LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper
Indiana University
Mozilla Research
October 31, 2013
LVars:
Lattice-based Data Structures
for Deterministic Parallelism

Lindsey Kuper
Indiana University

Mozilla Research
October 31, 2013
Jeff Dean
Google Fellow
Google, Inc.
The Tail at Scale: Achieving Rapid Response Times in Large Online Services

Peter Bailis
PhD Student
UC Berkeley
Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

Justin Sheehy
Chief Technology Officer
Basho Technologies
Maximum Viable Product

Pat Helland
Architect
Salesforce.com
Keystone: Between a ROC and a SOFT Place

Diego Ongaro
PhD Student
Stanford University
The Raft Consensus Algorithm

Jeff Hodges
Distributed Systems Engineer
Twitter
Practicalities of Productionizing Distributed Systems

Lindsey Kuper
PhD Student
Indiana University
LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein
Instigator
Code Climate
Distributed Systems Archaeology
grad student

Google Fellow
Google, Inc.
The Tail at Scale: Achieving Rapid Response Times in Large Online Services

PhD Student
UC Berkeley
Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

Chief Technology Officer
Basho Technologies
Maximum Viable Product

Architect
Salesforce.com
Keystone: Between a ROC and a SOFT Place

Diego Ongaro
PhD Student
Stanford University
The Raft Consensus Algorithm

Jeff Hodges
Distributed Systems Engineer
Twitter
Practicalities of Productionizing Distributed Systems

Lindsey Kuper
PhD Student
Indiana University
LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein
Instigator
Code Climate
Distributed Systems Archaeology
LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper
Indiana University

Mozilla Research
October 31, 2013
x = 3
put(x, 4)

x = 3
put(x, 0)

x = 3
put(x, 5)

x = 3
put(x, 5)

x = 3
put(x, 4)

x = 3
put(x, 6)

x = 3
put(x, 7)
x = 3
put(x, 4)

x = 3
put(x, 0)

x = 3
put(x, 4)

x = 3
put(x, 5)

x = 3
put(x, 6)

x = 3
put(x, 5)

x = 3
put(x, 0)

x = 3
put(x, 7)
Eventual consistency.
Eventual consistency.

Dynamo: Amazon’s Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels
Amazon.com

ABSTRACT
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors
D.4.2 [Operating Systems]: Storage Management; D.4.5 [Operating Systems]: Reliability; D.4.2 [Operating Systems]: Performance;
General Terms

1. INTRODUCTION
Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon’s platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornadoes. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon’s software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon’s platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon’s platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon’s platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs
Dynamo: Amazon’s Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels
Amazon.com

ABSTRACT
Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure based on a large number of Amazon ec2 instances and S3 buckets. Since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client’s experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart.

Amazon runs a world-wide e-commerce platform that serves tens of millions of customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon’s platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view items in their shopping cart even if disks are routes are flapping, or data centers are being added. Therefore, the service responsible for the cart requires that it can always write to and read from the cart, and that its data needs to be available in an infrastructure comprised of millions of servers in a wide range of data centers. Amazon has developed the Amazon Simple Storage Service (S3) to provide a scalable, reliable storage solution that can meet these requirements. Amazon Simple Storage Service is also available outside of Amazon and known as S3. S3 is the best known of Amazon’s cloud computing services.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.


Eventual consistency.
Conflict-Free Replicated Data Types*

Marc Shapiro1,5, Nuno Preguiça1,2, Carlos Baquero3, and Marek Zawirski1,4

1 INRIA, Paris, France
2 CITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPMC, Paris, France
5 LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both add and remove operations, and consider in-depth the more complex Graph data type. CRDTs can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard "strong consistency" approach serialises updates in a global total order [16]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8]. When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, eventual consistency promises better availability and performance [179]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, requires prior specific permission and/or a fee.


Eventual consistency.

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is complicated by both an infrastructure composed of thousands of

ware of the data schema it is method that is best suited for an application that maintains a "merge" the conflicting shopping cart.

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon’s platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service-oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to order items to their shopping cart even if disks are routes are flapping, or data centers are being added. Therefore, the service responsible for its carts requires that it can always write to and read data store, and that its data needs to be available data centers.

Challenges in an infrastructure comprised of millions of servers and tens of thousands of server and disk node failures can occur at any given time. As such Amazon’s software is constructed in a manner that treats failure as the normal case without impacting availability or performance. Further, because of the scale and scaling needs, Amazon has developed its own technologies, of which the Amazon Simple Storage Service (S3) is the most widely used. S3 is available outside of Amazon and known as the best commercially available cloud storage service.

Amazon’s platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application developers configure the data store appropriately. Dynamo is a key-value store that provides strong consistency and availability guarantees. It is a key-value store that provides strong consistency and availability guarantees. It is a key-value store that provides strong consistency and availability guarantees.

Amazon provides a simple primary-key only interface to meet the requirements of these applications. Dynamo uses a synthesis of well known techniques to achieve scalability and availability. Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs
Eventual consistency.
Conflict-Free Replicated Data Types*

Marc Shapiro1,5, Nuno Preguiça1,2, Carlos Baquero3, and Marek Zawirski1,4
1 INRIA, Paris, France
2 CITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPMC, Paris, France
5 LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both add and remove operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, eventual consistency promises better availability and performance [12][13]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas that provide best seller lists, user reviews, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well-known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Eventual consistency.
Conflicts-Free Replicated Data Types*

Marc Shapiro1,5, Nuno Preguiça1,2, Carlos Baquero3, and Marek Zawirska1,4

1 INRIA, Paris, France
2 CITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPMC, Paris, France
5 LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicable Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both add and remove operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or IP2P systems, eventual consistency promises better availability and performance [12]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

©2007 ACM 978-1-59593-591-5/07/0010...$5.00.

