC requires that R forms a *causal snapshot*.

Definition 2. *R is a causal snapshot* $\iff \forall (a_i, b_j) \in R, \nexists a_k \mid a_i \rightarrow a_k \wedge a_k \rightarrow b_j$.

That is, for any pair of keys a_i, b_j in R, if a_k is a dependency of b_j, then a_i is not allowed to happen before a_k; it can be equal to a_k, happen after a_k, or be concurrent with a_k. Note that TCC is stronger than CC+ because issuing a sequence of reads to a data store that guarantees CC+ does not ensure that the keys read are from the same causal snapshot.

In the social network example, the application explicitly specifies that Alice's photo update depends on her permission update. TCC then ensures that the system only reveals the *new* permission and the funny picture to the application, which rejects Bob's request to view the picture.

TCC also ensures *atomic visibility* of written keys; either all writes from a transaction are seen or none are. In our context, if a DAG writes a_i and b_j and another DAG reads a_i and b_k, TCC requires $b_k == b_j \vee b_j \rightarrow b_k$.

4.2 HydroCache

In this section, we introduce the design of HydroCache and discuss how it achieves TCC for individual functions executed at a single node. To achieve both causal snapshot reads and atomic visibility, each cache maintains a single *strict causal cut* C (abbreviated as *cut*) which we define below.

Definition 3. *C is a cut* $\iff \forall k_i \in C$,
$\forall d_j \in get\_tuple(k_i.deps), \exists d_k \in C \mid d_k == d_j \vee d_j \rightarrow d_k$.

A cut requires that for *any* dependency, d_j, of any key in C, there is a $d_k \in C$ such that either the two versions are equal or d_k happens after d_j. We formally define this notion:

Definition 4. *Given two versions of the same key k_i and k_j, we say that k_i supersedes k_j ($supersede(k_i, k_j)$) when $k_i == k_j \vee k_j \rightarrow k_i$. Similarly, given two sets of key versions T and S, let K be a set of keys that appear in both T and S. We say that T supersedes S if $\forall k \in K$, let $k_i \in T$ and $k_j \in S$, we have $supersede(k_i, k_j)$.*

Note that a cut differs from a causal snapshot in two ways. First, cuts are closed under dependency: If a key is in the cut, so are its dependencies. Second, the *happens-before* constraint in a cut is more stringent than in a causal snapshot: The key d_k must be equal-to or happen after every dependency d_j associated with other keys in the cut—concurrency is disallowed. We will see why this is important in Section 4.2.2.

Updating the Local Cut. HydroCache initially contains no data, so it trivially forms a cut, C. When a function requests a key b that is missing from C, the cache fetches a version b_j from Anna. Before exposing it to the functions, the cache checks to see if all of b_j's dependencies are superseded by keys already in C. If a dependency a_i is not superseded, the cache fetches versions of a, a_m, from Anna until a_i is superseded by a_m. HydroCache then recursively ensures that all dependencies of a_m are superseded. This process repeats until the dependencies of all new keys are superseded. At this point, the cache updates C by merging the new keys with keys in C and exposes them to the functions. If a cache runs out of memory during the merge of requested keys, the requesting function is rescheduled on another node.

The cache "subscribes" to its cached keys with Anna, and Anna periodically pushes new versions to "refresh" the cache. New data is merged into C following the same process as above. When evicting key k, all keys depending on k are also evicted to preserve the cut invariant.

In the rest of this section, we first discuss how HydroCache guarantees TCC at a single node—providing a causal snapshot for the read set (Section 4.2.1) and atomic visibility for the write set (Section 4.2.2). We then discuss garbage collection and fault tolerance in Section 4.2.3.

4.2.1 Causal Snapshot Reads

From Definition 3, we know that for any pair of keys a_i, b_j in a cut C, if $a_k \rightarrow b_j$, then a_i supersedes a_k—either $a_k == a_i \vee a_k \rightarrow a_i$. This is stronger than the definition of a causal snapshot (Definition 2), as that definition permits a_k to be concurrent with a_i. Since each function reads from the cut in its local cache, the read set trivially forms a causal snapshot.

As we show below, this stricter form of causal snapshot (disallowing a_k and a_i to be concurrent) also ensures atomic visibility. Therefore, from now on, we consider this type of causal snapshot and abbreviate it as *snapshot*.

4.2.2 Atomic Visibility

Say a function writes two keys, a_i and b_j; in order to make them atomically visible, HydroCache makes them mutually dependent—$a_i \rightarrow b_j$ and $b_j \rightarrow a_i$. If another function reads a snapshot that contains a_i and b_k, since $b_j \rightarrow a_i$, the snapshot ensures that $b_k == b_j \vee b_j \rightarrow b_k$, satisfying atomic visibility. When executing a DAG, in order to ensure that writes across functions are mutually dependent, all writes are performed at the end of the DAG at the sink function.

There is, however, a subtle issue. Recall that VCs consist of the IDs of threads that modify a key along with those threads' logical clocks. All functions performing writes through an executor thread *share* the same ID. This introduces a new challenge: Consider two functions F and G both using executor e to write key versions a_i and a_j. When e writes these two versions—say a_i first, then a_j—a_i may be overwritten since e attaches a larger logical clock to $VC(e)$ of a_j. However, a_i and a_j may in fact be logically concurrent ($a_i \sim a_j$) since F may not have observed a_j before writing a_i and vice versa. This violates atomic visibility: If a function writes a_i and b_k, a_i can be overwritten by a concurrent version from the same thread. For a later read of a and b, b_k is visible but a_i is lost.

To prevent this from happening, each executor thread keeps the latest version of keys it has written. When a function writes a_i at executor e, e inspects its dependency set to see whether this write depends on the most recent write of the same key, a_{latest}, performed by e. If so, e advances its logical clock, updates $VC_{a_i}(e)$, writes to Anna, and updates a_{latest} to a_i. If not, then $a_i \sim a_{latest}$. This is because since a_{latest} is not in a_i's dependency set, $a_{latest} \nrightarrow a_i$, and since e wrote a_{latest} before a_i, $a_i \nrightarrow a_{latest}$. Since the versions are not equal, we have $a_i \sim a_{latest}$. In this case, e first merges a_i and a_{latest} following Definition 1 to produce a_k, advances its logical clock and updates $VC_{a_k}(e)$, writes to Anna, and sets a_{latest} to a_k. Doing so prevents each executor from overwriting keys with a potentially concurrent version.

4.2.3 Discussion

Dependency Metadata Garbage Collection. Causal dependencies accumulate over time. For a key $b_j \mid a_i \rightarrow b_j$, we can safely garbage collect $\langle a_i, VC_{a_i} \rangle \in b_j.deps$ if all replicas of a supersede a_i. We run a background consensus protocol to periodically clear this metadata.

Fault Tolerance. When writes to Anna fail due to storage node failures or network delay, they are retried with the same key version, guaranteeing idempotence. Function and DAG executions are *at least once*. Heartbeats are used to detect node failures in the compute layer, and unfinished functions and DAGs at a failed node are re-scheduled at other nodes.

4.3 MTCC Protocols

Although the design introduced in Section 4.2 guarantees TCC for individual functions executed at a single node, this is insufficient for serverless applications. Recall that a DAG in Cloudburst consists of multiple functions, each of which can be executed at a different node. To achieve TCC for the DAG, we must ensure that a read set spanning *multiple* physical sites forms a distributed snapshot. A naïve approach is to have all caches coordinate and maintain a large distributed cut across all cached keys at all times. This is infeasible in a serverless environment due to the enormous traffic that protocol would generate amongst thousands of nodes.

In this section, we discuss a set of MTCC protocols we developed to address this challenge while minimizing coordination and data shipping overheads across caches. The key insight is that rather than eagerly constructing a distributed cut, caches collaborate to create a snapshot of each DAG's read set during execution. This leads to significant savings for two reasons. First, snapshots are constructed per-DAG; the communication to form these snapshots is combined with that of regular DAG execution, without any global coordination. Second, snapshots are restricted to holding the keys read by the DAG, whereas a cut must include all keys' transitive dependencies.

We start with the *centralized* (CT) protocol in which all functions in a DAG are executed at a single node. We then introduce three protocols—*optimistic* (OPT), *conservative* (CON), and *hybrid* (HB)—that allow functions to be executed across different nodes. Throughout the rest of this section, we assume the read set of each function is known, but we return to cases in which the read set is unknown in Section 4.3.6.

4.3.1 Centralized (CT)

Under CT, all functions in a DAG are executed at a single node, accessing a single cache. This significantly simplifies the challenge of providing TCC, as we do not need to worry about whether reads from cuts on different nodes form a snapshot. Before execution, the cache creates a snapshot from the local cut that contains the read set of all functions in the DAG. All reads are serviced from the created snapshot to ensure that they observe the same cut, regardless of whether the cut is updated while the DAG is executing.

The main advantage of CT is its simplicity: Since all functions are executed on the same node, there is no network cost for passing results across nodes. However, CT suffers from some key limitations. First, it constrains scheduling: Scheduling happens at the DAG level instead of at the level of individual functions. The distributed nature of our scheduler sometimes leads to load imbalances, as schedulers do not have a global view of resource availability. When the scheduling granularity becomes coarse (from function to DAG), the performance penalty due to load imbalance will be amplified. Second, CT requires the data requested by all functions to be co-located at a single cache. The overheads of fetching data from remote storage and constructing the cut can be significant if there are many cache misses or if the read set is large. Finally, CT limits the amount of parallelism in a DAG to the number of executor threads on a single node. In Section 4.4.2, we evaluate these limitations and quantify the trade-off between CT and our distributed protocols.

4.3.2 Towards Distributed Snapshots

The goal of our distributed protocols below is to ensure that each DAG observes a snapshot as computation moves across nodes. This requires care, as reading arbitrary versions from the various local cuts may not correctly create a snapshot.

Theorems

Before describing our protocols, we present simple theorems and proofs that allow us to combine data from each node to ensure the distributed snapshot property.

Definition 5 (Keysets and Versionsets)**.** *A keyset $\widetilde{R}$ is a set of keys without specified versions. A versionset R is a* binding *that maps each $k \in \widetilde{R}$ to a specific version k_i.*

In subsequent discussion, we notate keysets with a tilde above. As a mild abuse of notation, we will refer to the intersection of a keyset $\widetilde{K}$ with a versionset V; this is the maximal subset of V whose keys are found in the keyset $\widetilde{K}$. Each element in the versionset only contains the key identifier and its VC; dependency and payload information are not included.

