Type-Aware Programming Models for Distributed Applications

Brandon Holt

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Luis Ceze, Chair

Mark Oskin, Chair

Dan Ports

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Type-Aware Programming Models for Distributed Applications

Brandon Holt

Co-Chairs of the Supervisory Committee:
Associate Professor Luis Ceze
UW CSE

Associate Professor Mark Oskin
UW CSE

Modern applications are distributed: from the simplest interactive web applications to social networks with massive datacenters around the world. Even simple distributed applications depend on a complex ecosystem of servers, databases, and caches to operate. In order to scale services and handle turbulent internet traffic, developers of distributed applications constantly balance fundamental tradeoffs between parallelism and locality, replication and synchronization, consistency and availability. This task is made more difficult by the fact that each component operates independently by design, knowing little about the original intent of the application or its specific performance needs. Layers of abstraction between the application and its data prevent the system from adapting itself to better meet the requirements of the application.

Distributed application developers need interfaces that can communicate the structure and semantics of their programs to distributed systems that know how to use that information to optimize performance. Programmers should be able to improve data layout without completely re-architecting the system, and tell the system which data accesses should be less accurate or the highest priority. The system should be able to find concurrency and exploit it, leveraging weaker constraints to improve performance. Programmers should be protected from common mistakes, such as consistency bugs, by the languages and platforms they use.

This dissertation explores new programming models that use type systems and abstract data types to communicate application semantics to distributed systems. The new interfaces place minimal burden on programmers by using the abstract behavior of existing data structures to naturally express high-level properties. New runtime techniques and optimizations are proposed to correspond with each additional piece of information passed down to the underlying system. These techniques leverage concurrency both in massively data-parallel analytics workloads and in web-service workloads with abundant inter-request parallelism. First, we propose a way to automatically move computation closer to data, statically analyzing remote data accesses and improving locality through compiler-assisted lightweight thread migrations. Next, we present the design of global shared data structures that enable threads to cooperate rather than contend for access using distributed combining. Then we explore ways of exposing concurrency between transactions in distributed datastores using abstract properties of the datatypes, such as commutativity. Finally, we introduce a programming model, IPA, that makes it safer to trade off consistency for performance. Explicit performance and correctness constraints allow the system to adapt to changing conditions by relaxing the consistency of some operations, secure in the knowledge that the type system will enforce safety by requiring the developer to consider the effects of weak operations. Together, these programming models and techniques in this work contribute to the toolkit available to distributed application developers to make their lives easier and their software more robust.

Contents

1. Introduction

Modern applications have grown beyond what a single machine is capable of handling. It is now hard to even think of compelling new applications that require no external compute, cloud storage, or communication among users. From social networking sites and games (Facebook, Pokemon Go), to online retail services (Amazon, Etsy) and collaborative working tools (Github, Google Docs) — all of these are split between client devices and servers distributed among many machines spread among datacenters around the world.

Building software that spans these varied machines is fundamentally more challenging than traditional standalone applications. Consider the architecture of a modern web service such as Twitter: the mobile phone app communicates over the wide-area internet with frontend servers in one of Twitter's datacenters. This frontline of servers, however, merely routes requests to some other set of services which gather data for the user, such as their timeline of recent tweets, recommendations for users to follow, and advertisements. Each of these in turn is itself a distributed application, running across multiple machines, with data stored in each machine's memory, in other caching services, and in slower persistent storage. Designing this kind of software requires decisions about the protocols and APIs each component uses to communicate with the others, where each piece of functionality should reside, as well as countless other questions about storage systems to use and low-level implementation details.

In addition to purely functional design choices, distributed applications introduce many fundamentally difficult performance decisions, such as

- **Replication and consistency:** Data is often replicated for fault tolerance and high availability, but this creates a tension between availability (e.g. quick responses in the face of failure) and consistency (correct values that everyone agrees on).

- **Sharding and locality:** Not all data will fit on a single machine, so it must be split among many machines. But where should data be placed? What should go together? How much data must be moved to compute something?

- **Parallelism and synchronization:** More machines means more compute resources but requires more coordination to ensure correct results. At what point does the additional synchronization outweigh the benefit of parallelism? Which ordering constraints are actually necessary? Where are the true serialization bottlenecks?

Developing traditional standalone programs, we are used to a suite of tools that support building correct and efficient software. Most notably, we rely on programming languages and compilers to understand the programs we write and optimize them for the machine they run on. Type systems prevent common mistakes like assigning an incorrect value to a variable or dereferencing a null pointer. Compilers and runtime systems support programmers by freeing them from burdens such as explicit memory management. Finally, hardware automatically manages data, caching it close the cores that are likely to use it again, moving data around transparently to wherever it is needed.

When it comes to distributed applications, however, developers are on the hook. They must ensure they use APIs correctly, choose the correct level of consistency, and distribute their data among services explicitly. Most of the knowledge about application semantics is lost at the boundary between each service; only information that is explicitly shared through the provided interface can be leveraged by the system. Take a typical application using a key-value store for its persistent state. The application may have a rich hierarchy of classes and data structures that represent its data. However, if the key-value store's interface only provides `put` and `get` operations on byte strings, then most of this structure will be lost when the application sends its data to the storage system. Accordingly, the key-value store has no chance of optimizing for the particular use case of this application. It can't predict which data is more likely to be accessed together, or understand when two updates to the same key could be performed independently, or know which operations are the most important to get right. However, if the interface is expressive enough to provide more of these semantics from the application and the system is designed to take advantage of that additional knowledge to inform how it handles that data, then programmers can benefit from optimizations across the many dimensions of performance.

1.1. Overview

The overarching goal of this dissertation is to help programmers express their distributed application's needs to the system, and then design and build systems that can take advantage of that knowledge to improve performance. This has been borne out in several distinct projects which leverage different pieces of knowledge to optimize for different situations.

A portion of the work in this dissertation was done as part of a larger effort to build a latency-tolerant distributed shared memory runtime called Grappa. The core of the system is the subject of Jacob Nelson's dissertation [117] and is described in detail in [116]. The work described in this document proposes novel solutions to general problems, some of which leverage particular strengths of the Grappa platform or address weaknesses by extending its core functionality.

1.1.1. Alembic: Automatic locality extraction via migration

A common challenge faced by distributed applications is co-locating computation with the data that it operates on. In the Grappa runtime system there exist mechanisms to perform arbitrary computation on a remote machine where some piece of data resides. This chapter introduces a compiler extension that automatically determines which parts of each thread should execute remotely. New annotations are introduced to C++ to distinguish special "global" pointers, which the compiler uses in its locality analysis, eventually transforming threads into a sequence of continuations, implementing a form of compiler-assisted lightweight thread migration. The analysis and transformation passes are implemented in LLVM [98] and evaluated on several existing Grappa applications.

1.1.2. Distributed combining: Reducing contention with cooperation

Due to the extreme degree of parallelism possible in distributed systems such as the Grappa runtime, globally shared data structures must be designed carefully to handle high contention. Naive implementations of strongly consistent linearizable data structures require all operations to be serialized, limiting throughput to that of a single thread. However, much of this contention can be mitigated if threads cooperate among themselves locally to combine their operations before applying them in bulk on the contended data structure. This effectively distributes and parallelizes the synchronization process, leveraging the associativity of many operations. The proposed techniques for distributed data structure design are described for an environment like Grappa but could be applied to other distributed environments.

1.1.3. Claret: Exposing concurrency in transactions with ADTs

Just as combining above leveraged associativity to reduce contention, abstract properties of data types can also be used in the context of distributed transactions to reduce conflicts. This work uses the notion of *abstract data types* (ADTs) to communicate semantics between users, who model their application state using common data structures, and the transactional datastore which they use to store them. By exposing abstract properties to the transaction processor, this system is able to reduce conflicts and synchronization bottlenecks, improving the performance achievable with strong consistency. Adding distributed transaction support to a Redis-like [135] prototype datastore, this work shows that strong consistency can perform nearly as well as a non-transactional version.

1.1.4. Disciplined inconsistency: Safely trading consistency for performance

Distributed applications and web services, such as online stores or social networks, are expected to be scalable, available, responsive, and fault-tolerant. To meet these steep requirements in the face of high round-trip latencies, network partitions, server failures, and load spikes, applications use eventually consistent datastores that allow them to weaken the consistency of some data. However, making this transition is highly error-prone because relaxed consistency models are notoriously difficult to understand and test.

This chapter proposes a new programming model for distributed data that makes consistency properties explicit and uses a type system to enforce *consistency safety*. With the *Inconsistent, Performance-bound, Approximate* (IPA) storage system, programmers specify performance targets and correctness requirements as constraints on persistent data structures and handle uncertainty about the result of datastore reads using new *consistency types*. Built in Scala on top of the open-source Cassandra datastore [11], IPA is shown to prevent consistency-based programming errors and improve performance by adapting to changing network conditions.

1.2. Tasting notes

The projects above contribute to an ever-growing, though still impoverished, toolkit available to developers of distributed applications. This document is organized according to these discrete projects, but several common threads (pardon the parallelism pun) tie this work together. These same few fundamental challenges permeate much of distributed systems, yet luckily this is not strictly a zero-sum game; by using additional knowledge the programmer may already have, many of these challenges can be mitigated. The trick lies in exposing situations where prior solutions had to be overly conservative due to lack of insight. We will see many instances of these themes crop up throughout later chapters, but here we give you a hint of what to watch for.

1.2.1. Skew and contention

Natural phenomena have a tendency to follow power law distributions: from Zipf's Law which observed that the frequency of words in natural language follows a power law, to the power-law degree distributions that cause low diameter networks (colloquially "six degrees of separation") and cause Instagram to make special cases for every time Justin Bieber posts a selfie [108]. Network effects can amplify small signals into deluges of activity, causing memes to propagate virally through social networks, blogs, and news sites, inundating services under heavy load without warning, such as in February 2015 when a picture of a black and blue dress exploded across the internet bringing unprecedented traffic to social and news sites like BuzzFeed [118]. Systems with real-time components encounter spikes of activity as events occur in real life — in its early days, goals during World Cup games famously caused Twitter to crash and show the "fail whale" [81], and even in its 8th year of dealing with unpredictable traffic, Twitter briefly fell victim in 2014 after Ellen Degeneres posted a selfie at the Oscars which was retweeted at a record rate [15].

Skewed distributions in space and time must be taken into account when designing systems. Solutions that work for the average case (e.g. an average Twitter user) may not work for extremely popular users or viral memes. Increasingly interactive distributed applications result in a high degree of writes to data, which leads to contention on heavily accessed objects. Contention was a key factor in the design of each of the components of this dissertation. The Grappa runtime system, designed for such irregular workloads, required data structures which can handle high contention, which are explored in Chapter 3. Chapter 4 introduces a way to use information already in applications to reduce the impact of these high-contention workloads by exposing the high-level semantics of write operations to the underlying store. The IPA programming model in Chapter 5 helps applications adapt to changing conditions, ensuring the code handles both common and extreme cases.

1.2.2. Parallelism, locality, and synchronization

Data layout has always been important for application performance. However, distributed applications must be particularly concerned about where their data resides because retrieving data from remote machines can be orders of magnitude more expensive than main memory. This cost comes partly from simple latency resulting from physical distance, but software overhead within the complex network stack and operating system interactions make for significant cost even among neighboring machines.

Locality also plays a role in synchronization. In general, guaranteeing linearizability — where all observers agree on a single total order of operations on an object or record — requires sequencing concurrent operations. This is typically done by designating one owner (such as a single thread) which orders all operations that it receives. Note that this is analogous to how coherence protocols work in multi-processors. As an aside, multiple machines can coordinate to agree on a common ordering using expensive consensus protocols (e.g. Paxos [97]), but this is done for fault tolerance, not performance, and typically comes down to choosing a leader which can trivially order operations.

Because of this reliance on a single owner to order operations, applications must choose carefully where to place data and computation. Heavily modified data cannot easily be cached, but with the high cost of communication, it is important to ensure that computation happens as close as possible to the data it uses. Chapter 2 proposes a compiler (*Alembic*) that can leverage its knowledge of data accesses to move computation (essentially migrating threads) closer to data, even if it is spread among multiple machines. Both Chapter 3 and Chapter 4 use the idea of *combining* to get around the single-owner problem by pre-synchronizing operations in parallel before sequencing them.

1.2.3. Consistency and ordering

Consistency models[1] allow programmers to reason about the behavior of reads and writes to replicated data, particularly properties specifying the observable order of updates. The weakest, eventual consistency [158], merely guarantees that at some point in the future, all replicas will agree, but says nothing about when or which updates may be reflected. On the other hand, *read your writes* [151] guarantees, as the name implies, that reads will reflect at least the last write made by the same client.

Stronger consistency necessitates more synchronization and comes

[1]The term *consistency* means something different to everyone, from the "C" in "ACID", to the "C" in "CAP", not to mention to computer architects who believe in "sequential consistency". In this work, we use it to discuss the behavior of replicated data, especially in the context of consistency models such as *eventual consistency*.

in direct opposition to *availability* — this is known as the CAP Theorem [34, 67]. In the context of replicated datastores, this could mean waiting for several machines, potentially distributed over a wide geographic area, to coordinate and agree on the current state. This means that if a network partition or failure causes some messages to be delayed or lost, clients can see widely varying response times compared with weaker consistency that allows them to use whichever replica is closest or least-loaded.

The problem with consistency models is that they imply an ordering between operations that almost certainly does not reflect what a particular application needs. Sequential consistency or linearizability [76], which forces all operations to be totally ordered, imposes extreme restrictions on reorderings and precludes high availability. There are many models that allow applications to control consistency at a finer grain and even specify application-level properties. Some of these are discussed in detail in §5.2, in the context of disciplined inconsistency.

Another source of excessive ordering constraints can come about when there is insufficient information available to the system. Within an application, it may be obvious that the relative order of two actions is irrelevant, such as two people "liking" a post on Facebook. Yet if these actions are represented in the datastore as simple low-level `put` and `get` operations, then they will appear to conflict with one another and can lead to conflicts or consistency violations. Chapter 4 explores this issue in detail and proposes extensions to existing data storage systems that allow them to take advantage of the abstract semantics of operations.

1.2.4. Abstract data types

The systems stack involves many layers of abstractions, without which it would be nearly impossible to implement wide-area communication, durable storage despite hardware failures, or transparent scale-out based on demand. However, the inherent challenge with abstractions like this is the loss of information between layers. The simplest example is an application that stores its data in a key-value store using `put` and `get` operations. This interface is fully general, making it easy to swap in new storage systems, but this impoverished channel means that most of what the application knows about the structure of its data is lost in translation.

A core tenet of computer science, *abstract data types* (ADTs) hide the details of their concrete implementation, yet allow reasoning about abstract state and logical behavior, including algebraic properties such as commutativity and associativity. Developers are familiar with ADTs, constantly reasoning about the data structures needed to model application state and implement algorithms. Without burdening the developer, relevant properties about the semantics of operations can be passed on to other parts of the distributed system stack, allowing them to leverage that knowledge to improve performance.

For example, datastores with a richer interface, such as Redis, Riak, and Hyperdex [22, 58, 135], which support a wide variety of operations on common data structures such as lists and sets, provide much more information to the system. The concept of ADTs has long been used to extend databases: supporting indices and query planning for custom data types [148, 149], and concurrency control via abstract locks [14, 47, 75, 162]. Modern schema-less ("NoSQL") datastores deal with new challenges at scale and have evolved many database techniques to solve them, but they largely do not make use of the properties of their data to provide strong guarantees at scale. Chapter 4 introduces the Claret prototype datastore that demonstrates the promise of using ADT semantics to reduce conflicts between distributed transactions by recognizing which operations can safely execute concurrently.

In addition to reasoning about concurrency, semantics can also help reason about changes to state and the effects optimizations will have on observed values. Knowing, for example, the effect an `increment` operation has on a `Counter` can allow a storage system to make smarter decisions, such as allowing some increments to race with reads because the result will still be close enough. This is explored in more detail in Chapter 5.

1.3. Previously published material

This dissertation comprises work published elsewhere in conference papers and technical reports:

- Chapter 2: *Alembic: Automatic Locality Extraction via Migration*. Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2014. [79]
- Chapter 3: *Flat Combining Synchronized Global Data Structures.* Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. International Conference on PGAS Programming Models (PGAS), 2013. [78]
- Chapter 5: *Disciplined Inconsistency.* Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. University of Washington Technical Report. UW-CSE-16-06-01. [80]

2. Alembic: Automatic locality extraction via migration

2.1. Introduction

When targeting distributed systems, such as commodity clusters, application developers must deal with both parallelism and locality. Often these are at odds as placing more data on a single machine node improves locality but may decrease the ability to exploit parallelism across the entire system. This lack of separability forces programmers to reason about these two conflicting drivers of performance in tandem.

Partitioned Global Address Space (PGAS) [40, 42, 43] languages simplify the expression of parallel computations on large distributed systems. The programmer writes to a shared memory model, using *global pointers* which can reference memory anywhere in the system, and the runtime automatically handles the movement of data. While elegant, PGAS systems do not remove the fundamental conflict between parallelism and locality; in fact, they can easily lead to less efficient applications [49]. Compared with expressing all data movement manually, PGAS models may hide cases where the way the algorithm is expressed leads to excessive communication.

Thus even in a PGAS system, programmers wishing to exploit the last ounce of performance must manage locality themselves. The typical way to do this is to carefully layout data structures such that blocks of data accessed together are placed together and then when spawning new threads, explicitly place them where most of the data the thread will access is located. This is not an ideal solution for two reasons: (1) it can make otherwise-elegant PGAS implementations excessively complex with explicit *computation movement* – in effect, instead of explicitly moving data around with MPI invocations, the programmer is explicitly moving computation around using a variety of techniques (spawning new threads, continuations, etc); and (2) not all applications are amenable to easy partitions of computation and data – notably, irregular graph algorithms lack spatial locality, so placing the computation at any fixed location in the system guarantees several remote accesses and poor performance.

This chapter introduces Alembic, a compilation technique for PGAS systems that automatically extracts locality from programmer-created threads. Alembic statically analyzes code looking for sequences of memory references that go to the same machine. It then transforms a single programmer-written thread into a series of continuation threads, each spawned on the machine that hosts the majority of the data that continuation will access. Synchronization is added to ensure the sequential semantics the programmer has expressed are maintained, and necessary context state is packaged into messages that pass control between machines.

Alembic provides a substantial performance boost for PGAS code. In our evaluation, simple, elegant PGAS implementations of common graph algorithms such as breadth first search achieved only 13% of the performance of hand-optimized implementations where the programmer explicitly writes remote procedure calls. Alembic can transform these cleanly written algorithms into high performance locality-aware codes, achieving 82% of the performance of the hand-tuned implementation on average.

In summary, the novel contributions of this work are:

- An analysis to prove co-location of global memory accesses.
- An optimization system that identifies good candidates for code movement.
- An implementation based on LLVM of these techniques.
- An evaluation of the implementation in a PGAS environment on commodity cluster hardware.

Before delving into the details of the Alembic analysis, we first provide some context on PGAS programming models and the Grappa runtime on which this work is evaluated.

2.2. Background on distributed programming models

In general, the problem which Alembic addresses is an instance of *program partitioning*: the process of determining how to divide a program and its data over a number of machines in order to minimize the amount of communication required to move data and perform synchronization. The most primitive solution is embodied by the single-process-multiple-data (SPMD)[2] model used by many high-performance computing models, most notably the Message Passing Interface (MPI) standard. In this model, all communication is explicit and data is typically partitioned statically among processes all executing the same code. Over many years, a number of solutions have been proposed to simplify the process of building applications that span multiple machines by making them appear more like traditional shared memory programs.

2.2.1. Distributed Shared Memory

An early attempt at providing the illusion of shared memory, termed *distributed shared memory* (DSM), piggybacked on the virtual memory system of most processors. These systems [26, 41, 101] moved entire pages of data dynamically as requested by applications, so they required very regular access patterns and careful thought when determining which parts of the computation to execute where. Many DSM applications were attempted to be ported naively from their original shared memory implementations that were not designed to carefully avoid remote memory access.

2.2.2. PGAS Systems

Recognizing the huge performance cost of dynamically migrating pages that plagued DSM systems, Partitioned Global Address Space (PGAS) systems instead make the assumption that every piece of global memory

[2]SPMD is derived from Flynn's Taxonomy [63], an extension of the single-instruction-multiple-data (SIMD) concept but for multiprocessors, particularly distributed.

is owned by a particular entity which mediates all accesses to that piece of memory. This entity often corresponds to a physical locality domain, such as a node in a cluster. From this domain, any memory it owns can be accessed directly using simple loads and stores. Memory accesses to a different node are mediated by the host node where that memory is located. As in DSM systems, the distinction between local and remote memory is hidden from the programmer, typically via a *global pointer* abstraction. However, these languages — in particular, Unified Parallel C (UPC) [40], Chapel [42], and X10 [43] — provide more support to programmers to control data layout and placement of computation tasks.

Having a single node own each piece of memory makes it much simpler to maintain a single consistent view of shared data for programs. The PGAS runtime system ensures that program-level memory ordering is preserved through the various communication mechanisms. The PGAS model has been applied to a variety of system architectures, not just distributed-memory clusters, and as such often use different terminology. In this paper we adopt Chapel's *locale* [42] to refer to a particular set of computational and memory resources.

2.2.3. Grappa

Grappa is a PGAS-style programming model and runtime system designed for irregular applications. The primary factors that make an application *irregular* are unpredictable data-dependent access patterns and poor spatial and temporal locality. Examples of such applications include graph analytics on social networks, fraud detection, or meta-genomic analysis. To tackle these kinds of applications, the Grappa runtime, implemented as a C++11 library, uses massive parallelism to tolerate the latency of automatically aggregating communication. In Grappa, the programmer is expected to provide the runtime with many (potentially millions) of fine-grained tasks. The runtime then schedules these tasks on the available computational resources, overlapping the remote memory accesses from one thread with the productive execution of other threads. In Grappa, the unit of work being carried out is referred to as a *task.* The particular execution container (stack, context state, etc) that carries out the execution of a task is a *worker thread,* or just *worker*. We will use the term *thread* to refer to the more abstract notion of a sequential thread of execution.

In Grappa, a task is mostly executed by a single worker, but the runtime has also embraced a delegation-based execution model, similar in many ways to the CmPS model (described below), where arbitrary computations on remote data are shipped to where the data is in order to be executed. Delegate operations block the caller until they return their result in order to preserve a sequential thread of execution. In the existing system, delegate operations are specified explicitly and are the only way to access data on other locales. Composing delegate operations and choosing which code should be executed where becomes the dominating concern when writing and optimizing Grappa code, which this work attempts to automate.

2.2.4. Communication-Passing Style

As the name is intended to evoke, Communication-Passing Style (CmPS) [88] is an analog to continuation-passing style for distributed systems. The core idea is that rather than fetching data remotely, communication is done by sending a *continuation,* which contains everything necessary to resume execution, to the locale where the data resides. Transforming execution in this way preserves the same sequential execution expressed in the source program, but now, if there is more than one access to data on the same locale, no additional communication is necessary.

In this execution model, communication is still implicit; however, forcing migration on every access has downsides if a large amount of state must be carried over to continue execution. Therefore, the CmPS work formalized a notion they call *computation migration,* where most of the state is *frozen* and left behind, and the reduced continuation is sent, does its computation, and immediately returns to rejoin the rest of the state it left behind. CmPS programs explicitly mark when a frozen migration should be done.

CmPS uses a notion of *address spaces* associated with objects to reason about when migration is necessary, which includes ways to recognize when accesses to different objects refer to the same address space. We refer to this as *locality partitioning*.

The CmPS work established formal operational semantics for a functional language with distributed memory, forming the basis by which we reason that our own continuation-passing transformations are sound. Our Alembic transformation essentially applies the CmPS technique to an imperative, object-oriented context – our variant of C++ with PGAS extensions. Additionally, we design analyses to statically choose when to migrate to minimize communication.

2.3. Language extensions for locality

We start by introducing some concrete syntax and semantics to define the context for the rest of the techniques in this work and establish some common terminology. The aim of our particular implementation of the PGAS model is to stay within the confines of plain C++ as much as possible, both for ease of adoption as well as ease of implementation, so our extensions are confined to *attributes* that express where operations can execute. The syntax and semantics should not be particularly surprising to anyone familiar with PGAS languages, and the techniques we apply should be generalizable to other PGAS environments.

In order to interoperate with the existing Grappa runtime, which is a plain C++11 library, each of the constructs below maps to a C++ class and can be coerced between its "library" and "language" forms. This means that any part of the application can be written without relying on special compiler support, and just the region where the new syntax is used will be manipulated by the passes described in this paper.

2.3.1. Global Pointers

A fundamental primitive of PGAS languages is the *global pointer*, which encodes the locale where it is valid in addition to the address in the locale's memory. Like normal pointers, global pointers may refer to data on task stacks, static memory, or the heap. In Alembic, global pointers are expressed using a new `global` modifier on pointer types, e.g.: `int global* x`.[3]

[3]The syntax of pointer modifiers in C/C++ is undeniably confusing. Just as `int const*` indicates that the pointer cannot modify the `int` it points to, so

We encode the `global` attribute as a custom *address space*, part of the Embedded C extensions [86], which gets propagated into the compiler's intermediate code. We refer to pointers without any modifier as *local* pointers, signifying that they do not encode a particular locale, but are only guaranteed to be valid where they were generated.

Because global pointers are only valid on one particular locale, a dereference of one implies the chance of communication, since the actual load or store must be executed on the locale indicated by the global pointer. The PGAS language is responsible for ensuring this, typically by turning each global load or store into a `put` or `get` operation supplied by the runtime.

Global pointers are *deeply* global: pointers computed as offsets from a global pointer, via member accesses or array indexing, are also global. The locale of the resulting pointer, however, is not necessarily the same as the original pointer; it depends on the operation and the type of the object pointed to. These rules will be discussed in more detail in §2.4.1.

Method calls through global pointers are allowed. Because the receiver is now global, any references to the objects' fields must also be associated with the same locale. And local pointers used or returned by the method are only valid where that object resides, so they must also be made global. The details of this transformation will be covered in §2.4.4.

Global pointers can be explicitly constructed from a local pointer and a locale, or may come from allocating out of some global heap which is distributed in some fashion over the locales in the system. In both cases, the pointer must carry the information about how the object it refers to is distributed so that operations on the pointer, such as indexing off of it, can be resolved correctly. PGAS languages often provide a variety of choices for how to distribute arrays, such as Chapel's domain distributions. In Grappa, we have a simple block-cyclic heap with a fixed block size. Objects allocated from the heap must be aligned to the block size so they are not split between multiple locales. Elements of arrays allocated from the heap are distributed round-robin among locales.

`int global*` indicates that the object it points to may be remote. As with `const`, `global int*` would also be correct, but we prefer the first version.

2.3.2. Symmetric Pointers

Globally distributed data structures are an important part of PGAS environments. For instance, it can be useful to have a hash table that tasks on all locales can operate on and see a consistent view. These distributed objects can be implemented in various ways. In Grappa, we implement them using a handle, or *proxy,* to the global object on every locale. Methods called on these proxies from any locale observe the state of one globally distributed object. Internally, the implementation of these methods coordinates among all the other proxy objects and any additional global state to provide this illusion, allowing optimizations, such as buffered updates, to be hidden from the user behind this level of abstraction. These uses are discussed in more detail in Chapter 3.

In order for our language to handle these objects correctly, we introduce a notion of *symmetric* objects, referred to by symmetric pointers, which have a copy on every locale. Distinct from global pointers, which are valid on one locale only, a symmetric pointer has a valid address on *all* locales. In order to refer to one of these globally-distributed objects, all one needs is a symmetric pointer to its proxies. Methods called through these symmetric pointers go to whichever copy is on the current locale, which then takes care of maintaining the illusion of one distributed object. One additional constraint is that methods called using symmetric pointers must be executed entirely on one copy of the object – if the method is inlined, for example, we must ensure all the references to the symmetric pointer resolve to the same locale. This ensures that any state maintained internally in each proxy is kept in a consistent state.

Symmetric pointers can be obtained by using a special allocation from the global heap that ensures that all the copies are at the same offset. By obtaining an allocation in this way, the programmer is asserting that their object has symmetric semantics. Variables in the C++ global scope, because of the SPMD nature of the runtime, have the same static offset on every locale, so they may also be treated as distributed objects if they are explicitly annotated as `symmetric`.

2.3.3. "Anywhere" function annotation

As with unannotated pointers, by default, functions must be assumed to be local, so cannot be moved in a migration. The `anywhere` annotation applied to a function implies that it can safely be run from any locale. This is useful for functions that will take care of inter-locale communication themselves, similar to how symmetric objects work. Furthermore, this annotation is applied to functions whose semantics allow flexibility in where they execute, such as print statements and assertions, or runtime calls such as `spawn`. % In many cases, the compiler could determine by inspecting the function that the objects it references are symmetric and allow it to be treated as such; however, this annotation can be used to assert this up front, regardless of what the compiler is able to glean.

2.3.4. Tasking and synchronization

In this work, we use the tasking and synchronization provided by Grappa unchanged. We introduce some constructs here so that code examples throughout will make sense. As in many parallel frameworks, we express parallelism in the form of *tasks.* A task represents a small amount of sequential work to be run asynchronously some time after it is *spawned.* These short-lived, lightweight parallel threads of execution go by many names, such as *fibers, green threads,* or simply *asyncs.*

Tasks are expressed by passing a C++11 lambda to `spawn`; their initial state is made up of captured variables. In general, tasks may run asynchronously any time after they are spawned and must be explicitly synchronized to ensure they finish. This can be done via ad-hoc synchronization or more structured constructs. For instance, tasks spawned by parallel loops, described below, are typically synchronized using a phaser, which we describe next for reference.

2.3.4.1. Phased synchronization

A *phaser* [143] is a flexible, reusable global barrier where the number of registered events may be unknown at the start. This is particularly useful for phased rounds of computation where a large amount of parallel work will be recursively spawned, for instance while traversing a graph. Tasks may `enroll` with the phaser before starting and call `complete` when finished, while other tasks can block until the phase is done by calling `wait` on it. Phasers are implemented as symmetric objects in our system, so the same phaser is accessible from all locales.

2.3.4.2. Parallel loops

Parallel loops (we borrow the name `forall` used in UPC and Chapel [40, 42]) conceptually spawn a separate asynchronous task per iteration. Our parallel loops use a phaser to synchronize all spawned tasks and any additional asynchronous operations that should be completed before the loop is terminated. The non-blocking version, `forall<async>`, can be nested inside other loops and typically uses the phaser of the outermost loop to ensure all iterations complete.