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com. One of the reasons is that while its infrastructure is deployed across a vast number of sites, the underlying system is tightly coupled, with each site having a local copy of each data item. This can make it difficult to ensure data consistency, especially when different sites have different views of the same data. Furthermore, the system must be able to handle high levels of traffic and maintain high availability even in the face of failures.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. We have found that loosely coupled, service-based architectures are more resilient than tightly coupled, service-based architectures. In this model for storage technologies like key-value stores, customers should be able to manage their own data and have complete control over their data. Dynamo is a new distributed key-value store that is designed to provide high availability and low latency, even in the face of failures.

Dynamo is a new distributed key-value store that is designed to provide high availability and low latency, even in the face of failures. Dynamo uses a synthesis of well known techniques to achieve scalability and availability: It is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Eventual consistency.
Eventual consistency.
Deterministic Parallelism
Deterministic Parallelism

(observably)
What does this program do?
What does this program do?

```
3
putMVar

4
putMVar

num

MVar

v
```
What does this program do?
What does this program do?
What does this program do?

\[ p = \text{do} \]
What does this program do?

\[
p = \text{do}\n\text{num} \leftarrow \text{newEmptyMVar}
\]
What does this program do?

\[ p = \text{do} \]
\[ \text{num} \leftarrow \text{newEmptyMVar} \]
\[ \text{forkIO (putMVar num 3)} \]
What does this program do?

```haskell
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
```

Diagram:
- `num` is a new empty `MVar`.
- `forkIO (putMVar num 3)` creates a new `MVar` with value `3`.
- `forkIO (putMVar num 4)` creates a new `MVar` with value `4`.
- `takeMVar` is used to retrieve the value from `num`.
What does this program do?

\[
p = \text{do}
\begin{align*}
\text{num} & \leftarrow \text{newEmptyMVar} \\
\text{forkIO} \ (\text{putMVar} \ \text{num} \ 3) \\
\text{forkIO} \ (\text{putMVar} \ \text{num} \ 4) \\
\text{v} & \leftarrow \text{takeMVar} \ \text{num}
\end{align*}
\]
What does this program do?

\[
p = \text{do}
\]

\[
\text{num} \leftarrow \text{newEmptyMVar}
\]

\[
\text{forkIO} (\text{putMVar} \text{ num} 3)
\]

\[
\text{forkIO} (\text{putMVar} \text{ num} 4)
\]

\[
\text{v} \leftarrow \text{takeMVar} \text{ num}
\]

\[
\text{return} \text{ v}
\]
landin:lvarexamples lkuper$ make data-race-example
ghc -O2 data-race-example.hs -rtsopts -threaded
Linking data-race-example ...
while true; do ./data-race-example +RTS -N2; done
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v

Disallow multiple writes?
Disallow multiple writes?

```haskell
p = do
    num <- newEmptyMVar
    forkIO (putMVar num 3)
    forkIO (putMVar num 4)
    v <- takeMVar num
    return v
```
Disallow multiple writes?

 Tesler and Enea, 1968
 Arvind et al., 1989

\[ p = \text{do} \]
\[
\text{num} \leftarrow \text{newEmptyMVar}
\]
\[
\text{forkIO (putMVar num 3)}
\]
\[
\text{forkIO (putMVar num 4)}
\]
\[
\text{v} \leftarrow \text{takeMVar num}
\]
\[
\text{return v}
\]
Disallow multiple writes?

Tesler and Enea, 1968
Arvind et al., 1989

\[ p :: \text{Par Int} \]
\[ p = \text{do} \]
\[ \text{num} \leftarrow \text{new} \]
\[ \text{fork} \ (\text{put} \ \text{num} \ 3) \]
\[ \text{fork} \ (\text{put} \ \text{num} \ 4) \]
\[ v \leftarrow \text{get} \ \text{num} \]
\[ \text{return} \ v \]
Disallow multiple writes?

 Tesler and Enea, 1968
 Arvind et al., 1989

```
p :: Par Int
p = do
  num <- new
  fork (put num 3)
  fork (put num 4)
  v <- get num
return v
```

IVars

```
./ivar-example +RTS -N2
ivar-example: multiple put
```
Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
    num <- new
    fork (put num 3)
    fork (put num 4)
    v <- get num
    return v
```
Deterministic programs that single-assignment forbids

\[
p :: \text{Par Int}
\]
\[
p = \text{do}
\]
\[
\text{num} <- \text{new}
\]
\[
\text{fork} (\text{put} \quad \text{num} \quad 4)
\]
\[
\text{fork} (\text{put} \quad \text{num} \quad 4)
\]
\[
\text{v} <- \text{get} \quad \text{num}
\]
\[
\text{return} \quad \text{v}
\]
Deterministic programs that single-assignment forbids

\[
p :: \text{Par Int}
\]
\[
p = \begin{aligned}
&\text{do} \\
&\quad \text{num <- new} \\
&\quad \text{fork (put num 4)} \\
&\quad \text{fork (put num 4)} \\
&\quad v <- \text{get num} \\
&\quad \text{return v}
\end{aligned}
\]

`.repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put`
Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork put num 4
  fork put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```
Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork put num 4
  fork put num 4
  v <- get num
  return v

./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