Definition 6 (Keyset-Overlapping Cut)**.** *Given a versionset V and a keyset $\widetilde{K}$, we say that V is a keyset-overlapping cut for $\widetilde{K}$ when $\forall k_i \in V$, we have $k \in \widetilde{K} \wedge \forall d_j \rightarrow k_i$, if $d \in \widetilde{K}$, then $\exists d_{js} \in V \mid supersede(d_{js}, d_j)$.*

In essence, a keyset-overlapping cut for $\widetilde{K}$ is similar to a cut with the relaxation that it only consists of keys and dependencies that overlap with $\widetilde{K}$.

Lemma 1. *Given a keyset $\widetilde{K}$ and a cut C, $S = C \cap \widetilde{K}$ is a keyset-overlapping cut for $\widetilde{K}$.*

Proof. This follows directly from Definition 6. S is the intersection of the versionset C and $\widetilde{K}$; the fact that C is a cut ensures that the conditions of Definition 6 hold. □

Definition 7 (Versionset Union)**.** *Given two versionsets V_1 and V_2, their union $V_3 = V_1 \cup V_2$ includes all keys k_m such that:*

$$k_m = \begin{cases} k_i \in V_1 & \nexists k_j \in V_2 \\ k_i \in V_2 & \nexists k_j \in V_1 \\ k_i \cup k_j & \exists k_i \in V_1 \wedge \exists k_j \in V_2 \end{cases}$$

We show keyset-overlapping cuts are closed under union:

Theorem 1 (Closure Under Union)**.** *Given keyset $\widetilde{K}$, let $S_1 = C_1 \cap \widetilde{K}, S_2 = C_2 \cap \widetilde{K}$ be keyset-overlapping cuts for $\widetilde{K}$. Then $S_3 = S_1 \cup S_2$ is a keyset-overlapping cut for $\widetilde{K}$.*

Proof. Let $k_3 \in S_3$ and $d_3 \rightarrow k_3$. Since $S_3 = S_1 \cup S_2$, we know $k_3 = k_1 \cup k_2$ where $k_1 \in S_1$, $k_2 \in S_2$, and $d_3 = d_1 \cup d_2$ where $d_1 \rightarrow k_1$, $d_2 \rightarrow k_2$.

If $d \in \widetilde{K}$, according to Definition 6, $\exists d_{1s} \in S_1 \mid supersede\ (d_{1s}, d_1)$ and $\exists d_{2s} \in S_2 \mid supersede(d_{2s}, d_2)$. It follows that $supersede(d_{1s} \cup d_{2s}, d_1 \cup d_2)$, where $d_1 \cup d_2 = d_3$ and $d_{1s} \cup d_{2s} = d_{3s} \in S_3$; we have $supersede(d_{3s}, d_3)$. This holds for all dependencies in $\widetilde{K}$. We omit cases where $\nexists k_1 \in S_1$ or $\nexists k_2 \in S_2$ as they follow trivially from set union and Definition 6. Therefore, S_3 is a keyset-overlapping cut for $\widetilde{K}$. □

We conclude with a simple lemma that ensures the snapshot property that is the goal of our protocols in this section.

Lemma 2. *Every keyset-overlapping cut is a snapshot.*

Proof. Let S be a keyset-overlapping cut for keyset $\widetilde{K}$. For $(a_i, b_j) \in S$, if $a_k \rightarrow b_j$, since $a \in \widetilde{K}$, from Definition 6 we know $supersede(a_i, a_k)$. The same holds for other pairs of keys in S. Therefore, S is a snapshot. □

4.3.3 Optimistic (OPT)

OPT is our first MTCC protocol. The idea is to eagerly start running the functions in a DAG and check for violations of the snapshot property at the time of each function execution. If no violations are found—e.g. when updates are infrequent so the cuts at different nodes are roughly in sync—then no communication costs need be incurred by constructing a DAG-specific snapshot in advance. Even when violations are found at some node, we can potentially adjust the versionset

being read at that node to re-establish the snapshot property for the DAG. However, we will see that in some cases the violation cannot be fixed, and we must restart the DAG.

OPT validates the snapshot property in two cases: when an upstream function triggers a downstream function (linear flow), and when multiple parallel functions accumulate their results (parallel flow). We present an algorithm for each case.

Linear Flow Validation

In the linear flow case, we have an "upstream" function in the DAG that has completed with its readset bound to specific versions, and an about-to-be-executed "downstream" function whose readset is still unbound. If we are lucky, the current cut at the downstream node forms a snapshot with the upstream function's readset; if not, we will try to modify the downstream readset to suit.

Specifically, given a versionset R_u read until now in the DAG, a downstream function F_d must bind a keyset $\widetilde{R}_d$ to a versionset R_d, where $R_d \cup R_u$ is a snapshot. This require two properties: (Case I) R_d supersedes R_u's dependencies, $R_u.deps$ (see Definition 4), and (Case II) R_u supersedes $R_d.deps$.

Algorithm 6 shows the validation process. When an upstream function F_u triggers F_d, it sends the results (R_u, S_u) from running Algorithm 6 on F_u. R_u is the versionset read by F_u and any of its upstream functions. S_u is a versionset with two properties: It is a keyset-overlapping cut for the DAG's read set ($\widetilde{R}_{DAG}$), and $R_u \subseteq S_u$—all keys in R_u are present in S_u. Since S_u is a snapshot (Lemma 2), we know S_u supersedes the dependencies of R_u. We show later how S_u is constructed and prove its properties in Theorem 2. Recall that S_u and R_u only contain the id and vector clock for each key. Dependency metadata and payloads are *not* shipped across functions.

In lines 1-2 of Algorithm 6, we check if the upstream validation process decided to abort (a is an abort flag from the upstream). If so, we also abort. Otherwise, beginning on line 3, we ensure that the keys $\widetilde{R}_d$ to be read in F_d are available. For each $k \in \widetilde{R}_d$ that is not present in the local cut C, if k exists in S_u as k_i, we add it to the versionset R_{remote} (line 6); it will be fetched from upstream at the end of the algorithm. Otherwise, we update C to include k (line 8) following the Local Cut Update process described in Section 4.2.

Next, we begin handling the two cases mentioned above. In Case I (lines 9-12), we start by forming a candidate mapping for $\widetilde{R}_d$, R_{local}, by simply binding $\widetilde{R}_d$ to the overlap of the local cut C (line 9). We then check each element of R_{local} to see if it supersedes the corresponding element of $R_u.deps$. It is sufficient to check if each element of R_{local} supersedes the corresponding element of S_u, which in turn supersedes the element of $R_u.deps$. When we discover a violation, we add the corresponding key from S_u to R_{remote}.

Algorithm 6 Linear Flow Validation

Input: $a, S_u, R_u, \widetilde{R}_d, \widetilde{R}_{DAG}, C$

1: **if** $a == True$ **then**
2: return "Abort"
// Ensure all keys in $\widetilde{R}_d$ are available for execution
3: $R_{remote} \coloneqq \varnothing$
4: **for** $k \in \widetilde{R}_d$ **do**
5: **if** $k_i \notin C \wedge k_j \in S_u$ **then**
6: R_{remote}.add(k_j)
7: **else if** $k_i \notin C \wedge k_j \notin S_u$ **then**
8: C.update(k)
// Case I
9: $R_{local} \coloneqq C \cap \widetilde{R}_d$
10: **for** $k_i \in S_u$ **do**
11: **if** $k_j \in R_{local} \wedge !supersede(k_j, k_i)$ **then**
12: R_{remote}.add(k_i)
// Case II
13: $abort \coloneqq False$
14: $S\_map \coloneqq \{\}$ // an empty map
15: **for** $k_i \in R_{local}$ **do**
16: $S_i =$ RetrieveCut($k, \widetilde{R}_{DAG}, C$) // Algorithm 7
17: $S\_map[k_i] = S_i$
18: **for** $m_i \in S_i$ **do**
19: **if** $m_j \in R_u \wedge !supersede(m_j, m_i)$ **then**
20: // key k violates Case II. Try to move it to R_{remote}
21: **if** $k_j \notin S_u$ **then** // cannot read k from upstream to fix
22: $abort = True$
23: break
24: **else** // can read k from upstream to fix
25: R_{remote}.add(k_j)
26: R_{local}.remove(k_i)
27: **if** $abort$ **then**
28: return "Abort"
29: **else**
30: $S_d \coloneqq \varnothing$
31: **for** $k_i \in R_{local}$ **do**
32: $S_d = S_d \cup S\_map[k_i]$
33: S_d.version() // create temporary versions for keys in S_d
34: $R_d \coloneqq R_{local}$
35: **for** $k_i \in R_{remote}$ **do**
36: R_d.merge(fetch(k_i))
37: return $S_{new\_u} = S_u \cup S_d$, $R_{new\_u} = R_u \cup R_d$

In Case II (line 13), we ensure that R_u supersedes $R_d.deps$. To do so, we identify elements in

Algorithm 7 RetrieveCut

Input: $k, \widetilde{R}_{DAG}, C$
1: $C' := \varnothing$
2: $to\_check := \{k_i \in C\}$ // k_i is the version of k in C
3: **while** $to\_check \neq \varnothing$ **do** // transitively add dependencies
4: **for** $v_j \in to\_check$ **do** . // of k to construct the cut C'
5: **for** $d_k \in v_j.deps$ **do**
6: **if** $d_l \notin C'$ **then** // C' does not contain d
7: C'.add($d_m \in C$)
8: to_check.add($d_m \in C$)
9: to_check.remove(v_j)
10: return $C' \cap \widetilde{R}_{DAG}$ // a keyset-overlapping cut for $\widetilde{R}_{DAG}$

R_{local} whose dependencies are not superseded by the corresponding keys in R_u (line 19).

For $k_i \in R_{local}$, it is sufficient to construct S_i, a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that contains k_i and check if R_u supersedes S_i. S_i is created as follows: we first construct a cut C' from C that includes k_i and intersect C' with $\widetilde{R}_{DAG}$ to get S_i (Algorithm 7). According to Lemma 1, S_i is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. We remove elements not in $\widetilde{R}_{DAG}$ as they need not be checked for supersession. This optimization significantly reduces the amount of causal metadata we ship across nodes (line 10 of Algorithm 7).

