LVars:
Lattice-based Data Structures *for* Deterministic Parallelism

Lindsey Kuper
Indiana University

RICON West, San Francisco, CA, USA
October 29, 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

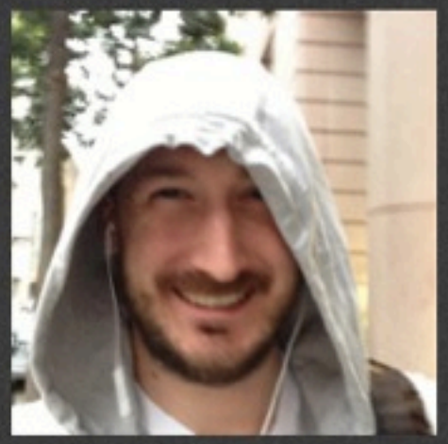
Keystone: Between a ROC and a SOFT Place

Diego Ongaro

PhD Student
Stanford University

The Raft Consensus Algorithm

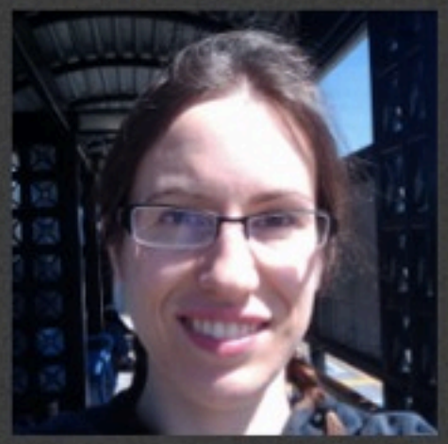
Jeff Hodges



Distributed Systems Engineer
Twitter

Practicalities of Productionizing Distributed Systems

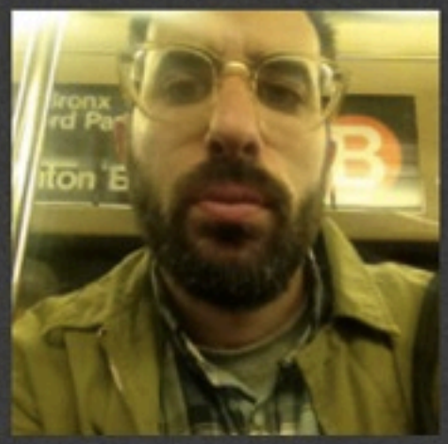
Lindsey Kuper



PhD Student
Indiana University

LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein



Instigator
Code Climate

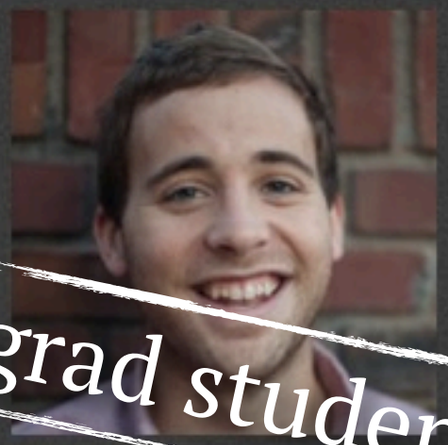
Distributed Systems Archaeology

Jeff Dean

Google Fellow
Google, Inc.

The Tail at Scale: Achieving Rapid Response Times in Large Online Services

Peter Bailis

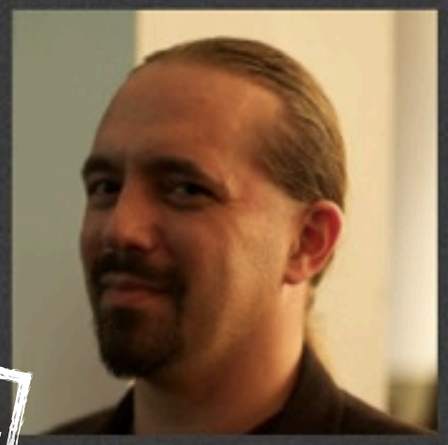


grad student

PhD Student
UC Berkeley

Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

Justin Sheehy



hief Technology Officer
Basho Technologies

Maximum Viable Product

Pat Helland

Architect
Salesforce.com

Keystone: Between a ROC and a SOFT Place

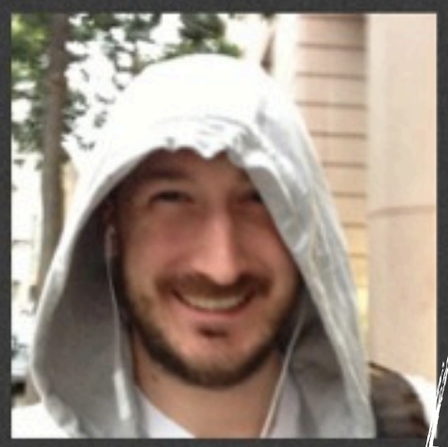
Diego Ongaro

grad student

PhD Student
Stanford University

The Raft Consensus Algorithm

Jeff Hodges



Distributed Systems Engineer
Twitter

Practicalities of Productionizing Distributed Systems

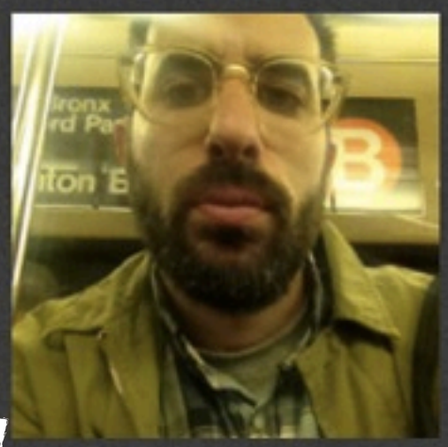
Lindsey Kuper

grad student

PhD Student
Indiana University

LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein



Instigator
Code Climate

Distributed Systems Archaeology

RICON | WEST

Home Blog Venue Speakers Schedule Training After Party Sponsors Register

Jeff Dean

person with real job

Google Fellow
Google, Inc.