```
do
  fork (insert t "0")
fork (insert t "1100")
fork (insert t "1111")
v <- get t
return v
```
Deterministic programs that single-assignment forbids

\[
p :: \text{Par Int}
\]
\[
p = \text{do}
\]
\[
\text{num} \leftarrow \text{new}
\]
\[
\text{fork} \quad \text{put} \quad \text{num} \quad 4
\]
\[
\text{fork} \quad \text{put} \quad \text{num} \quad 4
\]
\[
\text{v} \leftarrow \text{get} \quad \text{num}
\]
\[
\text{return} \quad \text{v}
\]

\[
./\text{repeated-4-ivar} +\text{RTS} -\text{N2}
\]
\[
\text{repeated-4-ivar: multiple put}
\]

\[
\textit{do}
\]
\[
\text{fork} \quad (\text{insert t} \quad "0")
\]
\[
\text{fork} \quad (\text{insert t} \quad "1100")
\]
\[
\text{fork} \quad (\text{insert t} \quad "1111")
\]
\[
\text{v} \leftarrow \text{get} \quad \text{t}
\]
\[
\text{return} \quad \text{v}
\]
LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
LVars: Multiple *monotonic* writes

- **Provably deterministic** [Kuper and Newton, FHPC ’13]
- Contents grow *monotonically* with each write
LVars: Multiple monotonic writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
- Contents grow monotonically with each write
- Pluggable application-specific types

```haskell
import Control.LVish
import Data.LVar.Set
```
LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```haskell
import Control.LVish
import Data.LVar.Pair
```
LVars: Multiple monotonic writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
- Contents grow monotonically with each write
- Pluggable application-specific types

```
import Control.LVish
import Data.LVar.Map
```
LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC '13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```haskell
import Control.LVish
import Data.LVar.Counter
```
LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```haskell
import Control.LVish
import Data.LVar.IVar
```
LVars: Multiple *monotonic* writes

- Provably deterministic [Kuper and Newton, FHPC ’13]
- Contents grow *monotonically* with each write
- Pluggable application-specific types

```haskell
import Control.LVish
import Data.LVar.IVar
```

- `cabal install lvish` today!
LVars: Multiple monotonic writes

Raise an error, since $3 \sqcup 4 = \top$

```plaintext
do
  fork (put num 3)
  fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

```plaintext
do
  fork (put num 4)
  fork (put num 4)
```
LVars: Multiple *monotonic* writes
Overlapping writes are no problem

do
fork (insert t "0")
fork (insert t "1100")
fork (insert t "1111")
v <- get t
return v
LVars: Threshold reads

\[
\begin{align*}
\text{nn} & \rightarrow \text{newPair} \\
& \text{fork (putFst nn 0)} \\
& \text{fork (putSnd nn 1)} \\
& v \leftarrow \text{getSnd nn} \\
& \text{return } v \quad \text{-- returns 1}
\end{align*}
\]
LVars: Threshold reads

\[ nn \]

\[ \text{do} \]

\[ nn \leftarrow \text{newPair} \]

\[ \text{fork (putFst } nn \ 0 ) \]

\[ \text{fork (putSnd } nn \ 1 ) \]

\[ v \leftarrow \text{getSnd } nn \]

\[ \text{return } v \quad -- \text{returns } 1 \]
LVars: Threshold reads

\[
\begin{align*}
nn &\quad \top \\
(0, 0) &\quad (0, 1) &\quad \cdots &\quad (1, 0) &\quad (1, 1) &\quad \cdots \\
(\bot, 0) &\quad (\bot, 1) &\quad \cdots &\quad (0, \bot) &\quad (1, \bot) &\quad \cdots \\
&\quad \downarrow &\quad \downarrow &\quad \downarrow &\quad \downarrow &\quad \cdots \\
&\quad \text{getSnd} &\quad \text{"tripwire"}
\end{align*}
\]

\[
\text{do}
\]

\[
\begin{align*}
\text{nn} &\leftarrow \text{newPair} \\
\text{fork} &\left( \text{putFst} \ nn \ 0 \right) \\
\text{fork} &\left( \text{putSnd} \ nn \ 1 \right) \\
v &\leftarrow \text{getSnd} \ nn \\
\text{return} &\ v \quad \text{-- returns} \ 1
\end{align*}
\]
LVars: Threshold reads

\[
\begin{align*}
\text{do} & \quad \text{nn} \leftarrow \text{newPair} \\
& \quad \text{fork (putFst nn 0)} \\
& \quad \text{fork (putSnd nn 1)} \\
& \quad \text{v} \leftarrow \text{getSnd nn} \\
& \quad \text{return v} \quad \text{-- returns 1}
\end{align*}
\]
LVars: Threshold reads

```do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v  -- returns 1
```
LVars: Threshold reads

do

nn <- newPair
fork (putFst nn 0)
fork (putSnd nn 1)
v <- getSnd nn
return v  -- reduces 1
LVars: Threshold reads

\[
\begin{align*}
\text{do} & \\
& \text{nn} \leftarrow \text{newPair} \\
& \text{fork (putFst nn 0)} \\
& \text{fork (putSnd nn 1)} \\
& v \leftarrow \text{getSnd nn} \\
& \text{return } v \quad \text{--- returns } 1
\end{align*}
\]
LVars: Threshold reads

nn