If R_u does not supersede S_i, then k_i cannot be included in R_d, and we try to use the upstream versions instead. If k does not exist upstream, we fail to form a snapshot and abort (lines 21-22). Otherwise, we add an older version $k_j \in S_u$ to R_{remote} (line 25) and remove k_i from R_{local} (line 26).

At this point we can construct S_d, a union of all keyset-overlapping cuts for $\widetilde{R}_{DAG}$ that supersedes dependencies of R_{local} (lines 31-32). By Theorem 1, S_d is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. The cache creates temporary versions that are stored locally for keys in S_d in case they need to be fetched by other caches during the distributed snapshot construction (line 33). Finally, we initialize R_d to R_{local} and fetch the keys in R_{remote} to merge into R_d. R_d is now provided to F_d for execution, and $S_{new\_u} = S_u \cup S_d$, $R_{new\_u} = R_u \cup R_d$ are used for validation in subsequent functions.

Correctness. S_{new_u} passed to subsequent functions must have the same properties as S_u. Recall that S_u has two properties. First, it is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. Second, $R_u \subseteq S_u$. For the first property, recall that S_d is also a keyset-overlapping cut for $\widetilde{R}_{DAG}$. Hence by Theorem 1, we know $S_{new\_u} = S_u \cup S_d$ is also a keyset-overlapping cut for $\widetilde{R}_{DAG}$. We now show $R_{new\_u} \subseteq S_{new\_u}$.

Theorem 2. *If the validation process in Algorithm 6 succeeds, let $R_{new\_u} = R_u \cup R_d$ and $S_{new\_u} = S_u \cup S_d$. Then $R_{new\_u} \subseteq S_{new\_u}$.*

Proof. We prove by induction. We first prove the base case where there is no upstream. In this case, $R_u = \varnothing, S_u = \varnothing$, so it is sufficient to show $R_d \subseteq S_d$. Recall by construction S_d contains all

Algorithm 8 Parallel Flow Validation

Input: $(S_1, R_1, a_1), (S_2, R_2, a_2), ..., (S_n, R_n, a_n)$

1: $a \coloneqq \bigvee_{i=1}^{n} a_i$
2: **if** $a == True$ **then**
3: return "Abort"
4: $S \coloneqq \bigcup_{i=1}^{n} S_i$
5: **for** $i \in [1, n]$ **do**
6: **for** $k_m \in R_i$ **do**
7: **if** $!supersede(k_m, k_n \in S)$ **then**
8: return "Abort"
9: $R \coloneqq \bigcup_{i=1}^{n} R_i$
10: return S, R

keys in R_{local} (line 34-35). Since there is no upstream function, we do not fetch any data to update R_d (line 39), so we have $R_d = R_{local}$. Hence, $R_{new\_u} \subseteq S_{new\_u}$.

Inductive hypothesis: Given R_u, S_u such that $R_u \subseteq S_u$, we want to show $R_{new\_u} \subseteq S_{new\_u}$. It is sufficient to show that $R_u \subseteq S_{new\_u}$ and $R_d \subseteq S_{new\_u}$. For the first part, let $k_i \in R_u$, $k_m \in S_d$ and $k_j \in (S_u \cup S_d)$. Since $R_u \subseteq S_u$, we know $k_i \in S_u$, and therefore $k_j = k_i \cup k_m$. Since Case II of the validation process ensures that $supersede(k_i, k_m)$, we know $k_i \cup k_m = k_i$, and therefore $k_i == k_j$. If $k_m \notin S_d$, it is trivially true that $k_i == k_j$. Hence, $R_u \subseteq S_{new\_u}$.

We now prove the second half of the inductive hypothesis. If $k \in \widetilde{R}_d$, there are two cases: either k is read from C (and potentially from upstream) or k is not read from C. In the first case, let $k_m \in C$. By construction, we know $k_m \in S_d$. Suppose $k_j \in S_u$ and $k_i \in R_d$. If k is also read from the upstream, then $k_i = k_m \cup k_j$. Otherwise, $k_i = k_m$. Note that since Case I ensures that $supersede(k_i, k_j)$, we can still express k_i as $k_m \cup k_j$. Therefore, regardless of whether k is read from the upstream, we always have $k_i = k_m \cup k_j$. Also, since $k_m \in S_d$ and $k_j \in S_u$, we know $k_i = (k_m \cup k_j) \in (S_u \cup S_d)$. We now have $R_d \subseteq S_{new\_u}$. If k is not read from C, $k_i \in R_d$ is read from S_u, so $k_i \in S_u$, and Case II ensures that $k_j \notin S_d$. Hence $R_d \subseteq S_{new\_u}$, and S_{new_u} has the same properties as S_u. □

Parallel Flow Validation

When multiple parallel upstream functions $(U_1, U_2, ..., U_n)$ accumulate their results to trigger a downstream function, we need to validate if the versionsets read across these parallel upstreams form a snapshot. To this end, we check if their read sets $(R_1, R_2, ..., R_n)$ supersede $(S_1, S_2, ..., S_n)$, each S_i being a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that contains each upstream U_i's read set R_i.

In Algorithm 8, we first check if any upstream function aborts due to linear flow validation. If so, we also abort. Otherwise, we create S, a union of all upstream S_is that contains the read sets of all parallel upstreams; it follows that S supersedes the dependencies of all parallel upstream read sets. For each k_m in each read set R_i, we check if k_m supersedes the corresponding $k_n \in S$. Since

Algorithm 9 CreateSnapshot

Input: $\widetilde{R}_i, \widetilde{R}_{DAG}, C$
1: **for** $k \in \widetilde{R}_i$ — $k \notin C$ **do**
2: C.update(k)
3: $S_i \coloneqq \varnothing$
4: **for** $k \in \widetilde{R}_i$ **do**
5: $S_i = S_i \cup$ RetrieveCut($k, \widetilde{R}_{DAG}, C$) // Algorithm 7
6: S_i.version() // create temporary versions for keys in S_i
7: return S_i

Algorithm 10 Distributed Snapshot Construction

Input: $(F_1, \widetilde{R}_1), (F_2, \widetilde{R}_2), ..., (F_n, \widetilde{R}_n), \widetilde{R}_{DAG}$
1: $S \coloneqq \varnothing$
2: $R \coloneqq []$ // empty list
3: **for** $i \in [1, n]$ **do**
4: $S_i = F_i$.GetCache.CreateSnapshot($\widetilde{R}_i, \widetilde{R}_{DAG}$) // Algorithm 9
5: $R_i = S_i \cap \widetilde{R}_i$
6: R.append(R_i)
7: $S = S \cup S_i$
8: $R_{remote} \coloneqq \{\}$ // an empty map
9: **for** $R_i \in R$ **do**
10: **for** $k_m \in R_i$ **do**
11: **if** !$supersede(k_m, k_n \in S)$ **then**
12: R_{remote}[i].add(k_n) // cache i needs to fetch k_n
13: **for** $i \in R_{remote}$ **do**
14: F_i.GetCache.Fetch(R_{remote}[i])

we are validating between parallel upstreams whose functions have already been executed, OPT cannot perform any "repair" as in Algorithm 6. Therefore, if validation fails, we abort. Note that S has exactly the same properties as S_i if validation succeeds. We omit the proof as it is almost the same as Theorem 2.

4.3.4 Conservative (CON)

CON is the opposite of OPT: Instead of lazily validating read sets as the DAG progresses, the scheduler coordinates with all caches involved in the DAG request to construct a distributed snapshot of $\widetilde{R}_{DAG}$ before execution. Each function's corresponding cache first creates S_i, a keyset-overlapping cut for $\widetilde{R}_{DAG}$, such that S_i contains the function's read set. According to Theorem 1 and Lemma 2, the distributed snapshot S can then be formed by taking the union of all S_is.

In Algorithm 10, the scheduler instructs each function F_i's cache to create S_i (line 4) via Algorithm 9. If a key in $\widetilde{R}_i$ is missing from C, the cache updates C to include the key (lines

1-2 of Algorithm 9). Then, for each key $k \in \widetilde{R}_i$, the cache creates a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that includes k and unions all the keyset-overlapping cuts to create S_i (line 5). It then creates temporary versions for keys in S_i in case they need to be fetched by other caches during the distributed snapshot construction (line 6). After that, the scheduler forms R_i, the versionset that F_i reads from the local cut, by binding $\widetilde{R}_i$ to the overlap of S_i (line 5 of Algorithm 10). After all S_is are created, the scheduler unions them to create S (line 7). It then inspects the R_i of each function. If a key $k_m \in R_i$ cannot supersede $k_n \in S$ (line 11), then the corresponding cache fetches from remote caches to match k_n (line 14). The scheduler only begins execution after all remote reads finish; each function reads from a partition of S.

4.3.5 Hybrid (HB)

The OPT protocol starts a DAG immediately without coordination but is susceptible to aborts, and there is no guarantee as to how many times it retries before succeeding. On the other hand, the CON protocol never aborts but has to pay the cost of coordinating with all caches involved in a request to construct a distributed snapshot.

The hybrid protocol (HB) combines the benefits of the OPT and CON protocols. HB (run by the scheduler) starts the OPT subroutine and simultaneously performs a simulation of OPT. This simulation is possible because OPT only needs the read set of each function and the local cut at each cache to perform validation, and the simulation process can get the same information by querying the caches. The simulation is much faster than executing the request because no functions are executed and no causal metadata is passed across nodes. The CON subroutine is activated only when the simulation aborts. HB includes some key optimizations that enable the two subroutines to cooperate to improve performance.

Pre-fetching. After the simulation, we know what data must be fetched from remote storage during each function's validation process (Algorithm 6). In this case, the HB protocol notifies caches involved in the request to pre-fetch relevant data before the OPT subroutine reaches these caches.

Early Abort. When our simulation aborts, HB notifies all caches to stop the OPT process to save unnecessary computation. This is especially useful for DAGs with parallel functions: A function is not aware that a sibling has aborted in Algorithm 6 until they "meet" and abort in Algorithm 8.

Function Result Caching. After each function is executed under OPT, its result and key versions read are stored in the cache. If OPT aborts, the function is re-executed under CON, and if CON's key versions match the original execution's, we skip execution and retrieves the result from the cache. This data is cleared immediately after the DAG finishes.