2.3.5. Example: HOPS

To motivate this work, we use a simple benchmark based on the HPCC random-access benchmark GUPS [82]. In GUPS, an array of random numbers, `B`, is used to index into another array, `A`, and atomically modify the element there. There are more elements in `B` than `A`, so most elements will be visited multiple times. The modified benchmark, which we call *HOPS,* additionally tracks which element from `B` first reaches a given element in `A`. This operation is meant to be representative of work done when visiting objects in irregular applications, and should look familiar to those who know the parent-claiming step of the Graph500 BFS benchmark [69]. In addition, we disregard the distribution of the `B` array when initially placing tasks in order to better demonstrate an opportunity to *hop* directly from one locale to another when migrating. Two implementations of HOPS are shown above in Listing 2.1, one using the extended C++ syntax, the other explicit communication, with the two migrated regions highlighted in each.

```
struct Counter { long count, winner; };

symmetric Phaser phaser;

void hops(Counter global* A,
          long global* B, size_t N) {
  forall<&phaser>(0, N, [=](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_and_add(&a->count, 1);
    if (prev == 0) a->winner = i;
  });
}

void hops(GlobalAddress<Counter> A,
          GlobalAddress<long> B, size_t N) {
  forall<&phaser>(0, N, [=](long i){
    Locale origin = here();
    phaser.enroll(1);
    delegate<async>(B+i, [=](long& b){
      delegate<async>(A+b, [=](Counter& a){
        long prev = fetch_and_add(&a.count, 1);
        if (prev == 0) a.winner = i;
        phaser.complete(origin, 1);
      });
    });
  });
}
```

Listing 2.1. *Managing nested delegates and synchronizing them is significantly more tedious and error-prone.* This listing shows code for the HOPS benchmark, a variant on GUPS that tracks which index from B incremented an element of A first. The top version uses the extended syntax and relies on compiler-generated communication; the bottom does explicit movement and synchronization. The first highlighted region (green, dashed border), indicates the first migration, to `B[i]`, the immediately-following region (purple, dotted border) indicates the second hop.

2.4. Alembic analysis

Since memory regions are owned by locales, we can think of accesses to that memory as points in the execution that are *anchored* to a particular locale (i.e., a `load` from a global pointer must occur on the locale it points to). These *anchor points* are constraints on the execution of the task, with the start of the task anchored wherever the runtime invokes it. Rather than thinking of remote accesses as necessary communication points, we can instead think of them merely as constraining execution of that part of the task to a particular locale. Many instructions are not anchored, meaning that we could choose to execute them at either location.

Tasks can be thought of as being divided into regions based on locality. At each transition between regions, a continuation is constructed and sent to where the next region is to be executed. These migrations may either be *blocking,* in which case control returns to the home locale immediately after, or *chained,* hopping from one locale directly to the next. Though executed on different locales, these regions still represent a single sequential task.

Considering task execution in this way enables many useful optimizations. Regions that include more than one anchor point can save on round-trips. Values produced and consumed on the same locale need not be communicated. Finally, when a continuation constitutes the remainder of the task, the migration can be *asynchronous,* immediately freeing up the worker executing it. We do not consider opportunities to further parallelize tasks, counting on the programmer to express the concurrency they desire with explicit task spawns.

The goal of our analysis is to choose how to divide tasks into locality regions and transform them into a series of continuation-passing migrations that minimize communication cost. Our analysis operates at the level of standard compiler optimizations, specifically, on LLVM's intermediate representation (IR) [98]. First, *locality partitioning* divides anchor points into sets proven to be on the same locale. Next, *region selection* enumerates and evaluates possible regions. Finally, a transform pass extracts the regions, computes continuations, and inserts runtime calls to do the migration. The following sections describe the steps in more detail.

2.4.1. Locality Partitioning Algorithm

Anchors are instructions that access memory, which restricts them to execute where that memory is. For most anchor points, the region of mem-

Expression	Locality	Operation
Local pointer:		
`p`	`here()`	identity
`p[4]`	`locale(p)`	array index
`p->f`	`locale(p)`	field offset
`p->adj()`	`locale(p)`	local pointer
`p->adj()+9`	`locale(p)`	local pointer index
`new T[4]`	`here()`	allocation
Global pointer:		
`g`	`locale(g)`	identity
`g[4]`	`unknown`	array index
`g->f`	`locale(g)`	field offset
`g->adj()`	`locale(g)`	local pointer
`g->adj()+9`	`locale(g)`	local pointer index
`make_global(p,3)`	`3`	constructor
`global_alloc<T>(4)`	`unknown`	allocation

Table 2.1. Locality of various pointer operations. In these examples, assume `T` is aligned to the block size, and the method `adj()` returns a local pointer.

ory is demarcated by a pointer and size. While the precise locale of the pointer will almost never be known statically, it is often possible to prove that two anchor points' locales are the same. The goal of *locality partitioning* is to find as many of these co-located anchors as possible.

We take an approach similar to *value partitioning* [7, 36] to divide anchor points into different locality sets. Value partitioning is a variant of *value numbering* which tries to divide value-producing instructions into *congruence classes* (or *sets*) for the purpose of eliminating redundant computations. Congruence is a recursive property, so in order for two values to be congruent, their respective operands must be in the same congruence sets. Value partitioning can be approached either from an *optimistic* perspective, where values are considered congruent until proven incongruent, or *pessimistic,* where values begin in their own sets and are merged when proven congruent. Both approaches are conservative.

Locality partitioning differs primarily in the definition of *congruence.* Rather than finding when operations compute the same value, we are concerned with finding when pointer values are guaranteed to be on the same locale. Table 2.1 shows a number of operations on pointers and the information available about their relative locality. For example, field offsets in block-size aligned objects are guaranteed to be on the same locale as the global pointer.

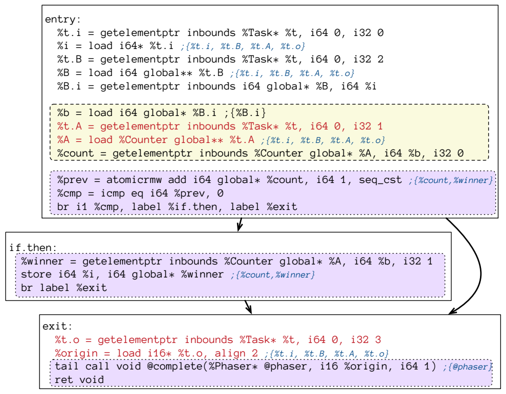
The locality rules supported by our C++ PGAS language use only local reasoning. One could imagine extending this in languages with more rich global locality information to prove co-locality in more situations. For example in Chapel [42], *domain distribution* information could be used to prove that elements with the same index in arrays with the same distribution have the same locale. To support such features, additional locality rules would simply need to be added.

The list of locality rules need not be exhaustive – any operation not covered will be conservatively placed in a new locality set. Some instructions may have no information available about the region of memory they access, such as an opaque function call. These must remain on the home locale to ensure that they are executed in the context they expect; the programming model ensures that they handle any necessary communication themselves.

The current implementation takes a pessimistic partitioning approach, initially placing all anchor points in distinct locality sets and merging sets when it proves they are on the same locale. This limits our analysis in the same way as for value partitioning: we must rely on visiting anchor points in a topological order, and therefore cannot use loop-carried information to prove co-locality. A future implementation could use the optimistic value partitioning approach if this was shown to be too limiting.

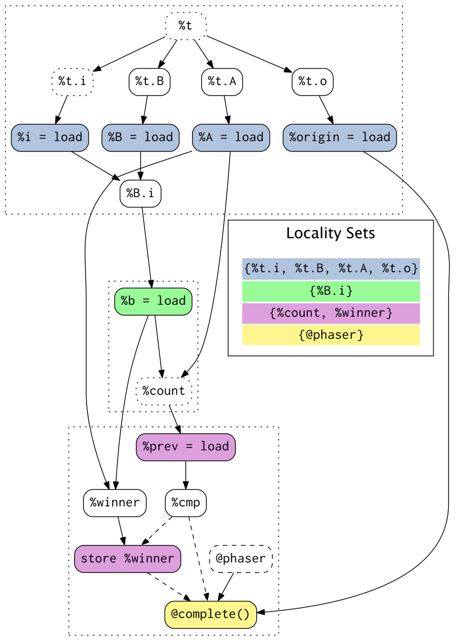
2.4.2. Region Selection

Once the anchor points have been classified, the next task is to choose where to execute the remaining unconstrained instructions. The goal is



```
entry:
  %t.i = getelementptr inbounds %Task* %t, i64 0, i32 0
  %i = load i64* %t.i ;{%t.i, %t.B, %t.A, %t.o}
  %t.B = getelementptr inbounds %Task* %t, i64 0, i32 2
  %B = load i64 global** %t.B ;{%t.i, %t.B, %t.A, %t.o}
  %B.i = getelementptr inbounds i64 global* %B, i64 %i

  %b = load i64 global* %B.i ;{%B.i}
  %t.A = getelementptr inbounds %Task* %t, i64 0, i32 1
  %A = load %Counter global** %t.A ;{%t.i, %t.B, %t.A, %t.o}
  %count = getelementptr inbounds %Counter global* %A, i64 %b, i32 0

  %prev = atomicrmw add i64 global* %count, i64 1, seq_cst ;{%count,%winner}
  %cmp = icmp eq i64 %prev, 0
  br i1 %cmp, label %if.then, label %exit

if.then:
  %winner = getelementptr inbounds %Counter global* %A, i64 %b, i32 1
  store i64 %i, i64 global* %winner ;{%count,%winner}
  br label %exit

exit:
  %t.o = getelementptr inbounds %Task* %t, i64 0, i32 3
  %origin = load i16* %t.o, align 2 ;{%t.i, %t.B, %t.A, %t.o}
  tail call void @complete(%Phaser* @phaser, i16 %origin, i64 1) ;{@phaser}
  ret void
```

(a) HOPS Iteration task in LLVM IR.



(b) Dependence graph for HOPS iteration.

Figure 2.1. Breakdown of the task executing a single iteration of the HOPS loop. In (a) we show the task's instructions, annotated with their locality set (in braces), and divided into regions. At the first horizontal line, the task migrates to {B.i} (in green). At the second line, execution migrates again for the atomic increment until the end of the task (purple). Bold, non-highlighted instructions are those that must be hoisted into the first (home) region. The corresponding value dependence graph is shown in (b) with nodes for instructions labeled with the value they produce. Here boxes are drawn around regions – arrows that cross these boundaries indicate values that will go into continuations.

to come up with a sequence of migrations, constrained by anchor points, that will result in the minimum amount of communication. Recall from §2.2.4 that the continuation must include everything needed to resume execution; the communication cost is the size of this continuation. In some situations it is preferable to leave some state on the original task's stack, migrate a smaller continuation, and return to pick up the rest (in CmPS this was a *freeze* operation).

This analysis divides the instructions in each *task* into *regions* by locality. All the anchors in a region are proven to be to the same locale. Non-anchor instructions, including symmetric pointer accesses and `anywhere` function calls, are placed in one region or another to minimize communication. Though it would likely lead to some improvement in communication, to simplify the problem, this pass does not consider duplicating instructions in more than one region, nor splitting the thread to expose additional parallelism. This means that regions do not overlap; the task is still a single thread of execution whose control and data jumps around the system. A more ambitious transformation which does allow for these is left for future work.

Before diving into the details of the algorithm, let us revisit the HOPS code in Listing 2.1, which will be used throughout this text to explain the mechanics of Alembic. A parallel loop from 0–N creates tasks for each iteration. Each task gets the random value stored at `B[i]`, a global access, and uses that to index into `A`, likely referring to another locale, on which it performs an atomic increment. The LLVM IR corresponding to this task, on which our analyses operate, is shown in Figure 2.1(a). Anchors are annotated with their locality sets and two distinct migrated regions are shown highlighted. The un-highlighted instructions in these regions must be hoisted to make the regions contiguous.

Anchor points are annotated with their locality set in blue. The locality regions we would like to infer are highlighted: the first migrated region, after the horizontal rule, to be executed at `B[i]`, and purple for the region at the element in `A`. However, instructions highlighted in red, which are anchored where the task started, currently prevent these regions from being contiguous.

Choosing the optimal migration policy is intractable: it would at the very least require full-program analysis, but would also depend on the layout of data, runtime load balancing, physical interconnect topology, and many other concerns. The hypothesis of this work is that automated decisions at the scope of a task, with the constraints provided by anchor points, are sufficient to compete with communication explicitly provided by the programmer. Even with the above restriction that instructions only appear once, instructions can still be reordered and because we do not know the optimum number of migrations, the problem reduces to finding a minimum k-cut (where *k* is the number of migrations) on the task's dependence graph, which is known to be NP-complete [65].

Instead, we implement a much simpler greedy algorithm that evaluates a restricted set of candidate regions with a simple cost heuristic. Rather than evaluating all possible reorderings, we determine independent regions for each anchor (migrating back home after each), reorder anchors only with the home region, and attempt to combine adjacent regions pairwise in a greedy fashion. Steps of the algorithm will be explained in the coming sections, but at a high level, it works as follows:

- For each anchor, expand a region, starting from the anchor, to its maximum allowed extent.
  - When encountering other anchors, determine if they: (i) share the same *locality set,* and can be included in the region, (ii) have a *symmetric* locality and can be included, (iii) can be *hoisted* to the home region, or (iv) represent a necessary end to the region.
  - At each step, find the inputs and outputs to the region and compute the *cost heuristic* (described below) for the current region, as if any *hoistable* instructions were moved before the region.
  - Keep track of the best sub-region.
- Skip anchors that have already been completely subsumed within another anchor's best region.
- For each pair of adjacent regions (those whose maximum extents overlap or are adjacent):
  - Compute the continuations needed to migrate directly between the two.
  - If the cost combined is less than the cost of the two separate migrations, then replace the two separate regions with a new *chained* region containing both.
- Mark regions whose exits are the end of the task as *async.*

The next few sections explore this algorithm in more detail.

2.4.2.1. Expanding the region

To find regions of code that can be executed at the location of a given anchor, we start from the anchor instruction and iteratively expand the region to include instructions that are valid to run on that locale. Any instructions proven to not touch memory are trivially allowed. Thanks to the previous analysis, any memory-access instructions, including calls to functions that may access memory, will be associated with a locality set. Any instructions in the same locality set as the current anchor are allowed. Symmetric pointers, explained in §2.3.2, are valid on any locale, so any symmetric anchors can also be included. For other accesses, we will attempt to hoist or localize them, which will be explained next. After determining that an instruction is valid, the cost function, explained below, is computed for the current region, and the minimum cost region is tracked.

Though regions may have multiple exits, in order for the continuation-passing transformation to work, they must have only a single entrance. To ensure this, basic blocks reached by the expanding region are visited in reverse postorder, and basic blocks with incoming edges not already in the region are disallowed and become exit points. This over-conservatively disallows loops from being subsumed within a region, which is a potential pitfall that could be remedied with further engineering.

2.4.2.2. Cost heuristic

The cost function attempts to encode the combination of communication and execution costs inherent in migrating the given region. In the coarsest view of the runtime system, the total amount of data moved is worth minimizing, but the execution overhead – time spent aggregating, sending, deaggregating, blocking and waking threads – is roughly *per application-level message*. These are aggregated into larger messages by the runtime, but overhead is associated with each independent task that issues a remote request. Therefore, our cost function has to take into account number of messages in addition to the amount of data moved.

Inputs and outputs to each region are computed from LLVM IR, which is in static single assignment (SSA) form by design, and used to compute the size of the continuation, or the total amount of data that needs to be moved, in each migration. This can be viewed as partitioning the program dependence graph, similar to how it is done in decoupled software pipelining [128], but attempting to minimize data crossing the partitions rather than exposing parallelism. Figure 2.1(b) shows the dependence graph for HOPS, with a node for each instruction labeled by the value it produces, and arrows showing uses of those values. Arrows that cross a region's bounding box represent values that must go into a continuation. Grappa's communication mechanisms currently only support POD types, allowing Alembic to statically determine the precise amount of data to be moved. More dynamic object-oriented features, such as subtype polymorphism or serialization of arbitrary additional data, would make this cost estimate more difficult.

Grouping two anchors with the same locale into one region eliminates a round-trip message. This is modeled in the cost heuristic by subtracting the cost of those messages for each anchor. If all exits from a region return `void`, this means it is the final region in the task and the return trip to get back to the task's home locale is unnecessary, saving an additional message, which we also model. The resulting heuristic equation is:

$$\begin{aligned} cost = \; & \texttt{sizeof}(inputs + outputs) \\ & - 2 * messageCost * numAnchorsIncluded \\ & - (messageCost, \texttt{ if } allExitsVoid) \end{aligned}$$

In §2.5.3, we evaluate this tradeoff empirically to come up with a reasonable setting for *messageCost* for our experimental platform.

2.4.2.3. Hoisting anchors

We saw earlier, in Figure 2.1(a), that sometimes the order in which memory accesses are scheduled is not ideal for migrating because the instruction scheduler is assuming a different cost model for memory accesses than what we have in mind. For example, the load of `%origin` in the exit block prevents what would otherwise be an asynchronous migration. It is a clear win in this case to hoist the load before both regions because it only costs the data movement of 2 additional bytes but saves in total messages sent by allowing an asynchronous migration. In the general case, one would need to explore every allowable reordering of anchor points to find the one that minimizes messages and continuation size. In our simplified search, we only attempt to move instructions into the first region (at the home locale), which is a clear case where reordering will be beneficial. Anytime an anchor with a different locality set is reached, we check whether it can be placed in the first region without violating dependences or locality. We do not consider opportunities to move the access into other regions, as it would greatly increase the complexity and search space, and we found it typically did not pay off in the situations we encountered.

We use LLVM's memory dependence analyses to determine if the memory operation clobbers or is clobbered by any instructions in the region or violates synchronization ordering. Additionally, to be hoisted, stores must not be conditional (must dominate all exits from the region). Typically this move has already been done by previous passes if it is possible. Finally, we must determine if, recursively, all of the operands that reside in the region can be hoisted.

If all of these criteria are met, then the operation can be marked as *hoistable.* When computing migration cost, candidate regions treat them as if they had been moved, but they are not actually moved unless the minimum-cost selection includes it. Hoisting instructions is done independent of prior region selections. There is a slight chance that this hoisting could have made prior migrations happen if they had known, but this is a performance, not a correctness, issue.

In the running example, hoisting both of the loads in HOPS leaves us with two contiguous regions back-to-back, allowing us to migrate directly between them. The phaser is symmetric, so calling `complete` on it can be done anywhere, so the migration can be asynchronous.

2.4.2.4. Localizing allocas

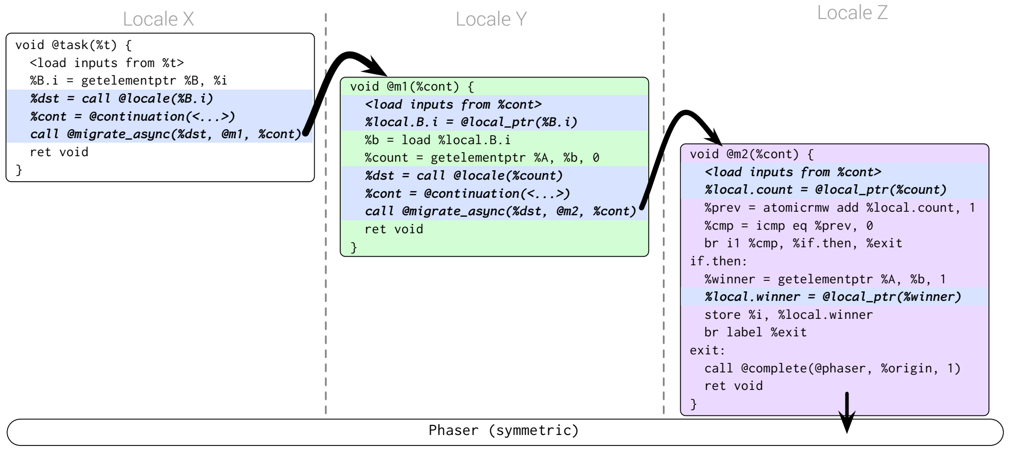
Rather than hoisting loads and stores before the region, in some cases it can be possible to instead change what memory they are referring to. In particular, temporary objects are typically allocated on the stack on entry to the function and used later. If we can prove that a piece of stack-allocated memory is only used inside a single region, then we can *localize* that temporary storage and put it in the migrated region, so that the loads and stores using it can be done locally after migration. To determine if this is the case, we examine all accesses to the region of memory specified by an `alloca` instruction, including double-checking with the alias analysis to ensure nothing else may be using that memory. If all of the accesses are resolvable, and they all occur in one region, then we can move the `alloca` inside the region.

This check can only be done after the region has been expanded to its maximum extent (see below). Our analysis speculatively allows the region to include accesses to stack-allocated memory, expands as far as possible, then does this check. If any allocated regions are not localizable, we mark them and redo the region expansion, this time not including those accesses. This may iterate more times, but each iteration will remove at least one speculatively-included anchor, so it will terminate quickly.

2.4.2.5. Chaining regions

After finding the maximum extents of all single-locale regions, we start evaluating how to stitch these regions together to form a single migrating thread of execution. We could simply migrate back to the home locale of the task after each region, and we would still benefit from moving multiple anchors on a single locale. However, if two regions to different locales are adjacent, additional benefit could come from hopping directly between the two. Migrating directly saves costly messages and wake-ups but may increase the size of the continuation.

To evaluate whether continuing directly to the next region will be beneficial, we use the same cost heuristic. We compute the continuation needed to execute the combined region, the continuation from the first region to the second, and the outputs of the combined region. If this combined cost is less than the sum of the individual region costs, which amounts to whether the continuation between the two regions is smaller than the cost of an additional message, then the two regions are chained. As we continue to consider adjacent pairs, longer chains of linked regions may be generated, resulting in a task that will seem to hop around between locales, following where its data is.



```
void @task(%t) {
  <load inputs from %t>
  %B.i = getelementptr %B, %i
  %dst = call @locale(%B.i)
  %cont = @continuation(<...>)
  call @migrate_async(%dst, @m1, %cont)
  ret void
}
```

```
void @m1(%cont) {
  <load inputs from %cont>
  %local.B.i = @local_ptr(%B.i)
  %b = load %local.B.i
  %count = getelementptr %A, %b, 0
  %dst = call @locale(%count)
  %cont = @continuation(<...>)
  call @migrate_async(%dst, @m2, %cont)
  ret void
}
```

```
void @m2(%cont) {
  <load inputs from %cont>
  %local.count = @local_ptr(%count)
  %prev = atomicrmw add %local.count, 1
  %cmp = icmp eq %prev, 0
  br i1 %cmp, %if.then, %exit
if.then:
  %winner = getelementptr %A, %b, 1
  %local.winner = @local_ptr(%winner)
  store %i, %local.winner
  br label %exit
exit:
  call @complete(@phaser, %origin, 1)
  ret void
}
```

Figure 2.2. Alembic transformation of the HOPS task doing multi-hop migration (much-simplified LLVM IR with types elided). Code added to do the transformation (bold and highlighted blue) includes: for each migration, construct a continuation and find the destination locale, and in each migrated region, extract local pointers from global pointers.

2.4.3. Transforming tasks

This section will explain at a high level how the original task is transformed according to the choices made by the analyses above, at the level of LLVM IR. Migration is done by extracting all of the instructions in the region into a separate function, sending the continuation in a message to the remote locale, which, on receipt, invokes the extracted function, and the output values from the region, needed for the next continuation, are collected. If the next region is to be executed back on the original task, these outputs are sent back to rejoin the rest of the stack. Otherwise, if the next hop is directly to another locale, then the continuation is constructed, and another migration is done.

All of the data movement is handled by a generic `migrate` call in the runtime. This function takes as input the destination locale, the function to run, a struct for the continuation, and a pointer to a struct for storing the outputs. The calling task blocks until the sequence of migrations returns to rejoin the stack. In the case where a migrated region includes the end of the task, the variant `migrate_async` is used, which immediately frees the worker to start another task while the migrated continuation finishes the previous task's execution remotely.

Extraction is done using a modified version of the LLVM `CodeExtractor` utility. All the basic blocks of the region to be extracted are cloned into a new function. All of the exits are redirected to a single `return` block which returns the output of a `phi` to differentiate which exit was taken. At the call site, this return value is used in a `switch` to jump to the correct exit. Before the call, the continuation is constructed on the task's stack, and after the call the outputs are loaded from the other struct passed to `migrate`.

Figure 2.2 shows how the HOPS code ends up being transformed. The initial task constructs a continuation with the values needed for both migrations, and computes the destination locale. Inside each migrated region, we load the inputs from the continuation. Finally, in each region, we extract and use the local pointer from global pointers which are now local. Because the two migrations make up the rest of the task, `migrate_async` can be used, which allows the initial task to return immediately, though the enclosing parallel loop waits for the final migrated region to signal `complete`.

2.4.4. Globalizing functions

As mentioned back in §2.3.1, methods can be called on objects via global pointers. However, this is not expressible in C++. We allow the C++ frontend to generate these method calls anyway and fix them ourselves.

To handle method calls on global pointers correctly, they must be made parametric on the pointer type of the receiver. This means constructing a new version of the method where the receiver is a global pointer instead. To do this, we clone the function, then propagate the changed pointer type through all the instructions, which may cause other pointer values to become global. Any local pointers referenced inside the method are wrapped up in a new global pointer with the locale of the receiver pointer, including the return value if the method returns a local pointer. Finally, we replace all calls where the receiver was cast from a global pointer with a call to the new *globalized* version.

In fact, because methods are really just functions, we apply this same transformation on any functions that accept a local pointer but are passed a global pointer instead.

2.4.5. Put/get generation

For comparison, we also implement a version of our compiler that generates just `put` and `get` operations. This is a fairly standard baseline for PGAS languages without any optimizations enabled. Each global memory access is replaced with a call to a corresponding remote operation in the API. After fixing up function calls with global pointer parameters (as described in §2.4.4), all of the global memory accesses are clearly delineated in the LLVM IR. We then simply find all instances of `load`, `store`, `cmpxchg`, and `atomicrmw` which have a global pointer operand and replace them with calls to the underlying PGAS library (in our case, `grappa_get`, `grappa_put`, `grappa_compare_and_swap`, etc). To maintain the sequential semantics implied by the original memory operations, these operations all block the calling task.

Beyond the generic memory access optimizations applied by LLVM, our compiler generates fairly naive puts and gets compared to optimized communication generated by other PGAS systems (see §2.6.3). However, the Grappa runtime dynamically aggregates messages from multiple tasks and tolerates remote access latency using massive multithreading which give much of the performance benefit of those other techniques but with some runtime cost.

2.5. Evaluation

Our goal in this evaluation is to quantify the extent to which these static migration analyses and transformations are able to match the performance of hand-tuned locality optimizations. First, we evaluate the performance of Alembic on 4 irregular application kernels. Then we probe more deeply into the effect of each optimization using the HOPS case study. Finally, we explore the tradeoff between asynchronous and blocking migrations in order to empirically choose a value for *messageCost.*

2.5.1. Application performance

The purpose of Alembic is to be able to automatically generate task migrations that are onerous to do by hand. We evaluate the analyses on 4 representative irregular application kernels which were implemented and optimized in previous work evaluating the Grappa runtime [115]. The existing implementations have explicit delegate calls to do communication and move parts of the computation to different locales. These delegates calls were tuned by hand to get the best performance out of the Grappa system, including changes to make them asynchronous, reduce the number of messages, and minimize data transferred.

In each application, we ported the most performance-critical sections, removing all explicit communication and instead using the C++ extensions described in §2.3 (e.g., `global*`). These sections now rely on Alembic to automatically generate communication for them. In the following sections we will briefly describe the applications, the sections ported, and the regions identified by Alembic.

2.5.1.1. BFS

Breadth-first-search is a common kernel used to evaluate irregular application scaling, and is the primary benchmark for the Graph500 rankings [69]. The benchmark does a search starting from a random vertex in a synthetic graph and constructs a tree out of parent vertices for each vertex traversed. We port the entire timed region; a snippet which does a single level of the traversal is shown in Listing 2.2.

Alembic determines that the atomic compare-and-swap and everything after it can be in an asynchronous migration. This includes pushing the vertex onto the next frontier, which can be moved because `GlobalQueue` is symmetric and safely handles push operations from any locale.

2.5.1.2. Connected Components

Another core graph analysis kernel is Connected Components (CC). We implement the three-phase CC algorithm [28] designed for the massively-parallel MTA-2 machine. The first phase does multiple recursive traversals in parallel, each labeling vertices with a color. Whenever two traver-

```
symmetric GlobalQueue frontier, next;

void bfs_level(Graph symmetric* g) {
  Vertex global* vs = g->vertices();
  while ( !frontier.empty() ) {
    VertexID i = frontier.pop();
    forall<async,&phaser>(adj(g,vs+i),[=](VertexID j){
      if (cmp_swap(&vs[j]->parent, -1, i))
        next.push(j);
    });
  }
  phaser.wait();
}
```

Listing 2.2. Code from BFS which does a single level of the traversal. Alembic identifies and transforms the highlighted region into an asynchronous migration.

sals encounter each other, an edge between the two colors is inserted in a global set. The second phase performs the classical Shiloach-Vishkin parallel algorithm [142] on the reduced graph formed by the edge set from the first phase, and the final phase propagates the component labels back out to the graph.

We port the first phase, which does the traversals and insertion into the hash set and takes the majority of execution time; a snippet is shown in Listing 2.3. Most of the iteration is able to be subsumed in a single asynchronous migration because the stack-allocated lambda which is passed to `spawn` is able to be localized, the set is symmetric, and `spawn` and `complete` are annotated with *anywhere*.

2.5.1.3. Pagerank

This kernel is a common centrality metric for graphs which iteratively computes the weighted sum of neighbors until convergence. The computation essentially amounts to a sparse matrix dense vector multiply for each iteration, which in our implementation is parallelized over vertices in the graph as well as over the adjacencies for each vertex. We report

```
GlobalHashSet symmetric∗ set;
Graph symmetric∗ g;

void explore(VertexID r, color_t color) {
  Vertex global∗ vs = g->vertices();
  phaser.enroll(vs[r].nadj)
  forall<async>(adj(g,vs+r), [=](VertexID j){
    auto& v = vs[j];
    if (cmp_swap(&v.color, -1, color)){
      spawn([=]{ explore(j, color); });
    } else if (v.color != color) {
      Edge edge(color, v.color);
      set->insert(edge);
      phaser.complete(1);
    }
  });
  phaser.complete(1);
}
```

Listing 2.3. The first phase of Connected Components where we assign colors and insert an edge into the set whenever two traversals conflict. Alembic detects 2 migrations, highlighted above. The second region is only able to be asynchronous because the alloca for `spawn` could be localized.

```
void spmv(Graph symmetric∗ g, double global∗ X,
          double global∗ Y) {
  forall(g, [vx,vy](VertexID i, Vertex& v) {
    forall<async>(adj(g,v), [=,&v]
          (int64_t localj, VertexID j){
      Y[i] += X[j] ∗ v->weights[localj];
    });
  });
}
```

Listing 2.4. Ported code from Pagerank. The index into weights is local, so just two chained migrations are needed to visit the element in `X` and then update the element in `Y`.

performance as throughput, comparable to Graph500's TEPS measure, computed as the number of edges in the graph over the average time per iteration.