```
do
    nn <- newPair
    fork (putFst nn 0)
    fork (putSnd nn 1)
    v <- getSnd nn
    return v -- returns 1
```
LVars: Threshold reads

The threshold set must be \textit{pairwise incompatible}

\textbf{do}

\[
nn \leftarrow \text{newPair} \\
\text{fork (putFst } \text{nn 0)} \\
\text{fork (putSnd } \text{nn 1)} \\
v \leftarrow \text{getSnd } \text{nn} \\
\text{return } v \quad \text{-- returns 1}
\]
Monotonicity enables deterministic parallelism

Kahn, 1974
Monotonicity enables deterministic parallelism

\[ f \text{ is monotonic iff, for a given } \leq, \]
\[ x \leq y \implies f(x) \leq f(y) \]
Monotonicity enables deterministic parallelism

Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation: a machine need not have all of its input to start computing, since future input concerns only future output.

The kind of parallel programming we have studied in this paper is severely limited: it can produce only determinate programs.

Kahn, 1974
Challenge problem

In a directed graph:

- find the connected component of all nodes within \( k \) hops of a vertex \( v \)
- and compute a function \( \text{analyze} \) over each vertex in that component
- making the set of results available asynchronously to other computations
Challenge problem

- We compared two implementations:
  - `Control.Parallel.Strategies`
  - Our prototype LVar library (tracking visited nodes in an LVar)
- Level-sync breadth-first traversal, $k = 10$
- Random graph; 320K edges; 40K nodes
- Varying:
  - number of cores
  - amount of work done by `analyze`
Challenge problem: Strategies vs. LVars

Speedup over seq with 1µs analyze function

- par/pseq (1µs)
- LVar (1µs)

Speedup over seq with 8µs analyze function

- par/pseq (8µs)
- LVar (8µs)

Speedup over seq with 16µs analyze function

- par/pseq (16µs)
- LVar (16µs)

Speedup over seq with 32µs analyze function

- par/pseq (32µs)
- LVar (32µs)
Challenge problem: Strategies vs. LVars

Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed, this is a crucial property since it allows parallel operation: a machine need not have all of its input to start computing, since future input concerns only future output.

- Average time from start of program to first invocation of `analyze`:
  - Strategies version: 64.64 ms
  - LVar version: 0.18 ms
Deterministic Parallelism
via monotonic writes
and threshold reads.
Logic and Lattices for Distributed Programming

ABSTRACT

In recent work, we observed that the problems with Convergent Modules (CMs) (1) are unsolvable in general, and (2) are uninteresting in practice. We propose a simple semantics for CMs, in which the elements of a CM are semilattices. This approach is based on the idea that CMs can be used to reason about distributed systems, and that semilattices provide a natural framework for this reasoning. We then show how to use semilattices to model CMs, and how to use CMs to model semilattices. We conclude by discussing the implications of our results for the design and implementation of CMs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the need for efficient and scalable programming becomes more important. One approach to achieving these goals is to use distributed systems, in which data is stored and managed by a collection of machines. Distributed systems have the advantage of being able to handle large amounts of data, and of providing a natural framework for reasoning about distributed programs. However, distributed systems also have the disadvantage of being difficult to reason about.

Convergent Modules (CMs) are an approach to programming that attempts to solve this problem by providing a simple semantics for CMs. CMs are based on the idea that a CM is a set of elements, each of which is a semilattice. This approach is based on the idea that CMs can be used to reason about distributed systems, and that semilattices provide a natural framework for this reasoning. We then show how to use semilattices to model CMs, and how to use CMs to model semilattices. We conclude by discussing the implications of our results for the design and implementation of CMs.

5. CONCLUSION

In this paper, we have presented a simple semantics for CMs, in which the elements of a CM are semilattices. We then showed how to use semilattices to model CMs, and how to use CMs to model semilattices. We concluded by discussing the implications of our results for the design and implementation of CMs.

The authors are grateful to the anonymous reviewers for their helpful comments. This work was supported by the National Science Foundation under grant CCF-1018711.

Neil Conway  
UC Berkeley  
nconway@cs.berkeley.edu

William R. Marczak  
Portland State University  
wm@cs.pdx.edu

Joseph M. Hellerstein  
UC Berkeley  
hellerstein@cs.berkeley.edu

Peter Alvaro  
UC Berkeley  
palvaro@cs.berkeley.edu
In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency guarantees but require the programmer to ensure lattice properties. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency guarantees but require the programmer to ensure lattice properties. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency guarantees but require the programmer to ensure lattice properties. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency guarantees but require the programmer to ensure lattice properties. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency guarantees but require the programmer to ensure lattice properties.
Logic and Lattices for Distributed Programming

Neil Conway  
UC Berkeley  
nrc@cs.berkeley.edu

William M. bigger  
UC Berkeley  
wrm@cs.berkeley.edu

Joseph M. Hellerstein  
UC Berkeley  
hellerstein@cs.berkeley.edu

Peter Alvaro  
UC Berkeley  
palvaro@cs.berkeley.edu

ABSTRACT

As cloud computing becomes increasingly common, the ability to manage and protect data and services becomes more critical. Many applications use distributed systems for scalability and fault tolerance. However, managing distributed systems can be challenging, especially for systems that scale gracefully during runtime.

In this paper, we present Bloom, a logic programming language that supports reasoning about distributed systems. Bloom provides a framework for reasoning about the correctness of distributed algorithms, and it can be used to verify the correctness of distributed systems.

1. INTRODUCTION

As cloud computing becomes increasingly common, the ability to manage and protect data and services becomes more critical. Many applications use distributed systems for scalability and fault tolerance. However, managing distributed systems can be challenging, especially for systems that scale gracefully during runtime.

2. CONVERGENT MODULES

Convergent Modules are a framework for reasoning about distributed systems. They provide a way to express a set of requirements that a distributed system must satisfy. In this paper, we present Bloom, a logic programming language that supports reasoning about Convergent Modules.

3. MONOTONICITY

Monotonicity is a property of a logic program that ensures that the program is consistent with the data it represents. In this paper, we present Bloom, a logic programming language that supports reasoning about Convergent Modules.

4. CRDTs