4.3.6 Discussion

We now discuss a few important properties of our MTCC protocols and how to handle cases when $\widetilde{R}_{DAG}$ is unknown.

Interaction with Autoscaling. Recall the naïve approach at the beginning of Section 4.3 in which all caches coordinate to maintain a large distributed cut at all times. Such an approach requires knowing the membership of the system, which dynamically changes in a serverless setting. This protocol will thus require expensive coordination mechanisms to establish cluster memberhsip before each cut update; the autoscaling policy cannot act while the cut is being updated.

On the other hand, none of protocols described in this section rely on node membership to achieve TCC. Each cache independently maintains its own cut, and only a small number of caches involved in a DAG request needs to communicate to ensure the snapshot property. Therefore, HydroCache guarantees TCC in a way that is orthogonal to autoscaling.

Repeatable Read. If two functions in a single request both read key k, it is natural to expect that they will read the same version of k [19]. The CT and CON protocols trivially achieve repeatable read because their snapshots are constructed prior to DAG execution. For OPT, repeatable read is a simple corollary of Theorem 2. Given R_u, R_d, S_{new_u}, $k_i \in R_u \Rightarrow k_i \in S_{new\_u}$, and $k_j \in R_d \Rightarrow k_j \in S_{new\_u}$. Hence $k_i == k_j$, and OPT and HB achieve repeatable read.

Versioning. Our protocols do not rely on aggressive multi-versioning; they create *temporary* versions (line 33 of Algorithm 6 and line 6 of Algorithm 9) so that remote caches can retrieve the correct versions of keys to construct snapshots. These versions are garbage collected after each request finishes, significantly reducing storage overhead.

Unknown Read Set. When the readset is unknown, CON can no longer pre-construct the distributed snapshot. Instead, we rely on OPT to "explore" the DAG's read set as the request progresses and validate if the keys read so far form a snapshot. When the validation fails, we invoke CON to construct a distributed snapshot for all keys read thus far *before* retrying the request. This way, the next trial of OPT will not abort due to causal inconsistencies between keys that we have already explored. We switch between OPT and CON until the read sets of all functions in the DAG are fully explored.

With an unknown $\widetilde{R}_{DAG}$, we can no longer perform the optimization in line 10 of Algorithm 7 to reduce causal metadata shipped across nodes. Finally, as OPT explores new keys, the protocol may abort multiple times. However, in practice, a DAG will likely only read a small number of keys that are updated very frequently. These keys are the primary culprits for aborts, and the number of aborts will roughly be bounded by the number of such write-heavy keys.

4.4 Evaluation

This section presents a detailed evaluation of HydroCache. We first study aspects of HydroCache in isolation: MTCC's performance (§4.4.2), a comparison to other consistency models (§4.4.3), and scalability (§4.4.4). We then evaluate HydroCache's broader benefits by comparing its performance and consistency against cache-less architectures (§4.4.5).

4.4.1 Experiment Setup and Workload

Our experiments were run in the `us-east-1a` AWS availability zone (AZ). Function schedulers were run on AWS `c5.large` EC2 VMs (2 vCPUs and 4GB RAM), and compute nodes used `c5.4xlarge` EC2 VMs (16 vCPUs and 32GB RAM); hyperthreading was enabled. Each compute node had 3 function executors that shared a HydroCache. Cloudburst was deployed with 3 scheduler nodes; the system's auto-scaling policy enforced a minimum of 3 compute nodes (9 Python execution threads and 3 caches total). Clients were run on separate machines in the same AZ.

Unless otherwise specified, for each experiment, we used 6 concurrent benchmark threads, each sequentially issuing 500 DAG execution requests. The cache "refresh" period for cut updates was set to 100ms. Our dataset was 1 million keys, each with an 8-byte payload. Caches were pre-warmed to remove the data retrieval overheads from the KVS.

Our benchmarks evaluate two DAG topologies: a linear DAG and a V-shaped DAG. Linear DAGs are chains of three sequentially executed functions, where the result of each upstream function is passed in as an argument to the downstream function. V-shaped DAGs also contain three functions, but the first two functions are executed in parallel, and their results are passed as arguments to the third function.

Every function takes two arguments, except for the sink function of the V-shaped DAG, which takes three arguments. The arguments are either a reference to key in Anna (drawn from a Zipfian distribution) or the result of an upstream function. At the end of a DAG, the sink function writes its result into a key randomly chosen from the DAG's read set; the write is causally dependent on the keys in the read set. Each function returns immediately to eliminate function execution overheads, and we assume the read set is known.

4.4.2 Comparison Across Protocols

In this section, we evaluate the protocols proposed in Section 4.3. Figure 4.2 shows the end-to-end DAG execution latency for our protocols, with varying topologies and Zipfian distributions (1.0, 1.25, and 1.5). Our experiments show that HB has the best performance of the three distributed protocols and highlight a trade-off between CT and HB, which we discuss in more detail. We omit discussion of the experiments with a Zipfian coefficient of 1.25, as the performance is in between that of the other two contention levels.

Centralized (CT) achieves the best median latency in all settings because each request has only one round of communication with a single cache to create its snapshot before execution. This avoids the additional overhead of passing causal metadata across caches; neither DAG topology nor workload contention affect performance. Furthermore, function results within a DAG are passed between threads rather than between nodes, avoiding expensive network latencies. Nonetheless, its 99th percentile latency is consistently worse than the conservative protocol's (CON) and the hybrid protocol's (HB), because requiring all functions to execute on a single node leads to more load imbalance (Section 4.3.1).

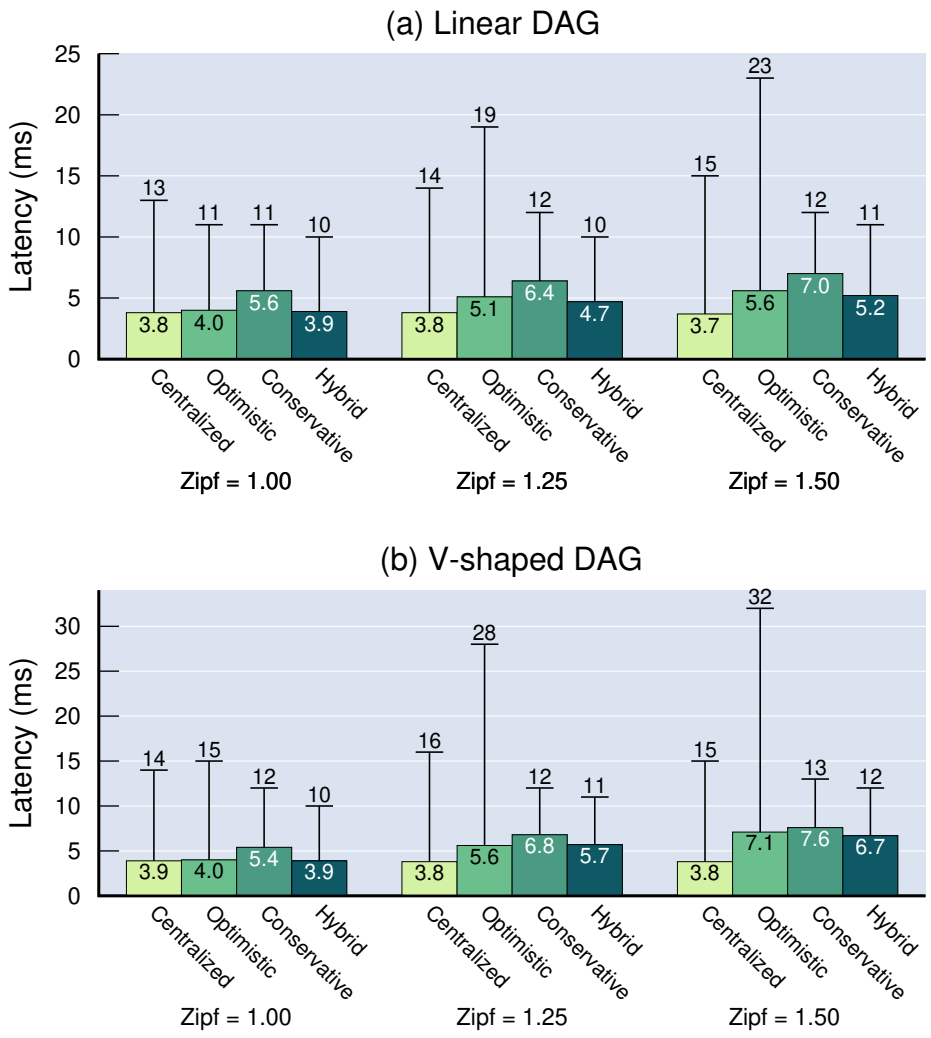

Figure 4.2: Median (bar) and P99 (whisker) latencies across different protocols for executing linear and V-shaped DAGs.

Optimistic (OPT) achieves excellent performance for linear DAGs (Figure 4.2 (a)) when the Zipfian coefficient is set to 1.0, a moderately contended distribution. Key accesses are spread across the entire key space, so each read likely accesses the most recent update, which has already been propagated and merged into the cut of all caches. As a result, 88% of DAG executions see local cuts that form a distributed snapshot without any intervention, significantly improving performance.

For the most contended workload (Zipf=1.5), the median latency increases by 40% due to increased data shipping costs (line 39 of Algorithm 6) to construct a snapshot; data shipping occurred in 82% of DAG executions. Correspondingly, 99th percentile latency was 2.1× worse. Under high contention, the probability of the OPT protocol's validation phase failing increases, which leads to

		HB Subroutines (Linear)		
		OPT Metadata Only	OPT Data Fetch	CON
Zipf	1.0	90%	7%	3%
	1.25	37%	55%	8%
	1.5	10%	81%	9%

Table 4.1: Percentage of different HB subroutines activated across contention levels for linear DAGs. The first column shows when OPT subroutine succeeds and only causal metadata is passed across nodes. The second column means the OPT subroutine succeeds but data is shipped across nodes to construct the snapshot. In the third column, OPT subroutine aborts and the DAG is finished by the CON subroutine with data shipping.

		HB Subroutines (V-shaped)		
		OPT Metadata Only	OPT Data Fetch	CON
Zipf	1.0	83%	5%	12%
	1.25	29%	17%	54%
	1.5	10%	12%	78%

Table 4.2: Percentage of different HB subroutines activated across different contention levels for V-shaped DAGs.

more aborts and retries. In this experiment, 8% of the DAGs aborted at least once, and in the worst case, a DAG was retried 7× before succeeding.