We port just this multiply section, shown in Listing 2.4, which makes up nearly all of the communication and execution time. This kernel is able to benefit from doing two continuation-passing migrations back-to-back to go from the original spawned task which is executed at the source vertex where the edge weight is, to the corresponding element in the source vector, and finally to the element in the resulting vector. That multi-hop migration can all be done with asynchronous migrations, eliminating the need for any blocking calls (except of course the main task which blocks on the phaser used to synchronize all this work).

2.5.1.4. IntSort

This benchmark comes from the NAS Parallel Benchmark Suite [16, 119]. The second-largest problem size, class D, ranks 0.5 billion random integers sampled from a gaussian distribution using a bucket sort algorithm. The performance metrics for NAS Parallel Benchmarks, including IntSort, are "millions of operations per second" (MOPS). For IntSort, this "operation" is ranking a single key, so it is roughly comparable to our TEPS measure.

We port the phase which scatters elements into buckets. This is done by essentially just appending individual elements to pre-allocated buckets, which involves a remote fetch-and-increment and a store. The entire remote end of the scatter is able to be done with an asynchronous migration.

2.5.1.5. Performance comparisons

To evaluate the impact automatic migration has on application performance, we performed experiments comparing compiler-generated communication against manually-optimized explicit delegate calls. As explained earlier, the manual implementations were optimized in previous work evaluating the Grappa runtime itself, so though they may not be the best possible implementation, they are the best known so far. For each application, we compare 2 variants of compiler-generated communication:

	Anchors	Migrations		Hoisted	Allocas	Symmetric
Application	(global)	(blocking)	(async)	accesses	localized	accesses
HOPS	3	0	2	2	0	1
BFS	4	2	1	1	0	5
CC	5	2	1	4	1	13
Pagerank	3	0	2	2	0	0
IntSort	5	0	1	0	0	0

Table 2.2. Static metrics: frequency of each optimization in each benchmark. Counts are for unique source-code instances, so more than one inlining location does not count in these metrics. Only the ported part is counted. HOPS and Pagerank's two asynchronous migrations are each chained.

individual puts and gets, as described in §2.4.5, and Alembic with the *messageCost* which will be chosen empirically in §2.5.3.

Experiments were run on a small cluster with 12 nodes, each with two 6-core Intel Westmere 2.66 GHz Xeon processors with hyperthreading disabled, 24 GB of memory, and 40 Gbit Mellanox ConnectX-2 InfiniBand interconnect. For these experiments we run 8 Grappa processes per node as this gives more reliable performance. The results in Figure 2.3 show the performance of each application as a throughput measurement (bigger is better). Also plotted is the total number of bytes transferred during execution, which is dominated by application data such as puts and gets or continuations.

It is clear that the naive put/get model is insufficient, despite the the runtime's efforts to mask latency. On average, manual delegates performed 7.6x better than put/get. This vast performance pitfall is due to the much larger number of round-trip messages that must be sent, and is echoed in the larger total amount of data moved. By looking at the static migration metrics in Table 2.2, we can get a sense for how many messages are saved. For instance, IntSort performs 5 remote accesses per scatter operation, which can be done manually with a single async delegate, a ratio of 10 messages to 1, so the performance difference should be drastic.

One the other hand, for Pagerank the discrepancy is smaller. The three remote accesses are transformed into two chained migrations, but the second one is back at the source locale, so the put/get implementation, which only does one get, moves less data than the transformed version. However, the additional scheduling overhead of waking the blocked task is such that the asynchronous version is still faster.

Alembic-generated migrations perform favorably with manual delegates, on average achieving 82% of their performance. The cause of this shortfall is visible in the total data moved metric – Alembic moves more data in each of the applications. Rather than doing template specialization and inlining as the C++ code does, Alembic currently uses a C-style interface for `migrate` which requires an additional function pointer and phaser pointer in each message, which, for messages on the order of 16-32 bytes, is significant.

Another situation where Alembic-generated continuations are larger than necessary is when it includes values which could be re-computed. One example is in IntSort, where rather than computing two field offsets from the base pointer, it includes both pointers in the continuation.

These shortcomings can of course be remedied with some engineering effort. A technique analogous to C++ template specialization could be used by the code extraction pass to make optimized versions of `migrate`, eliminating the need for the extra arguments and allowing opportunities to pass arguments through registers. Common register allocation techniques could be applied to determine when to *rematerialize* [35] values to save space.

2.5.2. HOPS Case Study

Our goal with this study is to explore how various optimizations implemented by Alembic affect its performance. Put/get does 3 blocking remote accesses, while the manually-optimized version and the Alembic version both do two chained asynchronous migrations. Figure 2.4 shows three metrics: execution time, total number of messages, and total data movement. We can see that blocking migrations end up moving more data even than put/get. This is because of the additional function pointer and phaser pointer explained earlier, which results in the greater amount of data for Alembic compared with manual. The message count metric matches our expectations closely: both blocking versions have the same number of messages, the next bar is allowed to do an async migration, saving a return trip, and the full Alembic additionally avoids yet another message by hopping directly between two locales. In the end, Alembic achieves 95% of hand-tuned performance for HOPS.

2.5.3. Measuring message cost

The heuristic which drives region selection, described in detail in §2.4.2.2, relies on having an estimate of the relative cost of each message. To get a rough idea of what a good setting for this message cost may be, we construct another variant of GUPS. The goal is to measure the tradeoff between making larger continuations, requiring larger messages for each migration, compared to the benefits of async migrations. For this experiment, GUPS is modified to do additional work after the increment to `A[B[i]]` – it copies an array of randomly-generated values into a static variable on the locale.

For the first experimental condition, we manually do an asynchronous migration containing the GUPS increment and the array computation, so the statically-sized data array must be included in the continuation, and synchronization is done via the default phaser. Alternatively, the second condition leaves the data array on the original stack, does a blocking migration to do the increment, and returns to do the array computation on the original locale. We then vary the size of the data array and measure the performance.

The results, shown in Figure 2.5, show that *blocking* performance is flat, because the communication, which dominates execution time, is constant. The asynchronous migration, however, varies greatly as the continuation's size changes. For smaller amounts of data, avoiding the return message and task wakeup is a clear win (3.5x better than blocking). As continuation size increases, there is an initial drastic drop in performance. This is due to some logic in Grappa's communication layer that optimizes for fitting messages plus some additional metadata in a single cacheline. After that initial drop, however, performance continues to degrade as more memory and network bandwidth is consumed. In these experiments on GUPS, the advantage shifts to the blocking version around 64 bytes of additional data.

Because of this slow degradation, it is safe to err on the larger side when choosing what to set *messageCost* to. In our case, we have chosen to set *messageCost* to 80, which is large enough that in our applications, whenever it is possible to migrate asynchronously, the compiler chooses to do so.

2.6. Related work

Program partitioning to reduce communication has been explored in a variety of systems previously. These can broadly be separated into solutions related to moving computation closer to data, offloading computation to a more capable locale, and other communication optimization techniques.

2.6.1. Computation migration

2.6.1.1. DSM systems

Computation migration was employed in multiple early DSM systems, most notably MCRL [83, 84] and Olden [39, 130], to improve performance for unpredictable access patterns. MCRL, and prior simulation work in the Prelude language, generated a continuation and appropriate messages to perform a lightweight migration, but only at user-annotated procedure calls. On the other hand, Olden performed a relatively heavyweight thread migration (registers and top stack frame) at every remote memory access. Both used heuristics similar to Alembic's to predict when to migrate – Olden used static analysis and annotations to determine how many accesses are co-located, and MCRL used the dynamic read and write load to determine how to balance work. Alembic's migrations are both lightweight like MCRL's, and may happen anywhere in a program, as in Olden. Alembic's aggressive instruction reordering and alloca localizing further improve the effectiveness of computation migration.

2.6.1.2. Traveling threads

The traveling thread execution model [113] is another notable instance of moving execution context to data. In this execution model, threads are split up into much smaller *threadlets,* consisting of just a few instructions, which represent a migration to execute that part of the code closer to the memory it accesses. This work is part of a larger effort to overcome the von Neumann bottleneck by leveraging processing-in-memory (PIM) technology [93]. Aimed at offloading small snippets of execution to the memory system, their notion of locale is extremely fine-grained, at the level of banks of physical memory. Some of the analyses they describe use a similar minimum-cut optimization strategy over the dataflow graph to determine where to split threadlets. Our work could be seen as implementing a form of traveling thread architecture in software on commodity clusters.

2.6.1.3. Charm++ & ParalleX

Charm++ [91] and ParalleX [90] are event-driven distributed-memory programming models based on sending messages between dynamically movable objects. These models allow for a form of computation migration via fine-grained asynchronous active messages. While these models provide opportunities for latency tolerance and scalability, reasoning about sequential control flow can be difficult. Alembic comes from the opposite direction, taking sequential tasks and turning them into asynchronous messages.

2.6.2. Computation offload

Automatic program partitioning has also been explored in the domain of mobile application offloading, where the goal is to reduce the load on resource-constrained clients. Wang and Li [160] partition statically based on a cost heuristic, but rather than using a fixed cost, specialize for multiple cost ranges and select among them at runtime. Other work in dynamic object-oriented languages [154, 161] has modeled communication patterns with object relation graphs, assigning costs according to a target platform and doing min-cut analyses to partition computation and place objects. In the interest of keeping sensitive data on the server, Chong et al. [46] used a similar notion of "anchoring" computation and optimizing communication based on those constraints, in this case for security.

2.6.3. Communication optimization

2.6.3.1. UPC

Unified Parallel C (UPC) [40] is a PGAS language with a number of compiler optimizations to make communication more efficient for the runtime. In addition to common optimizations such as redundancy elimination, the UPC compiler coalesces puts and gets [44] and tolerates latency by automatically making some remote memory operations asynchronous [45]. The latter optimization involves aggressive reordering of memory accesses and coordination of data dependences. Expressing global accesses as C++ pointer dereferences allows us to leverage built-in optimizations, such as simple redundancy elimination, but we do not do static coalescing. The Grappa runtime dynamically aggregates requests and uses programmer-specified parallel tasks to tolerate latency. These techniques, while improving performance by making communication more efficient, do not significantly affect total data movement as migration has the potential to.

2.6.3.2. FortranD

An early PGAS-like programming language, FortranD [77], used layout information to partition straight-line programs to place computation where its data is. Like modern PGAS languages, FortranD has ways to express at a high level how data is distributed across locales, which it uses to determine where to run iterations of loops and generate communication, using what they call the *owner computes* rule. For these techniques to be effective, they need global knowledge of layout, which is not always possible, especially for workloads where layout is dependent on the data.

2.6.3.3. Chapel & X10

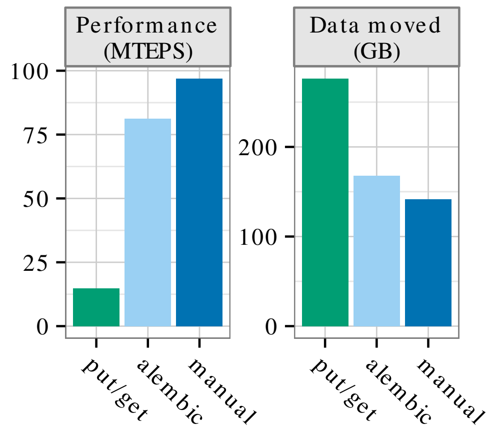
Chapel [42] and X10 [43], two PGAS languages in active development, employ a mix of techniques leveraging high-level information about data layout to optimize communication, such as coalescing communication into bulk operations and spawning tasks with their data [21, 137]. These languages also support explicitly running blocks of code on other locales (via `on` or `at` statements) which operate the same as Grappa's delegates. To the best of our knowledge, that work has not included automatically splitting up tasks and migrating them to improve locality, which is important when there is no "good" initial task placement, and allows the code to remain readable – free of cumbersome nested migration blocks.

2.7. Discussion

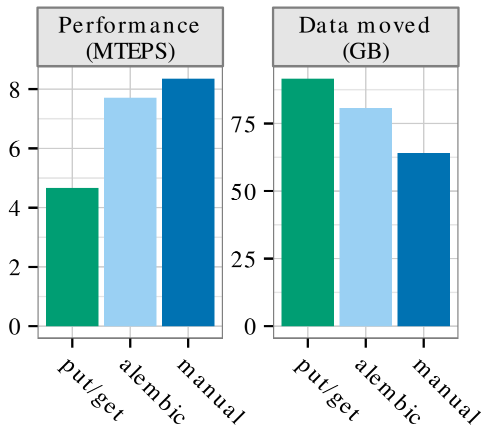
Rather than always moving data around the system, it can be significantly more efficient to move computation to data, provided the execution context is small and the cost of migrating execution is low. On the set of irregular application kernels evaluated in this work, Alembic provides performance close to that of hand-optimized migration – on average within 18% , and is 5.8× faster than naively generated communication. The key observation in this work is that potential migration points can be determined statically by the compiler, and execution context in many cases can be quite small. This is largely due to the nature of the Grappa programming model — programmers are encouraged to maximize concurrency by creating as many small threads as possible. The technique, which finds co-located memory accesses and chooses migrations that minimize communication, should be generally applicable to other PGAS environments and even in non-PGAS situations, such as determining code offload opportunities in mobile applications.

A few important caveats do apply, however. First of all, the size of the execution context is not always statically determinable; the applications evaluated use fixed-size data types with no dynamic allocations. Even a simple string value could break this fragile assumption. There is no obvious barrier to allowing Alembic continuations to contain dynamically sized context, but the determination of when to migrate may become more difficult and data-dependent. This leads to another potential downfall, inherent in any automatic solution: in some situations the compiler may make the wrong choice, and programmers may have a more difficult time tracking down this performance bug. Though the compiler-generated code is functionally correct, it can be difficult to tell what exactly the compiler chose to do and why. The programmer always has the option to write their own migrations manually using delegate operations, but as soon as they do so they introduce the chance for bugs and make the code less readable. Rather than rewriting code whenever one doesn't trust the compiler's decision, it would be advantageous to be able to understand the migrations chosen by the compiler and have ways of controlling the decisions. In order to build and debug our implementation, we built rudimentary tools that generated graph visualizations of the transformed tasks, in LLVM IR. These visualization tools could be improved and made available to users, but we have not done so yet.

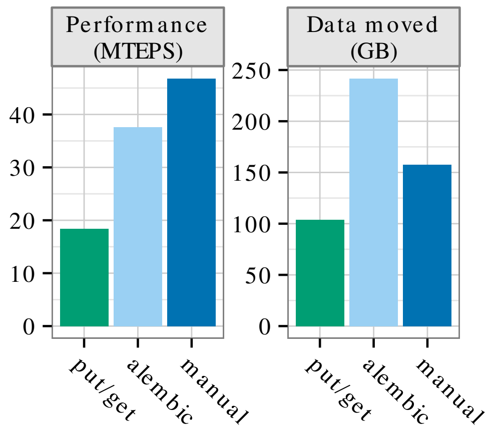
This chapter has tackled one of the core tradeoffs in distributed systems between locality and parallelism by automatically and efficiently moving computation closer to data when beneficial, freeing programmers from this burden. The next chapter will start to address synchronization and consistency and the role that datatypes can have in providing opportunities to increase parallelism without sacrificing consistency.



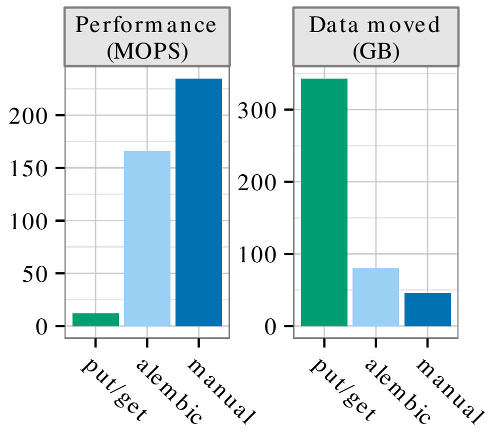
(a) BFS (scale-23 graph)



(b) CC (scale-23 graph)



(c) Pagerank (scale-23 graph)



(d) IntSort (Class D, 0.5M keys)

Figure 2.3. Application kernel performance: comparing manually-optimized movement against compiler-generated migration. Experiments were done on 12 nodes with 8 cores per node. Overall, Alembic performs competitively with manually-optimized communication, and significantly better than naive puts and gets. This performance is due in part to reduced data movement, which is also shown. The outlier, Pagerank, is explained in §2.5.1.5.

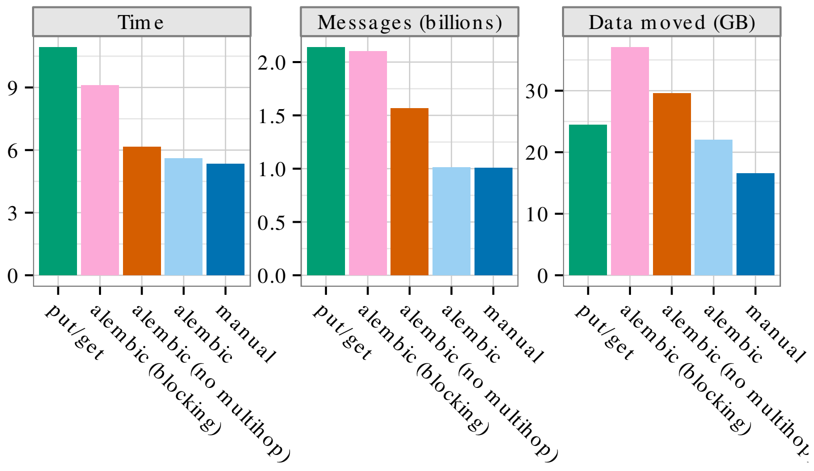


Figure 2.4. Performance of HOPS using manual communication, naive puts and gets, or Alembic migration, with various features disabled. We can see that only with all features enabled does Alembic produce the same number of messages as the manual version.

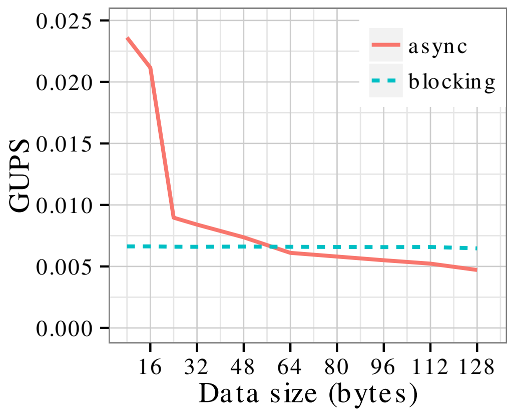


Figure 2.5. Exploring the tradeoff between asynchronous and blocking migrations. The async version must carry additional data with it, while blocking can leave its data behind. Past 64 bytes, blocking wins out, but performance degrades slowly.

3. Distributed combining: Reducing contention with cooperation

3.1. Introduction

Memory consistency models [3, 145] define the observable orders of accesses to memory. *Sequential consistency* (SC), which enforces that all accesses are committed in program order and appear to happen in some global serializable order, is commonly accepted as the easiest to reason about. To preserve SC, operations on shared data structures must be *linearizable* [76]; that is, appear to happen atomically in some global total order.

Enforcing linearizability is notoriously difficult to do in a scalable way due to *contention*. The simplest way is to have a single global lock to enforce atomicity and linearizability through simple mutual exclusion. Serializing accesses in this way limits performance to the speed of a single core, though in practice it is often even worse due to wasted work as threads repeatedly attempt to acquire contended locks. Lock-free concurrent data structures improve performance by using atomic primitives such as fetch-and-increment to implement necessary synchronization. Some examples include the Treiber stack [156] and Michael-Scott queues [109]. However, developing specialized lock-free data structures is itself an ongoing field of research, and with many concurrent updaters, even well-designed synchronization schemes suffer from contention as multiple threads fight to acquire write access to a cache line. With the massive amount of parallelism in a cluster of multiprocessors and with the increased cost of remote synchronization, the problem is magnified.

A general synchronization technique called *combining* encourages threads to *cooperate* rather than *contend*. Rather than every thread attempting to gain full write access to the shared data structure, with combining threads coordinate with each other first in some way to merge their operations before delegating one thread to perform a single combined operation on the shared data structure. Combining allows even a data structure with a single global lock to sometimes scale better than complicated concurrent fine-grained lock-free implementations. Moreover, instead of needing a new complex scalable synchronization mechanism for each new data structure, combining-based data structure implementations can all reuse the same underlying combining framework; each data structure just defines how to merge its operations. Several combining techniques exist, differing mainly in the structure used coordinate among threads: fixed trees [165], dynamic trees (or "funnels") [89, 141], randomized trees [4], or flat queues [72].

This chapter describes how the combining paradigm can be applied to distributed data structures in a partitioned global address space (PGAS) runtime such as Grappa (refer to §2.2 for an overview of PGAS models and Grappa). Our distributed combining approach allows threads on each node to cooperate locally to merge operations before applying them globally, reducing contention on linearizable global data structures. This enables a scalable sequentially consistent environment despite the massive amount of concurrency that the Grappa runtime uses to tolerate remote access latencies. Using a generic combining framework, we implement multiple global data structures — queue, stack, hash set, and hash map — and show that they are scalable, unlike naive lock-based implementations, with close to the same throughput as custom distributed synchronization schemes.

3.2. Distributed combining

One form of combining that has shown to be particularly effective for many-core shared memory systems is called *flat combining* [56, 72, 73] because it employs a single scalable queue that threads add operations to. A single thread then walks this queue, combining the operations in data-structure-specific ways, employing single-threaded optimizations now that it is free from interference. The benefits of flat combining break down into three components: improved locality, reduced synchronization, and data-structure specific optimizations. We will explore how this works in a traditional shared-memory system, and then describe how the same concepts can be applied to distributed memory.

3.2.1. Shared memory

By delegating all work to one core, locality can be improved and synchronization reduced. Consider a concurrent stack with an array of preallocated storage and an index keeping track of the current top of the stack. In a lock-based approach, whenever a thread attempts to push onto the stack, it must acquire a lock, put its value into the storage array, bump the top pointer, and then release the lock. When many threads contend for the lock, all but one will fail and have to retry. Each attempt consumes memory bandwidth and may force an expensive memory fence depending on the architecture, and as the number of threads increase, the fraction of successes plummets.

With flat combining, threads add their push and pop requests to a shared *publication list.* Then they each attempt to acquire the lock once; the thread that succeeds becomes the *combiner* while the rest that failed just sleep instead of retrying. The combiner walks the publication list, performs each request, returning results of any reads to the requester thread, and releases the lock when done. This allows the core running the combiner thread to keep the data structure in its cache, reducing thrashing. At first glance, it appears that we've only shifted the synchronization to the publication list instead of the stack. However, if a thread performs multiple operations, it can leave its publication record in the list and amortize the synchronization cost, and periodically, disused publication records will be garbage collected. This same publication list mechanism can be reused in other data structures, saving each from needing to implement this structure itself.

So far, this has improved locality and amortized synchronization, but we can do better if we take advantage of data-structure-specific properties that allow multiple operations to be performed together more efficiently than separately. As an example, we know that `pop` will take the last pushed value off the top of the stack. Rather than going to the effort of pushing a new value only to immediately pop it off, the combiner can recognize a pop following a push and immediately return the result of the pop to the caller, *annihilating* the matching push and pop operations so that they never show up on the shared data structure. In each round, only unmatched pushes or pops end up being applied to the shared data structure, drastically reducing the work done to the shared data structure. Other data structures can benefit in similar ways, such as tree structures sharing a single traversal for multiple inserts.

3.2.2. Distributed memory

In a distributed setting, particularly with Grappa, the amount of concurrency is much greater than in a shared memory multiprocessor, as is the cost of global synchronization. With thousands of threads per core, as is typical for Grappa applications, a reasonably sized cluster will likely have millions of workers, so contention will be a huge problem. However, because worker threads are cooperatively scheduled, some aspects, such as local synchronization, become easier. Figure 3.1 shows a linearizable global stack distributed over 3 nodes in a cluster. The `top` pointer is owned by one master node (in this case Node 0), conceptually protected by a lock. Usually, in order to do a `push`, a worker would need to perform a remote delegate operation on the node that owns `top`, find where to put the value, potentially do a remote operation to write the new value, then increment top and return control back to the worker. With lots of workers on each node issuing operations on the shared structure, the master node would get swamped in requests, and each worker would have to wait a long time as all the operations serialize on the master.

Employing the combining paradigm here, we wish to allow multiple workers to cooperate and merge their operations into larger combined ones, so that the stack typically handles at most one request per node rather than thousands. In order to cooperate, the threads need to all be able to find a publication list to add their operations to. To facilitate threads finding each other locally, rather than one global publication list, each node or core gets its own. In Grappa, the global stack allocates a

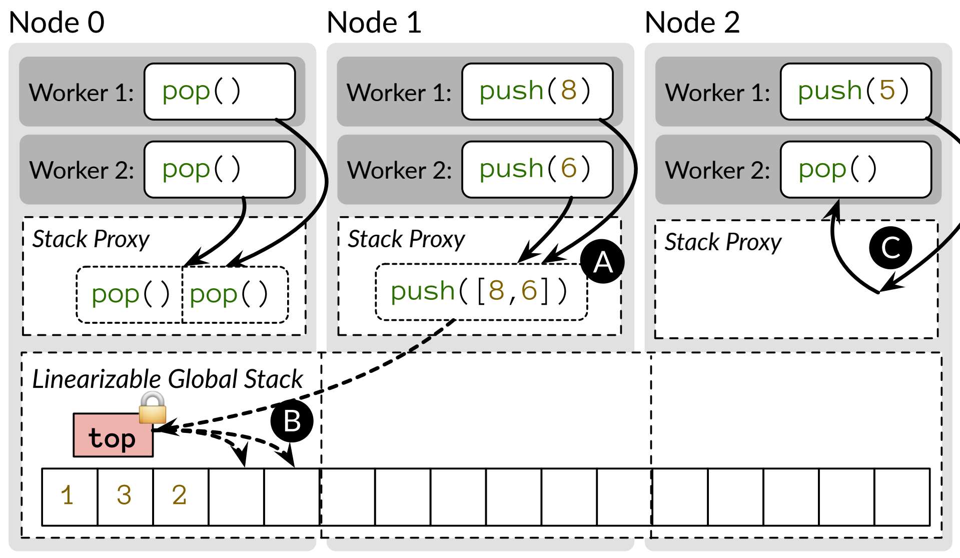


Figure 3.1. Example of distributed combining on a global stack. (A) Worker threads locally combine their operations using the combiner proxy. (B) The combined operation is executed on the shared data structure, using the global lock to ensure linearizability. (C) Some operations can be locally matched, such as this pop which matches with a push locally so that neither operation ever is ever committed to the global stack.

"proxy" object using a symmetric allocation (see §2.3.2), so that each node will have its own copy. Each proxy keeps track of the outstanding operations that have been added to the publication list, preferably in a way that local optimizations, such as matching pushes and pops, can be done. In the case of the global stack, each proxy is itself a little miniature stack, so that pops can be matched with local pushes. In Figure 3.1 (C), we see Worker 1's `push(5)` gets matched with Worker 2's `pop`, so that node does not need to synchronize with the master at all. However, Node 1 ends up with two unmatched pushes that get locally merged into a single operation that pushes two items at once. This combined operation can be applied on the global stack **(B)** with one message to bump `top`, then another message to add both items to the array. Similarly, on Node 0, two pops end up being sent to the global stack because they were unable to be matched locally.

3.2.3. Ensuring linearizability is maintained

In the context of Grappa, sequential consistency means that within a thread, operations will be in thread order and that all threads will observe the same global order. The Grappa memory model is essentially the same as the C++ memory model [29, 30, 87], guaranteeing sequential consistency for data-race free programs. To be conservative, delegate operations block the calling worker until they have become globally visible, ensuring they can be used for synchronization. As such, delegate operations within a task are guaranteed to be globally visible in program order and all tasks observe the same global order. Operations on synchronized data structures must provide the same guarantees. Because it is not immediately obvious that this distributed version of flat combining preserves sequential consistency, we now argue why it does.

To behave in accordance with sequential consistency, operations on a particular data structure must obey a consistent global order. One way to provide this is to guarantee *linearizability* [76] of operations, which requires that the operation appear to take effect globally at some instantaneous point during invocation of the API call. This ensures that a consistent total order can be imposed on operations on a single data structure. For operations to be globally linearizable, two conditions must be met:

1. The execution of local combined operations must be serializable with respect to each other. This will be true as long as each worker blocks until its operation is either satisfied locally or until the combined operation commits globally.

2. Combined operations must be committed atomically in some globally serializable order. This holds as long as the implementation of combined operations ensure atomicity correctly, either by holding a lock on the master or some other form of synchronization. Typically, combined operations use the same synchronization as individual operations before combining was introduced.

Provided these two conditions are upheld, it is straightforward to show that the overall data structure will be linearizable: each combined operation can be thought of as a batch of operations which get applied sequentially, and each batch itself is serializable with respect to each other. This serial “concatenation” of serial batches implies a total global order.

The validity of locally satisfied operations, such as when stack pushes and pops are matched with each other, is not obviously covered by the above argument. However, linearizability does in fact hold for these as well: because matched operations “annihilate” each other, they are not visible to any other threads. Therefore, it does not matter where they “appear” in the global order. As long as locally satisfied operations obey this invisibility criteria, then the data structure can still be considered linearizable.

Combining set and map operations exposes more nuances in the requirements for consistency. Insert and lookup operations performed by different tasks are inherently unordered unless synchronized externally. Therefore, a batch of these operations can be committed globally in parallel, since they are guaranteed not to conflict with each other. Note that two inserts to the same key can be merged, but both callers must block until the combined insert is complete to ensure that program order is preserved. Similarly, lookups can piggy-back on other lookups of the same key. Using intuition from store/write buffers in modern processors, it is tempting to allow a lookup to be satisfied locally by inspecting outstanding inserts. However, this would allow the local order in which keys were inserted to be observed. To preserve SC, that same order would need to be respected globally, forcing each batch to commit atomically with respect to other cores’ batches. Enforcing this would be prohibitively expensive, requiring locks to be held on all the affected buckets, so a cheaper solution is chosen: lookups only get their values from the global data structure, and batches can be performed in parallel.