CRDTs are a framework for reasoning about distributed systems that are not based on a single source of truth. In this paper, we present Bloom, a logic programming language that supports reasoning about Convergent Modules.

5. CALM

CALM is a framework for reasoning about distributed systems that are not based on a single source of truth. In this paper, we present Bloom, a logic programming language that supports reasoning about Convergent Modules.

6. RELATED WORK

Related work includes research on logic programming and distributed systems. In this paper, we present Bloom, a logic programming language that supports reasoning about Convergent Modules.

7. CONCLUSION

In this paper, we present Bloom, a logic programming language that supports reasoning about distributed systems. Bloom provides a framework for reasoning about the correctness of distributed algorithms, and it can be used to verify the correctness of distributed systems.

8. ACKNOWLEDGMENTS

We would like to thank the reviewers for their feedback on our paper. We also would like to thank the University of California, Berkeley, for providing financial support for our research.

9. REFERENCES


Logic and Lattices for Distributed Programming

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein
UC Berkeley, University of Portland

ABSTRACT

We present Bloom, a logic language for specifying and reasoning about software modules and their interactions. Bloom's key features include support for both observable and hidden state, and a lattice semantics that provides a powerful lens for reasoning about software module interactions. Bloom's state is guaranteed to be eventually consistent with respect to a lattice preorder that captures both causal and commutativity constraints. Bloom's lattice semantics can be automatically verified by Convergent Modules, a system we use to develop several practical distributed programs. Bloom's efficiency is demonstrated on a simple shopping cart application.

1. INTRODUCTION

In the world of distributed computing, we are witnessing a growing interest in systems that enable software to be easily developed, deployed, and maintained across a network of multiple computers. Cloud computing, in particular, promises to transform the way in which we design, develop, and deploy software. Cloud computing services allow developers to easily scale their applications by adding or removing computing resources on demand. This scalability makes cloud computing a powerful tool for building software that can handle large volumes of data and support a large number of users. However, the challenges of designing and implementing scalable systems remain significant.

We present Bloom, a logic language for specifying and reasoning about software modules and their interactions. Bloom's key features include support for both observable and hidden state, and a lattice semantics that provides a powerful lens for reasoning about software module interactions. Bloom's state is guaranteed to be eventually consistent with respect to a lattice preorder that captures both causal and commutativity constraints. Bloom's lattice semantics can be automatically verified by Convergent Modules, a system we use to develop several practical distributed programs. Bloom's efficiency is demonstrated on a simple shopping cart application.

In recent work, we observed that the problems with Convergent Modules are similar to those in database theory, in particular, the problem of maintaining eventual consistency in the presence of concurrent updates. To address these issues, we propose Bloom, a logic language that allows developers to specify software modules and their interactions, and automatically verify that these specifications are consistent with respect to a lattice preorder. Bloom's lattice semantics provides a powerful tool for reasoning about software module interactions.

In this paper, we present Bloom, an extension to Bloom, which allows developers to specify software modules and their interactions, and automatically verify that these specifications are consistent with respect to a lattice preorder. Bloom's lattice semantics provides a powerful tool for reasoning about software module interactions.

In logic languages such as Bloom, Convergent Modules can automatically verify that program modules achieve consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the CRDT framework of strongly consistent storage [—4 ,—]. In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees but require the programmer to ensure lattice properties easy to inspect and test, but provides only partial failure—a difficulty inherent to the CRDT framework. We propose Bloom, which allows developers to specify software modules and their interactions, and automatically verify that these specifications are consistent with respect to a lattice preorder.
July 2012

Logic and Lattices for Distributed Programming

Neil Conway
William R. Marczak
Palvaro
Joseph M. Hellerstein

ABSTRACT

As cloud computing becomes increasingly common, the demand for distributed systems grows. A key aspect of distributed systems is their ability to handle faults and maintain consistency in the face of uncertainty. In this paper, we present a calculus for reasoning about distributed algorithms that supports concurrent operations and is based on the algebraic notion of a lattice. We extend the standard lattice theory to include both partial and full consistency, and show how our framework can be applied to a variety of distributed algorithms.

1. INTRODUCTION

In this paper, we present a calculus for reasoning about distributed algorithms that supports concurrent operations and is based on the algebraic notion of a lattice. We extend the standard lattice theory to include both partial and full consistency, and show how our framework can be applied to a variety of distributed algorithms.

2. LATTICES AND CRDTs

In this section, we introduce the concepts of lattices and CRDTs (Conflict-Free Replicated Data Types). We show how these concepts can be used to reason about distributed algorithms, and present a calculus for reasoning about distributed algorithms that supports concurrent operations and is based on the algebraic notion of a lattice.

3. MONOTONIC LOGIC

In this section, we introduce the concept of monotonic logic and show how it can be used to reason about distributed algorithms. We present a calculus for reasoning about distributed algorithms that supports concurrent operations and is based on the algebraic notion of a lattice.

4. APPLICATIONS

In this section, we present several applications of our calculus to distributed algorithms, including a distributed algorithm for managing a shopping cart, a distributed algorithm for managing a courseware application, and a distributed algorithm for managing a student-study team assignment.

5. CONCLUSIONS

In this section, we summarize our results and present some directions for future work.
Logic and Lattices for Distributed Programming

Neil Conway
William R. Marczak
Joseph M. Hellerstein
Palvaro@cs.berkeley.edu
Maier@cs.pdx.edu
William R. Marczak
Brian Atkinson
Neil Conway
Joseph M. Hellerstein
Palvaro@cs.berkeley.edu
Maier@cs.pdx.edu

ABSTRACT