OPT performs similarly for V-shaped DAGs (Figure 4.2 (b)). The key difference is that increasing contention significantly degrades 99th percentile latencies. By design, OPT is unaware of the causal metadata required across the two parallel functions until parallel flow validation (Algorithm 8). The probability of validation failure is much higher since repair cannot be performed during Algorithm 8. For the most contended workload, 75% of DAGs were aborted at least once; in the worst case, a request required 14 retries.

Conservative (CON)'s median latency is 40% higher than OPT's due to the coordination prior to DAG execution. However, 99th percentile latency is more stable for high-contention workloads, with an increase of 1ms from the least to most contention. Each cache already has a snapshot for the DAG's read set before executing, so requests never abort.

Hybrid (HB) offers the best median and 99th percentile latency in all settings. To explain the performance improvements, Tables 4.1 and 4.2 show how often each protocol subroutine was activated for each topology and contention level.

Under moderate contention (Zipf=1.0), HB has OPT's advantages of immediately executing the DAG without coordination. We see that in a large majority of cases—90% for linear DAGs and 83% for V-shaped DAGs—no data fetching is required; the OPT subroutine of HB simply passes causal metadata along the DAG. Much of the DAG has already been executed under HB

by the time the CON protocol finishes constructing its snapshot. This explains the 44% and 38% improvements in median latency for linear DAGs and V-shaped DAGs, respectively.

Under high contention (Zipf=1.5), HB offers 35% better median latency than CON for linear DAGs. However, for V-shaped DAGs, the performance improvement is less than 15%. The reason, seen in Table 4.2, is that the OPT subroutine aborts frequently under high contention: In 78% of the cases, the CON subroutine is activated. Nevertheless, for the remaining 22% of requests, the OPT subroutine succeeds, leading to a moderate improvement in median latency.

Interestingly, the median latency of HB is even lower than OPT. There are two reasons for this: First, the CON subroutine prevents aborts to which OPT is susceptible. Second, while the OPT subroutine executes, the CON subroutine pre-fetches data to help OPT construct the snapshot (see Section 4.3.5). At the 99th percentile, HB matches the performance of CON and significantly outperforms OPT due to the avoided aborts.

***Takeaway**: HB consistently achieves the best performance among the distributed protocols, by taking advantage of optimistic execution while avoiding repeated aborts.*

Centralized vs. Hybrid

From the previous section, it is clear that HB is the best distributed protocol, but CT achieves better median latencies while compromising on 99th percentile latencies. We now turn to the question of whether our system should choose to use CT, as it is a simpler protocol that achieves reasonable performance.

As discussed in Section 4.3.1, CT has three key limitations: coarse-grained scheduling, forcing all data to a single node, and limited parallelism. In this section, we have not seen the data retrieval overhead as all caches were warmed up in advance. As we show in Section 4.4.5, the overhead of remote data fetches can be significant, especially for large data.

To better understand the limitation due to parallelism, we use a single benchmark thread to issue V-shaped DAGs with varying fanout (number of parallel functions), ranging from 1 to 9. To emphasize the performance gains from parallelism, each parallel function executes for 50ms, and the sink function returns immediately (for brevity, no figure is shown for this experiment). We observe that under a workload with moderate contention, HB's performance (median latency) is relatively stable, as it parallelizes sibling functions across nodes. However, CT executes all functions on the same node, so parallelism is limited to the three executors on that node. Therefore, we observe latency jumps as fanout grows from 3 to 4 and from 6 to 7. For DAGs with fanout greater than 7, HB outperforms CT by 3×.

***Takeaway**: Many factors affect the optimal protocol for a given workload, including load balancing, cache hits rates, and the degree of parallelism within a DAG. In general, the HB protocol offers the most flexibility.*

4.4.3 Consistency Overheads

In this section, we compare the performance of the HB protocol against two weaker consistency protocols (last-writer-wins (LWW) and Bolt-on Causal Consistency (BCC) [17]) and one strong

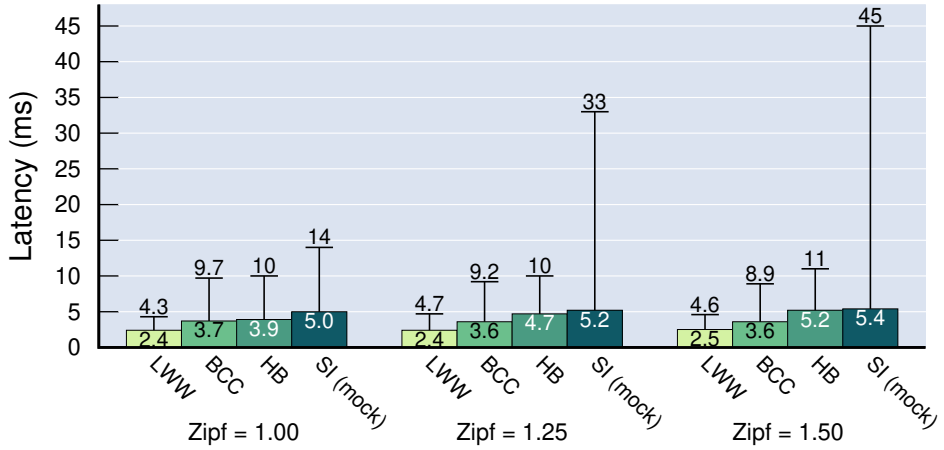

Figure 4.3: Median and P99 latencies between LWW, BCC, HB, and simulated SI protocols.

consistency protocol (Snapshot Isolation (SI)). We begin by discussing LWW and BCC and return to SI in Section 4.4.3. We evaluate linear DAGs in this experiment and vary the workload's contention level.

The LWW protocol attaches a timestamp to each write, and concurrent updates are resolved by picking the write with the largest timestamp. LWW only guarantees eventual replica convergence for individual keys but offers the best performance as there are no constraints on which key versions can be read. BCC only guarantees CC+ (see Section 4.1.2) for keys read within individual functions.

As shown in Figure 4.3 LWW and BCC are insensitive to workload skew. No causal metadata need be passed across nodes, and no data is shipped to construct a distributed snapshot. HB, as discussed previously (Section 4.4.2), incurs higher overheads under high contention due to the data fetching overhead incurred by the OPT subroutine and the coordination overhead incurred by the CON subroutine.

Under moderate contention (Zipf=1.0), HB matches the performance of BCC and is 62% slower than LWW. Under high contention, HB is 44% slower than BCC and 2× slower than LWW. However, Table 4.1 shows that 90% of DAGs require data shipping across caches under high contention. Since BCC does not account for multiple caches, over 90% of the BCC requests violated the TCC guarantee.

In addition to latency, we measure the maximum causal metadata storage overhead for each key in the working set for the HB protocol. Under moderate contention (Zipf=1.0), the median metadata overhead is 120 bytes and the 99th percentile overhead is 432 bytes. Under high contention (Zipf=1.5), the median and the 99th percentile overheads increase to 300 bytes and 852 bytes, respectively. Under high contention, both the keys' vector clock lengths and dependency counts increase: The 99th percentile vector clock length is 9 and the dependency count is 7.

Snapshot Isolation

HydroCache relies on Anna for its storage consistency; both components are coordination-free by design. However, there are databases that provide "strong" isolation with serverless scaling. Notably, AWS Serverless Aurora [91] provides Snapshot Isolation (SI) via its PostgreSQL configuration. SI is stronger than TCC in two ways: (1) it guarantees that reads observe all committed transactions, and (2) at commit time, SI validates writes and aborts transactions that produce write-write conflicts. TCC allows transactions to observe stale data and also allows concurrent updates, asynchronously resolving conflicts via the convergent conflict resolution policy (set union in our case).

To determine whether a strongly consistent serverless framework could compete with HydroCache and Anna, we conduct two experiments. As a baseline, we replace Anna with Aurora-PostgreSQL (with SI) and measure performance. We warm the Aurora database by querying every key once in advance of the experiment, so that all future requests hit Aurora's buffer cache and there are no disk accesses on the critical path. Second, we perform a more apples-to-apples comparison using HydroCache and Anna, replaying the cache misses and abort/retry patterns observed in Aurora.

In the first experiment, we observe that SI is over an order of magnitude slower than HB at both the median and the 99th percentile across all contention levels. (Due to space constraints, no figure is shown for this experiment.) There are four likely reasons for this gap. The first three echo the guarantees offered by SI. (a) Guarantee (1) requires that when a transaction first reads any key k, it must bypass our cache and fetch k from an up-to-date database replica. (b) Both guarantees (1) and (2) require coordination inside Aurora to ensure that replicas agree on the set of committed transactions. (c) Guarantee (2) causes transactions to abort/retry. The fourth reason is a matter of system architecture: (d) Aurora is built on PostgreSQL, almost certainly resulting in more query overhead than HydroCache and Anna.

The absolute numbers from this experiment do not provide much insight into the design space due to reason (d). However, workload traces can be used to simulate the performance of SI in HydroCache and Anna.

Therefore, in our second experiment, we take the Aurora trace (accounting for cache misses and abort/retry count) and run it under LWW (our fastest option) using HydroCache and Anna. This is a *lower bound* on the latency of a full SI implementation, as it doesn't account for coordination overheads (reason (b)). The overhead of coordination protocols such as two-phase commit is at least 17ms as reported by Google Spanner [31] and worsens as the system scales.

The SI (mock) bars in Figure 4.3 show the simulated results. Median latency is worse than HB's due to the added network round-trip overhead during cache misses. While HB's 99th percentile latency is insensitive to workload skew, SI's tail latency is over 3x worse as we increase the contention from Zipf=1.0 to 1.5. Under high contention, a large number of transactions concurrently update a single key and only one is allowed to commit. All other transactions are aborted and retried, contributing to the high tail latency. In fact, 23% of the transactions are retried at least once, and in the worst case, a transaction is retried 18 times before committing. Thus, SI does not meet our goal of low-latency function serving.