3.2.4. Evaluation

To evaluate the impact of distributed combining on performance, we implemented a generic combining framework for the Grappa runtime and used it to implement some common global data structures:

- Global vector-based types: a Queue and a Stack, using a single global lock to ensure linearizability.

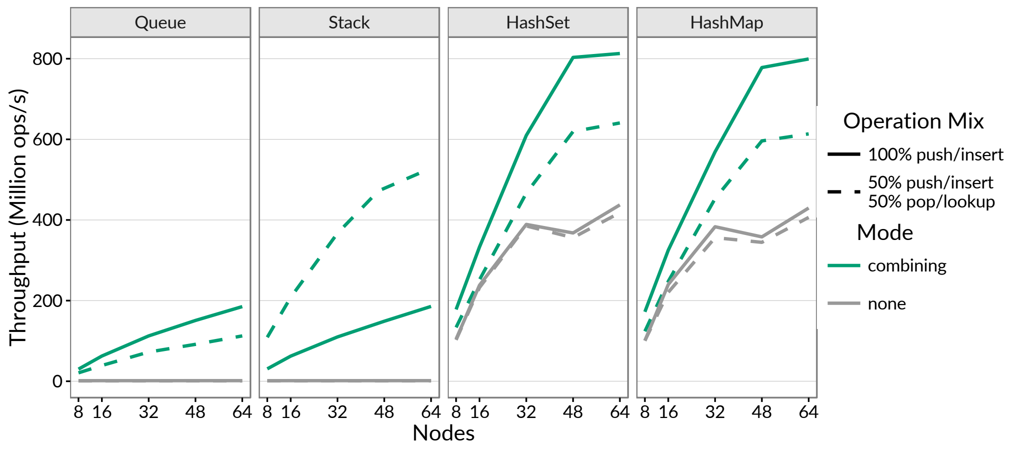


Figure 3.2. *Raw performance of data structures:* throughput results for two different workloads, with all threads in the system performing operations on a single shared data structure. We see that the stack and queue, with a single global lock, perform terribly (~1M ops/s) without combining as expected, but with combining they scale nearly linearly. Global hashset and hashmaps both have locks per bucket, allowing them to scale better, but combining results in a 2x improvement in peak throughput.

- HashMap (and HashSet, which shares most of the implementation), with chaining and no re-hashing, using fine-grained per-bucket locks.

We measured the performance of the data structures with simple stress tests with different workload characteristics, as well as in two graph analytics benchmarks. Experiments were run on a cluster of AMD Interlagos processors. Nodes have 32 2.1-GHz cores in two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards connected via a QLogic switch.

3.2.4.1. Raw data structure performance

We start with a microbenchmark to measuring the performance of the global data structures in isolation: all threads on all cores in the system randomly perform operations on a single shared data structure instance, with and without combining enabled. Throughput results are shown in Figure 3.2.

Without combining, neither the stack nor the queue scale at all, reaching their peak throughput of around 1 million operations per second with less than 8 nodes. However, with combining, both the queue and stack have nearly linear scaling out to 64 nodes. The two perform identically with a workload of all push operations because they perform the same work, but on a mixed workload with push and pop operations, the queue performs somewhat worse due to fewer combining opportunities. On the other hand, the stack achieves significantly higher throughput on mixed workloads due to local matching of pushes and pops, resulting in a 500x throughput improvement with combining, 2.5x better than with 100% pushes.

The HashSet and HashMap both scale significantly better than the stack and queue without combining because they have per-bucket locks, so synchronization is distributed over all the nodes, as long as the workload is uniformly distributed. However, they reach their peak throughput of around 400M operations per second at 32 nodes. With combining enabled, there are opportunities for inserts to combine locally, resulting in a 2x improvement for the 100% insert workload. Because lookups cannot be satisfied by local inserts, we see a smaller improvement on the mixed workload.

3.2.4.2. Graph analytics benchmarks

We also evaluate the usage of these data structures in two common graph analytics benchmarks:

- Breadth First Search (BFS) benchmark used for the Graph500 ranking [69]. Our BFS algorithm employs a global queue which represents the frontier of vertices to be visited in each level. Our implementation employs the direction-optimizing algorithm by Beamer et al [23]. The hand-optimized implementation uses a custom asynchronous "bag" data structure that keeps data local when possible rather than enforcing strict ordering of the queue.

- Connected Components (CC) is another important graph analysis kernel. We implement the three-phase CC algorithm [28] that was de-

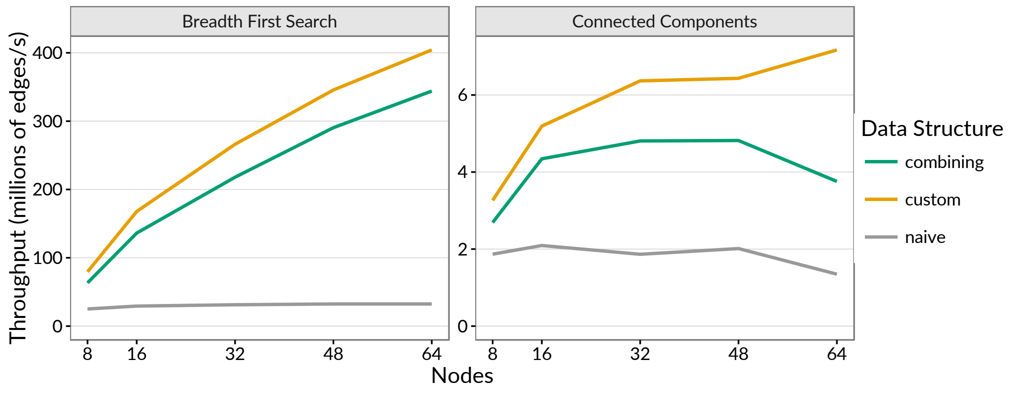


Figure 3.3. *Performance of graph analytics kernels* with different implementations of the core data structure. We see that without combining, the naive data structure performs poorly, but combining performs nearly as well as a custom data structure that relies on additional asynchrony.

signed for the massively parallel MTA-2 machine because of the architecture's similarity to Grappa. The algorithm involves the creation of a reduced graph by building a set of edges that may be at the boundary between components. We use the global HashSet for building this reduced edge set, compared with a custom approach that uses a different set on each node and merges them later, avoiding much of the communication required for the global set.

Both kernels are run on a synthetic power-law graph specified by the Graph500 benchmark, with a scale of 26 (64 million vertices, 1 billion edges). Throughput is reported using the Graph500 metric, "traversed edges per second" which is simply a rate metric based on the size of the graph; we use the same metric for connected components, though the algorithm does significantly more work than a single graph traversal.

Our results, shown in Figure 3.3, show that without combining, the naive implementations of these common data structures severely hinder the performance and scaling of these straightforward graph algorithms. However, when employing combining, the same data structures are able to perform nearly as well as special customized data structures. For BFS, combining eliminates most of the additional synchronization required for maintaining queue order, performing almost as well as the “bag” data structure. For CC, we again see that the combining HashSet achieves roughly twice the throughput, though it does not scale quite as well as the custom version which uses separate sets. This custom approach requires knowing that the sets can be built independently and merged later without sacrificing correctness or losing too much optimization opportunity, which is not always the case.

3.3. Discussion

Distributed combining is an important technique for reducing contention on globally shared data structures. It leverages the *associativity* of operations to allow them to be merged in parallel before being applied globally to the shared data structure. Combining has previously been used in shared memory multiprocessors to improve locality, reduce synchronization, and leverage data-structure specific optimizations. This work applied those techniques to a distributed environment where locality is even more essential and synchronization is costly. The technique works by essentially parallelizing the synchronization required for enforcing linearizability and distributing it among the many nodes in the system. This allows even simple locking protocols to scale, making it simpler to implement correct, scalable data structures.

Another interesting observation of this work is that there can be significant performance improvements in instances where the common ADT, such as a set or queue, is more strict than necessary for the application — both of the hand-optimized graph analytics kernels used custom data structures that required less synchronization. Even though the traditional BFS algorithm calls for *queues* of vertices to visit, the optimized distributed implementation recognized that the order of vertices in each level is unimportant, so uses a custom “bag” data structure instead, which can be implemented with significantly less synchronization in a distributed environment. Similarly, the connected components algorithm called for one shared *set* data structure, but the optimized distributed version recognized that it was beneficial to reduce communication and synchronization by keeping many separate sets on each node and merging them later.

The next chapter builds on this idea of recognizing which data type best expresses the desired semantics. It will introduce a new programming model based on using ADTs to communicate application-level properties to the underlying storage system, and an extensible implementation that allows custom ADTs to be added to reduce synchronization as much as possible. In that work, we will see another application of distributed combining to improve the performance of distributed transactions.

4. Claret: Exposing concurrency in transactions with ADTs

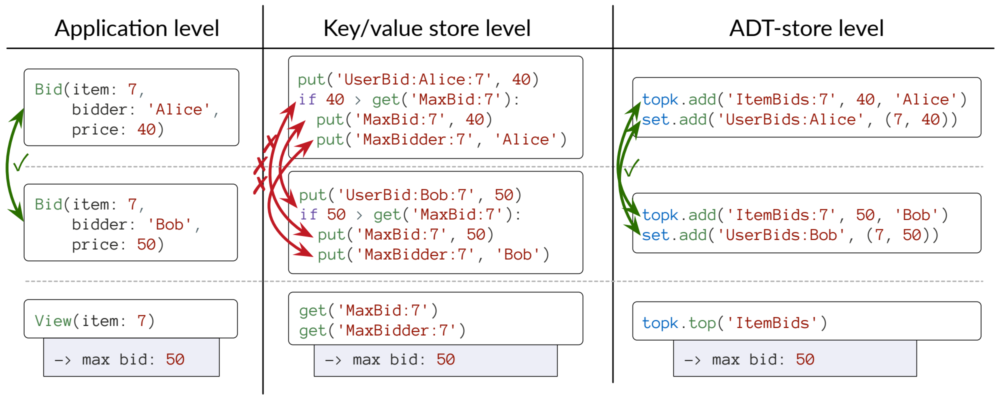
4.1. Introduction

4.1.1. Ubiquitous power laws

In today's online ecosystem, power laws govern everything from the number of followers a user has to the popularity of a given post while real-time events and network effects can lead to sudden traffic spikes at unpredictable times. Because of their interactive nature, this irregular, highly skewed access pattern leads to contention in datastores as many clients attempt to concurrently update the same records. Even content consumption generates write traffic as providers track user behavior to personalize their experience, target ads, or collect statistics [33].

To discuss this more concretely throughout the rest of this chapter, we will use an eBay-like online auction service, based on the well-known RUBiS benchmark [10]. At its core, this service allows users to put items up for auction, browse auctions by region and category, and place bids on open auctions. While running, an auction service is subjected to a mix of requests to open and close auctions but is dominated by bidding and browsing actions. Studies of real-world auction sites [5, 6, 106] have observed that many aspects of them follow power laws. First of all, the number of bids per item roughly follow Zipf's Law (a *zipfian* distribution). However, so do the number of bids per bidder, amount of revenue per seller, number of auction wins per bidder, and more. Furthermore, there is a drastic increase in bidding near the end of an auction window as bidders attempt to out-bid one another, so there is also a realtime component.

An auction site's ability to handle these peak bidding times is crucial:



```
Bid(item: 7,
    bidder: 'Alice',
    price: 40)

Bid(item: 7,
    bidder: 'Bob',
    price: 50)

View(item: 7)
 -> max bid: 50
```

```
put('UserBid:Alice:7', 40)
if 40 > get('MaxBid:7'):
  put('MaxBid:7', 40)
  put('MaxBidder:7', 'Alice')

put('UserBid:Bob:7', 50)
if 50 > get('MaxBid:7'):
  put('MaxBid:7', 50)
  put('MaxBidder:7', 'Bob')

get('MaxBid:7')
get('MaxBidder:7')
 -> max bid: 50
```

```
topk.add('ItemBids:7', 40, 'Alice')
set.add('UserBids:Alice', (7, 40))

topk.add('ItemBids:7', 50, 'Bob')
set.add('UserBids:Bob', (7, 50))

topk.top('ItemBids')
 -> max bid: 50
```

Figure 4.1. At the application level, it is clear that bid transactions commute, but when translated down to put and get operations, this knowledge is lost. Using an ADT like a `topk` set preserves this commutativity information.

a slowdown in service caused by a popular auction may prevent bidders from reaching their maximum price (especially considering the automation often employed by bidders). The ability to handle contentious bids directly impacts revenue, as well as being responsible for user satisfaction. Additionally, this situation is not suitable for weaker consistency, so we must find ways to satisfy performance needs without sacrificing strong consistency.

4.1.2. Finding concurrency

To avoid catastrophic failures and mitigate poor tail behavior, significant engineering effort must go into handling these challenging high-contention scenarios. Writes are such a problem because they impose ordering constraints requiring synchronization in order to have any form of consistency. Luckily, many of these ordering constraints are actually irrelevant from the perspective of the application. For example, it is not necessary to keep track of the order in which people retweeted Ellen's famous celebrity selfie at the Oscars [15]. In particular, some operations are commutative, meaning the order in which they occur does not change the observable outcomes. If the system knew exactly which constraints were relevant to the application, then it could expose significantly more concurrency, allowing it to handle spikes without sacrificing correctness.

Consider the auction service example again. At the application level, it should be clear that bids on an item can be reordered with one another, provided that the correct maximum bid can still be tracked. When the auction closes, or whenever someone views the current maximum bid, that imposes an ordering which bids cannot move beyond. In the example in Figure 4.1, it is clear that the maximum bid observed by the `View` action will be the same if the two bids are executed in either order. That is to say, the bids *commute* with one another.

If we take the high-level `Bid` action and implement it on a typical key/value store, we lose that knowledge. The individual `get` and `put` operations used to track the maximum bid conflict with one another. Executing with transactions will still get the right result but only by ensuring mutual exclusion on all involved records for the duration of each transaction, serializing bids per item.

4.1.3. Using abstract data types with Claret

In this work, we propose a new way to leverage *abstract data types* (ADTs) to improve the performance of distributed transactions in key-value datastores by exposing concurrency that was already present. ADTs allow users and systems alike to reason about their logical behavior, including algebraic properties like commutativity, rather than the low-level operations used to implement them. The datastore can leverage this higher-level knowledge to avoid conflicts, allowing transactions to interleave and execute concurrently without changing the observable behavior. Programmers benefit from the flexibility and expressivity of ADTs, reusing data structures from a common library or implementing custom ADTs for their specific use case.

Our prototype ADT-store, *Claret*, demonstrates how ADT awareness can be added to a datastore to make strongly consistent distributed transactions practical. It is the first non-relational system to leverage ADT semantics to reduce conflicts between distributed transactions. Rather than requiring a relational data model with a fixed schema, Claret encourages programmers to use whatever data structures naturally express their application.

Prior work in databases and distributed systems have used properties such as commutativity to reduce coordination, by either relying on a predefined relational schema [14, 47, 75, 162] or restricting operations to only those that are coordination-free [8, 112, 139]. However, schemaless NoSQL datastores like Redis [135] are becoming increasingly popular for building web services, in part because they allow more flexibility and control over data structures and are more easy to scale. In addition to Redis, many [11, 22, 58, 159] support common collection types like lists, sets, and maps, with custom operations for each type. However, these datastores typically do not provide support for distributed transactions[4] due to the high overhead of naive implementations.

Claret uses the logical properties of data types to communicate application-level semantics to the system so it can perform optimizations on both the client and server side. In §4.3, we show how commutativity can be used to avoid false conflicts (*boosting*) and ordering constraints (*phasing*), and how associativity can be applied to reduce load on the datastore (*combining*).

On high-contention workloads, the combined optimizations achieve up to a 50x improvement in peak transaction throughput over traditional concurrency control on a synthetic microbenchmark, up to 2x two applications: an auction benchmark based on Rubis [10] and a Twitter clone based on Retwis [136]. While Claret's optimizations help most in high-contention cases, its performance on workloads with little contention is unaffected. Claret's transactions achieve 67-82% of the throughput possible without transactions, which represents an upper bound on the performance of our datastore.

This work makes the following contributions:

- Design of an *extensible ADT-store*, Claret, with interfaces to express logical properties of new ADTs
- Implementation of optimizations leveraging ADT semantics: *transac-*

[4]Hyperdex is the exception, though its Warp concurrency system [59] uses an orthogonal approach called transaction chopping, not knowledge of ADT operations.

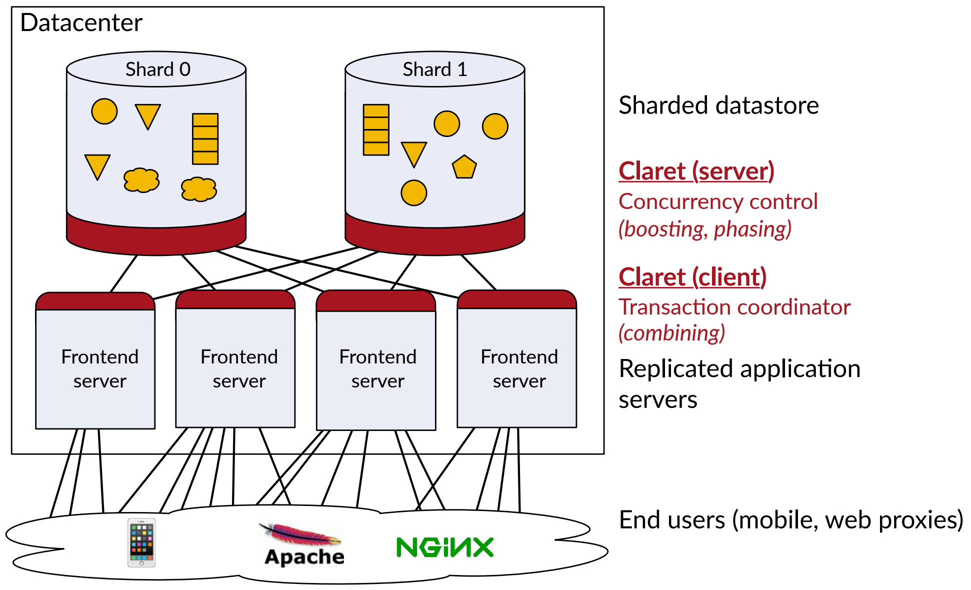


Figure 4.2. *System model:* End-user requests are handled by replicated stateless application servers which all share a sharded datastore. Claret operates between these two layers, extending the datastore with ADT-aware concurrency control (*Claret server*) and adding functionality to the app servers to perform ADT operations and coordinate transactions (*Claret client*).

tion boosting, *operation combining*, and *phasing*

- Evaluation of the impact of these optimizations on raw transaction performance and benchmarks modeling real-world contention

In this chapter, we describe the design of the system and evaluate the impact ADT-enabled optimizations have on transaction performance. But first, we must delve more deeply into what causes contention in real applications.

4.2. System model

The concept of ADTs could be applied to many different datastores and systems. For Claret, we focus on one commonly employed system architecture, shown in Figure 4.2: a sharded datastore shared by many stateless replicated application servers within a single datacenter. For horizontal scalability, datastores are divided into many shards, each containing a subset of the key space (often using consistent hashing), running on different hosts (nodes or cores). Frontend servers are the *clients* in our model, implementing the core application logic and exposing it via APIs to end users that may be mobile clients or web servers. These servers are replicated to mitigate failures, but each instance may handle many concurrent end-user connections, mediating access to the backing datastore where application state resides.

Claret operates between application servers and the datastore. Applications model their state using ADTs and operations on them, as they would in Redis, but differing from Redis, Claret strongly encourages the use of transactions to ease reasoning about consistency. Clients are responsible for coordinating their transactions, retrying if necessary, using multithreading to handle concurrent end-user requests. A new ADT-aware concurrency control system is added to each shard of the core datastore. ADT awareness is used in both the concurrency control system and the client, which will be explained in more depth in §4.3.

4.2.1. Programming model

The Claret programming model is not significantly different than traditional key/value stores, especially for users of Redis [135]. Rather than just strings with two available operations, `put` and `get`, records can have any of a number of different types, each of which have operations associated with them. Each record has a type, determined by a tag associated with its key so invalid operations are prevented on the client. The particular client bindings employed are not essential to this work; our code examples will use Python-like syntax similar to Redis's Python bindings though our actual implementation uses C++. An example of an ADT implementation of the Bid transaction is shown in Figure 4.1.

Programmers express application-level semantics by choosing the most specific ADT for their needs, either by choosing from the built-in ADTs (Table 4.1) or implementing their own (see §4.4). In Figure 4.1, we saw that `Bid` transactions should commute; we just need know the current high bid. Redis has a `zset` type representing a sorted or ranked set: it associates a score with each item and allows elements to be retrieved by score. A `topk` set is a specialization of `zset` optimized to keep track of only the highest-ranked items. A `topk` meets our needs for bids perfectly. Furthermore, `topk.add` operations commute so `Bid` transactions no longer conflict.

4.2.2. Consistency model

Weak consistency models require programmers to understand and guard against all potentially problematic interleavings. With Claret, programmers instead focus on choosing ADTs that best represent their desired behavior, naturally exposing opportunities for optimization. Individual operations in Claret are strictly linearizable, committing atomically on the shard that owns the record. Each record, including aggregate types, behaves as a single object living on one shard. Atomicity is determined by the granularity of ADT operations. Custom ADTs can allow arbitrarily complex application logic to be atomic, provided they can be localized to a single object, but in general, composing operations requires transactions.

4.2.3. Transaction model

Claret implements interactive distributed transactions with strict serializable isolation, similar to Spanner [52], with standard `begin`, `commit`, and `abort` functions and automatic retries. Claret supports general transactions: clients are free to perform any operations on any records within the scope of the transaction. It uses strict two-phase locking, acquiring a lock for each record before accessing it during transaction execution.

We support arbitrary ADT operations in transactions by splitting them into two parts, *execute* and *commit*, which both run on the shard holding the record. *Execute* attempts to acquire the lock for the record; when it succeeds, it executes the operation *read-only* to compute a result. Operations without a return value do nothing after acquiring the lock, simply returning control to the calling transaction. Once locks for all operations in a transaction have been acquired, a *commit* is sent to each participating shard to run the *commit* part of all the operations, performing any mutation on the record and releasing the lock. Similar to Spanner [52], clients do not read their own writes; due to the buffering of mutations, operations always observe the state prior to the beginning of the transaction.

We chose a lock-based approach; we briefly explain in §4.3.1 how this approach could similarly benefit optimistic concurrency control (OCC).

Claret does not require major application modifications to express concurrency. From the clients' view, there are no fundamental differences between using Redis and Claret (except the addition of distributed transactions and custom types). Under the hood, however, Claret will use its knowledge about ADTs to improve performance in a number of ways that we describe next.

4.3. Leveraging data types

Abstract data types decouple their abstract behavior from their low-level concrete implementation. Abstract operations can have properties such as commutativity, associativity, or monotonicity, which define how they can be reordered or executed concurrently, while the concrete implementation takes care of performing the necessary synchronization. Knowledge of these properties can be used in many ways to improve performance. First, we show how commutativity can be used in the concurrency control system to avoid false conflicts (*boosting*) and ordering constraints (*phasing*). Then we give an example of how associativity can be applied to reduce the load on the datastore (*combining*).

4.3.1. Transaction boosting

To ensure strong isolation, all transactional storage systems implement some form of concurrency control. A common approach is strict two-phase-locking (S2PL), where a transaction acquires locks on all records in the execution phase before performing any irreversible changes. However, in distributed systems, holding locks is costly because large round-trip latencies cause them to be held for long periods, depriving the system of much of its potential parallelism. Allowing operations to *share* locks is essential to providing reasonable throughput and latency for transactions. Reader/writer locks are commonly used to allow transactions reading the same record to execute concurrently, but they force transactions writing the record to wait for exclusive access.

Abstract locks [14, 47, 75, 138, 162] generalize reader-writer locks to any operations that can logically run concurrently. When associated with an ADT, they allow operations that commute to hold the lock at the same time. For the `topk` set, `add` operations can all hold the lock at the same time, but reading operations such as `size` must wait. The same idea can be applied to OCC: operations only cause conflicts if the abstract lock doesn't allow them to execute concurrently with other outstanding operations.

Known as *transaction boosting* [74] in the transactional memory literature, using abstract locks is even more important for distributed transactions which support massively more parallelism but have much longer latencies. Abstract locks directly increase the number of transactions which can execute concurrently. In OCC-based systems, boosting can reduce the abort rate because fewer operations conflict with one another.

Boosting is essential for highly contended records, such as bids on a popular auction as it is about to close. If all the bids are serialized because they are thought to conflict, then fewer can complete and the final price could be lower. Using a `topk` set whose `add` operations naturally commute allows more bids to complete.

4.3.2. Phasing

Sometimes the order that operations happen to arrive causes problems with abstract locks. In particular, they only help if the operations that commute with each other arrive together; poor interleaving can result in little effect. *Phasing* reorders operations so that commuting operations execute together, similar to batching.

Each ADT defines a *phaser* that is responsible for grouping operations into *phases* that execute together. The interface will be described in more detail in §4.4.1.2. Each record (or rather, the lock on the record), has its own phaser which keeps track of which mode is currently executing and keeps queues of operations in other modes waiting to acquire the lock. The phaser then cycles through these modes, switching to the next when all the operations in a phase have committed. Abstract locks, which have more distinct modes, benefit more than reader/writer locks, which must still serialize all writes.

By reordering operations, phasing has an effect on fairness. Queueing on locks improves fairness compared to our baseline retry strategy that can lead to starvation. However, because phasing allows operations that commute to be executed before earlier blocked operations, it can lead to fairness issues. If a record has a steady stream of read operations, it may never release the lock to allow mutating operations. To prevent this, we cap phases at a maximum duration whenever there are blocked operations. In our experiments, we only observed this as an issue at extreme skew. The latency of some operations may increase as they are forced to wait for their phase to come. However, reducing conflicts often reduces the latency of transactions overall.

4.3.3. Combining

Associativity is another useful property of operations. If we consider *commutativity*, used in boosting and phasing above, as allowing operations to be executed in a different order on a record, then *associativity* allows us to merge operations together *before* applying them to the record. This technique, *combining* [72, 141, 165], can drastically reduce contention on shared data structures and improve performance whenever the combined operation is cheaper than all the individual operations. As discussed in detail in the Chapter 3, combining can be extremely useful in distributed environments, effectively parallelizing synchronization for a single data structure over multiple machines.

For distributed datastores, where the network is typically the bottleneck, combining can reduce server load. If many clients wish to perform operations on one record, each of them must send a message to acquire the lock. Even if they commute and so can hold the lock concurrently, the server handling the requests can get overloaded. In our model, however, "clients" are actually frontend servers handling many different end-user requests. With combining enabled, Claret keeps track of all the locks currently held by transactions on one frontend server. Whenever a client performs a combinable operation, if it finds its lock already in the table, it simply merges with the operation that acquired the lock, without needing to contact the server again.

For correctness, transactions sharing combined operations must all commit together. This also means that they must not conflict on any of their other locks, otherwise they would deadlock, and this applies transitively through all combined operations. Claret handles this by merging the lock sets of the two transactions and aborting a transaction and removing it from the set if it later performs an operation that conflicts with the others.

Tracking outstanding locks and merging lock sets adds overhead but offloads work to clients, which are easier to replicate to handle additional load than datastore shards. In our evaluation (§4.5), we find that combining is most effective at those critical times of extreme contention when load is highly skewed toward one shard.

4.4. Expressing abstract behavior

As in any software design, building Claret applications involves choosing the right data structures. There are many valid ways to compose ADTs to model application data, but to achieve the best performance, one should express as much high-level abstract behavior as possible through ADT operations.

Typically, the more specialized an ADT is, the more concurrency it can expose, so finding the closest match is essential. For example, one could use a counter to generate unique identifiers, but counters must return numbers in sequence, which is difficult to scale (as implementers of TPC-C [155], which explicitly requires this, know well). A special `UniqueID` type succinctly communicates that non-sequential IDs are acceptable, allowing a commutative implementation.

Claret has a library of pre-defined ADTs (Table 4.1) used to imple-

Data type	Description
`UIdGenerator`	Create unique identifiers (not necessarily sequential) (`next`)
`Dict`	Map which allows setting or getting multiple fields atomically
`ScoredSet`	Set with unique items ranked by a score (`add`, `size`, `range`)
`TopK`	Like `ScoredSet` but keeps only highest-ranked items (`add`, `max`, ...)
`SummaryBag`	Container where only summary statistics of items can be retrieved (`add`, `mean`, `max`)

Table 4.1. Library of built-in data types.

ment the applications in this paper. Reusing existing ADTs saves implementation time and effort, but may not always expose the maximum amount of concurrency. Custom ADTs can express more complex application-specific properties, but the developer is responsible for specifying the abstract behavior for Claret. The next sections will show how ADT behavior is specified in Claret to expose abstract properties of ADTs for our optimizations to leverage.

4.4.1. Commutativity Specification

Commutativity is not a property of an operation in isolation. A *pair* of operations commute if executing them on their target record in either order will produce the same outcome. Using the definitions from [94], whether or not a pair of method invocations commute is a function of the methods, their arguments, their return values, and the *abstract state* of their target. We call the full set of commutativity rules for an ADT its *commutativity specification.* An example specification for a *Set* is shown in Table 4.2. However, we need something besides this declarative representation to communicate this specification to Claret's concurrency controller.

Method:	**And:**	**Commute when:**
`add(x): void`	`add(y)`	$\forall x, y$
`remove(x): void`	`remove(y)`	$\forall x, y$
	`add(y)`	$x \neq y$
`size(): int`	`add(x)`	$x \in Set$
	`remove(x)`	$x \notin Set$
`contains(x): bool`	`add(y)`	$x \neq y \vee y \in Set$
	`remove(y)`	$x \neq y \vee y \notin Set$
	`size()`	$\forall x$

Table 4.2. Abstract Commutativity Specification for Set.

4.4.1.1. Abstract lock interface

In Claret, each data type describes its commutativity by implementing the *abstract lock* interface shown in Listing 4.1. This imperative interface allows data types to be arbitrarily introspective when determining commutativity. In our implementation, clients must acquire locks for each operation before executing them. When the datastore receives a lock request for an operation on a record, the concurrency controller queries the abstract lock associated with the record using its `acquire` method, which checks the new operation against the other operations currently holding the lock to determine if it can execute concurrently (commutes) with all of them.