The Tail at Scale: Achieving Rapid Response Times in Large Online Services

Peter Bailis

grad student

PhD Student
UC Berkeley

Bad As I Wanna Be: Coordination and Consistency in Distributed Databases

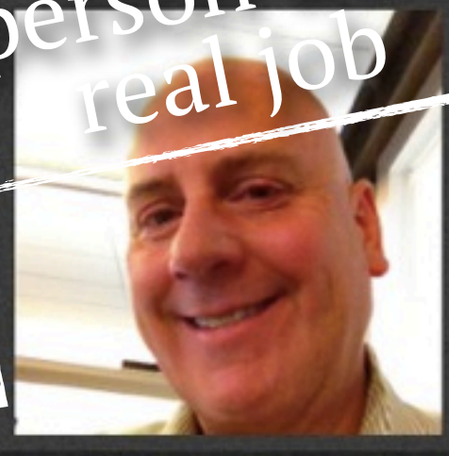
Justin Sheehy

person with real job

Chief Technology Officer
Basho Technologies

Maximum Viable Product

Pat Helland



person with real job

Architect
Salesforce.com

Keystone: Between a ROC and a SOFT Place

Diego Ongaro

grad student

PhD Student
Stanford University

The Raft Consensus Algorithm

Jeff Hodges

person with real job

Distributed Systems Engineer
Twitter

Practicalities of Productionizing Distributed Systems

Lindsey Kuper

grad student

PhD Student
Indiana University

LVars: lattice-based data structures for deterministic parallelism

Michael Bernstein

person with real job

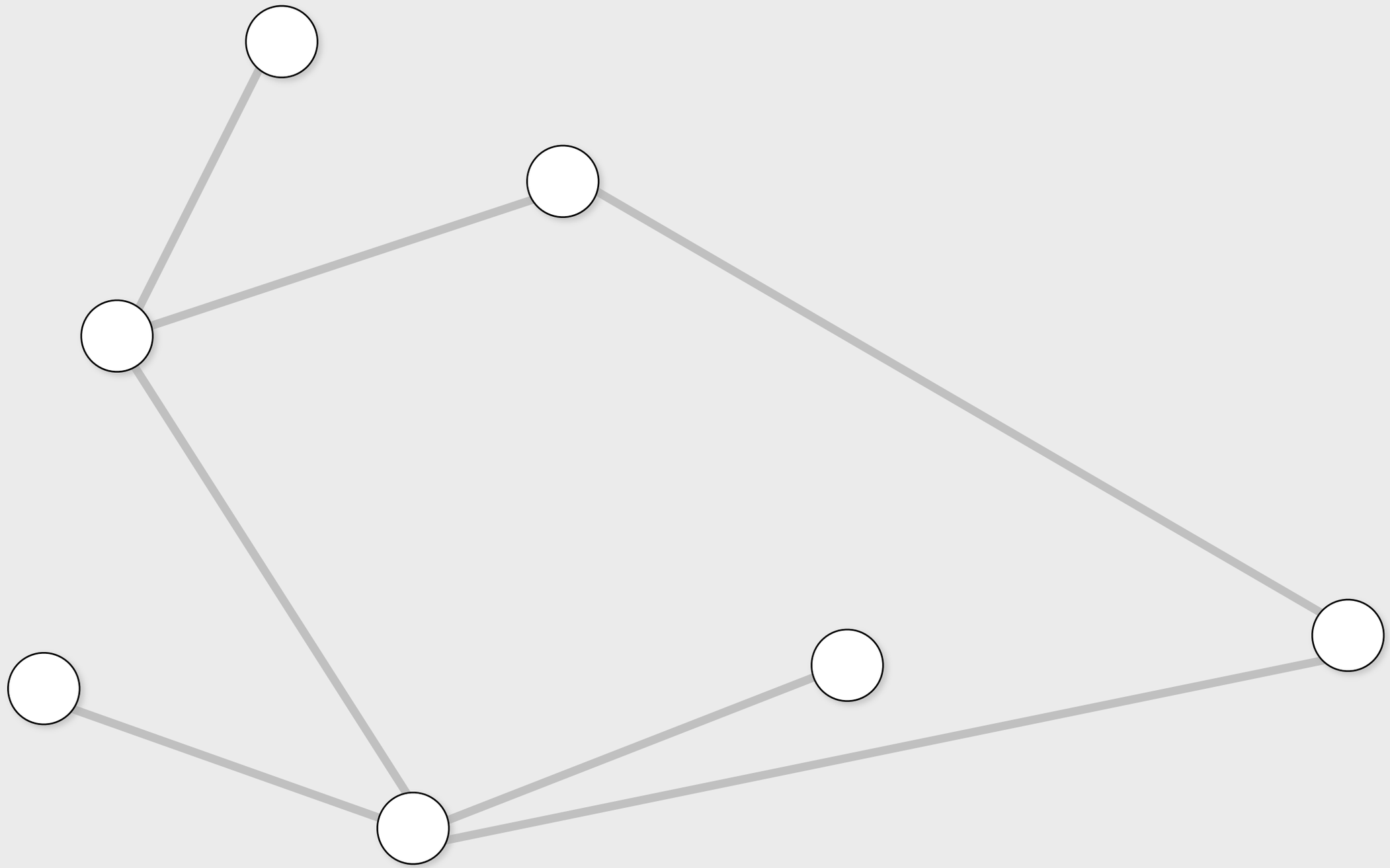
Instigator
Code Climate

Distributed Systems Archaeology

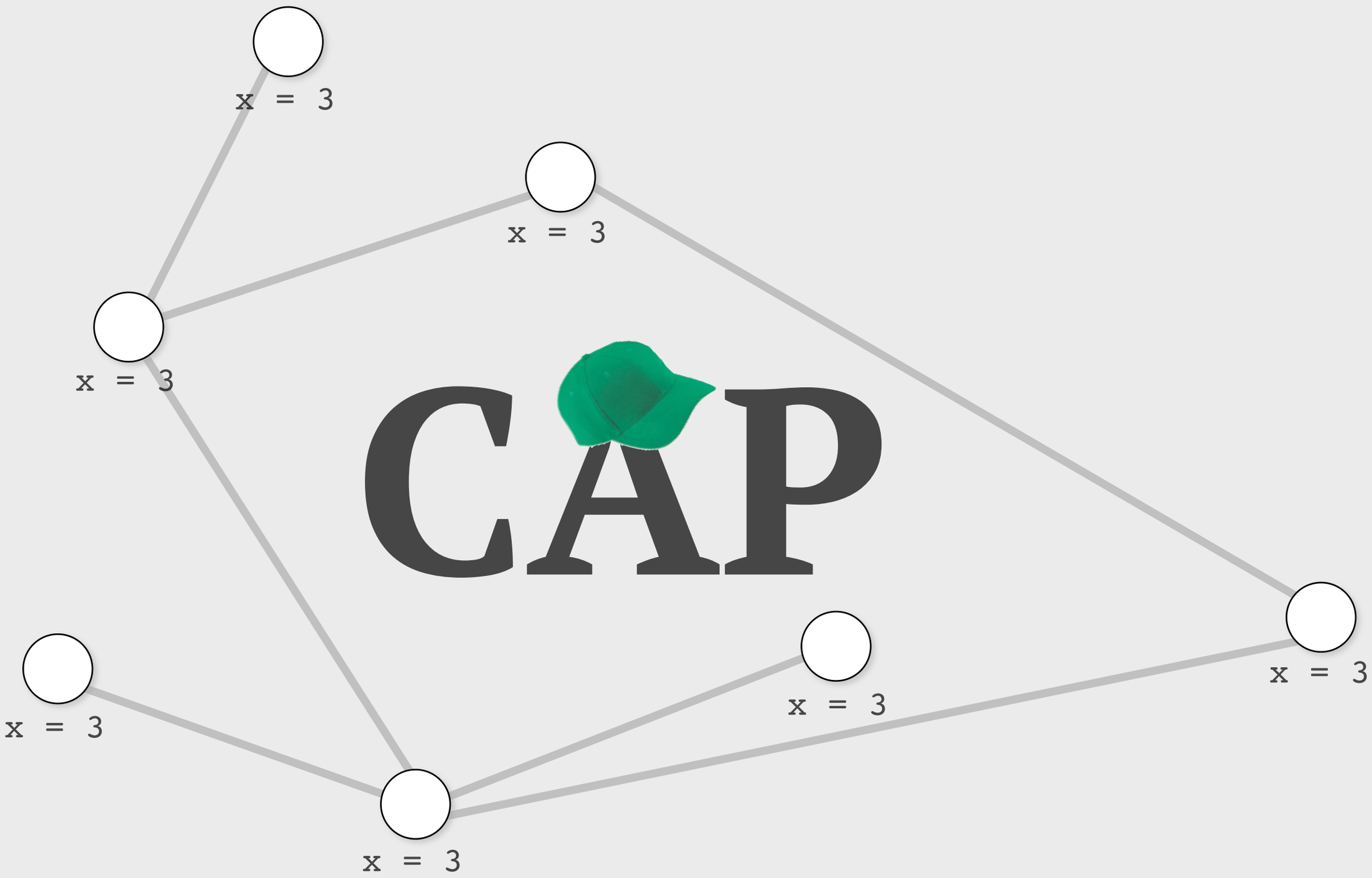
LVars:
Lattice-based Data Structures *for* Deterministic Parallelism

Lindsey Kuper
Indiana University

RICON West, San Francisco, CA, USA
October 29, 2013



CAP


x = 3
x = 3
x = 3
CAP
x = 3
x = 3
x = 3

x = 3
put(x, 4)

x = 3
put(x, 5)

x = 3
put(x, 0)

CAP

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, 5)

x = 3
put(x, 0)