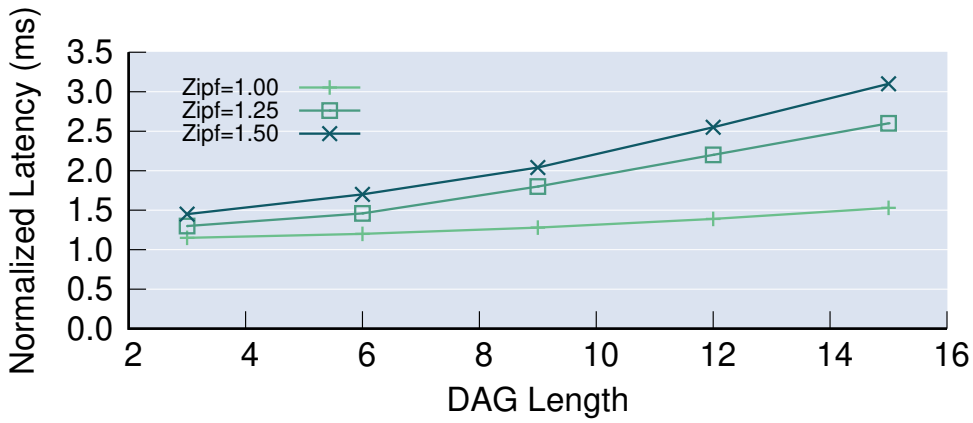

Figure 4.4: Normalized median latency as we increase the length of a linear DAG.

***Takeaway**: MTCC protocols incur a reasonable overheads—at most 2× performance penalty and sub-KB metadata overheads—compared to weaker protocols while preventing anomalies on up to 90% of requests. MTCC also avoids the cache misses and expensive aborts caused by snapshot isolation.*

4.4.4 Scalability

Thus far, we have only studied small DAGs of length at most 3. In this section, we explore the scalability of our MTCC protocol as DAG length increases. Figure 4.4 reports the performance of the HB protocol as a function of the length of a linear DAG for different contention levels. We normalize (divide) the latency by the length of the DAG. Ideally, we would expect a constant normalized latency for each skew.

In practice, as we increase DAG length from 3 to 15, the normalized latency grows under all contention levels. Longer DAGs with larger read sets require creating larger snapshots, so the volume of causal metadata being passed along the DAG increases linearly with respect to DAG length. Furthermore, the protocol must communicate across a larger number of caches to construct the snapshot as the DAG grows. Under moderate contention (Zipf=1.0), the normalized latency grows from 1.15ms to 1.53ms (a 33% increase), while under high contention (Zipf=1.5), the normalized latency doubles.

The reason for the difference across contention levels is that the OPT subroutine avoids data fetches for 91% of requests under moderate contention. Under high contention, 75% of requests require OPT to ship data across caches, and 12% of requests require the CON subroutine to communicate with all caches, significantly increasing latencies.

***Takeaway**: Our MTCC protocols scale well from short to long DAGs under moderate contention, with normalized latencies only increasing by 33%; however, high contention workloads require large amounts of expensive data shipping, leading to a 2× increase in normalized latency.*

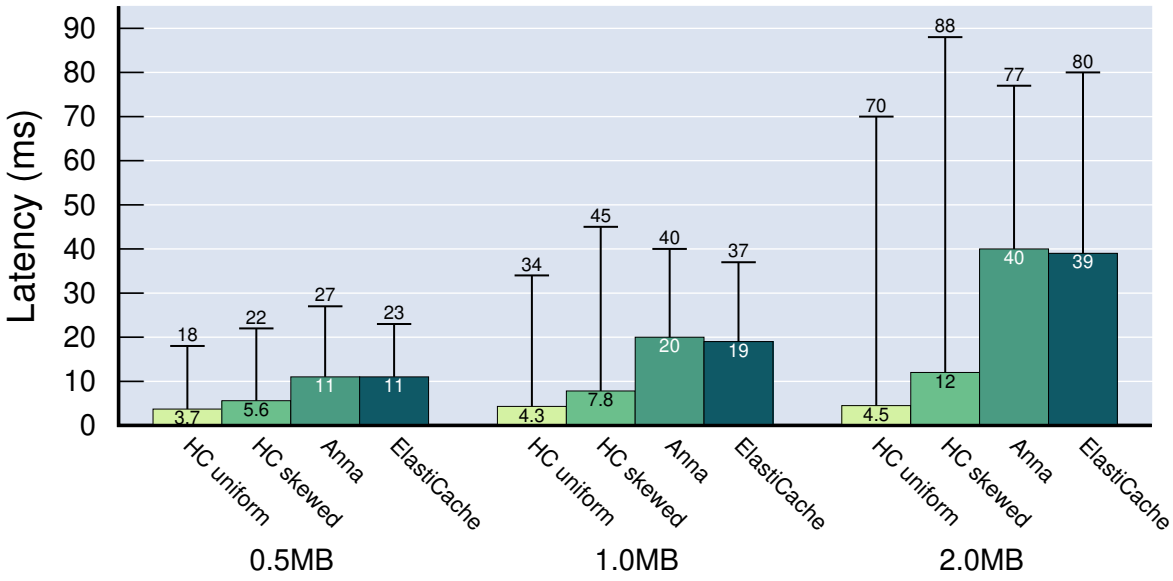

Figure 4.5: Median and P99 latency comparison against architectures without HydroCache.

4.4.5 Life Without HydroCache

Finally, we investigate the benefits of HydroCache (using the HB protocol) relative to a cache-less architecture. We study two different architectures: One fetches data directly from Anna, and another fetches data from AWS ElastiCache (using Redis in cluster mode). Although ElastiCache is not an autsocaling system, we include it in our evaluation because it is an industry-standard, memory-speed KVS [82].

Caching Benefits

We first study the performance benefits of HydroCache. We begin with caches empty, and execute linear DAGs with 3 functions. In previous experiments, the reads and writes were drawn from the same distribution. Here, we draw the read set from a Zipfian distribution of 100,000 keys with a coefficient of 1.5. We study two, less-skewed distributions for the write set—a uniform distribution and a Zipfian distribution with coefficient 0.5. This variation in distributions is commonly observed in many social network applications [39].

Writing new key versions forces HydroCache to exercise the MTCC protocol, which highlights its performance overhead under different levels of write skew. To focus on the benefit of caching large payloads, we configure each DAG executed with HydroCache to write a small (1 byte) payload to avoid the overhead of expensive writes to the KVS. To prevent this write from clobbering the original large payload—thereby reducing the cost of future KVS reads—we attach the 1 byte payload with a vector clock concurrent with what is stored in Anna. This payload will be *merged*

	Write Skew					
	Uniform	0.5	0.75	1.0	1.25	1.5
HydroCache	0%	0%	0%	0%	0%	0%
Anna	0.02%	0.4%	1.9%	6.6%	15%	21%
ElastiCache	0.03%	0.4%	2.0%	6.6%	14%	20%

Table 4.3: Rates of inconsistencies observed with varying write skew for HydroCache/Anna, Anna only, and ElastiCache.

with the original payload via set union so that further reads to the same key still fetch both payloads. Since ElastiCache does not support payload merge, we simplify the workload of cache-less architectures by making the DAG read-only.

Figure 4.5 compares HydroCache, Anna, and ElastiCache as we increase payload size from 0.5MB to 2MB. Anna and ElastiCache exhibit similar performance for all payload sizes. HydroCache with uniform random writes outperforms both systems by 3× at median for small payloads and 9× for large payloads. 99th percentile latencies are uniform across systems because cache misses incur remote data fetches.

With a slightly skewed write distribution (Zipf=0.5), our protocol is forced to more frequently ship data between executor nodes to construct a distributed snapshot. For large payloads, the median latency is only 3.5× better than other systems, and 99th percentile latency is in fact 14% higher.

***Takeaway**: HydroCache improves performance over cache-less architectures by up to an order-of-magnitude by avoiding expensive network hops for large data accesses.*

Consistency Benefits

We measure how many inconsistencies HydroCache prevents relative to Anna and ElastiCache, neither of which guarantee TCC. To track violations in the other systems, we embed causal metadata directly into each key's payload when writing it to storage. At the end of a request, we extract the metadata from the read set to check whether the versions formed a snapshot.

Table 4.3 shows the rates of TCC violations for different write skews. The number of inconsistencies increases significantly as skew increases. For the two most skewed distributions, over 14% of requests fail to form a causal snapshot.

***Takeaway**: In addition to improving performance, HydroCache prevents up to 21% of requests from experiencing causal consistency anomalies that occur in state-of-the-art systems like Redis and Anna.*

4.5 Related Work

Many recent storage systems provide causal consistency in various settings, including COPS [67], Eiger [68], Orbe [36], ChainReaction [7], GentleRain [35], Cure [4] and Occult [74]. However,

these are fixed-deployment systems that do not meet the autoscaling requirements of a serverless setting. [74, 4, 7, 36, 35] rely on linear clocks attached to each data partition to version data, and they use a fixed-size vector clock comprised of the linear clocks to track causal dependencies across keys. The size of these vector clocks is tightly coupled with system deployment—specifically, the shard and replica counts. Correctly modifying this metadata when the system autoscales requires an expensive coordination protocol, which we rejected in HydroCache's design. [67] and [68] reveal a new version only when all of its dependencies have been retrieved. This design is susceptible to "slowdown cascades" [74], in which a single straggler node limits write visibility and increases the cost of write buffering.

By extending the Bolt-On Causal Consistency approach [17], HydroCache guarantees TCC at the caching layer, separate from the complexity of the autoscaling storage layer. Each cache creates its own causal cut without coordination, eliminating the possibility of a slowdown cascade, which removes concerns about autoscaling at the compute tier. Our cache tracks dependencies via individual keys' metadata rather than tracking the linear clocks of fixed, coarse-grained data partitions. This comes at the cost of increased dependency metadata overhead; we return to this in Section 4.6.

Another causal system that employs a client-side caching approach is SwiftCloud [116]. That work assumes that clients are resource-poor entities like browsers or edge devices, and the core logic of enforcing causal consistency is implemented in the data center to which client caches connect. We did not consider this design because constructing a causal cut for an entire datacenter is expensive, especially in a serverless setting where the system autoscales. Moreover, SwiftCloud is not designed to guarantee causal consistency across *multiple* caches, one of the main contributions of our work.

4.6 Conclusion and Takeaways

Disaggregating compute and storage services allows for an attractive separation of concerns regarding autoscaling resources in a serverless environment. However, disaggregation introduces performance and consistency challenges for applications written on FaaS platforms. In this work, we presented HydroCache, a distributed cache co-located with a FaaS compute layer that mitigates these limitations. HydroCache guarantees transactional causal consistency for individual functions at a single node, and we developed new multisite TCC protocols that offer the same guarantee for compositions of functions spanning multiple physical nodes. This architecture enables up to an order-of-magnitude performance improvements while also preventing a plethora of anomalies to which existing cloud infrastructure is susceptible.