Implementations of this interface typically keep a set of the current lock-holders. They determine which operations are currently permitted to share the lock by dividing them into *modes*. For example, reader/writer locks have a *read* mode for all read-only operations and an *exclusive* mode for the rest, while abstract locks have additional modes, such as an *append* mode for sets which allows all `add`s. More fine-grained tracking in `acquire` can expose more concurrency; for instance, `contains` can execute during *append* if the item already exists.

```
class zset:
  class abstract_lock:
    # return True if `op` can execute on `record`
    # concurrently with other lock holders;
    # adds txn_id to set of lock holders
    def acquire(record, op, txn_id):
      if (op.is_add() and self.mode == ADD) or
         (op.is_read() and self.mode == READ):
        self.holders.add(txn_id)
        return True
      # ...

    # called when a transaction commits or aborts,
    # releasing its locks, remove `txn_id` from
    # lock holders
    def release(txn_id):
      self.holders.remove(txn_id)
      if self.holders.empty():
        self.mode = None
```

Listing 4.1. Interface for expressing commutativity for a data type. Typical implementations use *modes* to easily determine sets of allowed operations, and a *set* of lock-holders to keep track of outstanding operations.

4.4.1.2. Phaser interface

Phasing requires knowing how to divide operations into *phases*, similar to the *modes* for locks, but rather than tracking which operations currently hold the lock, the *phaser* associated with a record tracks all the operations waiting to acquire the lock. When an operation fails to acquire the lock, the controller *enqueues* it with the phaser. When a phase completes, the abstract lock *signals* the phaser, requesting operations for a new phase.

The simplest implementations keep queues corresponding to each mode. Listing 4.2, for example, shows `adders`, which will contain all operations that may insert into the set (just `add`), and `readers`, which includes any read-only operations (`size`, `contains`, `range`, etc). As with abstract locks, more complicated phaser implementations can allow operations to be in multiple modes or use more complex state-dependent logic to determine which operations to signal.

```
class zset:
  class phaser:
    def enqueue(self, op):
      if op.is_read():
        self.readers.push(op)
      elif op.is_add():
        self.adders.push(op)
      # ...

    def signal(self, prev_mode):
      if prev_mode == READ:
        self.adders.signal_all()
      elif prev_mode == ADD:
        self.readers.signal_all()
      # ...
```

Listing 4.2. Phaser interface (example implementation): `enqueue` is called after an operation fails to acquire a lock, `signal` is called when a phase finishes (all ops in the phase commit and release the lock).

4.4.2. Combiners

Finally, ADTs wishing to perform combining (§4.3.3) must implement a *combiner* to tell Claret how to combine operations. Combiners only have one method, `combine`, which attempts to match the provided operation against any other outstanding operations (operations that have acquired a lock but not committed yet).

Remember from §4.2 that operations are split into *execute* and *commit*. Combining is only concerned with the *execute* part of the operation. Operations that do not return a value (such as `add`) are simple to combine: any commuting operations essentially share the acquired lock and commit together. Operations that return a value in *execute* (any read), can only be combined if they can share the result. For `zset.size`, all concurrent transactions should read the same size, so combined `size` ops can all return the size retrieved by the first one. The `range` operation on a `zset` is a more complex example of sharing results: if one operation's range is a subset of another, the two can be combined, sharing the output of the larger range query.

To avoid excessive matching, only operations declared *combinable* are compared. The client-side library keeps track of outstanding combinable operations with a map of combiners indexed by key. Combiners are registered after a lock is acquired and removed when the operation commits. Before sending an acquire request for a combinable operation, the client checks the map for that key. If none is found, or the combine fails, it is sent to the server as usual. If it succeeds, the result is returned immediately, and Claret handles merging the two transactions as described before in §4.3.3.

4.4.3. Adding a custom ADT

The ADTs provided by Claret are all implemented using these interfaces to communicate their commutativity and associativity to Claret's concurrency control system. To implement a custom ADT that can take advantage of all of the optimizations, programmers must simply implement an abstract lock, phaser, and combiner. ADT designers could choose to use simple implementations similar to those in Claret that just divide operations statically into modes, or experts could use more state-dependent logic to provide fine-grained concurrency if needed.

4.5. Evaluation

To understand the potential performance impact of the optimizations enabled by ADTs, we built a prototype in-memory ADT-store in C++, dubbed Claret. Our design follows the architecture previously described in Figure 4.2, with keys distributed among shards by consistent hashing, though related keys may be co-located using tags as in Redis. Records are kept in their in-memory representations, avoiding serialization. Clients are multi-threaded, to model frontend servers handling many concurrent end-user requests, and use ethernet sockets and protocol buffers to communicate with the data servers.

As discussed before, Claret uses two-phase locking (2PL) to isolate transactions. The baseline uses reader/writer locks. When boosting is enabled, these are replaced with abstract locks. By default, operations retry whenever they fail to acquire a lock; with phasing, replies are sent whenever the lock is finally acquired, so retries are only used to resolve deadlocks or lost packets.

Our prototype does not provide fault-tolerance or durability. These could be implemented by replicating shards or logging to persistent storage. Synchronizing with replicas or the filesystem should increase the time spent holding locks, so lower throughput and higher latency are expected. Claret increases the number of clients which can simultaneously hold locks, so adding fault tolerance would be expected to reinforce our findings.

We wish to understand the impact our ADT optimizations have on throughput and latency under various contention scenarios. First, we use a microbenchmark to directly tune contention by varying operation mix and key distribution. We then move on to explore scenarios modeling real-world contention in two benchmarks – Rubis: an online auction service, and Retwis: a Twitter clone.

The following experiments are run on a Linux cluster running Ubuntu 15.10. Each node has dual 6-core 2.5 GHz Intel Xeon E5-2680-v3 processors with 64 GB of memory, connected by a 40Gb ethernet. Experiments are run with 4 single-threaded shards running on 4 different nodes, with 4 multithreaded clients on 4 separate nodes. Average round-trip times between nodes for UDP packets are 35 μs, with negligible packet loss.

4.5.1. Raw Operation Mix

This microbenchmark performs a random mix of operations, similar to YCSB or YCSB+T [51, 55], that allows us to explicitly control the degree of contention. Each transaction executes a fixed number of operations (4), randomly selecting either a read operation (`set.size`), or a commutative write operation (`set.add`), and keys selected randomly with a Zipfian distribution from a pool of 10,000 keys. By varying the percentage of `add`s, we control the number of potential conflicting operations. Importantly, `add`s commute with one another, but not with `size`, so even with boosting, conflicts remain. The Zipf parameter, α, determines the shape of the distribution; a value of 1 corresponds to Zipf's Law, lower values are shallower and more uniform, higher values more skewed. YCSB sets α near 1; we explore a range of parameters.

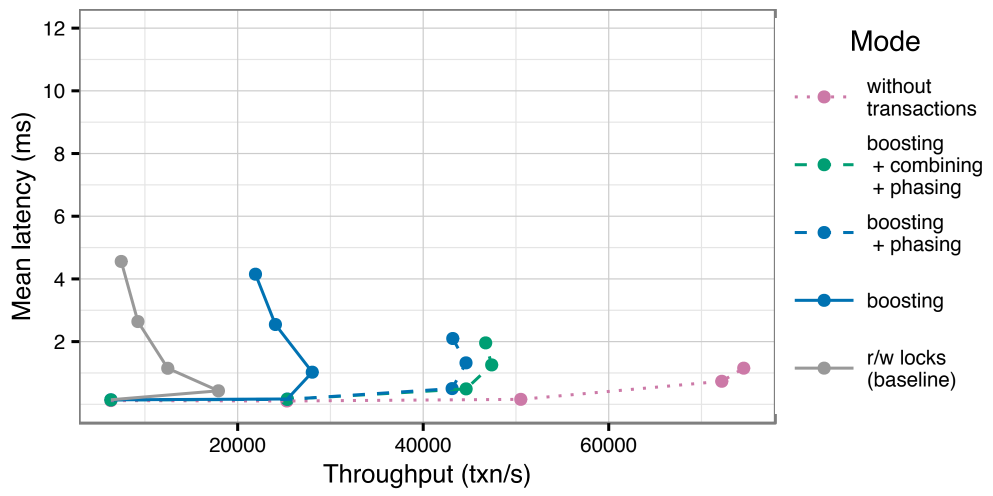


Figure 4.3. Raw mix workload (50% read, zipf: 0.6), increasing number of clients, plotted as throughput vs. latency. Boosting (abstract locks) eliminates conflicts between `adds`, phasing reorders operations more efficiently. Results in a 2.6x throughput improvement, 63% of the performance without transactions.

We start with a 50% read, 50% write workload and a modest zipfian parameter of 0.6, and vary the number of clients. Figure 4.3 shows a throughput versus latency plot with lines showing each condition as we vary the number of clients (from 8 to 384). The baseline, using traditional r/w locks, reaches peak throughput with few clients before latencies spike; throughput suffers as additional clients create more contention. Abstract locks (*boosting*) expose more concurrency, increasing peak throughput. Adding phasing (dashed lines) improvements peak throughput because it improves operation fairness while also improving the chances of commuting.

The dotted pink line in Figure 4.3 shows performance of the same workload with operations executed independently, without transactions (though performance is still measured in terms of the "transactions" of groups of 4 operations). These operations execute immediately on the records in a linearizable [76] fashion without locks. This serves as a reasonable upper bound on the throughput possible with our servers. Claret's

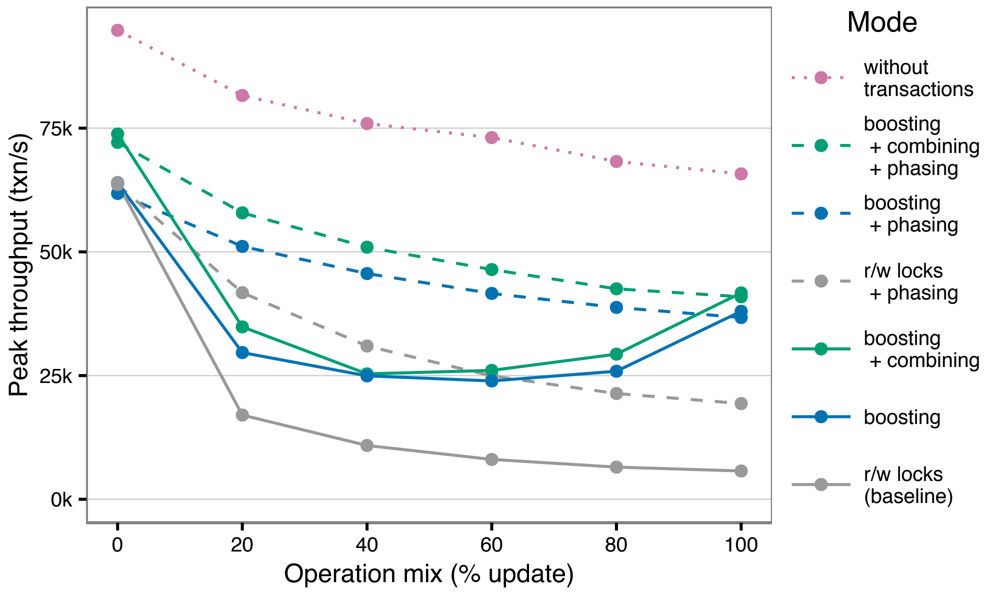


Figure 4.4. *Peak throughput, varying operation mix.* Boosting is increasingly important with a higher fraction of adds. Phasing is essential for any mixed workload.

transactions achieve 63% of that throughput on this workload.

Varying operation mix. Figure 4.4 shows throughput as we vary the percentage of commutative write operations (`add`), with keys selected with a modest 0.6 zipfian distribution. Boosting becomes more important as the fraction of commutative `add`s increases. Phasing has a significant impact for any mixed workload, as it helps commutative operations run concurrently; tracing the execution, we observed that records regularly alternated between `add` and `read` phases. Combining shows a modest improvement for all workloads, even for read-only and write-only, because it occasionally allows transactions to share locks (and the result of reads) without burdening the server.

Varying key distribution. Figure 4.5 shows throughput with a 50/50 operation mix, controlling contention by adjusting the zipfian skew parameter used to choose keys. At low zipfian, the distribution is mostly

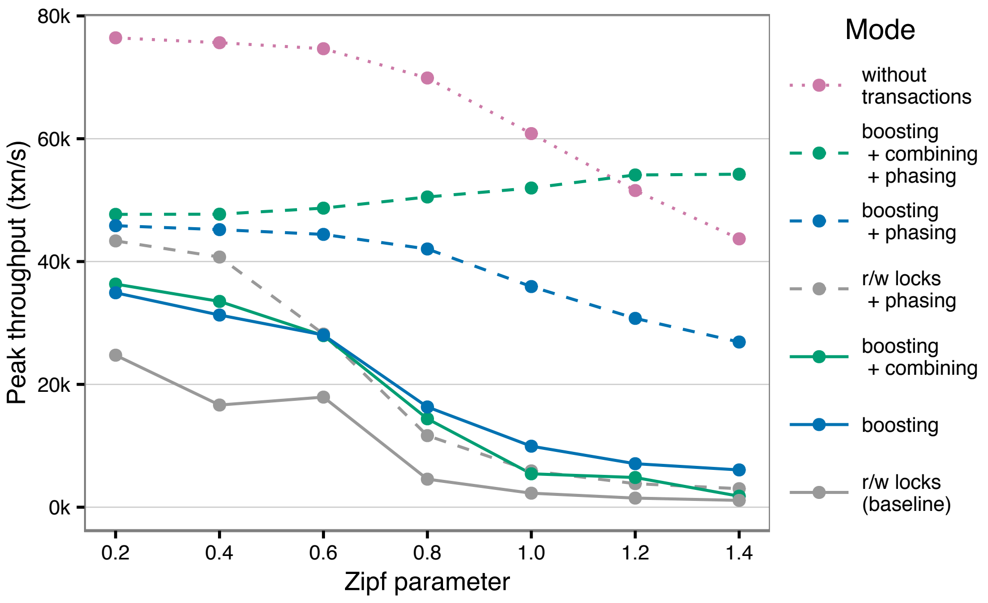
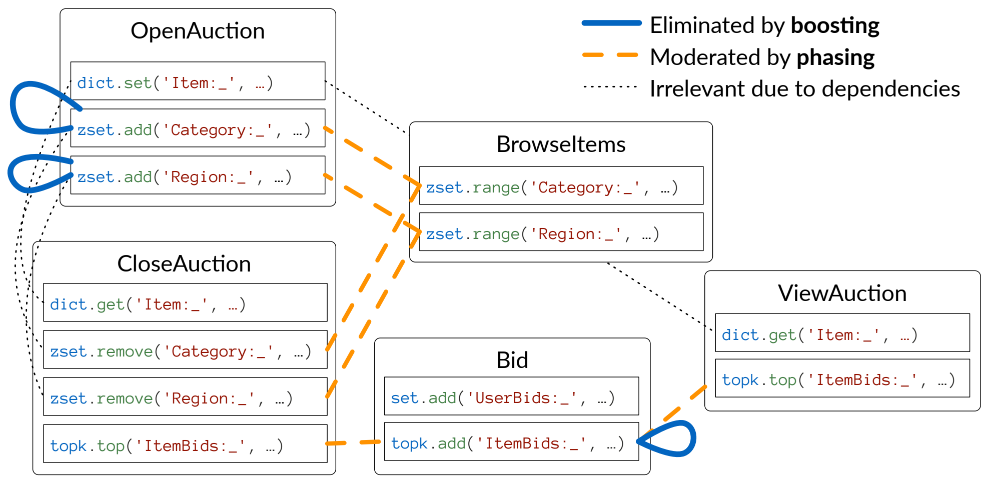


Figure 4.5. *Peak throughput, varying key distribution.* Higher Zipf parameter results in greater skew and contention; boosting and phasing together expose concurrency. At extreme skew, combining reduces load on hot records, which our non-transactional mode cannot do.

uniform over the 10,000 keys, so most operations are concurrent simply because they fall on different keys, and Claret shows little benefit. As the distribution becomes more skewed, transactions contend on a smaller set of popular records. With less inter-record concurrency, we rely on abstract locks (boosting) to expose concurrency within records.

At high skew, there is a steady drop in performance simply due to serializing operations on the few shards unlucky enough to hold the popular keys. However, skew increases the chance of finding operations to combine with, so combining is able to offload significant load from the hot shards. Our implementation of combining requires operations to be split into the acquire and commit phases, so it cannot be used without transactions. Though distributions during normal execution are typically more moderate, extreme skew models the behavior during exceptional situa-



Eliminated by **boosting**
Moderated by **phasing**
Irrelevant due to dependencies
OpenAuction
dict.set('Item:_', …)
zset.add('Category:_', …)
zset.add('Region:_', …)
CloseAuction
dict.get('Item:_', …)
zset.remove('Category:_', …)
zset.remove('Region:_', …)
topk.top('ItemBids:_', …)
BrowseItems
zset.range('Category:_', …)
zset.range('Region:_', …)
Bid
set.add('UserBids:_', …)
topk.add('ItemBids:_', …)
ViewAuction
dict.get('Item:_', …)
topk.top('ItemBids:_', …)

Figure 4.6. Overview of important Rubis transactions implemented with ADTs. Lines show conflicts between operations, many of which are either eliminated due to commutativity by boosting or mediated by phasing.

tions like BuzzFeed's viral dress.

4.5.2. RUBiS

The RUBiS benchmark [10] imitates an online auction service like the one described in §4.1.1. The 8 transaction types and their frequencies are shown in Table 4.3; `ViewAuction` and `Bid` dominate the workload. The benchmark specifies a workload consisting of a mix of these transactions and the average bids per auction. However, the distribution of bids (by item and time) was unspecified.

Our implementation models the bid distributions observed by subsequent studies [5, 6], with bids per item following a power law and the frequency of bids increasing exponentially at the end of an auction. Otherwise, we follow the parameters specified in [10]: 30,000 items, divided into 62 regions and 40 categories, with an average of 10 bids per item, though in our case this is distributed according to a zipfian with $\alpha = 1$.

Figure 4.7 shows results for two different workloads: read-heavy and

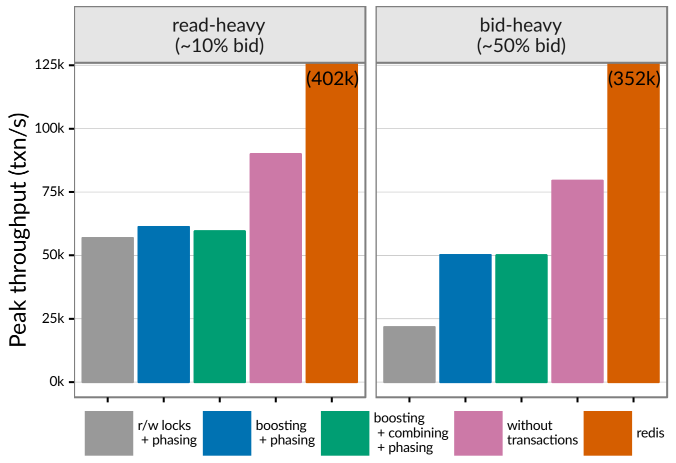


Figure 4.7. Peak throughput of Rubis. For read-heavy workloads, boosting makes little difference, but in times of high contention, during bidding spikes, commutativity between bids is essential to keep performance up.

bid-heavy. In the read-heavy workload, bids do not often come in at a high enough rate to require commutativity, so phasing alone suffices. However, during heavy bidding times, commutativity is essential: Claret maintains nearly the same throughput in this situation as the read-heavy workload, roughly 2x better than r/w locks and 68% of non-transactional performance. Considering the importance of getting bids correct, this seems an acceptable tradeoff. For comparison, we ran the same workload on Redis; its peak throughput was 400k transactions per second, a little over 4x the throughput of the Claret prototype. This is due to inefficiencies in the Claret's sorted set implementation as well as worse cache behavior.

Figure 4.6 predicted which conflicts should be affected by boosting; in Figure 4.8 we validate those predictions by plotting the actual number of conflicts for the most significant edges in the conflict graph. Using a log scale, it is apparent that boosting all but eliminates Bid-Bid conflicts, but Bid-View conflicts have gone up; now that there are more bids, the chances of conflicting with `ViewAuction` have increased. The introduc-

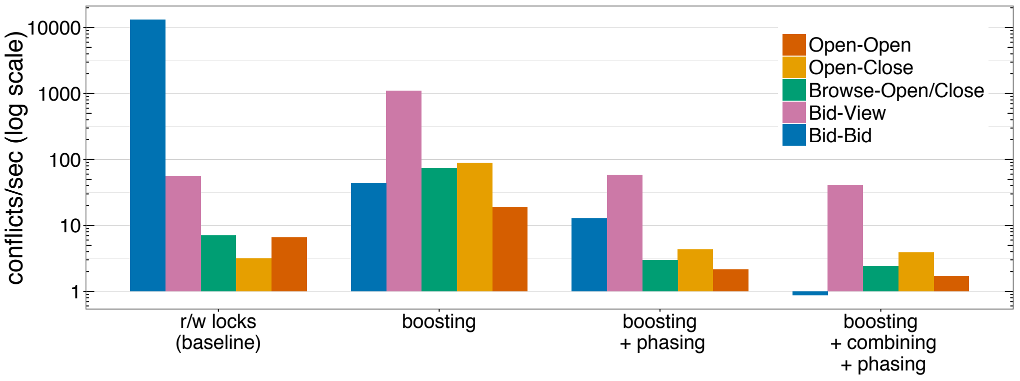


Figure 4.8. Breakdown of conflicts between Rubis transactions (minor contributors omitted) with 256 clients on bid-heavy workload (averaged). As predicted by Figure 4.6, boosting drastically reduces Bid-Bid conflicts, and phasing drastically reduces the remaining conflicts.

tion of phasing and combining eliminate much of the remaining conflicts.

Overall, we can see that boosting and phasing are crucial to achieving reasonable transaction performance in Rubis even during heavy bidding. If an auction service is unable to keep up with the rate of bidding, it will result in a loss of revenue and a lack of trust from users, so a system like Claret could prove invaluable to them.

4.5.3. Retwis

Retwis is a simplified Twitter clone designed originally for Redis [135]. Data structures such as lists and sets are used track each user's followers and posts and keep a materialized up-to-date timeline for each user (as a `zset`). It is worth noting that in our implementation, `Post` and `Repost` are each a single transaction, including appending to all followers' timelines, but when viewing timelines, we load each post in a separate transaction.

Retwis doesn't specify a workload, so we simulate a realistic workload using a synthetic graph with power-law degree distribution and a simple user model. We use the Kronecker graph generator from the

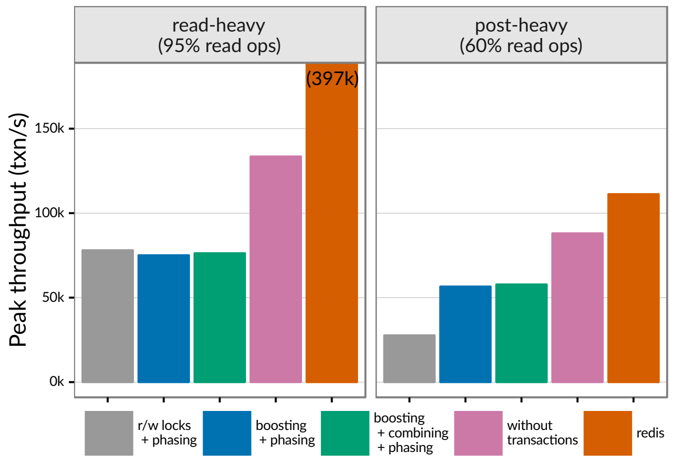


Figure 4.9. Peak throughput of Retwis. Read-heavy workload does not benefit from boosting, but during times of heavy posting, boosting is 2x better, still within 66% of non-transactional throughput. We also see that Redis write-heavy throughput is comparable to Claret's.

RUBiS	View/Browse	Bid	Open/Close	Comment	NewUser/ViewUser
bid-heavy	52%	45%	1%	1%	1%
read-heavy	82%	10%	4%	2%	2%
Retwis	Timeline	Repost	Post	Follow	NewUser
bid-heavy	82%	8%	6%	3%	<1%
read-heavy	91%	5%	2%	1%	1%

Table 4.3. Transaction mix for benchmark workloads.

Graph 500 benchmark [69], which is designed to produce the same power-law degree distributions found in natural graphs. These experiments generate a graph with approximately 65,000 users and an average of 16 followers per user.

Our simple model of user behavior determines when and which posts to repost. After each timeline action, we rank the posts by how many reposts they already have and repost the most popular ones with probability determined by a geometric distribution. The resulting distribution of reposts follows a power law, approximating the viral propagation effects observed in real social networks.

Figure 4.9 shows throughput on two workloads, listed in Table 4.3. The read-heavy workload models steady-state Twitter traffic, while the post-heavy workload models periods of above-average posting, such as during live events. We only show the results with phasing because the non-phasing baseline had too many failed transactions. On the read-heavy workload, r/w locks are able to keep up reasonably well; after all, reading timelines is easy as long as they do not change frequently. However, the post-heavy workload shows that when contention increases, the performance of r/w locks falls off much more drastically than with boosting. Combining even appears to pay off when there are enough clients to find matches.

Unlike auctions, many situations in Twitter are tolerant of minor inconsistencies, making it acceptable to implement without transactions in order to aid scalability. This tradeoff is clear when performance is as flat as the baseline performance is in these plots. However, with Claret's optimizations, it is able to achieve up to 82% of the non-transactional performance. In situations where inconsistent timelines are more likely to be noticed, such as conversations, it may be worth paying this overhead.

Evaluation summary. We find that leveraging commutativity via boosting and phasing is clearly beneficial under all of our simulated scenarios, showing greater benefit under more extreme contention resulting from high skew or heavy writing. Combining appears mostly ineffectual in our benchmarks, but does not hinder performance. In the most dire circumstances of extreme contention, having combining as an optional release valve for offloading work is useful. Moreover, these improvements come with a programming model largely identical to Redis's, which is sufficient for many applications that only require simple ADTs already built into Redis. The Claret prototype system is not as highly optimized as Redis itself, roughly 4x slower on read-heavy workloads and marginally slower on write-heavy workloads. However, its performance is similar enough that our findings should be expected to hold within Redis itself.

4.6. Related work

4.6.1. Improving Transaction Concurrency

Several recent systems have explored ways of exposing more concurrency between transactions. An old technique, transaction chopping [140], statically analyzes transactions and splits them to reduce the time locks are held for.

Recent systems Salt and Callas [163, 164] introduced ways of separating transactions that often conflict from the rest so they can be handled differently. In Salt, problematic transactions were rewritten to operate with weaker consistency, which Callas improved upon to maintain the same ACID properties by using runtime pipelining within a group. Boosting and phasing could both be applied within one of these groups along with runtime pipelining to expose concurrency among frequently conflicting transactions. It would be interesting to explore whether Claret's ADT knowledge could be used to inform Callas's grouping decisions, which are crucial to performance.

4.6.2. ADTs and Commutativity

The importance of commutativity is well known in databases and distributed systems. ADTs were first used in databases in the 1980s to support indices for custom data types [148, 149], and for concurrency control [61, 75, 138, 162]. That work introduced abstract locks to allow databases to leverage commutativity and also allowed user-defined types to express their abstract behavior to the database. Another classic system [147] used type-based locks in an early form of distributed transactions.

To improve scalability and flexibility, NoSQL stores gave up the knowledge afforded by predefined schemas, but recent work has shown how these systems can still leverage commutativity. Lynx [167] allows users to annotate parts of transactions as commutative. HyFlow [92] combines multi-versioning and commutativity to reorder commutative transactions before others, but requires annotating entire transactions as commutative. ADTs naturally express commutativity as part of application design and abstract locks expose more fine-grained concurrency.

Doppel [114], a multicore in-memory database, allows commutative operations on highly contended records to be performed in parallel *phases* with a technique called *phase reconciliation*; Claret's *phasers* are more general in that they expose pairwise operation commutativity on arbitrary ADTs, but do not parallelize across cores.

In eventually consistent datastores, commutativity can improve convergence. RedBlue consistency [99] exploits the convergence of commutative "blue" operations to avoid coordination. Conflict-free (or convergent) replicated data types (CRDTs/CvRDTs) [139] force operations to commute by defining merge functions that resolve conflicts automatically and have been implemented for production in Riak [22]. Claret's strictly linearizable model exposes concurrency without relaxing consistency because CRDT behavior is often still counterintuitive.

4.7. Discussion

Claret mitigates contention in real-world workloads by allowing programmers to expose application-level semantics with ADTs, which the datastore leverages in the transaction protocol and client library. These optimizations lead to significant speedups for transactions in high contention scenarios, without hurting performance on lighter workloads, making them competitive with non-transactional performance.

Our prototype implementation has significantly lower throughput than Redis, especially on read-heavy workloads. This is largely due to the heavily optimized Redis server, written in C, employs aggressive tactics to prevent unnecessary dynamic allocation, optimize locality, and accelerate serialization, as well as highly tuned implementations of the core data structures. The non-transactional Claret performance numbers give an idea of the raw difference in performance between the two systems. Our experiments showed that the average latency of completing transactions is similar. The majority of the difference in performance between Claret and Redis appears to be in the implementation of the core data structures like the `ScoredSet` compared with Redis's `zset` implementation, with some additional overhead in Claret's Protobuf-based serialization. We hypothesize that if Redis itself was extended with Claret's boosting, combining, and phasing optimizations, then it would exhibit a similar performance pattern, where distributed transactions would have a roughly 20-30% performance penalty. This would be significantly better than Redis's existing transaction support, which is OCC-based and only works for a single Redis server. Preliminary evaluation of an approach to add Claret's features to Redis using its Lua scripting capability has shown promise.

We have seen that exposing already-existing concurrency between transactions can drastically improve performance. However, there is a limit to the amount of true concurrency that is available in these applications. For example, though retweets commute with one another, if a Twitter user *views* the retweet count, that must be serialized with respect to retweets. If the application could express where approximation or inconsistency is acceptable, e.g., where it can tolerate an imprecise retweet count, then even more concurrency would be available. The next chapter introduces a new programming model that lets programmers express error tolerances, and uses types to ensure that they deal with the ramifications.

5. Disciplined Inconsistency: Safely trading off consistency for performance

5.1. Introduction

To provide good user experiences, modern datacenter applications and web services must balance the competing requirements of application correctness and responsiveness. For example, a web store double-charging for purchases or keeping users waiting too long (each additional millisecond of latency [64, 103]) can translate to a loss in traffic and revenue. Worse, programmers must maintain this balance in an unpredictable environment where a black and blue dress [118] or Justin Bieber [108] can change application performance in the blink of an eye.