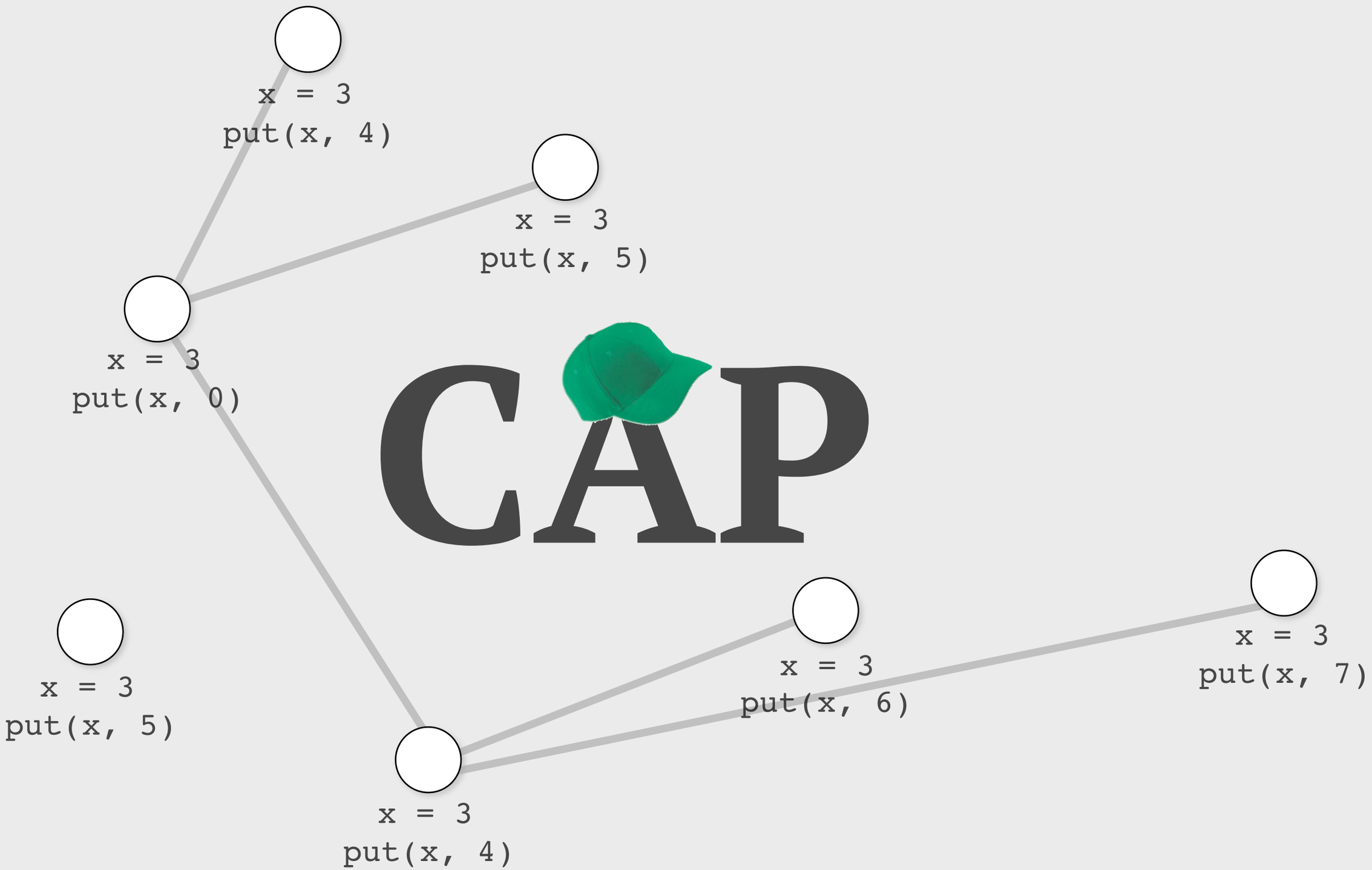
CAP

x = 3
put(x, 5)

x = 3
put(x, 4)

x = 3
put(x, 6)

x = 3
put(x, 7)



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

Algorithms, Management, Measurement, Performance, Design, Reliability.

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.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

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 tornados. 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

***Eventual* consistency.**

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.

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

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.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

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 d items to their shopping cart even if disks are routes are flapping, or data centers are being nados. Therefore, the service responsible for ng carts requires that it can always write to and ata store, and that its data needs to be available data centers.

ures in an infrastructure comprised of millions of r standard mode of operation; there are always a ant number of server and network components t any given time. As such Amazon's software be constructed in a manner that treats failure normal case without impacting availability or

bility and scaling needs, Amazon has developed ge technologies, of which the Amazon Simple lso available outside of Amazon and known as obably the best known. This paper presents the mentation of Dynamo, another highly available ibuted data store built for Amazon's platform. o manage the state of services that have very quirements and need tight control over the 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

***Eventual* consistency.**

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[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 [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

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

ware of the data schema it
thod that is best suited for
application that maintains
"merge" the conflicting
opping cart.

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.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

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 d items to their shopping cart even if disks are routes are flapping, or data centers are being nados. Therefore, the service responsible for ng carts requires that it can always write to and ata store, and that its data needs to be available data centers.

ures in an infrastructure comprised of millions of standard mode of operation; there are always a ant number of server and network components t any given time. As such Amazon's software be constructed in a manner that treats failure normal case without impacting availability or

bility and scaling needs, Amazon has developed ge technologies, of which the Amazon Simple lso available outside of Amazon and known as obably the best known. This paper presents the mentation of Dynamo, another highly available ibuted data store built for Amazon's platform. o manage the state of services that have very quirements and need tight control over the 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

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[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 [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