Convergent Modules (CMs) are an approach to consistent, distributed consistency with a new set of guarantees. Their nice properties hold for a large module, resulting in software that is easy to test, maintain, and trust. Like CRDTs, CMs are non-blocking and do not require coordination. Unlike CRDTs, CMs provide standard semantic guarantees such as idempotence and associativity, and are confederated. The main promise of CMs is the ability to reason about consistency in the presence of arbitrary conflicts, in effect by providing strong consistency without coordination. CMs can be used to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo.

The user of the libStatebox is an open-source library that merges conflicting guarantees regarding message reordering and retry. For example, encapsulated modules whose public methods provide certain guarantees but require the programmer to ensure lattice properties for their own operations. This is outside the scope of CRDT guarantees. Taken together, the problems with Convergent Modules (CMs) and CRDTs present two main problems: (1) the programmer bears responsibility for ensuring lattice properties for their own operations, which is not for application logic in general. As an example of this, consider a shopping cart system.

Students are updated consistently to reflect the purchases made by another application replica. The application assigns students into study teams. It uses two CRDTs: one for Students and another for Teams. Concurrently, the problems with CMs and CRDTs bear responsibility for ensuring lattice properties for their own operations, which is not for application logic in general. As an example of this, consider a shopping cart system. Students are updated consistently to reflect the purchases made by another application replica. The application assigns students into study teams. It uses two CRDTs: one for Students and another for Teams. Concurrently, the problems with CMs and CRDTs hold for a large module, resulting in software that is easy to test, maintain, and trust. Like CRDTs, CMs are non-blocking and do not require coordination. Unlike CRDTs, CMs provide standard semantic guarantees such as idempotence and associativity, and are confederated. The main promise of CMs is the ability to reason about consistency in the presence of arbitrary conflicts, in effect by providing strong consistency without coordination. CMs can be used to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo.

In this paper we present Bloom, an extension to Bloom that supports the power of Bloom to support message reordering and retry. Bloom is an open-source library that merges conflicting guarantees regarding message reordering and retry. For example, encapsulated modules whose public methods provide certain guarantees but require the programmer to ensure lattice properties for their own operations. This is outside the scope of CRDT guarantees. Taken together, the problems with Convergent Modules (CMs) and CRDTs present two main problems: (1) the programmer bears responsibility for ensuring lattice properties for their own operations, which is not for application logic in general. As an example of this, consider a shopping cart system. Students are updated consistently to reflect the purchases made by another application replica. The application assigns students into study teams. It uses two CRDTs: one for Students and another for Teams. Concurrently, the problems with CMs and CRDTs hold for a large module, resulting in software that is easy to test, maintain, and trust. Like CRDTs, CMs are non-blocking and do not require coordination. Unlike CRDTs, CMs provide standard semantic guarantees such as idempotence and associativity, and are confederated. The main promise of CMs is the ability to reason about consistency in the presence of arbitrary conflicts, in effect by providing strong consistency without coordination. CMs can be used to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo.

In the past, people have used Bloom as an open-source library to achieve strong consistency. Bloom is an open-source library that merges conflicting guarantees regarding message reordering and retry. For example, encapsulated modules whose public methods provide certain guarantees but require the programmer to ensure lattice properties for their own operations. This is outside the scope of CRDT guarantees. Taken together, the problems with Convergent Modules (CMs) and CRDTs present two main problems: (1) the programmer bears responsibility for ensuring lattice properties for their own operations, which is not for application logic in general. As an example of this, consider a shopping cart system. Students are updated consistently to reflect the purchases made by another application replica. The application assigns students into study teams. It uses two CRDTs: one for Students and another for Teams. Concurrently, the problems with CMs and CRDTs hold for a large module, resulting in software that is easy to test, maintain, and trust. Like CRDTs, CMs are non-blocking and do not require coordination. Unlike CRDTs, CMs provide standard semantic guarantees such as idempotence and associativity, and are confederated. The main promise of CMs is the ability to reason about consistency in the presence of arbitrary conflicts, in effect by providing strong consistency without coordination. CMs can be used to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo.
July 2012

updates to data items in a key-value store

the user of the LIple Statebox is an open-source library that merges conflicting guarantees regarding message reordering and retry.

For example, encapsulated modules whose public methods provide certain guarantees. Recent research on Convergent Modules has received significant attention in strongly consistent storage.

Two different frameworks without incurring the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations.

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage for database systems.

In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency.

In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency. In recent work, we observed that the use of CRDTs ensures that all replicas will eventually agree over time. It never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.

Monotonic Logic encourages the safe composition of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent. Over time, it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.

In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency. In recent work, we observed that the use of CRDTs ensures that all replicas will eventually agree over time. It never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.

Monotonic Logic encourages the safe composition of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent. Over time, it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.

In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency. In recent work, we observed that the use of CRDTs ensures that all replicas will eventually agree over time. It never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent. Over time, it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.

Monotonic Logic encourages the safe composition of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent. Over time, it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent. Over time, it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which establishes that all monotonic programs are guaranteed to be eventually consistent.