In order to meet these performance requirements, distributed systems programmers must routinely make tradeoffs between consistency and performance [17, 34, 67]. Recognizing this, many existing storage systems support configurable consistency levels that allow programmers to set the consistency of individual operations [11, 22, 99, 146]. These allow programmers to weaken consistency guarantees only for data that is not critical to application correctness, retaining strong consistency for vital data. Some systems further allow adaptable consistency levels at runtime, where guarantees are only weakened when necessary to meet availability or performance requirements (e.g., during a spike in traffic or datacenter failure) [150, 152]. Unfortunately, using these systems correctly is challenging. Programmers can inadvertently update strongly consistent data in the storage system using values read from weakly consistent operations, propagating inconsistency and corrupting stored data.

Over time, this *undisciplined* use of data from weakly consistent operations lowers the consistency of the storage system to its weakest level.

This chapter proposes a more disciplined approach to inconsistency through the *Inconsistent, Performance-bound, Approximate (IPA)* storage system. IPA introduces the following concepts:

- *Consistency Safety*, a new property that ensures that values from weakly consistent operations cannot flow into stronger consistency operations without explicit endorsement from the programmer. IPA is the first storage system to provide consistency safety.

- *Consistency Types*, a new type system in which *type safety implies consistency safety*. Consistency types define the consistency and correctness of the returned value from every storage operation, allowing programmers to reason about their use of different consistency levels. IPA's type checker enforces the disciplined use of IPA consistency types statically at compile time.

- *Error-bounded Consistency.* IPA is a data structure store, like Redis [135] or Riak [22], allowing it to provide a new type of weak consistency that places *numeric error bounds* on the returned values. Within these bounds, IPA automatically adapts to return the strongest IPA consistency type possible under the current load.

We implemented an IPA prototype based on Scala and Cassandra and show that IPA allows the programmer to trade off performance and consistency, safe in the knowledge that the type system has checked the program for consistency safety. We demonstrate experimentally that these mechanisms allow applications to dynamically adapt correctness and performance to changing conditions with three applications: a simple counter, a Twitter clone based on Retwis [136] and a Ticket sales service modeled after FusionTicket [1].

5.2. Trading off consistency (in the past)

Before delving into what can go wrong with inconsistency, we must first introduce the concepts behind replication and consistency, and describe some of the techniques previously available to programmers for trading off consistency. Distributed systems often employ data replication as a way to improve availability and fault tolerance. With multiple copies of data, some of them may be lost or corrupted, but the true original data can still be recovered. However, this requires keeping these copies up to date with each other. How these replicas are kept synchronized with each other, which replicas clients are required to coordinate with, and what order operations will appear to execute in, are all part of the *consistency model*.

5.2.1. The consistency model zoo

Consistency models are analogous to memory models from computer architecture; they define the allowable reorderings and visibility of operations in a distributed system, but they differ in one crucial way: *consistency models expose the existence of replication*. Due to the reliability and speed of CPU cache hierarchies, memory models can afford to assume that *coherence* will enforce the illusion of one copy of memory. In distributed systems, where failure is a real possibility and synchronization is expensive, it is often necessary to expose replication in the consistency model. This makes them significantly more difficult to reason about — as if memory models were not complex enough as it is — and has led to a minor Cambrian explosion of consistency models.

The strongest consistency model, *strict serializability* (roughly defined as Lamport's *sequential consistency* [96] combined with Herlihy's *linearizability*[5] [76]) guarantees that operations appear to occur in a global serial order that all observers agree on and that corresponds to real time. This, and any form of consistency that requires enforcing a *global total order* is theoretically impossible to enforce with *high availability* due to the possibility of network partitions (this is the essence of the CAP theorem [34, 67]). In practice, strict serializability may not be wholly impractical for the average case, but ensuring it in *all* cases is prohibitively expensive.

At the other extreme, *eventual consistency*, the least common denominator among consistency models, simply guarantees that if update operations stop occurring, all replicas will eventually reflect the same state [158].

[5]Not to be confused with *lionizable* (fictional), which would definitely be the king of the zoo.

Under this model, programmers cannot count on subsequent operations reflecting the same state, because those operations could go to any replica at any time, and those replicas are continuously receiving updates from other nodes.

There are a whole family of models similar to eventual consistency which add various ordering constraints:

- *Monotonic writes* (MW) ensure that writes from a client are serialized, enforcing *ordering* between writes.
- *Monotonic reads* (MR) ensures that reads will not observe earlier values than have been seen by a particular client already, strengthening visibility.
- *Read-your-writes* (RYW) ensures that a client will at least observe its own effects, primarily strengthening one aspect of visibility.
- *Causal consistency*[6] ensures that operations from different clients causally following a write will observe that write (by some definition of *causation* which the system must track). This means that operations will be *visible to* and *ordered with* each other when applicable.

Each of the above models restricts *ordering* and *visibility* differently, making some cases easier for programmers to reason about, while reducing the flexibility and therefore performance of the system. For instance, some require *sticky sessions* [151], which forces clients to continue communicating with a particular replica, even if it is not the fastest, or lowest latency, or most up-to-date one available. Alternately, the Conit consistency model [166] breaks down the consistency spectrum into numerical error, order error, and staleness, permitting a combinatorial number of variants.

Several datastores allow consistency levels to be specified on a per-operation basis: research systems Gemini [99] (RedBlue consistency), and Walter [146], and production systems Cassandra [11] and Riak [22]. However, they leave programmers to determine where to use stronger consistency in order to achieve their correctness goals, a very error-prone task. Recent work has explored ways of *automatically* choosing the correct consistency level or coordination strategy based on annotations.

[6]Inattentive or lazy typing can lead to the variant, *casual* consistency, inadvertently favored by some datastores which shall remain nameless.

5.2.2. Annotating ordering constraints

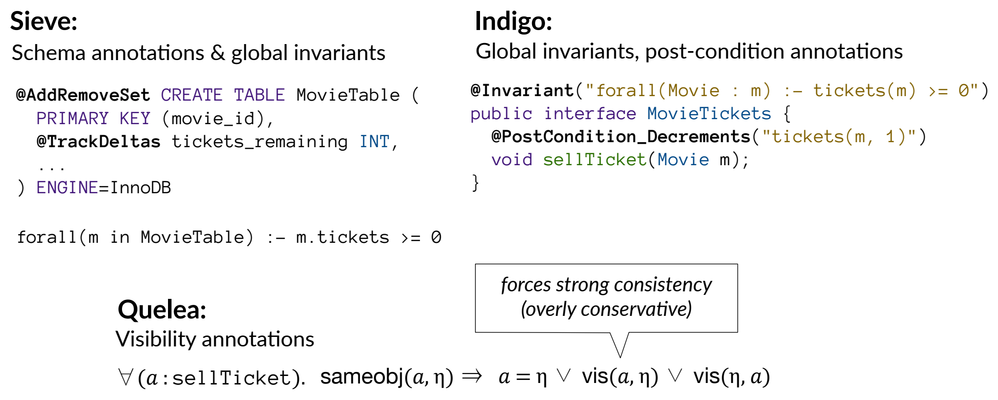
Rather than choosing consistency levels manually, some programming models have been proposed that allow applications to specify their desired ordering constraints in better ways.

Sieve [100] builds on top of Gemini, automatically determining how to implement the desired semantics with causal consistency and adding additional synchronization wherever strong consistency is needed. It relies on programmer-specified global invariants and annotations on the relational database schema to select the desired merge semantics in case of conflicts. These annotations echo the variants of CRDTs (see §5.2.4).

Quelea [144] has programmers write *contracts* to describe ordering constraints between operations and then automatically selects the correct consistency level for each operation to satisfy all of the contracts. Contracts are specified in terms of low-level consistency primitives such as *visibility* and *session order*. For example, to ensure a non-negative bank account balance, a contract indicates that all `withdraw` operations must be visible to one another, forcing the operation to be executed with sequential consistency. Because correctness properties are specified *independent of a particular consistency model*, or set of consistency levels, they are *composable* with each other and *portable* to other datastores supporting different consistency options. However, the low-level primitives used in contracts may not be intuitive for programmers and still require reasoning about all the possible anomalies between operations. These primitives are unable to capture other forms of coordination, sometimes leading to more conservative ordering constraints than necessary.

Indigo [19] takes a different approach to expressing application requirements: instead of specifying visibility and ordering constraints, programmers write *invariants over abstract state and state transitions*, and annotate *post-conditions* on actions to express their side-effects in terms of the abstract state. They then perform a static analysis to determine where concurrent execution could violate the invariants and add coordination logic to avoid those conflicts. Supported constraints include numeric constraints, such as lower or upper bounds on counts, as well as integrity constraints and general compositions of these.

Figure 5.1 shows how annotations would be written in these three


Sieve:
Schema annotations & global invariants
Indigo:
Global invariants, post-condition annotations
Quelea:
Visibility annotations
forces strong consistency (overly conservative)

```
@AddRemoveSet CREATE TABLE MovieTable (
  PRIMARY KEY (movie_id),
  @TrackDeltas tickets_remaining INT,
  ...
) ENGINE=InnoDB

forall(m in MovieTable) :- m.tickets >= 0
```

```
@Invariant("forall(Movie : m) :- tickets(m) >= 0")
public interface MovieTickets {
  @PostCondition_Decrements("tickets(m, 1)")
  void sellTicket(Movie m);
}
```

$\forall(a:\texttt{sellTicket}).\ \text{sameobj}(a,\eta) \Rightarrow a=\eta \vee \text{vis}(a,\eta) \vee \text{vis}(\eta,a)$

Figure 5.1. *Annotating application invariants.* Annotations are used to determine where coordination is necessary and what consistency is required to enforce it.

systems to implement our running ticket sales example. In this case, the desired invariant is that tickets are not over-sold – that is, the count of remaining tickets should be non-negative. Sieve and Indigo can enforce this invariant directly as written. Quelea's visibility-based contracts cannot tightly describe this invariant; instead, they must be conservative and force ticket sales to be strongly consistent. These approaches provide some ways to express the orderings required by the application and let the system enforce them. However, these assume that everything in applications needs to always be correct, and do not take performance needs into consideration.

5.2.3. Expressing performance targets

A long history of systems have been built around the principle that applications may be willing to tolerate slightly stale data in exchange for improved performance, including databases [27, 124, 127, 131] and distributed caches [120, 125]. These systems generally require developers to explicitly specify staleness bounds on each transaction in terms of absolute time (although Bernstein et al.'s model can generate these from error bounds when a value's maximum rate of change is known). Other systems, including PRACTI [24], PADS [25], and WheelFS [150], have given developers ways of expressing their desired performance and correctness requirements through *semantic cues* and policies.

One system of particular interest, called Pileus, supports *consistency-based service-level agreements (SLAs)* [152]. Consistency SLAs specify a target latency and consistency level (e.g. 100 ms with read-my-writes). Each operation specifies a set of desired SLAs associated with relative *utility* scores. The runtime monitors the performance of the system, predicts which SLAs are likely to be met, and chooses which to attempt in order to maximize utility. By monitoring current performance, Pileus attempts to predict which SLA to target to maximize utility, typically to achieve the best consistency possible within a certain latency. It then returns both the value and the achieved consistency to the application, which they can choose to take into account when using the value. Allowing users to specify their desired latencies and consistencies directly to the system is powerful. However, because it is so fine-grained, the burden of choosing target latencies and consistency for each operation could be quite high, and it is difficult to compose a sequence of operations and SLAs to achieve an overall target latency or correctness criteria.

5.2.4. Restricting value uncertainty

Convergent (or *conflict-free*) *replicated data types* (CRDTs) [139] are data types that have commutative merge functions defined for them. Resolving conflicts deterministically requires making choices about the semantics of concurrent updates, leading to a proliferation of CRDTs for various use cases. Even simple data structures like Sets have multiple variants that resolve non-commuting operations differently, e.g., a "grow-only set" (*G-Set*) where `remove` is simply disallowed, or an "observed-remove set" (*OR-Set*) where `add` wins over `remove` when causally concurrent.

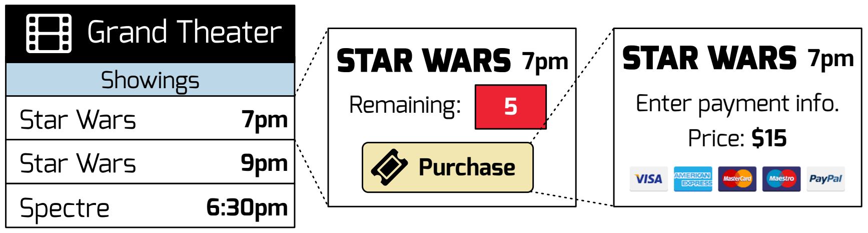
CRDTs are essential for handling concurrent updates to data structures; without them it is very difficult to predict the final state of eventually consistent data. Riak [22] implements several data types and encourages their use. Like ADTs, CRDTs also provide a well-defined set of possible values that a variable can hold, restricted to changes made by supported operations. This is a clear advantage over simple *registers* that are completely overwritten on each write, making them unpredictable when the order of updates is uncertain. CRDTs can still suffer from many of the effects of eventual consistency, such as temporary divergence when accessing different replicas, and there is still no guarantee of timeliness of update propagation, so they could stay divergent for some time.

Bloom [8, 50] is a language and programming paradigm that can be viewed as a generalization of CRDTs. The *CALM principle* (Consistency And Logical Monotonicity) underlying this work formalized the requirements for an entire program to be eventually consistent, obviating any need for coordination. It is built around the notion of monotonicity — programs compute sets of facts that grow over time so that information is never lost and convergence can be guaranteed. In practice, some coordination is needed to allow sequential reasoning in places, which the Bloom language can determine automatically. These facts can be encoded as sets or other collections with suitable *merge functions*, such as CRDTs.

These models can help programmers stay sane in the dangerous realm of eventual consistency. However, they cannot completely replace the benefits of strong consistency and linearizability. Interleaving eventually consistent CRDTs with strongly consistent operations can still lead to confusing inconsistencies and integrity constraint violations. The next section describes some situations where this can be problematic, motivating the need for safer ways to compose consistency models.

5.3. The case for consistency safety

Unpredictable Internet traffic and unexpected failures force modern datacenter applications to trade off consistency for performance. In this section, we demonstrate the pitfalls of doing so in an undisciplined way. As an example, we describe a movie ticketing service, similar to AMC or Fandango. Because ticketing services process financial transactions, they must ensure correctness, which they can do by storing data in a strongly consistent storage system. Unfortunately, providing strong consistency for every storage operation can cause the storage system and application to collapse under high load, as several ticketing services did in October



```
// adjust price based on number of tickets left
def computePrice(ticketsRemaining: Int): Float

// called from purchaseTicket & displayEvent
def getTicketCount(event: UUID): Int =
  // use weak consistency for performance
  readWeak(event+"ticket_count")

def purchaseTicket(event: UUID) = {
  val ticket = reserveTicket(event)
  val remaining = getTicketCount(event)
  // compute price based on inconsistent read
  val price = computePrice(remaining)
  display("Enter payment info. Price: ", price)
}
```

Figure 5.2. Ticket sales service. To meet a performance target in `displayEvent`, developer switches to a weak read for `getTicketCount`, not realizing that this inconsistent read will be used elsewhere to compute the ticket price.

2015, when tickets became available for the new Star Wars movie [53].

To allow the application to scale more gracefully and handle traffic spikes, the programmer may chose to weaken the consistency of some operations. As shown in Figure 5.2, the ticket application displays each showing of the movie along with the number of tickets remaining. For better performance, the programmer may want to weaken the consistency of the read operation that fetches the remaining ticket count to give users an estimate, instead of the most up-to-date value. Under normal load, even with weak consistency, this count would often still be correct because propagation is typically fast compared to updates. However, eventual consistency makes no guarantees, so under heavier traffic spikes, the values could be significantly incorrect and the application has no way of knowing by how much.

While this solves the programmer's performance problem, it introduces a data consistency problem. Suppose that, like Uber's surge pricing, the ticket sales application wants to raise the price of the last 100 tickets for each showing to \$15. If the application uses a strongly consistent read to fetch the remaining ticket count, then it can use that value to compute the price of the ticket on the last screen in Figure 5.2. However, if the programmer reuses `getTicketCount`, which used a weak read to calculate the price, then this count could be arbitrarily wrong. The application could then over- or under-charge some users depending on the consistency of the returned value. Worse, the theater expects to make \$1500 for those tickets with the new pricing model, which may not happen with the new weaker read operation. Thus, programmers need to be careful in their use of values returned from storage operations with weak consistency. Simply weakening the consistency of an operation may lead to unexpected consequences for the programmer (e.g., the theater not selling as many tickets at the higher surge price as expected).

In this work, we propose a programming model that can prevent using inconsistent values where they were not intended, as well as introduce mechanisms that allow the storage system to dynamically adapt consistency within predetermined performance and correctness bounds.

5.4. IPA Programming Model

We propose a programming model for distributed data that uses types to control the consistency–performance trade-off. The *Inconsistent, Performance-bound, Approximate* (IPA) type system helps developers trade consistency for performance in a disciplined manner. This section presents the IPA programming model, including the available consistency policies and the semantics of operations performed under the policies. §5.5 will explain how the type system's guarantees are enforced.

5.4.1. Overview

The IPA programming model consists of three parts:

ADT Method \ Policies:	`Consistency(Weak)`	`LatencyBound(_)`	`ErrorTolerance(_)`
`Counter``.read()`	`Inconsistent[Int]`	`Rushed[Int]`	`Interval[Int]`
`Set``.size()`	`Inconsistent[Int]`	`Rushed[Int]`	`Interval[Int]`
`Set``.contains(x)`	`Inconsistent[Bool]`	`Rushed[Bool]`	N/A
`List[T]``.range(x,y)`	`Inconsistent[List[T]]`]'	`Rushed[List[T]]`	N/A
`UUIDPool``.take()`	`Inconsistent[UUID]`	`Rushed[UUID]`	N/A
`UUIDPool``.remain()`	`Inconsistent[Int]`	`Rushed[Int]`	`Interval[Int]`

Table 5.1. *Consistency policies* determine the *consistency type* returned by ADT operations.

- Abstract data types (ADTs) implement common data structures (such as `Set[T]`) on distributed storage.
- Consistency policies on ADTs specify the desired consistency level for an object in application-specific terms (such as latency or accuracy bounds).
- Consistency types track the consistency of operation results and enforce consistency safety by requiring developers to consider weak outcomes.

Programmmers annotate ADTs with consistency policies to choose their desired level of consistency. The *consistency policy* on the *ADT operation* determines the *consistency type* of the result. Table 5.1 shows some examples; the next few sections will introduce each of the policies and types in detail. Together, these three components provide two key benefits for developers. First, the IPA type system enforces *consistency safety*, tracking the consistency level of each result and preventing inconsistent values from flowing into consistent values. Second, the programming interface enables performance–correctness trade-offs, because consistency policies on ADTs allow the runtime to select a consistency level for each individual operation that maximizes performance in a constantly changing environment. Together, these systems allow applications to adapt to changing conditions with the assurance that the programmer has expressed how it should handle varying consistency.

5.4.2. Abstract Data Types

The base of the IPA type system is a set of abstract data types (ADTs) for distributed data structures. ADTs present a clear abstract model through a set of operations that query and update state, allowing users and systems alike to reason about their logical, algebraic properties rather than the low-level operations used to implement them. Though the simplest key-value stores only support primitive types like strings for values, many popular datastores have built-in support for more complex data structures such as sets, lists, and maps. However, the interface to these datatypes differs: from explicit sets of operations for each type in Redis, Riak, and Hyperdex [22, 58, 85, 135] to the pseudo-relational model of Cassandra [95]. IPA's extensible library of ADTs allows it to decouple the semantics of the type system from any particular datastore, though our reference implementation is on top of Cassandra, similar to [144].

Besides abstracting over storage systems, ADTs are an ideal place from which to reason about consistency and system-level optimizations. The consistency of a read depends on the write that produced the value. Annotating ADTs with consistency policies ensures the necessary guarantees for all operations are enforced, which we will expand on in the next section.

Custom ADTs can express application-level correctness constraints. IPA's `Counter` ADT allows reading the current value as well as increment and decrement operations. In our ticket sales example, we must ensure that the ticket count does not go below zero. Rather than forcing all operations on the datatype to be linearizable, this application-level invariant can be expressed with a more specialized ADT, such as a `BoundedCounter`, giving the implementation more latitude for enforcing it. IPA's library is *extensible*, allowing custom ADTs to build on common features; see §5.6.3.

5.4.3. Consistency Policies

Previous systems [11, 22, 99, 146, 152] require annotating each read and write operation with a desired consistency level. This per-operation approach complicates reasoning about the safety of code using weak consistency, and hinders global optimizations that can be applied if the system

knows the consistency level required for future operations. The IPA programming model provides a set of consistency policies that can be placed on *ADT instances* to specify consistency properties for the lifetime of the object.
Consistency policies come in two flavors: static and dynamic.

Static policies are fixed, such as `Consistency(Strong)` which states that operations must have strongly consistent behavior. Static annotations provide the same direct control as previous approaches but simplify reasoning about correctness by applying them globally on the ADT.

Dynamic policies specify a consistency level in terms of application requirements, allowing the system to decide at runtime how to meet the requirement for each executed operation. IPA offers two dynamic consistency policies:

- A latency policy `LatencyBound(x)` specifies a target latency for operations on the ADT (e.g., 20 ms). The runtime can choose the consistency level for each issued operation, optimizing for the strongest level that is likely to satisfy the latency bound.
- An accuracy policy `ErrorTolerance(x%)` specifies the desired accuracy for read operations on the ADT. For example, the `size` of a `Set` ADT may only need to be accurate within 5% tolerance. The runtime can optimize the consistency of write operations so that reads are guaranteed to meet this bound.

Dynamic policies allow the runtime to extract more performance from an application by relaxing the consistency of individual operations, safe in the knowledge that the IPA type system will enforce safety by requiring the developer to consider the effects of weak operations.

Static and dynamic policies can apply to an entire ADT instance or on individual methods. For example, one could declare `List[Int] with LatencyBound(50 ms)`, in which case all read operations on the list are subject to the bound. Alternatively, one may wish to declare a `Set` with relaxed consistency for its `size` but strong consistency for its `contains` predicate. The runtime is responsible for managing the interaction between these policies. In the case of a conflict between two bounds, the system can be conservative and choose stronger policies than specified without affecting correctness.

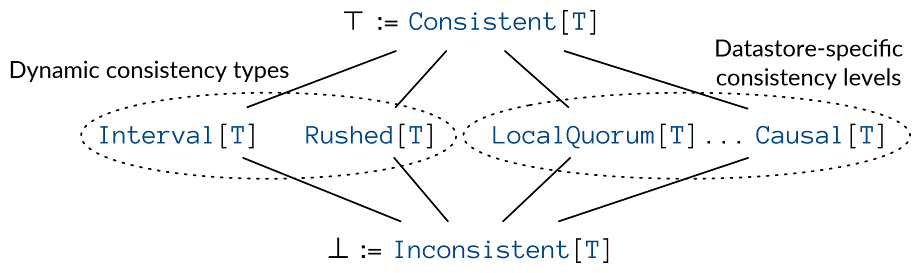

⊤ := Consistent[T]
Dynamic consistency types
Datastore-specific consistency levels
Interval[T]
Rushed[T]
LocalQuorum[T] . . . Causal[T]
⊥ := Inconsistent[T]

Figure 5.3. IPA Type Lattice parameterized by a type `T`.

In the ticket sales application, the `Counter` for each event's tickets could have a relaxed accuracy policy, `ErrorTolerance(5%)`, allowing the system to quickly read the count of tickets remaining. An accuracy policy is appropriate here because it expresses a domain requirement—users want to see accurate ticket counts. As long as the system meets this requirement, it is free to relax consistency and maximize performance without violating correctness. The `List` ADT used for events has a latency policy that also expresses a domain requirement—that pages on the website load in reasonable time.

5.4.4. Consistency Types

The key to consistency safety in IPA is the consistency types—enforcing type safety directly enforces consistency safety. Read operations of ADTs annotated with consistency policies return instances of a *consistency type*. These consistency types track the consistency of the results and enforce a fundamental non-interference property: results from weakly consistent operations cannot flow into computations with stronger consistency without explicit endorsement. This could be enforced dynamically, as in dynamic information flow control systems, but the static guarantees of a type system allow errors to be caught at compile time.

The consistency types encapsulate information about the consistency achieved when reading a value. Formally, the consistency types form a lattice parameterized by a primitive type `T`, shown in Figure 5.3. Strong read operations return values of type `Consistent[T]` (the top element), and so (by implicit cast) behave as any other instance of type `T`. Intuitively, this equivalence is because the results of strong reads are known to be consistent, which corresponds to the control flow in conventional (non-distributed) applications. Weaker read operations return values of some type lower in the lattice (*weak consistency types*), reflecting their possible inconsistency. The bottom element `Inconsistent[T]` specifies an object with the weakest possible (or unknown) consistency. The other consistency types are partially ordered as shown in Figure 5.3, according to the subtyping relation $\prec$ defined by:

$$\frac{\tau \text{ is weaker than } \tau'}{\tau'[T] \prec \tau[T]}$$

The only possible operation on `Inconsistent[T]` is to *endorse* it. Endorsement is an upcast, invoked by `endorse(x)`, to the top element `Consistent[T]` from other types in the lattice:

$$\frac{\Gamma \vdash e_1 : \tau[T] \qquad T \prec \tau[T]}{\Gamma \vdash \texttt{endorse}(e_1) : T}$$

The core type system statically enforces safety by preventing weaker values from flowing into stronger computations. Forcing developers to explicitly endorse inconsistent values prevents them from accidentally using inconsistent data where they did not determine it was acceptable, essentially inverting the behavior of current systems where inconsistent data is always treated as if it was safe to use anywhere. However, endorsing values blindly in this way is not the intended use case; the key productivity benefit of the IPA type system comes from the other consistency types which correspond to the dynamic consistency policies in §5.4.3 which allow developers to handle dynamic variations in consistency, which we describe next.

5.4.5. Rushed types

The weak consistency type `Rushed[T]` is the result of read operations performed on an ADT with consistency policy `LatencyBound(x)`. `Rushed[T]` is a *sum* (or *union*) *type*, with one variant per consistency level available to the implementation of `LatencyBound`. Each variant is itself a consistency type (though the variants obviously cannot be `Rushed[T]` itself). The effect is that values returned by a latency-bound object carry with them their actual consistency level. A result of type `Rushed[T]` therefore requires the developer to consider the possible consistency levels of the value.

For example, a system with geo-distributed replicas may only be able to satisfy a latency bound of 50 ms with a local quorum read (that is, a quorum of replicas within a single datacenter). In this case, `Rushed[T]` would be the sum of three types `Consistent[T]`, `LocalQuorum[T]`, and `Inconsistent[T]`. A match statement destructures the result of a latency-bound read operation:

```
set.contains() match {
  case Consistent(x) => print(x)
  case LocalQuorum(x) => print(x+", locally")
  case Inconsistent(x) => print(x+"???")
}
```

The application may want to react differently to a local quorum as opposed to a strongly or weakly consistent value. Note that because of the subtyping relation on consistency types, omitted cases can be matched by any type lower in the lattice, including the bottom element `Inconsistent(x)`; other cases therefore need only be added if the application should respond differently to them. This subtyping behavior allows applications to be portable between systems supporting different forms of consistency (of which there are many).

5.4.6. Interval types

Tagging values with a consistency level is useful because it helps programmers tell which operation reorderings are possible (e.g. strongly consistent operations will be observed to happen in program order). However, accuracy policies provide a different way of dealing with inconsistency by expressing it in terms of value uncertainty. They require knowing the *abstract behavior* of operations in order to determine the change in abstract state which results from each reordered operation (e.g., reordering increments on a Counter has a known effect on the value of reads).

The weak consistency type `Interval[T]` is the result of operations performed on an ADT with consistency policy `ErrorTolerance(x%)`. `Interval[T]` represents an interval of values within which the true (strongly consistent) result lies. The interval reflects uncertainty in the true value created by relaxed consistency, in the same style as work on approximate computing [31].

The key invariant of the `Interval` type is that the interval must include the result of some linearizable execution. Consider a `Set` with 100 elements. With linearizability, if we `add` a new element and then read the `size` (or if this ordering is otherwise implied), we *must* get 101 (provided no other updates are occurring). However, if `size` is annotated with `ErrorTolerance(5%)`, then it could return any interval that includes 101, such as $[95, 105]$ or $[100, 107]$, so the client cannot tell if the recent `add` was included in the size. This frees the system to optimize to improve performance, such as by delaying synchronization. While any partially-ordered domain could be represented as an interval (e.g., a Set with partial knowledge of its members), in this work we consider only numeric types.

In the ticket sales example, the counter ADT's accuracy policy means that reads of the number of tickets return an `Interval[Int]`. If the entire interval is above zero, then users can be assured that there are sufficient tickets remaining. In fact, because the interval could represent many possible linearizable executions, in the absence of other user actions, a subsequent purchase must succeed. On the other hand, if the interval overlaps with zero, then there is a chance that tickets could already be sold out, so users could be warned. Note that ensuring that tickets are not over-sold is a separate concern requiring a different form of enforcement, which we describe in §5.6.3. The relaxed consistency of the interval type allows the system to optimize performance in the common case where there are many tickets available, and dynamically adapt to contention when the ticket count diminishes.

5.4.7. Lower bounds

Weak consistency types enforce consistency safety by ensuring developers address the worst case results of weak consistency. However, the weak consistency types are *lower bounds* on weakness: one valid implementation of a system using IPA types is to always return strongly consistency values. Moreover, the runtime guarantees that if every value returned has strong consistency, then the execution is linearizable, as if the system were strongly consistent from the outset.

5.5. Enforcing consistency policies

The consistency policies introduced in the previous section allow programmers to describe application-level correctness properties. Static consistency policies (e.g. Strong) are enforced by the underlying storage system; the annotated ADT methods simply set the desired consistency level when issuing requests to the store. The dynamic policies each require a new runtime mechanism to enforce them: parallel operations with latency monitoring for latency bounds, and reusable reservations for error tolerance. But first, we briefly review consistency in Dynamo-style replicated systems.

To be sure of seeing a particular write, *strong* reads must coordinate with a majority (*quorum*) of replicas and compare their responses. For a write and read pair to be *strongly consistent* (in the CAP sense [34]), the replicas acknowledging the write (W) plus the replicas contacted for the read (R) must be greater than the total number of replicas ($W + R > N$). This can be achieved, for example, by writing to a quorum ($(N+1)/2$) and reading from a quorum (`QUORUM` in Cassandra), or writing to N (`ALL`) and reading from 1 (`ONE`) [54]. To support the `Consistency(Strong)` policy, the designer of each ADT must choose consistency levels for its operations which together enforce strong consistency.