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 wo... conseq... platfor... is impl...

Amazo... of mil... server... strict c... perfor... growth... of the... outage... customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

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. ..., loosely coupled, service ...ndreds of services. In this ...ed for storage technologies ...e, customers should be able ...ping cart even if disks are ... or data centers are being ...he service responsible for ... it can always write to and ... data needs to be available

...re comprised of millions of ...peration; there are always a ...r and network components ...s such Amazon's software ... manner that treats failure ...t impacting availability or

...eds, Amazon has developed ... which the Amazon Simple ...e of Amazon and known as ...wn. This paper presents the ...o, another highly available ...ilt for Amazon's platform. ... of services that have very ...ed tight control over the ...ency, cost-effectiveness and ...as a very diverse set of ...quirements. A select set of ...logy that is flexible enough ...heir data store appropriately ...ve high availability and ...st effective manner.

...'s platform that only need ...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

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[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 [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

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 wo… conseq… platfor… is impl…

Amazo… of mil… server… strict c… perfor… growth… of the… outage… customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

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. …l, loosely coupled, service …undreds of services. In this …ed for storage technologies …e, customers should be able …ping cart even if disks are … or data centers are being …he service responsible for … it can always write to and … data needs to be available …

…re comprised of millions of …peration; there are always a …r and network components …s such Amazon's software … manner that treats failure …t impacting availability or …

…eds, Amazon has developed … which the Amazon Simple …e of Amazon and known as …wn. This paper presents the …o, another highly available …ilt for Amazon's platform. … of services that have very …ed tight control over the …ency, cost-effectiveness and …as a very diverse set of …quirements. A select set of …logy that is flexible enough …heir data store appropriately …ve high availability and …st effective manner.

…'s platform that only need …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

Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[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 [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

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 wo… conseq… platfor… is impl…

Amazo… of mil… server… strict c… perfor… growth… of the… outage… customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

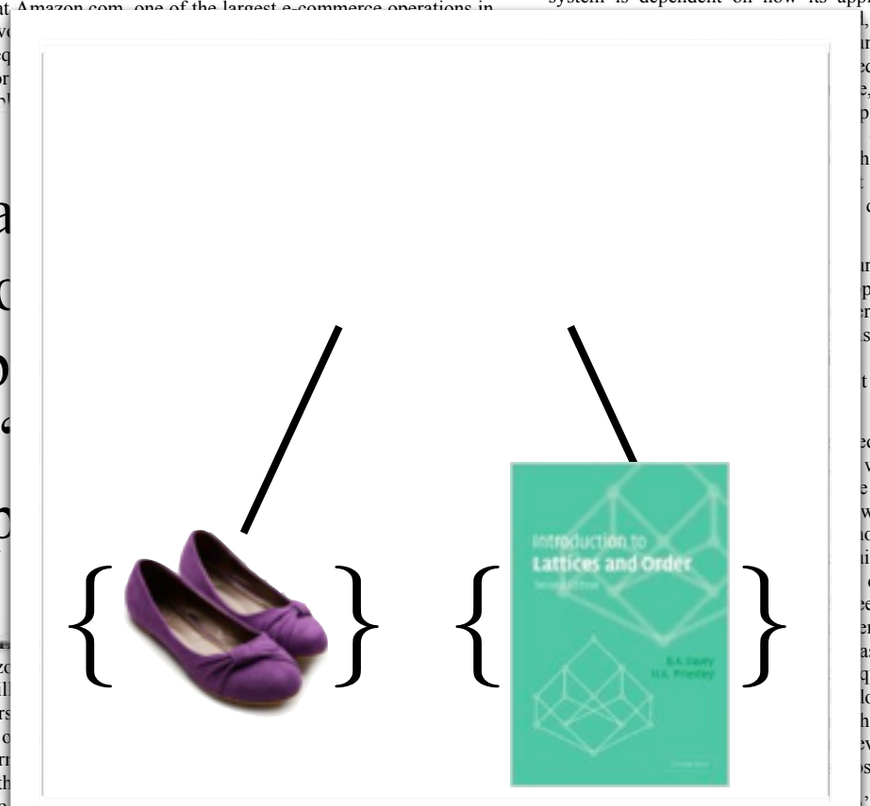
One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. …l, loosely coupled, service …undreds of services. In this …ed for storage technologies …e, customers should be able …ping cart even if disks are … or data centers are being …he service responsible for … it can always write to and … data needs to be available …