Logic and Lattices for Distributed Programming

Neil Conway, William R. Marczak, and Joseph M. Hellerstein

Abstract

Recent approaches to transactional memory (TM) have enabled developers to reason about distributed consistency in the presence of faults and replication. However, these approaches are not well suited for reasoning about the correctness of application logic because they focus on the guarantees provided by underlying hardware and software, rather than on the correctness of the logic itself.

In this paper, we present Bloom, a logic language that allows developers to reason about the correctness of application logic in the presence of faults and replication. Bloom is based on the theory of lattices, which is a powerful formalism for reasoning about distributed consistency.

We show how Bloom can be used to reason about the correctness of application logic in the presence of faults and replication. We demonstrate the power of Bloom by using it to reason about the correctness of a variety of distributed systems, including a replicated, fault-tolerant courseware application.

1. Introduction

As cloud computing becomes increasingly common, the necessity of ensuring consistent data models in distributed systems is becoming more important. However, most traditional approaches to ensuring consistency are not well suited for reasoning about the correctness of application logic, because they focus on the guarantees provided by underlying hardware and software, rather than on the correctness of the logic itself.

Bloom addresses this problem by allowing developers to reason about the correctness of application logic in the presence of faults and replication. Bloom is based on the theory of lattices, which is a powerful formalism for reasoning about distributed consistency.

We show how Bloom can be used to reason about the correctness of application logic in the presence of faults and replication. We demonstrate the power of Bloom by using it to reason about the correctness of a variety of distributed systems, including a replicated, fault-tolerant courseware application.

2. Related Work

There has been a lot of recent work on reasoning about distributed consistency. For example, CRDTs (Conflict-Free Replicated Data Types) have been widely used in practice, but there is little theoretical understanding of how to reason about the correctness of application logic in the presence of faults and replication.

One approach to reasoning about distributed consistency is to use a formalism for reasoning about distribution, such as the theory of lattices. Another approach is to use a formalism for reasoning about the correctness of application logic, such as the theory of lattices.

Bloom combines the benefits of both of these approaches by allowing developers to reason about the correctness of application logic in the presence of faults and replication. Bloom is based on the theory of lattices, which is a powerful formalism for reasoning about distributed consistency, and it is well suited for reasoning about the correctness of application logic.

3. Bloom

Bloom is a logic language that allows developers to reason about the correctness of application logic in the presence of faults and replication. Bloom is based on the theory of lattices, which is a powerful formalism for reasoning about distributed consistency.

Bloom is a simple, easy-to-use language that allows developers to reason about the correctness of application logic in the presence of faults and replication. Bloom is well suited for reasoning about the correctness of application logic because it is based on the theory of lattices, which is a powerful formalism for reasoning about distributed consistency.

We show how Bloom can be used to reason about the correctness of application logic in the presence of faults and replication. We demonstrate the power of Bloom by using it to reason about the correctness of a variety of distributed systems, including a replicated, fault-tolerant courseware application.
July 2012

August 2012

long, dark winter

May 2013

June 2013

“It’s very interesting stuff.”

“I’d love to link a few co-workers to your blog post...”
Logic and Lattices for Distributed Programming

**Abstract**

In this paper, we present Bloom, an extension to Bloom that takes inspiration from both these traditions. Bloom generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary language constructs. We show how the Bloom interpreter can be generalized to support lattices and extend the power of CALM analysis to whole programs containing arbitrary language constructs. We show how the Bloom interpreter can be generalized to support lattices and extend the power of CALM analysis to whole programs containing arbitrary language constructs.

1. INTRODUCTION

In recent work, we observed that the logic languages such as Bloom and MONOTONICITY have been used to model distributed systems. We also noticed that the logic languages such as Bloom and MONOTONICITY have been used to model distributed systems. We also noticed that the logic languages such as Bloom and MONOTONICITY have been used to model distributed systems.

2. CONVERGENT MODULES

In recent years, there has been interest in achieving application-level consistency in large-scale distributed systems. In this approach, convergence is not for application logic in general. As an example of this approach, consider the following:

   Consider a small module that records the number of students in a class. It updates the number of students in the class whenever a student is added or removed. This approach has roots in research in database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency.

3. CRDTS

CRDTs: one for application-level state is inconsistent unless the derived value for application logic is not for application logic in general. As an example of this, consider the following:

   Concurrent Modules. In this approach, convergence is not for application logic in general. As an example of this, consider the following:

   Concurrent Modules. In this approach, convergence is not for application logic in general. As an example of this, consider the following:

   Concurrent Modules. In this approach, convergence is not for application logic in general. As an example of this, consider the following:

   Concurrent Modules. In this approach, convergence is not for application logic in general. As an example of this, consider the following:
July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013

October 2013

“<b>It’s very interesting stuff.”</b>

FHPC ’13

POPL ’14

“I’d love to link a few co-workers to your blog post...”

Logic and Lattices for Distributed Programming

Neil Conway

William R. Marczak

Joseph M. Hellerstein

Palvaro@cs.berkeley.edu

David Maier

Maier@cs.pdx.edu
Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

ABSTRACT

Recent research has generated a variety of new models for representing and reasoning about distributed systems. In this paper, we describe a concrete, algebraic semantics for a new logic called Monotonic Logic (CALM). This logic integrates Bloommon, and shows how Bloom can be generalized to support lattices and extends the power of logic programming. Finally, we show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using CALM analysis to whole programs containing arbitrary lattices.