5.5.1. Latency bounds

The time it takes to achieve a particular level of consistency depends on current conditions and can vary over large time scales (minutes or hours) but can also vary significantly for individual operations. During normal operation, strong consistency may have acceptable performance while at peak traffic times the application would fall over. Latency bounds specified by the application allow the system to *dynamically* adjust to maintain comparable performance under varying conditions.

Our implementation of latency-bound types takes a generic approach: it issues read requests at different consistency levels in parallel. It composes the parallel operations and returns a result either when the strongest operation returns, or with the strongest available result at the specified time limit. If no responses are available at the time limit, it waits for the first to return.

This approach makes no assumptions about the implementation of read operations, making it easily adaptable to different storage systems. Some designs may permit more efficient implementations: for example, in a Dynamo-style storage system we could send read requests to all replicas, then compute the most consistent result from all responses received within the latency limit. However, this requires deeper access to the storage system implementation than is traditionally available.

5.5.1.1. Monitors

The main problem with our approach is that it wastes work by issuing parallel requests. Furthermore, if the system is responding slower due to a sudden surge in traffic, then it is essential that our efforts not cause additional burden on the system. In these cases, we should back off and only attempt weaker consistency. To do this, the system monitors current traffic and predicts the latency of different consistency levels.

Each client in the system has its own Monitor (though multi-threaded clients can share one). The monitor records the observed latencies of reads, grouped by operation and consistency level. The monitor uses an exponentially decaying reservoir to compute running percentiles weighted toward recent measurements, ensuring that its predictions continually adjust to current conditions.

Whenever a latency-bound operation is issued, it queries the monitor to determine the strongest consistency likely to be achieved within the time bound, then issues one request at that consistency level and a backup at the weakest level, or only weak if none can meet the bound. In §5.7.2.1 we show empirically that even simple monitors allow clients to adapt to changing conditions.

5.5.2. Error bounds

Enforcement of error bounds is built on the concepts of *escrow* and *reservations* [66, 121, 126, 129]. These techniques have been used in storage systems to enforce hard limits, such as an account balance never going negative, while permitting concurrency. The idea is to set aside a pool of permissions to perform certain update operations (we'll call them *reservations* or *tokens*), essentially treating *operations* as a manageable resource. If we have a counter that should never go below zero, there could be a number of *decrement* tokens equal to the current value of the counter. When a client wishes to decrement, it must first acquire sufficient tokens before performing the update operation, whereas increments produce new tokens. The insight is that the coordination needed to ensure that there are never too many tokens can be done *off the critical path*: tokens can be produced lazily if there are enough around already, and most importantly for this work, they can be *distributed* among replicas. This means that replicas can perform some update operations safely without coordinating with any other replicas.

5.5.2.1. Reservation Server

Reservations require mediating requests to the datastore to prevent updates from exceeding the available tokens. Furthermore, each server must locally know how many tokens it has without synchronizing. We are not aware of a commercial datastore that supports custom mediation of requests and replica-local state, so we need a custom middleware layer to handle reservation requests, similar to other systems which have built stronger guarantees on top of existing datastores [18, 20, 144].

Any client requests requiring reservations are routed to one of a number of *reservation servers*. These servers then forward operations when permitted along to the underlying datastore. All persistent data is kept in the backing store; these reservation servers keep only transient state tracking available reservations. The number of reservation servers can theoretically be decoupled from the number of datastore replicas; our implementation simply colocates a reservation server with each datastore server and uses the datastore's node discovery mechanisms to route requests to reservation servers on the same host.

5.5.2.2. Enforcing error bounds

Reservations have been used previously to enforce hard global invariants in the form of upper or lower bounds on values [20], integrity constraints [19], or logical assertions [104]. However, enforcing error tolerance bounds presents a new design challenge because the bounds are constantly shifting. Consider a `Counter` with a 10% error bound, shown in Figure 5.4. If the current value is 100, then 10 increments can be done before synchronization is required. However, we have 2 reservation servers, so these 10 reservations are distributed among them, allowing each to do some increments without synchronizing. As long as only 10 outstanding increments are allowed anywhere, reads are guaranteed to see values within 10% of the correct value.

If there are no tokens available locally, update operations like increment must block until tokens become available. Once replicas have synchronized updates with each other, the consumed tokens get returned to the reservation servers. In practice, usually the reservation servers force a synchronization — in Cassandra this can be done with a strong read (`ALL`) which performs read repair — ensuring that all replicas agree before allowing more updates.

Read operations for error-bounded ADTs must also be routed through reservation servers: the server does a weak read from any replica, then determines the interval based on how many reservations there are. For the read in Figure 5.4, there are 10 tokens total, but Server 2 knows that it has not used its local tokens, so it knows that there cannot be more than 5 outstanding increments. The server does a weak read from its closest replica, gets the value 100 because the increments have not yet propagated, and returns the interval $[100, 105]$. If instead the read went to Server 1, it would observe the value 102, but it would not know that Server 2 hasn't used its tokens, so the interval would be $[102, 109]$.

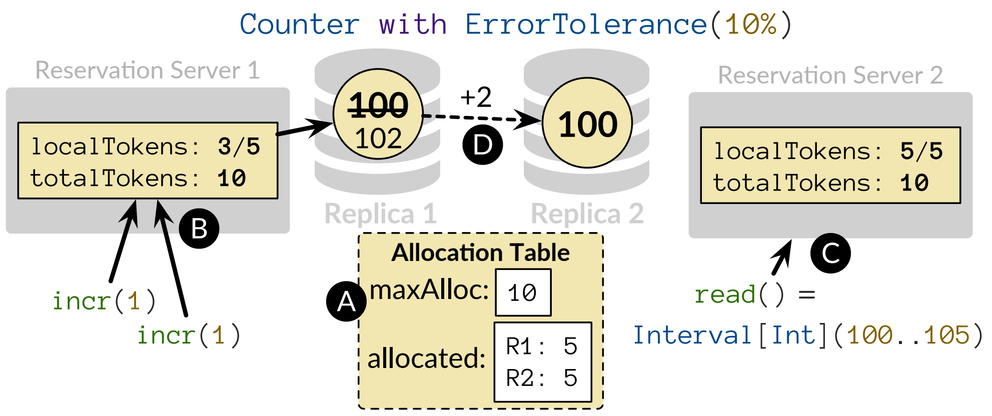


Figure 5.4. *Enforcing error bounds on a Counter:* (A) Each replica has some number of tokens allocated to it, must add up to less than the max (in this case, 10% of the current value). (B) Reservation Server 1 has sufficient tokens available, so both increments consume a token and proceed to Replica 1. (C) Reads return the range of possible values, determined by total number of allocated tokens; in this case, it reads the value 100, knows that there are 10 tokens total, but 5 of them (local to RS2) are unused, so it returns 100..105. (D) *Eventually,* when the increments have propagated, reservation server reclaims its tokens.

5.5.2.3. Narrowing bounds

Error tolerance policies set an *upper bound* on the amount of error; ideally, the interval returned will be more precise than the maximum error when conditions are favorable, such as when there are few update operations. Rather than assuming the total number of tokens is always the maximum allowable by the error bound, we instead keep an *allocation table* for each record that tracks the number of tokens allocated to each reservation server. If a reservation server receives an update operation and does not have enough tokens allocated, it updates the allocation table to allocate tokens for itself. The allocation table must preserve the invariant that the total does not exceed the maximum tokens allowed by the current value. For example, for a value of 100, 10 tokens were allowed, but after 1 decrement, only 9 tokens are allowed. Whenever this occurs, the server that changed the bound must give up the "lost" token out of its own allocations. As long as these updates are done atomically (in Cassandra, this is done using linearizable conditional updates), the global invariant holds. Because of this synchronization, reading and writing the allocation table is expensive and slow, so we use long leases (on the order of seconds) within each reservation server to cache their allocations. When a lease is about to expire, the server preemptively refreshes its lease in the background so that writes do not block unnecessarily.

For each type of update operation there may need to be a different pool of reservations. Similarly, there could be different error bounds on different read operations. It is up to the designer of the ADT to ensure that all error bounds are enforced with appropriate reservations. Consider a `Set` with an error tolerance on its `size` operation. This requires separate pools for `add` and `remove` to prevent the overall size from deviating by more than the bound in either direction, so the interval is $[v - \texttt{remove.delta}, v + \texttt{add.delta}]$ where v is the size of the set and `delta` computes the number of outstanding operations from the pool. In some situations, operations may produce and consume tokens in the same pool – e.g., `increment` producing tokens for `decrement` – but this is only allowable if updates propagate in a consistent order among replicas, which may not be the case in some eventually consistent systems.

5.6. Implementation

5.6.1. Backing datastore

IPA is implemented mostly as a client-side library to an off-the-shelf distributed storage system, though reservations are handled by a custom middleware layer which mediates accesses to any data with error tolerance policies. Our implementation is built on top of Cassandra, but IPA could work with any replicated storage system that supports fine-grained consistency control, which many commercial and research datastores do, including Riak [22].

IPA's client-side programming interface is written in Scala, using the asynchronous futures-based Phantom [122] library for type-safe access to Cassandra data. Reservation server middleware is also built in Scala using Twitter's Finagle framework [157]. Communication is done between clients and Cassandra via prepared statements, and between clients and reservation servers via Thrift remote-procedure-calls [13]. Due to its type safety features, abstraction capability, and compatibility with Java, Scala has become popular for web service development, including widely-used frameworks such as Akka [102] and Spark [12], and at established companies such as Twitter and LinkedIn [2, 37, 71].

5.6.2. Type system

The IPA type system, responsible for consistency safety, is also simply part of our client library, leveraging Scala's sophisticated type system. The IPA type lattice is implemented as a subclass hierarchy of parametric classes, using Scala's support for higher-kinded types to allow them to be destructured in match statements, and implicit conversions to allow `Consistent[T]` to be treated as type `T`. We use traits to implement ADT annotations; e.g. when the `LatencyBound` trait is mixed into an ADT, it wraps each of the methods, redefining them to have the new semantics and return the correct IPA type.

5.6.3. Provided by IPA

IPA comes with a library of reference ADT implementations used in our experiments, but it is intended to be extended with custom ADTs to fit more specific use cases. Our implementation provides a number of primitives for building ADTs, some of which are shown in Figure 5.5. To support latency bounds, there is a generic `LatencyBound` trait that provides facilities for executing a specified read operation at multiple consistency levels within a time limit. For implementing error bounds, IPA provides a generic reservation pool which ADTs can use. Figure 5.5 shows how a Counter with error tolerance bounds is implemented using these pools. The library of reference ADTs includes:

- `Counter` based on Cassandra's counter, supporting increment and decrement, with latency and error bounds

```
trait LatencyBound {
  // execute readOp with strongest consistency possible
  // within the latency bound
  def rush[T](bound: Duration,
              readOp: ConsistencyLevel => T): Rushed[T]
}
/* Generic reservaton pool, one per ADT instance.
   `max` recomputed as needed (e.g. for % error) */
class ReservationPool(max: () => Int) {
  def take(n: Int): Boolean // try to take tokens
  def sync(): Unit      // sync to regain used tokens
  def delta(): Int      // # possible ops outstanding
}
/* Counter with ErrorBound (simplified) */
class Counter(key: UUID) with ErrorTolerance {
  def error: Float // % tolerance (defined by instance)
  def maxDelta() = (cassandra.read(key) * error).toInt
  val pool = ReservationPool(maxDelta)

  def read(): Interval[Int] = {
    val v = cassandra.read(key)
    Interval(v - pool.delta, v + pool.delta)
  }
  def incr(n: Int): Unit =
    waitFor(pool.take(n)) { cassandra.incr(key, n) }
}
```

Figure 5.5. Some of the reusable components provided by IPA and an example implemention of a Counter with error bounds.

- `BoundedCounter` CRDT from [20] that enforces a hard lower bound even with weak consistency. Our implementation adds the ability to bound error on the value of the counter and set latency bounds.
- `Set` with `add`, `remove`, `contains` and `size`, supporting latency bounds, and error bounds on `size`.
- `UUIDPool` generates unique identifiers, with a hard limit on the number of IDs that can be taken from it; built on top of `BoundedCounter` and supports the same bounds.
- `List`: thin abstraction around a Cassandra table with a time-based clustering order, supports latency bounds.

Figure 5.5 shows Scala code using reservation pools to implement a Counter with error bounds. The actual implementation splits this functionality between the client and the reservation server.

5.7. Evaluation

The goal of the IPA programming model and runtime system is to build applications that adapt to changing conditions, performing nearly as well as weak consistency but with stronger consistency and safety guarantees. To that end, we evaluate our prototype implementation under a variety of network conditions using both a real-world testbed (Google Compute Engine [68]) and simulated network conditions. We start with simple microbenchmarks to understand the performance of each of the runtime mechanisms independently. We then study two applications in more depth, exploring qualitatively how the programming model helps avoid potential programming mistakes in each and then evaluating their performance against strong and weakly consistent implementations.

5.7.1. Simulating adverse conditions

To control for variability, we perform our experiments with a number of simulated conditions, and then validate our findings against experiments run on globally distributed machines in Google Compute Engine. We use a local test cluster with nodes linked by standard ethernet and Linux's Network Emulation facility [153] (`tc netem`) to introduce packet delay and loss at the operating system level. We use Docker containers [57] to enable fine-grained control of the network conditions between processes on the same physical node.

Table 5.2 shows the set of conditions we use in our experiments to explore the behavior of the system. The *uniform 5ms* link simulates a well-provisioned datacenter; *slow replica* models contention or hardware problems that cause one replica to be slower than others, and *geo-distributed* replicates the latencies between virtual machines in the U.S., Europe, and Asia on Amazon EC2 [9]. These simulated conditions are validated by experiments on Google Compute Engine with virtual machines in four datacenters: the client in *us-east*, and the storage replicas in *us-central*, *europe-west*, and *asia-east*. We elide the results for *Local* (same rack in our

Network Condition	Latencies (ms)		
Simulated	*Replica 1*	*Replica 2*	*Replica 3*
Uniform / High load	5	5	5
Slow replica	10	10	100
Geo-distributed (EC2)	1 ± 0.3	80 ± 10	200 ± 50
Actual	*Replica 1*	*Replica 2*	*Replica 3*
Local (same rack)	<1	<1	<1
Google Compute Engine	30 ± <1	100 ± <1	160 ± <1

Table 5.2. Network conditions for experiments: latency from client to each replicas, with standard deviation if high.

testbed) except in Figure 5.13 because the differences between policies are negligible, so strong consistency should be the default there.

5.7.2. Microbenchmark: Counter

We start by measuring the performance of a simple application that randomly increments and reads from a number of counters with different IPA policies. Random operations (`incr(1)` and `read`) are uniformly distributed over 100 counters from a single multithreaded client (allowing up to 4000 concurrent operations).

5.7.2.1. Latency bounds

Latency bounds provide predictable performance while maximizing consistency. We found that when latencies and load are low it is often possible to achieve strong consistency. Figure 5.6 shows the average operation latency with strong and weak consistency, as well with both 10ms and 50ms latency bounds.

As expected, there is a significant cost to strong consistency under all network conditions. IPA cannot achieve strong consistency under 10ms in any case, so the system must always default to weak consistency. With a 50ms bound, IPA can achieve strong consistency in conditions when

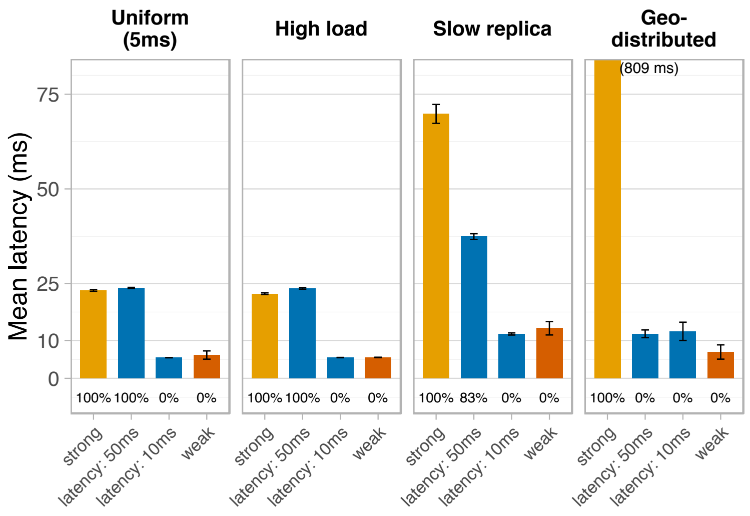


Figure 5.6. *Counter with latency bounds:* Mean latencies are below the bound. Beneath each bar is % of reads that were strong, which we see is never possible for the 10ms bound, but 50ms bound achieves mostly strong, only resorting to weak when network latency is high.

network latency is low (i.e., the single datacenter case). Cassandra assigns each client to read at a different replica for load balancing, so, with one slow replica, IPA will attempt to achieve strong consistency for all clients but not succeed. In our experiments, IPA was able to get strong consistency 83% of the time. Finally, with our geo-distributed environment, there are no 2 replicas within 50ms of our client, so strong consistency is never possible within our bounds; as a result, IPA adapts to only attempt weak in both cases.

Figure 5.7 shows the 95th percentile latencies for the same workload. The tail latency of the 10ms bound is comparable to weak consistency, whereas the 50ms bound overloads the slow server with double the requests, causing it to exceed the latency 5% of the time. There is a gap between latency-bound and weak consistency in the geo-distributed case because the `weak` condition uses weak reads *and* writes, while our rushed types, in order to have the option of getting strong reads without requir-

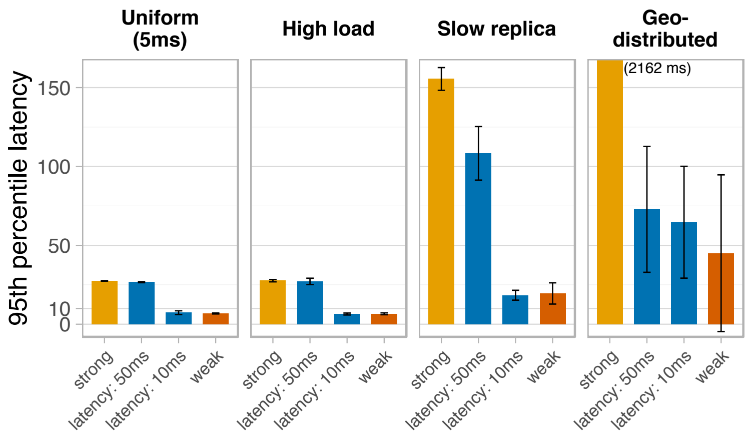


Figure 5.7. *Counter with latency bounds:* 95th percentile latency is improved by bounds, though they sometimes exceed the bounds due to unpredictability of the platform.

ing a read of `ALL`, must do `QUORUM` writes.

Note that, without consistency types, it would be challenging for programmers to handle the varying consistency of returned values in changing network conditions. However, IPA's type system not only gives programmers the tools to reason about different consistency levels, it enforces consistency safety.

5.7.2.2. Error bounds

This experiment measures the cost of enforcing error bounds using the reservation system described in §5.5.2, and its precision. Reservations move synchronization off the critical path: by distributing write permissions among replicas, reads can get strong guarantees from a single replica. Note that reservations impact write performance, so we must consider both in our experiments.

Figure 5.8 shows latencies for error bounds of 1%, 5%, and 10%, plotting the average of read *and* increment operations. As expected, tighter error bounds increase latency because it forces more frequent synchro-

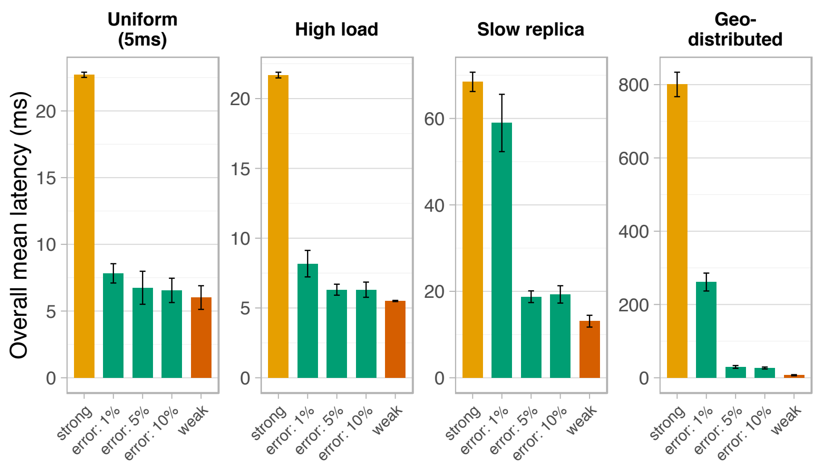


Figure 5.8. *Counter with error tolerance.* We see that wider error bounds reduce mean latency of increment and read operations because fewer synchronizations are required, matching *weak* around 5-10%.

nization between replicas. The 1% error bound provides most of the benefit, except in the slow replica and geo-distributed environments where it forces synchronization frequently enough that the added latency slows down the system. 5-10% error bounds provide latency comparable to weak consistency. In the geo-distributed case, the coordination required for reservations makes even the 10% error bound 4× slower than weak consistency, but this is still 28× faster than strong consistency.

While we have verified that error-bounded reads remain within our defined bounds, we also wish to know what error occurs in practice. We modified our benchmark to observe the actual error from weak consistency by incrementing counters a predetermined amount and reading the value; results are shown in Figure 5.9. We plot the percent error of weak and strong against the actual observed *interval width* for a 1% error bound, going from a read-heavy (1% increments) to write-heavy (all increments, except to check the value).

First, we find that the mean interval is less than the 1% error bound because, for counters that are less popular, IPA is able to return a more

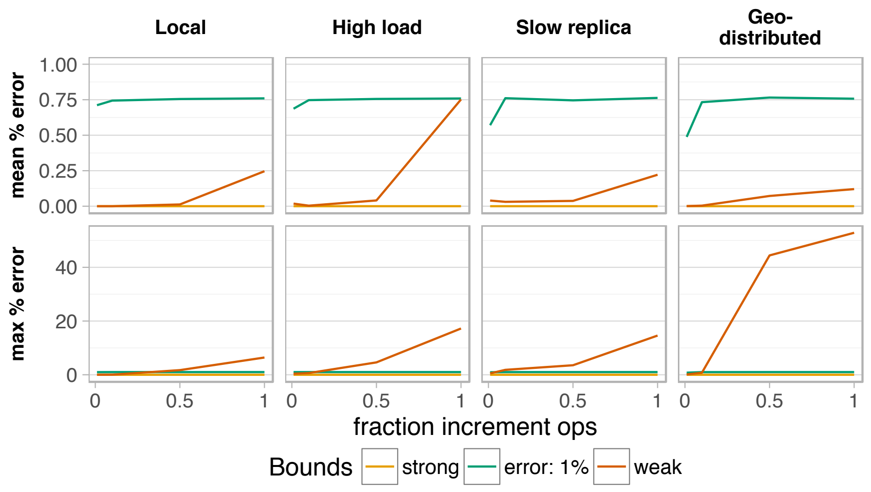


Figure 5.9. *Counter with error tolerance:* Observed % error for weak and strong, compared with the actual interval widths returned for 1% error tolerance. Mean actual error observed was less than 1% but can be as high as 60% without bounds.

precise interval. At low write rate, this interval becomes even smaller, down to .5% in the geo-distributed experiment. Next, we find that the mean error with weak consistency is also much less than 1%; however the maximum error that we observed is up to 60% of the actual value. This result motivates the need for error bounded consistency to ensure that applications do not see drastically incorrect values from weakly consistent operations. Further, using the `Interval` type, IPA is able to give the application an estimate of the variance in the weak read, which is often more precise than the upper bound set by the error tolerance policy.

5.7.3. Applications

Next, we explore the implementation of two applications in IPA and compare their performance against Cassandra using purely strong or weak consistency on our simulated network testbed and Google Compute Engine.

```
// creates a table of pools, so each event gets its own
// 5% error tolerance on `remaining` method, weak otherwise
val tickets = UUIDPool() with Consistency(Weak)
                with Remaining(ErrorTolerance(0.05))

// called from displayEvent (& purchaseTicket)
def getTicketCount(event: UUID): Interval[Int] =
  tickets(event).remaining()

def purchaseTicket(event: UUID) = {
  // UUIDPool is safe even with weak consistency (CRDT)
  endorse(tickets(event).take()) match {
    case Some(ticket) =>
      // imprecise count returned due to error tolerance
      val remaining = getTicketCount(event)
      // use maximum count possible to be fair
      val price = computePrice(remaining.max)
      display("Ticket reserved. Price: $" + price)
      prompt_for_payment_info(price)
    case None =>
      display("Sorry, all sold out.")
  }
}
```

Figure 5.10. *Ticket service* code demonstrating consistency types.

5.7.3.1. Ticket service

Our Ticket sales web service, introduced in §5.3, is modeled after FusionTicket [1], which has been used as a benchmark in recent distributed systems research [163, 164]. We support the following actions:

- `browse`: List events by venue
- `viewEvent`: View the full description of an event including number of remaining tickets
- `purchase`: Purchase a ticket (or multiple)
- `addEvent`: Add an event at a venue.

Figure 5.10 shows a snippet of code from the IPA implementation which can be compared with the non-IPA version from Figure 5.2. Tickets are modeled using the `UUIDPool` type, which generates unique identifiers to reserve tickets for purchase. The ADT ensures that, even with weak consistency, it never gives out more than the maximum number of tickets, so it is safe to `endorse` the result of the `take` operation as long as one is okay with the possibility of a false negative. Rather than just using a weak read as in the original example, in IPA we can bound the inconsistency of the remaining ticket count using an error tolerance annotation on the tickets pool. Now to compute the price of the reserved ticket, we call `getTicketCount` and get an `Interval`, forcing us to decide how to handle the range of possible ticket counts. We decide to use the `max` value from the interval to be fair to users; the 5% error bound ensures that we don't sacrifice too much profit this way.

To evaluate the performance, we run a workload modelling a typical small-scale deployment: 50 venues and 200 events, with an average of 2000 tickets each (gaussian distribution centered at 2000, stddev 500); this ticket-to-event ratio ensures that some events run out tickets. Because real-world workloads exhibit power law distributions [51], we use a moderately skewed Zipf distribution with coefficient of 0.6 to select events.

Figure 5.11 shows the average latency of a workload consisting of 70% `viewEvent`, 19% `browse`, 10% `purchase`, and 1% `addEvent`. We plot with a log scale because strong consistency has over 5× higher latency. The `purchase` event, though only 10% of the workload, drives most of the latency increase because of the work required to prevent over-selling tickets. We explore two different IPA implementations: one with a 20ms latency bound on all ADTs aiming to ensure that both `viewEvent` and `browse` complete quickly, and one where the ticket pool size ("tickets remaining") has a 5% error bound. We see that both perform with nearly the same latency as weak consistency. With the low-latency condition (*uniform* and *high load*), 20ms bound does 92% strong reads, 4% for *slow replica*, and all weak on both *geo-distributed* conditions.

Figure 5.11 also shows results on Google Compute Engine *(GCE)*. We see that the results of real geo-replication validate the findings of our simulated geo-distribution results.

On this workload, we observe that the 5% error bound performs well even under adverse conditions, which differs from our findings in the microbenchmark. This is because ticket `UUIDPool`s begin *full*, with many tokens available, requiring less synchronization until they are close to

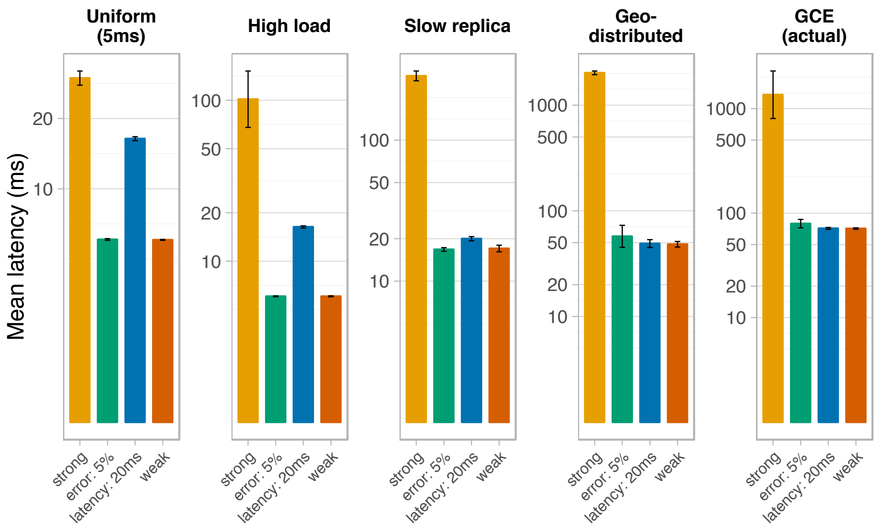


Figure 5.11. *Ticket service: mean latency, **log scale.*** Strong consistency is far too expensive (>10× slower) except when load and latencies are low, but 5% error tolerance allows latency to be comparable to weak consistency. The 20ms latency-bound variant is either slower or defaults to weak, providing little benefit. Note: the ticket `Pool` is safe even when weakly consistent.

running out. Contrast this with the microbenchmark, where counters started at small numbers (average of 500), where a 5% error tolerance means fewer tokens.

5.7.4. Twitter clone

Our second application is a Twitter-like service based on the Redis data modeling example, Retwis [136]. The data model is simple: each user has a `Set` of followers, and a `List` of tweets in their timeline. When a user tweets, the tweet ID is eagerly inserted into all of their followers' timelines. Retweets are tracked with a `Set` of users who have retweeted each tweet.

Figure 5.13 shows the data model with policy annotations: latency bounds on followers and timelines and an error bound on the retweets.

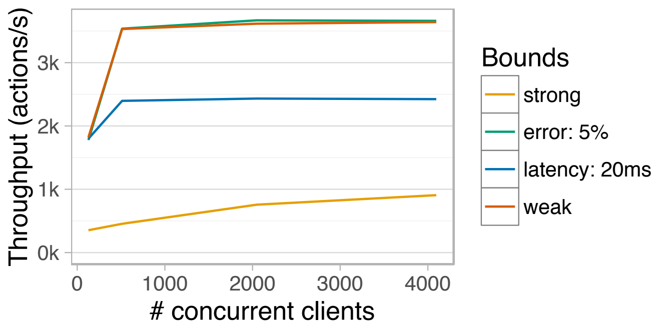


Figure 5.12. *Ticket service:* throughput on Google Compute Engine globally-distributed testbed. Note that this counts *actions* such as `tweet`, which can consist of multiple storage operations. Because error tolerance does mostly weak reads and writes, its performance tracks *weak*. Latency bounds reduce throughput due to issuing the same operation in parallel.