…re comprised of millions of …peration; there are always a …r and network components …s such Amazon's software … manner that treats failure …t impacting availability or …

…eds, Amazon has developed … which the Amazon Simple …e of Amazon and known as …wn. This paper presents the …o, another highly available …ilt for Amazon's platform. … of services that have very …ed tight control over the …ency, cost-effectiveness and …as a very diverse set of …quirements. A select set of …logy that is flexible enough …heir data store appropriately …ve high availability and …st effective manner.

…'s platform that only need … 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



Eventual consistency.

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[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 [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

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 wo... conseq... platfor... is impl...

Amazo... of mil... server... strict c... perfor... growth... of the... outage... customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

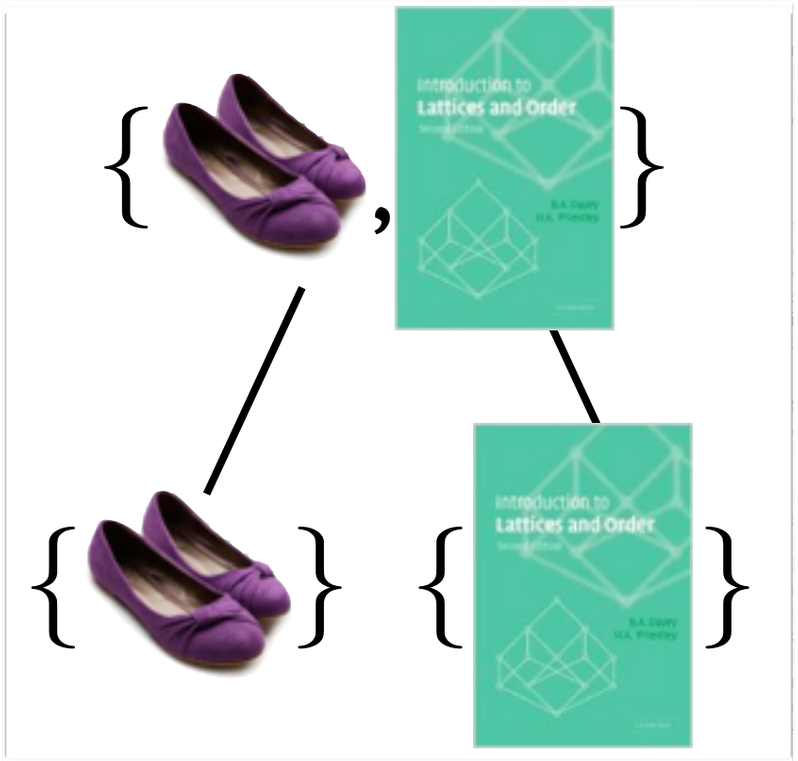
One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. ..., loosely coupled, service ...ndreds of services. In this ...ed for storage technologies ...e, customers should be able ...ping cart even if disks are ... or data centers are being ...he service responsible for ... it can always write to and ... data needs to be available

...re comprised of millions of ...peration; there are always a ...r and network components ...s such Amazon's software ... manner that treats failure ...t impacting availability or

...eds, Amazon has developed ... which the Amazon Simple ...e of Amazon and known as ...wn. This paper presents the ...o, another highly available ...ilt for Amazon's platform. ... of services that have very ...ed tight control over the ...ency, cost-effectiveness and ...as a very diverse set of ...quirements. A select set of ...logy that is flexible enough ...heir data store appropriately ...ve high availability and ...st effective manner.

...'s platform that only need ...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



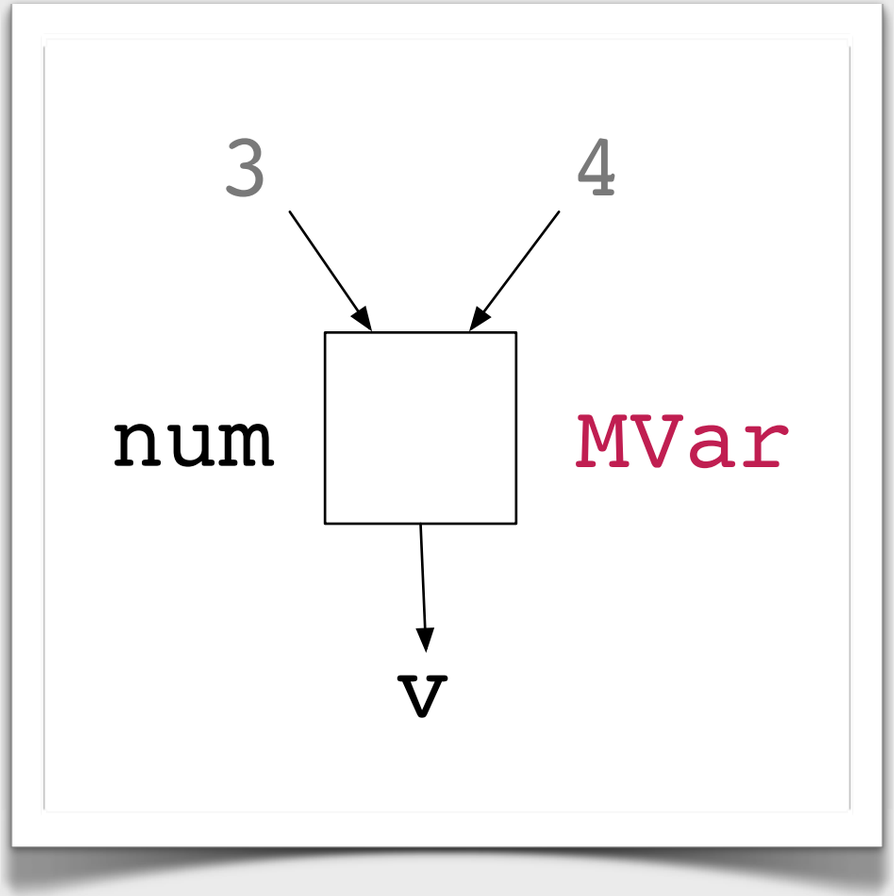
Eventual consistency.