1. INTRODUCTION

In logic languages such as Bloom, CALM analysis can automate program understanding by inspecting and testing programs. An example of this is the Bloom interpreter, which proves that logically monotonic programs are guaranteed to be eventually consistent. This is not for application logic in general, but for application logic in general, as an example of this is the Bloom interpreter, which proves that logically monotonic programs are guaranteed to be eventually consistent. This is not for application logic in general, but for application logic in general, as an example of this is the Bloom interpreter, which proves that logically monotonic programs are guaranteed to be eventually consistent. This is not for application logic in general, but for application logic in general, as an example of this is the Bloom interpreter, which proves that logically monotonic programs are guaranteed to be eventually consistent.
Dynamo: Amazon’s Highly Available Key-value Store
Guanghe DeCandia, David Hastorun, Madhav Jha, Guranteep Singh, and David DeWitt

Logic and Lattices for Distributed Programming
M.11 Carney, William B. Marzec, Peter Asaro, and E. Leibensperger

In recent work, we observed that the problems with Convergent Modules are simply semantic guarantees. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee. Large CRDTs fling an eventuality, lattice properties easy to inspect and test, but provide only a simple semantic guarantee.
Abstract

In contemporary computer systems, parallelism is usually expressed as concurrent execution of multiple threads, a model which has proven to be an excellent abstraction for many applications. One of the cornerstones of this model is the assumption of determinism, namely, the guarantee that a program will always produce the same externally observable result on multiple runs. We demonstrate the viability of our approach by implementing a language in which this assumption is expressed directly, enabling an expressive and useful style of parallel programming.

We prove that in a language where communication takes place byyconstruction parallel programming models ofy

4 Statebox is an open-source library that merges conflicting
encapsulated modules whose public methods provide certain
recent research).

4 Logic and Lattices for Distributed Programming

1 Introduction

Logic and Lattices for Distributed Programming

MALCOLM CONWAY, NEIL FONG, ANDREW KREUTZ

ABSTRACT

In contemporary computer systems, parallelism is usually expressed as concurrent execution of multiple threads, a model which has proven to be an excellent abstraction for many applications. One of the cornerstones of this model is the assumption of determinism, namely, the guarantee that a program will always produce the same externally observable result on multiple runs. We demonstrate the viability of our approach by implementing a language in which this assumption is expressed directly, enabling an expressive and useful style of parallel programming.

We prove that in a language where communication takes place byyconstruction parallel programming models ofy

4 Statebox is an open-source library that merges conflicting
encapsulated modules whose public methods provide certain
recent research).
1. INTRODUCTION

In recent work, we observed that the current programming frameworks for building distributed systems present two main problems. First, the programmer must develop and manage a collection of libraries for each distributed algorithm they wish to build. This can be burdensome, especially for small-scale applications. Second, the programmer must also manage the interaction of these libraries with other components of the distributed system, such as network communication protocols. This can be complex and error-prone, especially in a large-scale distributed system.

To address these problems, we propose a new model for programming distributed systems. In this model, we introduce a new programming construct called a LVar (Lattice-based Variable). LVars are a generalization of existing programming constructs such as shared memory variables and database transactions. They allow programmers to define shared state in a way that is both flexible and predictable. Additionally, they provide a natural way to express determinism in parallel programs.

1.1 LVar Features

LVars provide several key features that make them well-suited for building distributed applications. First, LVars support dependence, a notion of shared state that is stronger than that provided by existing programming constructs. Dependence ensures that data is only accessed by one thread at a time, which can help prevent data races and other nondeterministic behavior.

Second, LVars support a concept of "freeze," which allows programmers to "lock" a LVar in a way that guarantees determinism. Once a LVar is frozen, it cannot be accessed by any thread except for the one that froze it. This is useful for ensuring that data is accessed in a consistent order, which can be important for debugging and testing.

Finally, LVars are designed to be easy to reason about. They provide a clear separation of concerns between the programmer and the runtime system. The programmer specifies what operations can be performed on a LVar, while the runtime system ensures that these operations are performed in a consistent order. This abstraction can help simplify the design and implementation of distributed systems.

1.2 Related Work

Many algorithms are presented explicitly as fixpoints of monotonic logic. Monotonic logic is a formalism for reasoning about data that is both powerful and easy to reason about. It is also well-suited for reasoning about data races and other forms of nondeterminism.

In this paper, we present Bloom, a new model for building distributed systems. Bloom combines the features of LVars and monotonic logic to provide a powerful and flexible model for building distributed applications. We demonstrate the effectiveness of Bloom by implementing several practical distributed programs using it. We show that Bloom is easy to use and provides significant benefits over existing frameworks.
Freeze After Writing
Quasi-Deterministic Parallel Programming with LVars

1. INTRODUCTION

...
Freeze After Writing
Quasi-Deterministic Parallel Programming with LVars

LVars: Lattice-based Data Structures
for Deterministic Parallelism

Logic and Lattices for Distributed Programming

Conflict-Free Replicated Data Types

Dynamo: Amazon's Highly Available Key-value Store

Dynamo: Amazon's Highly Available Key-value Store

LVars: Lattice-based Data Structures
for Deterministic Parallelism

LVars: Lattice-based Data Structures
for Deterministic Parallelism

LVars: Lattice-based Data Structures
for Deterministic Parallelism

LVars: Lattice-based Data Structures
for Deterministic Parallelism

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming

Logic and Lattices for Distributed Programming
Thank you!

Email: lkuper@cs.indiana.edu
Research blog: composition.al
Project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples

Special thanks to: Ryan Newton, Aaron Turon, Neel Krishnaswami, Sam Tobin-Hochstadt, Amr Sabry, Vincent St-Amour, Neil Conway, Sam Elliott, Mike Bernstein, and the IU PL Wonks.

Photo by kakadu on Flickr. Thanks!