This ensures that when tweets are displayed, the retweet count is not grossly inaccurate. As shown in `displayTweet`, highly popular tweets with many retweets can tolerate approximate counts – they actually abbreviate the retweet count (e.g. “2.4M”) – but average tweets, with less than 20 retweets, will get an exact count. This is important because for regular people, they will notice if a friend’s retweet is not reflected in the count, whereas Ellen Degeneres’s record-breaking celebrity selfie, which nearly brought down Twitter in 2014 [15], can scale because a 5% error tolerance on 3.4 million retweets provides significant slack.

The code for `viewTimeline` in Figure 5.13 demonstrates how latency-bound `Rushed[T]` types can be destructured with a match statement. In this case, the timeline (list of tweet IDs) is retrieved with a latency bound. Tweet content is added to the store before tweet IDs are pushed onto timelines, so with strong consistency we know that the list of IDs will all be able to load valid tweets. However, if the latency-bound type returns with weak consistency (`Inconsistent` case), then this *referential integrity* property may not hold. In that case, we must guard the call to `displayTweet` and retry if any of the operations fails (e.g., if the retweet set wasn’t created yet).

```
class User(id: UserID, name: String,
  followers: Set[UserID] with LatencyBound(20 ms),
  timeline: List[TweetID] with LatencyBound(20 ms))

class Tweet(id: TweetID, user: UserID, text: String,
  retweets: Set[UserID] with Size(ErrorTolerance(5%)))

def viewTimeline(user: User) = {
  // `range` returns `Rushed[List[TweetID]]`
  user.timeline.range(0,10) match { // use match to unpack
    case Consistent(tweets) =>
      for (tweetID <- tweets)
        displayTweet(tweetID)
    case Inconsistent(tweets) =>
      // tweets may not have fully propagated yet
      for (tweetID <- tweets)
        // guard load and retry if there's an error
        Try { displayTweet(tweetID) } retryOnError
  }
}

def displayTweet(id: TweetID, user: User) = {
  val rct: Interval[Int] = tweets(id).retweets.size()
  if (rct > 1000) // abbreviate large counts (e.g. "2k")
    display("${rct.min/1000}k retweets")
  else if (rct.min == rct.max) // count is precise!
    display("Exactly ${rct.min} retweets")
  //...
  // here, `contains` returns `Consistent[Boolean]`
  // so it is automatically coerced to a Boolean
  if (tweets(id).retweets.contains(user))
    disable_retweet_button()
}
```

Figure 5.13. Twitter data model with policy annotations, `Rushed[T]` helps catch referential integrity violations and `Interval[T]` represents approximate retweet counts.

We simulate a realistic workload by generating a synthetic power-law graph, using a Zipf distribution to determine the number of followers per user. Our workload is a random mix with 50% `timeline` reads, 14% `tweet`, 30% `retweet`, 5% `follow`, and 1% `newUser`.

We see in Figure 5.14 that for all but the local (same rack) case, strong consistency is over 3× slower. Our implementation, combining latency

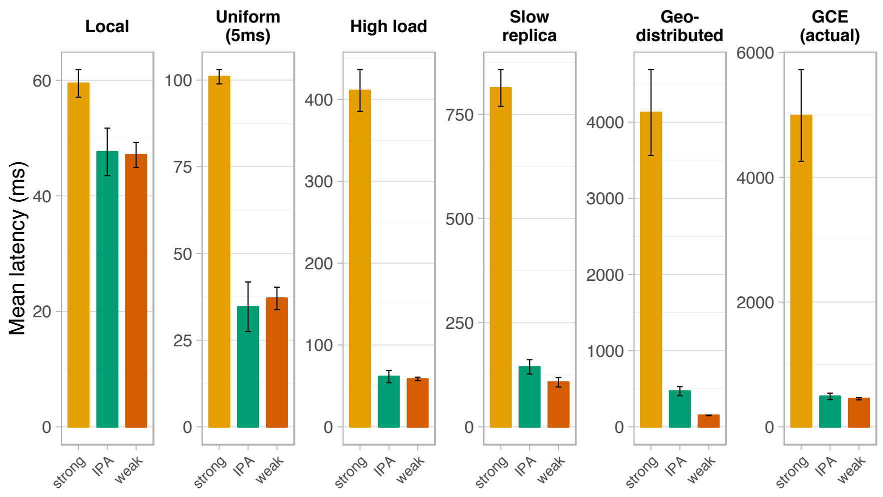


Figure 5.14. *Twitter clone: mean latency (all actions).* The IPA version performance comparably with weak consistency in all but one case, while strong consistency is 2-10× slower.

and error-bounds, performs comparably with weak consistency but with stronger guarantees for the programmer. Our simulated geo-distributed condition turns out to be the worst scenario for IPA's Twitter, with latency over 2× slower than weak consistency. This is because weak consistency performed noticeably better on our simulated network, which had one very close (1ms latency) replica that it used almost exclusively.

5.8. Discussion

At a high level, one can see *disciplined inconsistency* as a marriage of *approximate computing* research with the field of distributed systems. Coming from the field of Computer Architecture, approximate computing [132] is the idea that a lot of work in computer systems goes toward ensuring that everything is perfectly precise and reliable, yet not all computations have such strict needs. By allowing programs to be less correct in parts, performance and energy efficiency can be improved. The same is already recognized to be true in distributed systems, with eventual consistency as the poster child of allowing programs to be incorrect in order to let them perform well and scale. However, distributed systems are lacking in the second piece of approximate computing, which is disciplined programming models that ensure that important parts of the programs remain precise. IPA's consistency safety type system is inspired by EnerJ [32, 133] and Rely [38, 110], which annotate and track the flow of approximate values to prevent them from interfering with precise computation.

Disciplined inconsistency differs from approximate computing in several crucial ways. First of all, people already frequently use inconsistency in real applications, so this work, rather than introducing the idea of approximation, is actually pushing back against approximation, encouraging safer use of inconsistency only where necessary. Furthermore, the majority of approximations employed in hardware irretrievably lose data [60, 70, 134], whereas inconsistency is transient: the "precise" value can be retrieved whenever desired simply by synchronizing more aggressively.

The dynamic policies of IPA build on prior systems techniques, incorporating them into the type system to make them safer to use. IPA's latency bound policies were inspired by the *consistency-based SLAs* of Pileus [152]. In Pileus, reads return their consistency level, but do not help developers use that information; on the other hand, `Rushed` types ensure that developers consider weak outcomes. Error tolerance bounds are enforced using a technique that extends reservations [66, 121, 126, 129] and borrows ideas from Bounded Counter CRDTs [20], which are similar to allocation tables. Interval types used to express error-bounded values are similar in usage to Uncertain<T>'s probability distributions [31] and to interval analysis [111].

In summary, the IPA programming model provides programmers with disciplined ways to trade consistency for performance in distributed applications. By specifying application-specific performance and accuracy targets in the form of latency and error tolerance bounds, they tell the system how to adapt when conditions change and provide it with opportunities for optimization. Meanwhile, consistency types guarantee consistency safety, ensuring that all potential weak outcomes are handled and allowing applications to make choices based on the accuracy of the values the system returns. The policies, types and enforcement systems implemented in this work are only a sampling of the full range of possibilities within the framework of *Inconsistent, Performance-bound, and Approximate* types.

6. Conclusion

6.1. Retrospective

6.1.1. Vertical abstractions

The techniques employed in this work required new abstractions that cut across the traditional layered abstractions of the system stack, tying low-level implementations more directly into the high-level behavior of applications. There is still a tension between communicating too much to the lower level, leading to couplings that make it more difficult to swap in alternative backends and increasing the complexity of the simpler low-level systems. The trick here is to find the correct granularity and specificity to incorporate into the abstraction. One of the keys to this dissertation is using *abstract data types* because they strike a balance between generality — common ADTs like sets and maps are fairly universal — and specificity — properties like commutativity provide useful optimization opportunities while not tying them to concrete implementation details.

6.1.2. Unnecessary synchronization

Much of the improvements presented in this dissertation come down to reducing synchronization, either by recognizing additional concurrency in the form of commutativity or by helping programmers use weaker, more approximate, semantics. This is because synchronization is at the core of most scalability problems, and many applications could scale much better if only they knew which ordering constraints were actually essential. Techniques like distributed combining and abstract locking can help improve performance, but the best optimizations come from better understanding the desired behavior and choosing data structures that correctly capture only what is necessary. We saw this in the BFS kernel in §3.2.4.2 how the list of unordered bags better captured the needs of the algorithm than a fully ordered queue. Similarly, Claret (Chapter 4) was designed to be extensible for this very reason, to allow developers to create their own ADTs that better fit their needs. IPA (Chapter 5) takes this to a new level by introducing approximation into the mix, intentionally making some data inconsistent where accuracy is not essential, in order to improve performance.

6.1.3. Real workloads

It is always important to recognize and consider the benchmarks used to evaluate work. Some challenges — such as extreme contention, traffic spikes, and other irregular access patterns — will only be apparent if the workloads used to evaluate systems capture this, exhibiting features like skewed power-law distributions, network and timing effects, and adverse network conditions. If the techniques explored in this dissertation had only ever been evaluated on read-heavy workloads or uniform distributions, it would have appeared that many of them were unnecessary. However, we know that real web services have to deal with the unpredictable barrage of internet traffic, and graph analytics applications must operate on real social graphs with a host of challenging properties. In order to approximate the behavior of real applications for use in our evaluations, we have simplified the workloads, using realistic synthetic graphs in all of the chapters of this disseration, and simple user models based on power laws in Chapter 4 and Chapter 5. An obvious next step would be to attempt to use these techniques in real production systems to validate that they are still effective there.

6.2. Open problems

Research often opens new avenues of inquiry more than it closes them. The techniques described in this dissertation contribute to the toolkit available to distributed application developers, but that toolkit is by no means complete yet. There are ample opportunities to improve upon the systems that have been presented. In addition, it would be useful to step back a bit and evaluate how the proposed programming model changes actually affect the productivity of developers.

6.2.1. Locality and migration opportunities

Choosing where execution and data should reside in large distributed applications is still a difficult challenge. Alembic addressed a need in PGAS languages to automatically move parts of computation closer to the data it accesses. However, similar migration opportunities exist in many other distributed systems. Typical mobile applications must be divided between client-side code running on the device and applications servers running in the cloud, potentially further divided among many independent services and data storage platforms. In such a complicated environment, changing data layout to improve communication — such as adding a layer of caching, denormalizing, or materializing queries — may require significant changes to the code. Many current platform-as-a-service (PaaS) frameworks [62, 107, 123] encourage most application code to be in the client application. This provides an opportunity for automatically extracting parts of the client application to execute on the server side. We have done preliminary work for offloading parts of Redis applications automatically generating server-side Lua code to replace client-side Redis commands.

6.2.2. Checking and synthesis of implementations

As discussed above, the vertical abstractions proposed in this disseration have a cost in terms of added complexity within the underlying systems. Currently, the burden is on the expert developer who designs each ADT to ensure that it is correctly implemented, and as we incorporate more intelligence into the systems, there is more that can go wrong. In Claret, new ADTs express their commutativity and associativity by declaring abstract locks and combiners, but there is currently no way to checks that it is actually safe to commute those operations. IPA's ADTs also require the expert developer to ensure that all the consistency constraints are met, such as ensuring that all possible combinations of read and write operations still maintain strong consistency. It would be beneficial to have some guarantee that the ADT implementations are valid, similar to the commutativity tests generated by Commuter [48]. If designed well enough, this could enable more developers to create their own ADTs.

Beyond checking that implementations are correct, it would be even better if correct implementations could be synthesized from high-level descriptions. An example of this field of work is Cozy [105], which synthesizes collections from SQL-like queries. In the context of IPA, it may be possible to synthesize approximate versions of data structures based on some description of its semantics and the desired properties. A system like Cozy that knows the semantics and costs a priori would be well-positioned to choose the best ways to relax the implementation to improve performance. Currently, the consistency policies of IPA require manual help from ADT designers to be used; it would greatly improve the usability of the tool if these annotations could be freely applied to any ADT operation, giving programmers a way to pick and choose exactly which approximations their application desires.

6.2.3. Productivity and usability studies

The programming models proposed in this work are generally intended to improve the expressivity of applications and reduce the chances that programmers will make mistakes, without burdening them with excessive annotation or verification work. By leveraging the data structures already present in the applications, we gain information without requiring significant changes. However, it would be informative to directly study whether or not these programming models affect the productivity of developers. In the context of IPA especially, it would be useful to see if developers of distributed applications find the dynamic policies and consistency types useful in developing more robust software. The difficulty lies in evaluating the effect on productivity, comparing the additional work of wrangling consistency types with the potential benefits of ensuring consistency safety. It would also be useful to port larger, existing applications and evaluate them on production workloads because that is where the true benefit is. For example, would consistency types have caught already-known bugs, or does using IPA reduce the number of consistency violations that users observe? The Claret programming model is very similar to that of Redis; with some work, it should be possible to enhance Redis with Claret's optimizations, enabling efficient distributed transactions to be added to existing Redis applications.

A closely related concern is how applications can and should deal with inconsistent data. For the most part, inconsistency shows itself in subtle ways to end users. It could be that when the page is refreshed, some posts are reordered, or that the count of "likes" on a post is not the same in two different views. Some user interface designs have found ways to incorporate uncertainty in ways that are useful to end users, such as abbreviating counts so that the precise number is not visible (e.g., "3.7M retweets"), or pending changes being greyed out until they are correctly synchronized. A survey of such user interfaces could provide ideas as to how programmers should be dealing with inconsistency in their applications and may inform new abstractions that are more natural than the `Rushed` and `Interval` types sketched in IPA.

6.2.4. Wrap-up

This dissertation has proposed several advancements to make it easier to build performant and robust distributed applications. The techniques described contribute to an ever-growing body of tools, programming systems, frameworks, and languages for building distributed applications. These various programming models, consistency models, and systems — past, present, and future — are not in competition with one another; rather, they build on one another. They re-examine lessons learned in the past, reviving the ideas about ADT databases and bringing them into the modern world of web services, replicated weakly consistent datastores, and graph analytics. By exposing more knowledge about the high-level requirements of applications to lower layers of the system stack, this dissertation has shown that there is tremendous opportunity in leveraging semantics and breaking old abstractions, and there will be many more to come.

Bibliography

[1] Fusion ticket. http://fusionticket.org.

[2] Scala in the enterprise. http://www.scala-lang.org/old/node/1658, March 2009.

[3] S. V. Adve and H.-J. Boehm. Memory models: A case for rethining parallel languages and hardware. *Communications of the ACM*, 53 (8): 90–101, 2010.

[4] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par 2010-Parallel Processing*, pages 151–162. Springer, 2010.

[5] Vasudeva Akula and Daniel A Menascé. An analysis of bidding activity in online auctions. In *E-Commerce and Web Technologies*, pages 206–217. Springer, 2004.

[6] Vasudeva Akula and Daniel A. Menascé. Two-level workload characterization of online auctions. *Electronic Commerce Research and Applications*, 6 (2): 192–208, June 2007. doi:10.1016/j.elerap.2006.07.003.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11. ACM, 1988. ISBN 0-89791-252-7. doi:10.1145/73560.73561.

[8] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *Conference on Innovative Data Systems Research (CIDR)*, CIDR, pages 249–260. Citeseer, 2011.

[9] Amazon Web Services, Inc. Elastic compute cloud (ec2) cloud server & hosting – aws. https://aws.amazon.com/ec2/, 2016 .

[10] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*. IEEE, 2002. doi:10.1109/wwc.2002.1226489.

[11] Apache Software Foundation. Cassandra. http://cassandra.apache.org/, 2015.

[12] Apache Software Foundation. Apache spark - lightning-fast cluster computing. http://spark.apache.org/, 2016a.

[13] Apache Software Foundation. Apache thrift. https://thrift.apache.org/, 2016b.

[14] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Transactions on Database Systems*, 17 (1): 163–199, March 1992. doi:10.1145/128765.128771.

[15] Lisa Baertlein. Ellen's Oscar 'selfie' crashes Twitter, breaks record. http://www.reuters.com/article/2014/03/03/us-oscars-selfie-idUSBREA220C320140303, March 2014.

[16] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, Horst D Simon, V Venkatakrishnan, and Sisira K Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5: 63–73, 1991.

[17] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7 (3): 181–192, November 2013a. ISSN 2150-8097. doi:10.14778/2732232.2732237.

[18] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-2037-5. doi:10.1145/2463676.2465279.

[19] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys, pages 6:1–6:16, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3238-5. doi:10.1145/2741948.2741972.

[20] Valter Balegas, Diogo Serra, Sergio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. *34th International Symposium on Reliable Distributed Systems (SRDS 2015)*, September 2015b.

[21] R. Barik, Jisheng Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory X10 programs. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 1101–1113, May 2011. doi:10.1109/IPDPS.2011.105.

[22] Basho Technologies, Inc. Riak. `http://docs.basho.com/riak/latest/`, 2015.

[23] Scott Beamer, Krste Asanovi□, and David Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[24] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267680.1267685`.

[25] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Mike Dahlin, and Robert Grimm. Pads: A policy architecture for

distributed storage systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 59–73, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558982.

[26] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 168–176, New York, NY, USA, 1990. ACM. ISBN 0-89791-350-7. doi:10.1145/99163.99182.

[27] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed currency serializability for middle-tier caching and replication. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, June 2006. ACM.

[28] Jonathan W Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium. IPDPS 2007. IEEE International*, pages 1–14. IEEE, 2007.

[29] H.-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html, December 2007.

[30] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[31] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 14*, ASPLOS. Association for Computing Machinery (ACM), 2014. doi:10.1145/2541940.2541958.

[32] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 470–487, 2015. doi:10.1145/2814270.2814301.

[33] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7 (13): 1441–1451, August 2014. ISSN 2150-8097. doi:10.14778/2733004.2733016.

[34] Eric A. Brewer. Towards robust distributed systems. In *Keynote at PODC (ACM Symposium on Principles of Distributed Computing)*. Association for Computing Machinery (ACM), 2000. doi:10.1145/343477.343502.

[35] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 311–321. ACM, 1992. ISBN 0-89791-475-9. doi:10.1145/143095.143143.

[36] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27 (6): 701–724, 1997.

[37] Travis Brown. Scala at scale at Twitter (talk). http://conferences.oreilly.com/oscon/open-source-2015/public/schedule/detail/42332, July 2015.

[38] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, pages 33–52, 2013. doi:10.1145/2509136.2509546.

[39] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 29–38, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi:10.1145/209936.209941.

[40] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[41] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi:10.1145/121132.121159.

[42] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21 (3): 291–312, August 2007. ISSN 1094-3420. doi:10.1177/1094342007078442.

[43] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538. ACM, 2005. ISBN 1-59593-031-0. doi:http://doi.acm.org/10.1145/1094811.1094852.

[44] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi:10.1109/PACT.2005.13.

[45] Wei-Yu Chen, Dan Bonachea, Costin Iancu, and Katherine Yelick. Automatic nonblocking communication for partitioned global ad-

dress space programs. In *International Conference on Supercomputing, Proceedings*, pages 158–167. ACM, 2007.

[46] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi:10.1145/1294261.1294265.

[47] Panos K. Chrysanthis, S. Raghuram, and Krithi Ramamritham. Extracting concurrency from objects. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, SIGMOD. Association for Computing Machinery (ACM), 1991. doi:10.1145/115790.115803.

[48] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. ACM, 2013. doi:10.1145/2517349.2522712.

[49] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47. ACM, 2005.

[50] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC 12*, SoCC. ACM Press, 2012. doi:10.1145/2391229.2391230.

[51] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC 10*. Association for Computing Machinery (ACM), 2010. doi:10.1145/1807128.1807152.

[52] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 251–264, 2012. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`.

[53] Hayley C. Cuccinello. 'star wars' presales crash ticketing sites, set record for fandango. `http://www.forbes.com/sites/hayleycuccinello/2015/10/20/star-wars-presales-crash-ticketing-sites-sets-record-for-fandango/`, October 2015.

[54] Datastax, Inc. How are consistent read and write operations handled? `http://docs.datastax.com/en/cassandra/3.x/cassandra/dml/dmlAboutDataConsistency.html`, 2016.

[55] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *IEEE International Conference on Data Engineering Workshops (ICDEW)*, March 2014. doi:10.1109/icdew.2014.6818330.

[56] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi:10.1145/1989493.1989502.

[57] Docker, Inc. Docker. `https://www.docker.com/`, 2016.

[58] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery (ACM), August 2012. doi:10.1145/2342356.2342360.

[59] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell University, November 2013.

[60] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2012. URL http://sampa.cs.washington.edu/papers/truffle-web.pdf.

[61] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41 (1): 65–156, August 1990. doi:10.1016/0022-0000(90)90034-i.

[62] Firebase. Firebase, 2016. https://www.firebase.com/.

[63] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21 (9): 948–960, 1972. doi:10.1109/TC.1972.5009071.

[64] Brady Forrest. Bing and google agree: Slow pages lose users. Radar, June 2009. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[65] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[66] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8 (2): 3–10, 1985.

[67] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2): 51, June 2002. doi:10.1145/564585.564601.

[68] Google, Inc. Compute engine — google cloud platform. https://cloud.google.com/compute/, 2016.

[69] Graph500. Graph 500. http://www.graph500.org/, July 2012.

[70] Qing Guo, Karin Strauss, Luis Ceze, and Henrique Malvar. High-density image storage using approximate memory cells. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[71] Susan Hall. Employers can't find enough scala talent. http://insights.dice.com/2014/04/04/employers-cant-find-enough-scala-talent/, March 2014.

[72] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 355–364. ACM, 2010a.

[73] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*, pages 79–93. Springer, 2010b.

[74] Maurice Herlihy and Eric Koskinen. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 207–216, 2008. ISBN 978-1-59593-795-7. doi:10.1145/1345206.1345237.

[75] Maurice P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS, pages 201–210, New York, NY, USA, 1988. ACM. ISBN 0-89791-263-2. doi:10.1145/308386.308440.

[76] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12 (3): 463–492, July 1990. doi:10.1145/78969.78972.

[77] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35 (8): 66–80, August 1992. ISSN 0001-0782. doi:10.1145/135226.135230.

[78] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models (PGAS)*, PGAS, October 2013. URL http://sampa.cs.washington.edu/papers/holt-pgas13.pdf.

[79] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. Alembic: Automatic locality extraction via migration. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[80] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency. Technical Report UW-CSE-16-06-01, University of Washington, 2016.

[81] Mat Honan. Killing the fail whale with twitter's christopher fry. http://www.wired.com/2013/11/qa-with-chris-fry/, November 2013.

[82] HPCC. HPCC random-access benchmark http://icl.cs.utk.edu/hpcc/hpcc_results.cgi.

[83] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 239–248, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. doi:10.1145/155332.155357.

[84] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. Dynamic computation migration in DSM systems. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-89791-854-1. doi:10.1145/369028.369119.

[85] Hyperdex. Hyperdex. http://hyperdex.org/, 2015.

[86] ISO/IEC. Programming languages - C - Extensions to support embedded processors. Technical Report 18037, 2006.

[87] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language - C++ (Committee Draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[88] Suresh Jagannathan. Communication-passing style for coordination languages. In *Coordination Languages and Models*, pages 131–149. Springer, 1997.

[89] Simon Kahan and Petr Konecny. MAMA!: A memory allocator for multithreaded architectures. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 178–186, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi:10.1145/1122971.1122999.

[90] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. Parallex: An advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[91] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi:10.1145/165854.165874.

[92] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Enhancing Concurrency in Distributed Transactional Memory through Commutativity. In *EuroPar 2013*, EuroPar, pages 150–161. 2013. doi:10.1007/978-3-642-40047-6_17.

[93] Peter M. Kogge. Of piglets and threadlets: Architectures for self-contained, mobile, memory programming. In *Innovative Architecture for Future Generation High-Performance Processors and Systems, Proceedings*, pages 130–138. IEEE, 2004.

[94] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the Commutativity Lattice.

In *Conference on Programming Language Design and Implementation*, PLDI, pages 542–555, 2011. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993562.

[95] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44 (2): 35–40, April 2010. ISSN 0163-5980. doi:10.1145/1773912.1773922.

[96] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28 (9): 690–691, September 1979. doi:10.1109/tc.1979.1675439.

[97] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32, 2001.

[98] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

[99] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li`.

[100] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2`.

[101] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7 (4): 321–359, November 1989. ISSN 0734-2071. doi:10.1145/75104.75105.

[102] Lightbend Inc. Akka. `http://akka.io/`, 2016.

[103] Greg Linden. Make data useful. Talk, November 2006. `http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html`.

[104] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 503–517, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed`.

[105] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *PLDI 2016, Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation*, Santa Barbara, CA, USA, June 2016.

[106] Daniel A. Menascé and Vasudeva Akula. Improving the performance of online auctions through server-side activity-based caching. *World Wide Web*, 10 (2): 181–204, February 2007. doi:10.1007/s11280-006-0011-8.

[107] Meteor. Meteor, 2016. `http://www.meteor.com`.

[108] Cade Metz. How instagram solved its justin bieber problem, November 2015. URL `http://www.wired.com/2015/11/how-instagram-solved-its-justin-bieber-problem/`.

[109] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[110] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*, pages 309–328, 2014. doi:10.1145/2660193.2660231.

[111] Ramon E. Moore. *Interval analysis*. Prentice-Hall, 1966.

[112] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP. Association for Computing Machinery (ACM), 2013. doi:10.1145/2517349.2517350.

[113] Richard Cameron Murphy. *Traveling Threads: A New Multithreaded Execution Model*. PhD thesis, University of Notre Dame, 2006.

[114] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, SOSP, pages 511–524, Broomfield, CO, October 2014. ISBN 978-1-931971-16-4.

[115] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical Report UW-CSE-14-02-01, University of Washington, February 2014. URL `http://sampa.cs.washington.edu/papers/grappa-tr-2014-02.pdf`.

[116] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Brigg, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, July 2015. URL `http://sampa.cs.washington.edu/papers/grappa-usenix-2015.pdf`.

[117] Jacob Eric Nelson. *Latency-Tolerant Distributed Shared Memory For Data-Intensive Applications*. PhD thesis, University of Washington, 2015.

[118] Dao Nguyen. What it's like to work on buzzfeed's tech team during record traffic. http://www.buzzfeed.com/daozers/what-its-like-to-work-on-buzzfeeds-tech-team-during-record-t, February 2015.

[119] NPB3.3. NAS parallel benchmark suite 3.3. http://www.nas.nasa.gov/publications/npb.html, 2012.

[120] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, USA, May 1999. ACM.

[121] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11 (4): 405–430, December 1986. doi:10.1145/7239.7265.

[122] outworkers ltd. Phantom by outworkers. http://outworkers.github.io/phantom/, March 2016.

[123] Parse. Parse, 2016. http://www.parse.com.

[124] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Ontario, Canada, October 2004.

[125] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010. USENIX.

[126] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys 03*, MobiSys. Association for Computing Machinery (ACM), 2003. doi:10.1145/1066116.1189038.

[127] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, CO, USA, May 1991. ACM.

[128] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi:10.1145/1356058.1356074.

[129] Andreas Reuter. *Concurrency on high-traffic data elements*. ACM, New York, New York, USA, March 1982.

[130] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17 (2): 233–263, March 1995. ISSN 0164-0925. doi:10.1145/201059.201065.

[131] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS — a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, Hong Kong, China, August 2002.

[132] Adrian Sampson. *Hardware and Software for Approximate Computing*. PhD thesis, University of Washington, 2015.

[133] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 164–174, 2011. doi:10.1145/1993498.1993518.

[134] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *International Sympo-*

sium on Microarchitecture (MICRO), December 2013. URL `http://sampa.cs.washington.edu/papers/approxstorage-micro2013.pdf`.

[135] Salvatore Sanfilippo. Redis. `http://redis.io/`, 2015a.

[136] Salvatore Sanfilippo. Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. `http://redis.io/topics/twitter-clone`, 2015b.

[137] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, Sung-Eun Choi, and B.L. Chamberlain. Global data re-allocation via communication aggregation in Chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 235–242, October 2012. doi:10.1109/SBAC-PAD.2012.18.

[138] Peter Martin Schwarz. *Transactions on Typed Objects*. PhD thesis, Pittsburgh, PA, USA, 1984. AAI8506303.

[139] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400, 2011. ISBN 978-3-642-24549-7.

[140] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems*, 20 (3): 325–363, September 1995. doi:10.1145/211414.211427.

[141] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60 (11): 1355–1387, 2000.

[142] Yossi Shiloach and Uzi Vishkin. An O(N log(N)) parallel max-flow algorithm. *Journal of Algorithms*, 3 (2): 128–146, 1982.

[143] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *International Conference on Supercomputing*, ICS '08, pages 277–288. ACM, 2008. ISBN 978-1-60558-158-3. doi:10.1145/1375527.1375568.

[144] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, PLDI. Association for Computing Machinery (ACM), 2015. doi:10.1145/2737924.2737981.

[145] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, 2011.

[146] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles - SOSP'11*, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[147] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. *ACM SIGOPS Operating Systems Review*, 19 (5): 127–146, December 1985. doi:10.1145/323627.323641.

[148] Michael Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering, February 5-7, 1986, Los Angeles, California, USA*, pages 262–269, 1986.

[149] Michael Stonebraker, W. Bradley Rubenstein, and Antonin Guttman. Application of abstract data types and abstract indices to CAD data bases. In *Engineering Design Applications*, pages 107–113, 1983.

[150] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'09, pages 43–58, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1558977.1558981`.

[151] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS. Institute of Electrical & Electronics Engineers (IEEE), 1994. doi:10.1109/pdis.1994.331722.

[152] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*. ACM Press, 2013. doi:10.1145/2517349.2522731.

[153] The Linux Foundation. netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`, November 2009.

[154] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43759-8. doi:10.1007/3-540-47993-7_8.

[155] Transaction Processing Performance Council. TPC-C. `http://www.tpc.org/tpcc/`, 2015.

[156] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[157] Twitter, Inc. Finagle. `https://twitter.github.io/finagle/`, March 2016.

[158] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52 (1): 40, January 2009. doi:10.1145/1435417.1435432.

[159] Voldemort. Voldemort. `http://www.project-voldemort.com/voldemort/`, 2015.

[160] Cheng Wang and Zhiyuan Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 119–130, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi:10.1145/996841.996857.

[161] Lei Wang and Michael Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *International Conference on Parallel and Distributed Systems (ICPADS'08)*, pages 369–376. IEEE, 2008.

[162] W. E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. In *International Conference on System Sciences*, pages 205–214, 1988. ISBN 0-8186-0842-0.

[163] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie`.

[164] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2015. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.

[165] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100 (4): 388–395, 1987.

[166] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20 (3): 239–282, 2002.

[167] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achiev-

ing Serializability with Low Latency in Geo-distributed Storage Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.