Deterministic Parallelism

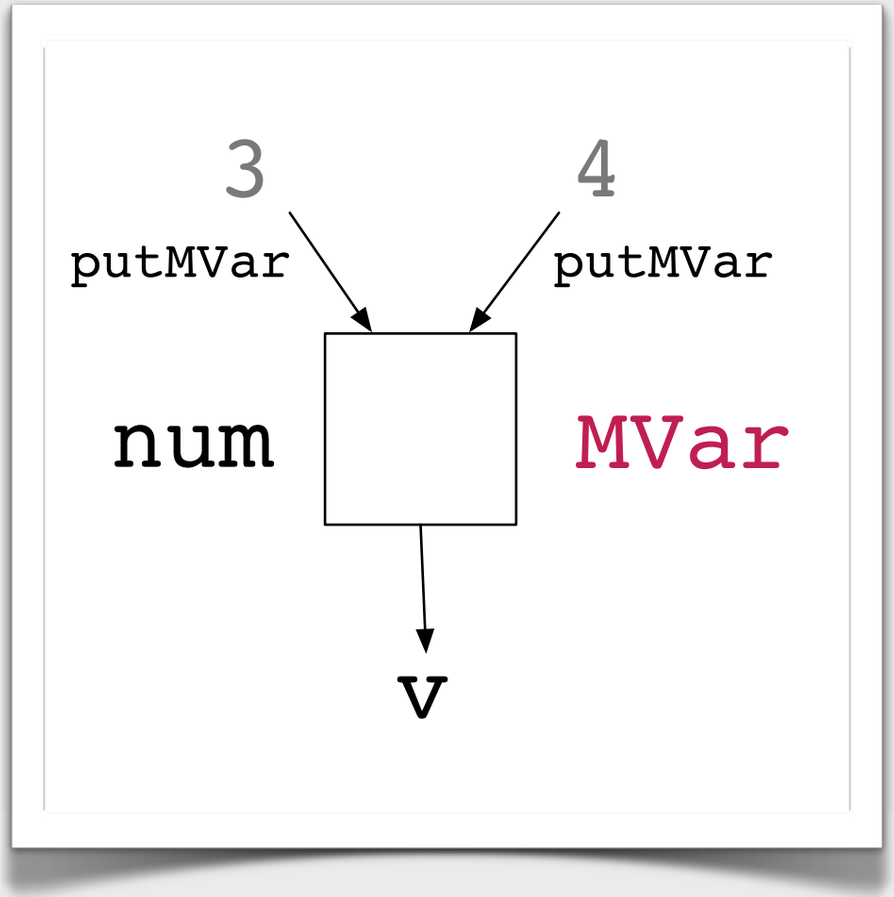
(observably)

Deterministic Parallelism

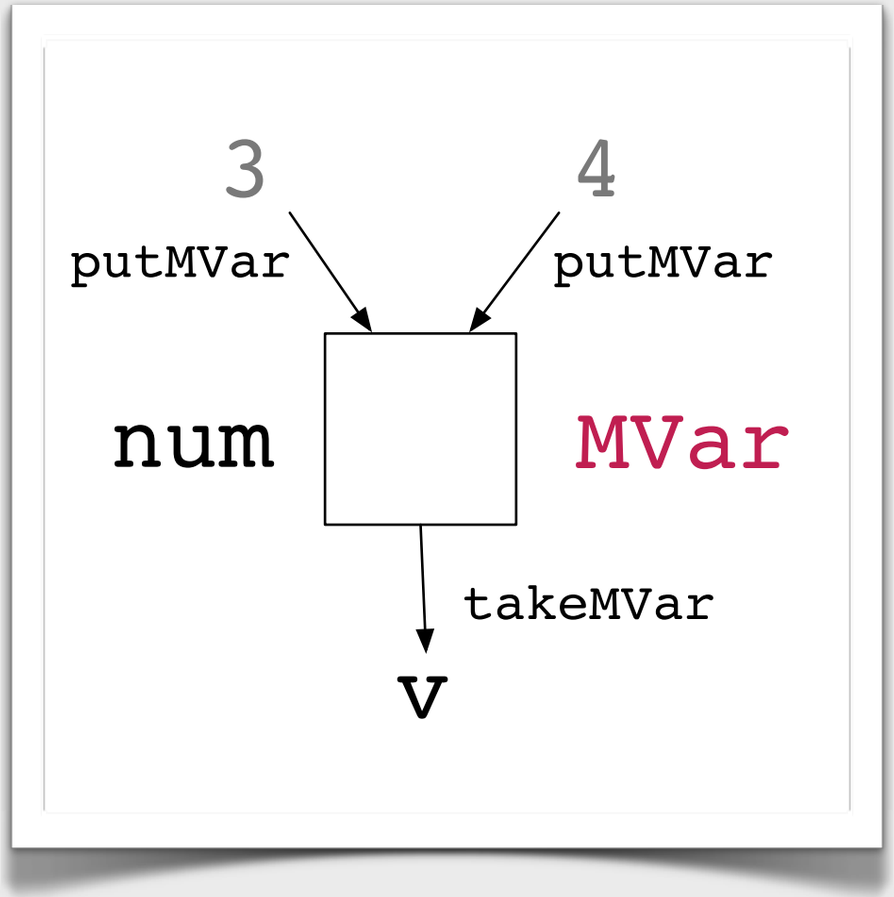
What does this program do?