Through this work, we learned that resource disaggregation is not at odds with low latency and consistency. By rearchitecting the system with a caching layer and employing efficient consistency protocols, we managed to combine features for high-performance request handling, independent scaling of compute and storage layers, and robust consistency within a single FaaS system.

Chapter 5

Summary and Open problems

We have seen tremendous progress in cloud technologies over the past decade. Hardware virtualization and VM technology allow every organization to harness the power of large computing clusters. Innovations around pay-as-you-go FaaS platforms further allow programmers without deep systems expertise to deploy and serve applications with a few clicks without having to worry about resource configurations, scaling, fault-tolerance, and systems security. However, as we discussed throughout this dissertation, state-of-the-art systems still face key limitations that prevent them from meeting the demanding requirements of serverless computing. In this chapter, we summarize our solutions to these challenges and highlight interesting future research directions.

Any-scale high performance

As cloud providers continue to roll out powerful machines, it becomes crucial to design systems that efficiently utilize the processing capacity as much as possible. In Chapter 2, we showed that traditional shared memory architectures incur significant coordination overheads. This cost is pronounced especially under high-contention write-heavy workloads; in these cases, up to 90% of the CPU time is devoted to lock acquisition or atomic retry. That is why we built Anna, which eliminates coordination by employing wait-free execution that lets each thread access private memory and asynchronously gossip updates via message passing in the background. The most elegant aspect of this design is that it requires no change as the system scales from a single multi-core machine to a distributed cluster. Moreover, the coordination-free execution model leads to an opportunity for Anna to support a wide spectrum of consistency models via lattice composition, allowing the system to support a diverse set of applications.

It is worth noting, however, that coordination-free consistency levels put burden on programmers to reason about the appropriate consistency models required to prevent application anomalies. An open research problem is how to let the programmers conveniently declare a set of application-level invariants and let the system automatically configure the most performant consistency model that satisfies these constraints.

SLO-driven autoscaling

As we discussed in Chapter 3, the majority of systems today are designed for a specific point in the cost-performance trade-off space. However, real-world workloads dynamically change over time, which introduces systems challenges around elastically scaling the deployment to maintain high performance and reduce resource inefficiency. To this end, we have studied various autoscaling mechanisms and policies in the context of Anna. The resulting serverless KVS allows users to specify high-level SLOs around performance, cost budget, and fault-tolerance. The system seamlessly adapts to shifting workloads, outperforming competing solutions such as DynamoDB by orders-of-magnitude in terms of cost-efficiency.

Despite the promising results, we believe there are a few directions in policies and SLOs worth exploring further. First, our current policy design is entirely reactive, taking action based on current state. To improve this, it would be interesting to explore *proactive* policies that anticipate upcoming workload changes and act in advance [96, 88, 80, 70]. By combining advanced predictive techniques with Anna's swift adaptivity, we believe Anna will further excel in meeting SLOs & SLAs. Second, the system administrator currently defines a single latency objective corresponding to an overall average. For any system configuration, there are adversarial workloads that can defeat this SLO. For example, in Section 3.5.4, a larger load spike could have forced Anna above its stated SLO for a long period. Therefore, it would be interesting to use pricing and incentives to design SLOs, SLAs and policies for both expected- and worst-case scenarios.

Low-latency with robust consistency

The key advantage of today's FaaS systems is the disaggregation of compute and storage. Decoupling the two layers allows them to scale independently, which significantly improves resource efficiency on a wide range of workloads from compute-intensive applications such as online model serving to data-intensive applications such as retail shopping cart management. Unfortunately, the current design comes at the cost of high latency I/O and poor consistency guarantees. In our HydroCache project (Chapter 3), we have redesigned the FaaS infrastructure from the ground up and shown that it is possible to simultaneously achieve resource disaggregation, low latency request serving, and support transactional causal consistency (the strongest amongst coordination-free models). With novel distributed caching and consistency protocols, HydroCache reduces the amount of cross-node data fetching while preventing a large number of consistency anomalies otherwise observed in traditional FaaS systems.

To further expand the applicability of FaaS platforms, we believe it is crucial to add support for strong consistency levels. Certain pieces of an application that require strong guarantees will be deployed on a set of nodes with limited elasticity, which achieve strong consistency via coordination protocols such as Paxos and two-phase commit. Other components will run on top of a coordination-free layer, enjoying the benefits of low latency and autoscaling. We think the problem of automating this deployment process while minimizing developer intervention is an exciting avenue for future research.

Bibliography

[1] *10Gb Ethernet Tests*. http://zeromq.org/results:10gbe-tests/. 2018.

[2] Swarup Acharya et al. "Broadcast disks: data management for asymmetric communication environments". In: *Mobile Computing*. Springer, 1995, pp. 331–361.

[3] *Akamai*. https://www.akamai.com.

[4] Deepthi Devaki Akkoorath et al. "Cure: Strong semantics meets high availability and low latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 405–414.

[5] Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 923–935.

[6] Mohammad Alizadeh et al. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 63–74. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851192. URL: http://doi.acm.org/10.1145/1851182.1851192.

[7] Sérgio Almeida, João Leitão, and Luís Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 85–98. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465361. URL: http://doi.acm.org/10.1145/2465351.2465361.

[8] Peter Alvaro et al. "Consistency Analysis in Bloom: a CALM and Collected Approach." In: *CIDR*. 2011.

[9] Amazon Web Services. *Amazon DynamoDB Developer Guide (API Version 2012-08-10)*. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html. Accessed May 3, 2018. Aug. 2012.

[10] Hrishikesh Amur et al. "Robust and Flexible Power-proportional Storage". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 217–228. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807164. URL: http://doi.acm.org/10.1145/1807128.1807164.

[11] Ganesh Ananthanarayanan et al. “Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters”. In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys ’11. Salzburg, Austria: ACM, 2011, pp. 287–300. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966472. URL: http://doi.acm.org/10.1145/1966445.1966472.

[12] *Apache Cassandra*. https://cassandra.apache.org/. 2018.

[13] *Amazon Web Services*. https://aws.amazon.com.

[14] *AWS Lambda - Case Studies*. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[15] *Microsoft Azure Cloud Computing Platform*. http://azure.microsoft.com.

[16] *Azure DocumentDB*. https://azure.microsoft.com/en-us/services/documentdb/. 2018.

[17] Peter Bailis et al. “Bolt-on Causal Consistency”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: ACM, 2013, pp. 761–772. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465279. URL: http://doi.acm.org/10.1145/2463676.2465279.

[18] Peter Bailis et al. “Coordination Avoidance in Database Systems”. In: *PVLDB* 8.3 (2015).

[19] Peter Bailis et al. “Highly Available Transactions: Virtues and Limitations”. In: *PVLDB* 7.3 (2013), pp. 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237. URL: http://dx.doi.org/10.14778/2732232.2732237.

[20] Peter Bailis et al. “The potential dangers of causal consistency and an explicit solution”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 22.

[21] Paul Barham et al. “Xen and the Art of Virtualization”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. ISBN: 1581137575. DOI: 10.1145/945445.945462. URL: https://doi.org/10.1145/945445.945462.

[22] Andrew Baumann et al. “The Multikernel: A New OS Architecture for Scalable Multicore Systems”. In: *SOSP*. 2009.

[23] Philip A Bernstein et al. “Indexing in an Actor-Oriented Database.” In: *CIDR*. 2017.

[24] Silas Boyd-Wickizer et al. “Non-scalable locks are dangerous”. In: *OLS*. 2012.

[25] Longbin Chen et al. “E2FS: an elastic storage system for cloud computing”. In: *The Journal of Supercomputing* 74.3 (Mar. 2018), pp. 1045–1060. ISSN: 1573-0484. DOI: 10.1007/s11227-016-1827-3. URL: https://doi.org/10.1007/s11227-016-1827-3.

[26] Christopher James Clark. “Courtship dives of Anna’s hummingbird offer insights into flight performance limits”. In: *Proceedings of the Royal Society of London B: Biological Sciences* (2009), rspb20090508.

[27] Neil Conway et al. “Logic and Lattices for Distributed Programming”. In: *SoCC*. 2012.

[28] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[29] Brian F. Cooper et al. “PNUTS: Yahoo!’s Hosted Data Serving Platform”. In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: https://doi.org/10.14778/1454159.1454167.

[30] George Copeland et al. “Data placement in Bubba”. In: *ACM SIGMOD Record*. Vol. 17. 3. ACM. 1988, pp. 99–108.

[31] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013). ISSN: 0734-2071. DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245.

[32] Jeff Dean. “Challenges in Building Large-Scale Information Retrieval Systems”. In: *WSDM*. 2009.

[33] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[34] *Kubernetes - Build, Ship, and Run Any App, Anywhere*. https://www.docker.com.

[35] Jiaqing Du et al. “GentleRain: Cheap and scalable causal consistency with physical clocks”. In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–13.

[36] Jiaqing Du et al. “Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 11:1–11:14. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523628. URL: http://doi.acm.org/10.1145/2523616.2523628.

[37] Jose M Faleiro and Daniel J Abadi. “Rethinking serializable multiversion concurrency control”. In: *PVLDB* 8.11 (2015).

[38] Jose M. Faleiro and Daniel J. Abadi. “Latch-free Synchronization in Database Systems: Silver Bullet or Fool’s Gold?” In: *CIDR*. 2017.

[39] Feng Fu, Lianghuan Liu, and Long Wang. “Empirical analysis of online social networks in the age of Web 2.0”. In: *Physica A: Statistical Mechanics and its Applications* 387 (Jan. 2008), pp. 675–684. DOI: 10.1016/j.physa.2007.10.006.

[40] Hector Garcia-Molina and Kenneth Salem. *Sagas*. Vol. 16. 3. ACM, 1987.

[41] *Google Cloud Platform*. https://cloud.google.com.

[42] AJ Ross, Adrian Hilton, and Dave Rensin. *SLOs, SLIs, SLAs, oh my - CRE life lessons*. https://cloudplatform.googleblog.com/2017/01/availability-part-deux--CRE-life-lessons.html. Accessed May 3, 2018. Jan. 2017.