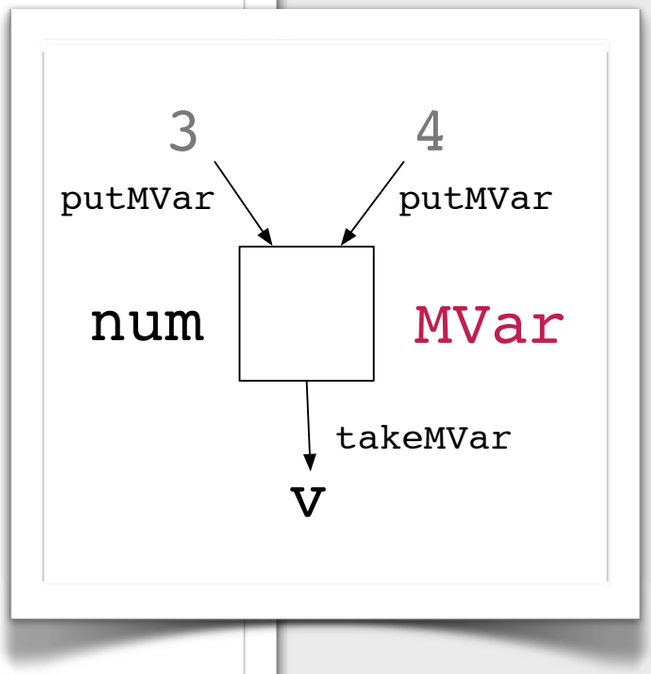
What does this program do?



What does this program do?

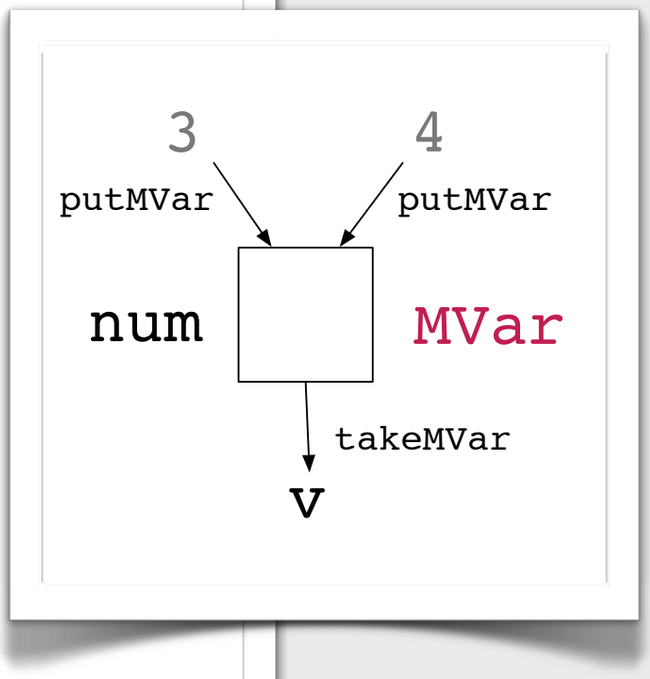


What does this program do?



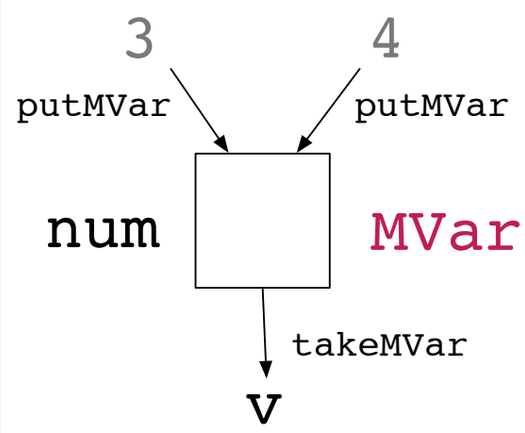
What does this program do?

```
p = do
```



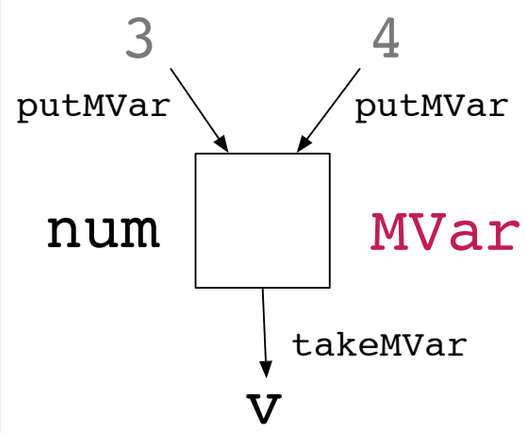
What does this program do?

```
p = do
   num <- newEmptyMVar
```



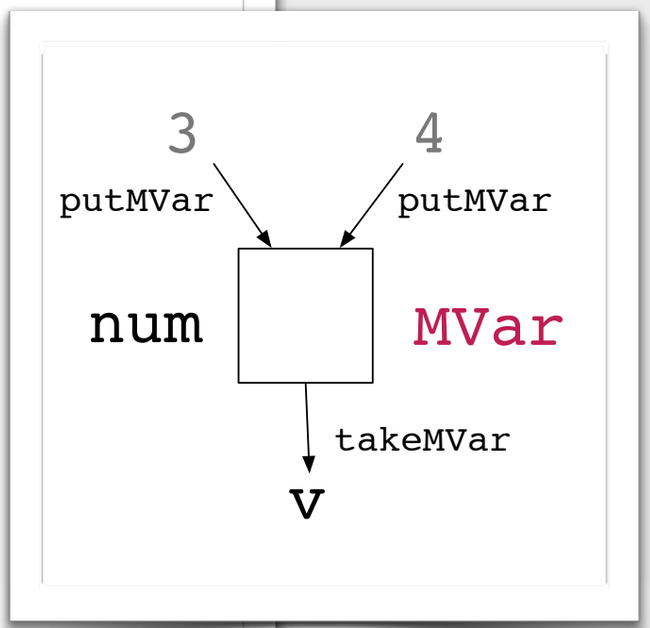
What does this program do?