[43] *HBase*. https://hbase.apache.org. 2018.

[44] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019. URL: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[45] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[46] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[47] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. Boston, MA, USA. 2010.

[48] *Intel Thread Building Blocks*. https://www.threadingbuildingblocks.org/. 2018.

[49] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[50] Elena Kakoulli and Herodotos Herodotou. "OctopusFS: A Distributed File System with Tiered Storage Management". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 65–78. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064023. URL: http://doi.acm.org/10.1145/3035918.3064023.

[51] Robert Kallman et al. "H-store: A High-performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1496–1499. ISSN: 2150-8097.

[52] David Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[53] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. "BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 485–500. ISBN: 978-1-931971-29-4. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khandelwal.

[54] Andrey Kolesnikov and Martin Kulas. "Load modeling and generation for IP-Based networks: a unified approach and tool support". In: *International GI/ITG Conference, MMB & DFT*. Springer. 2010, pp. 91–106.

[55] *Kubernetes: Production-Grade Container Orchestration*. http://kubernetes.io.

[56] Kubernetes. *Set up High-Availability Kubernetes Masters*. https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/. Accessed May 3, 2018.

[57] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788. URL: http://doi.acm.org/10.1145/2723372.2742788.

[58] Rivka Ladin et al. "Providing High Availability Using Lazy Replication". In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 360–391. ISSN: 0734-2071. DOI: 10.1145/138873.138877. URL: http://doi.acm.org/10.1145/138873.138877.

[59] *Common Lambda Application Types and Use Cases*. https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html.

[60] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998).

[61] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[62] Kasper Green Larsen et al. "Heavy hitters via cluster-preserving clustering". In: *CoRR* abs/1604.01357 (2016). arXiv: 1604.01357. URL: http://arxiv.org/abs/1604.01357.

[63] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. "The Bw-Tree: A B-tree for new hardware platforms". In: *ICDE*. 2013.

[64] Haoyuan Li et al. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 6:1–6:15. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670985. URL: http://doi.acm.org/10.1145/2670979.2670985.

[65] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server.” In: *OSDI*. Vol. 14. 2014, pp. 583–598.

[66] Hyeontaek Lim et al. “MICA: a holistic approach to fast in-memory key-value storage”. In: *NSDI*. 2014.

[67] Wyatt Lloyd et al. “Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS”. In: *SOSP’11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. Oct. 2011, pp. 401–416. DOI: 10.1145/2043556.2043593.

[68] Wyatt Lloyd et al. “Stronger semantics for low-latency geo-replicated storage”. In: *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 313–328.

[69] David Lomet and Betty Salzberg. “Access Methods for Multiversion Data”. In: *SIGMOD Rec.* 18.2 (June 1989), pp. 315–324. ISSN: 0163-5808. DOI: 10.1145/66926.66956. URL: http://doi.acm.org/10.1145/66926.66956.

[70] Lin Ma et al. “Query-based Workload Forecasting for Self-Driving Database Management Systems”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. 2018, pp. 631–645. DOI: 10.1145/3183713.3196908. URL: https://db.cs.cmu.edu/papers/2018/mod435-maA.pdf.

[71] P. Mahajan, L. Alvisi, and M. Dahlin. *Consistency, Availability, Convergence*. Tech. rep. TR-11-22. Computer Science Department, University of Texas at Austin, May 2011.

[72] Amit Manjhi, Suman Nath, and Phillip B. Gibbons. “Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’05. Baltimore, Maryland: ACM, 2005, pp. 287–298. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066191. URL: http://doi.acm.org/10.1145/1066157.1066191.

[73] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. “Cache craftiness for fast multicore key-value storage”. In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, pp. 183–196.

[74] Syed Akbar Mehdi et al. “I Can’t Believe It’s Not Causal! Scalable Causal Consistency with No Slowdown Cascades”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 453–468. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi.

[75] *Memcached*. https://www.memcached.org. 2018.

[76] *MongoDB*. https://www.mongodb.com. 2018.

[77] Edward Oakes et al. “SOCK: Rapid Task Provisioning with Serverless-Optimized Containers”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 57–70. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[78] Diego Ongaro and John K Ousterhout. "In search of an understandable consensus algorithm." In: *USENIX Annual Technical Conference*. 2014.

[79] Ruoming Pang et al. "Zanzibar: Google's Consistent, Global Authorization System". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 33–46. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/pang.

[80] Andrew Pavlo et al. "Self-Driving Database Management Systems". In: *CIDR 2017, Conference on Innovative Data Systems Research*. 2017. URL: https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf.

[81] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. "SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads". In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT '13. Genoa, Italy: Association for Computing Machinery, 2013, pp. 430–441. ISBN: 9781450315975. DOI: 10.1145/2452376.2452427. URL: https://doi.org/10.1145/2452376.2452427.

[82] Tilmann Rabl et al. "Solving Big Data Challenges for Enterprise Application Performance Management". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1724–1735. ISSN: 2150-8097. DOI: 10.14778/2367502.2367512. URL: http://dx.doi.org/10.14778/2367502.2367512.

[83] Ananth Rao et al. "Load balancing in structured P2P systems". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 68–79.

[84] Sylvia Ratnasamy et al. *A scalable content-addressable network*. Vol. 31. 4. ACM, 2001.

[85] Michel Raynal and Mukesh Singhal. "Logical Time: Capturing Causality in Distributed Systems". In: *Computer* 29.2 (Feb. 1996), pp. 49–56. ISSN: 0018-9162. DOI: 10.1109/2.485846. URL: https://doi.org/10.1109/2.485846.

[86] *Redis*. http://redis.io/. 2018.

[87] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. "Design Principles for Scaling Multi-core OLTP Under High Contention". In: *SIGMOD*. 2016.

[88] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting". In: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 500–507. ISBN: 978-0-7695-4460-1. DOI: 10.1109/CLOUD.2011.42. URL: https://doi.org/10.1109/CLOUD.2011.42.

[89] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[90] *Seastar / ScyllaDB, or how we implemented a 10-times faster Cassandra*. https://goo.gl/E7cxGW. 2016.

[91] *Amazon Aurora Serverless*. https://aws.amazon.com/rds/aurora/serverless/.

[92] Jason Sewall et al. "PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors". In: *PVLDB* 4.11 (2011).

[93] Vivek Shah and Marcos Antonio Vaz Salles. "Reactors: A Case for Predictable, Virtualized Actor Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 259–274. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3183752. URL: http://doi.acm.org/10.1145/3183713.3183752.

[94] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. "Architectural Implications of Function-as-a-Service Computing". In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: ACM, 2019, pp. 1063–1075. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358296. URL: http://doi.acm.org/10.1145/3352460.3358296.

[95] Marc Shapiro et al. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[96] Ahmad Al-Shishtawy and Vladimir Vlassov. "ElastMan: Elasticity Manager for Elastic Key-value Stores in the Cloud". In: *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. CAC '13. Miami, Florida, USA: ACM, 2013, 7:1–7:10. ISBN: 978-1-4503-2172-3. DOI: 10.1145/2494621.2494630. URL: http://doi.acm.org/10.1145/2494621.2494630.

[97] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[98] Mukesh Singhal and Ajay Kshemkalyani. "An Efficient Implementation of Vector Clocks". In: *Inf. Process. Lett.* 43.1 (Aug. 1992), pp. 47–52. ISSN: 0020-0190. DOI: 10.1016/0020-0190(92)90028-T. URL: http://dx.doi.org/10.1016/0020-0190(92)90028-T.

[99] Vikram Sreekanti et al. "A Fault-Tolerance Shim for Serverless Computing". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387535. URL: https://doi.org/10.1145/3342195.3387535.

[100] Vikram Sreekanti et al. "Cloudburst: Stateful Functions-as-a-Service". In: *ArXiv* abs/2001.04592 (2020).

[101] Ion Stoica et al. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[102] Michael Stonebraker. “The Design of the POSTGRES Storage System”. In: *Proceedings of the 13th International Conference on Very Large Data Bases*. VLDB ’87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 289–300. ISBN: 0-934613-46-X. URL: http://dl.acm.org/citation.cfm?id=645914.671639.

[103] *Storm*. https://github.com/apache/storm.

[104] *Swarmify*. https://swarmify.com.

[105] D. B. Terry et al. “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”. In: *SOSP*. 1995.

[106] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. “Sierra: Practical Power-proportionality for Data Center Storage”. In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys ’11. Salzburg, Austria: ACM, 2011, pp. 169–182. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966461. URL: http://doi.acm.org/10.1145/1966445.1966461.

[107] Dana Van Aken et al. “Automatic database management system tuning through large-scale machine learning”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 1009–1024.

[108] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. “Towards elastic transactional cloud storage with range query support”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 506–514.

[109] Florian M. Waas. “Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database - (Invited Talk)”. In: *Business Intelligence for the Real-Time Enterprise - Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers*. Ed. by Umesh Dayal, Malu Castellanos, and Timos Sellis. 2008, pp. 89–96. DOI: 10.1007/978-3-642-03422-0_7. URL: https://doi.org/10.1007/978-3-642-03422-0_7.

[110] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-conditioned, Scalable Internet Services”. In: *SOSP*. 2001.

[111] John Wilkes et al. “The HP AutoRAID Hierarchical Storage System”. In: *ACM Trans. Comput. Syst.* 14.1 (Feb. 1996), pp. 108–136. ISSN: 0734-2071. DOI: 10.1145/225535.225539. URL: http://doi.acm.org/10.1145/225535.225539.

[112] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. “Autoscaling tiered cloud storage in Anna”. In: *Proceedings of the VLDB Endowment* 12.6 (2019), pp. 624–638.

[113] 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*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 83–97. ISBN: 9781450367356. DOI: 10.1145/3318464.3389710. URL: https://doi.org/10.1145/3318464.3389710.

[114] Chenggang Wu et al. "Anna: A kvs for any scale". In: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[115] Lianghong Xu et al. "SpringFS: Bridging Agility and Performance in Elastic Distributed Storage". In: *Proc. of the 12th USENIX FAST*. Santa Clara, CA: USENIX, 2014, pp. 243–255. ISBN: ISBN 978-1-931971-08-9.

[116] Marek Zawirski et al. "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications". In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 75–87. ISBN: 978-1-4503-3618-5. DOI: 10.1145/2814576.2814733. URL: http://doi.acm.org/10.1145/2814576.2814733.