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



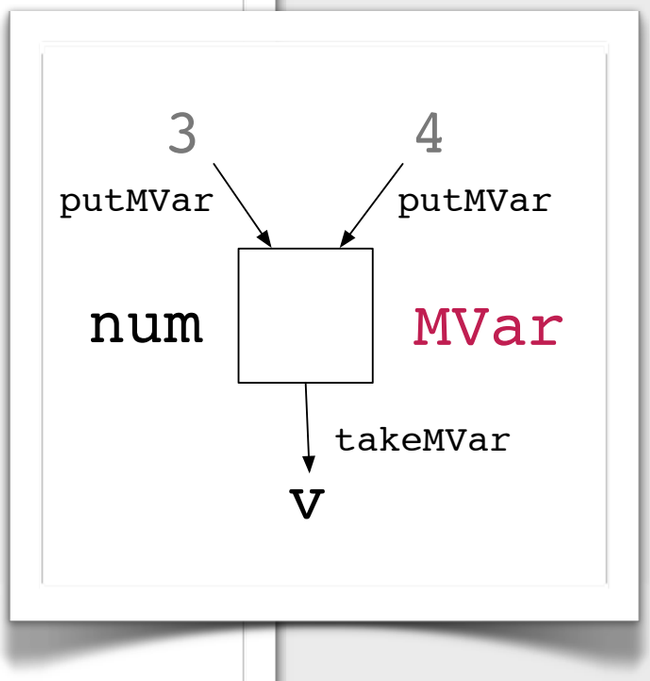
What does this program do?

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



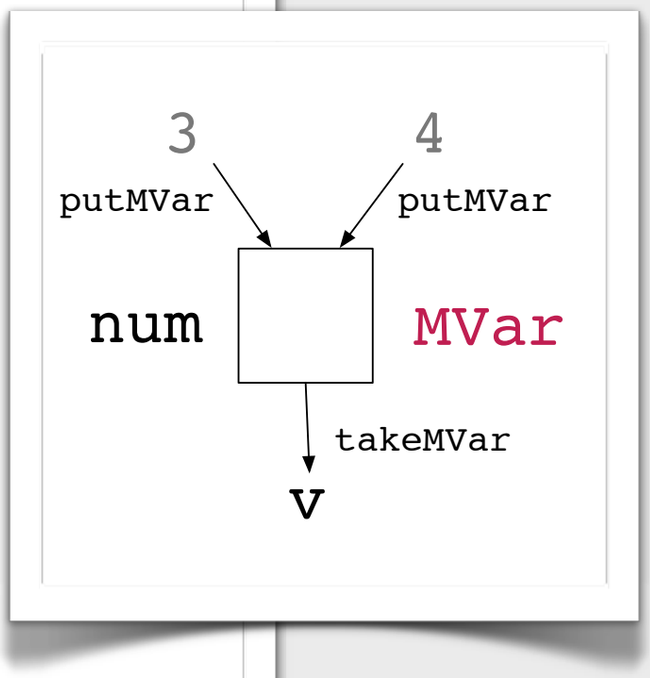
What does this program do?

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



What does this program do?

```
p = do
   num <- newEmptyMVar
   forkIO (putMVar num 3)
   forkIO (putMVar num 4)
   v <- takeMVar num
   return v
```



Terminal — bash — 88×23

bash

```
landin:lvar-examples 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
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```
p = do
   num <- newEmptyMVar
   forkIO (putMVar num 3)
   forkIO (putMVar num 4)
   v <- takeMVar num
   return v
```

Disallow multiple writes?

```
p = do
   num <- newEmptyMVar
   forkIO (putMVar num 3)
   forkIO (putMVar num 4)
   v <- takeMVar num
   return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

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

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

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

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

```
./ivar-example +RTS -N2
ivar-example: multiple put
```

Deterministic programs that single-assignment forbids

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

Deterministic programs that single-assignment forbids

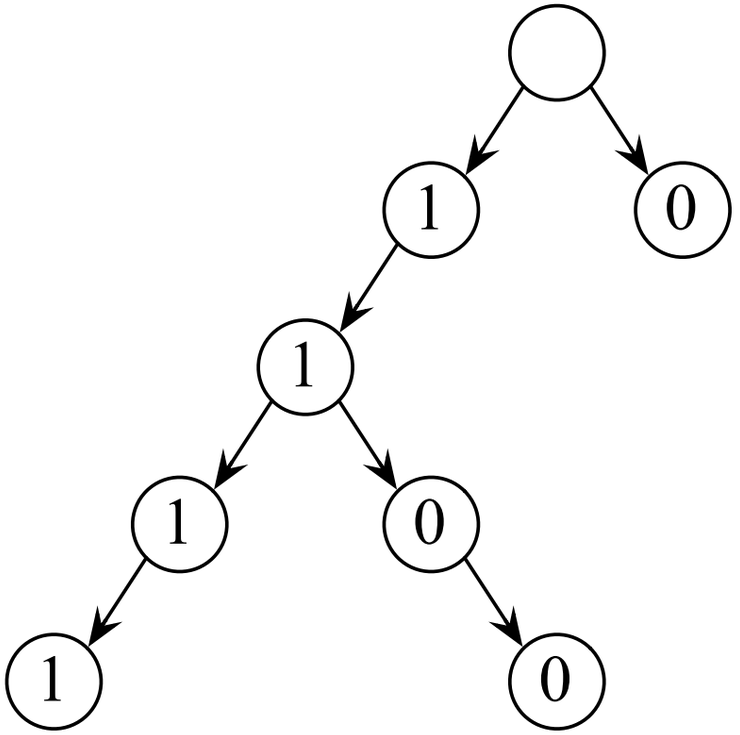
```
p :: Par Int
p = do
   num <- new
   fork    (put       num 4)
   fork    (put       num 4)
   v <- get          num
   return v
```

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

Deterministic programs that single-assignment forbids

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

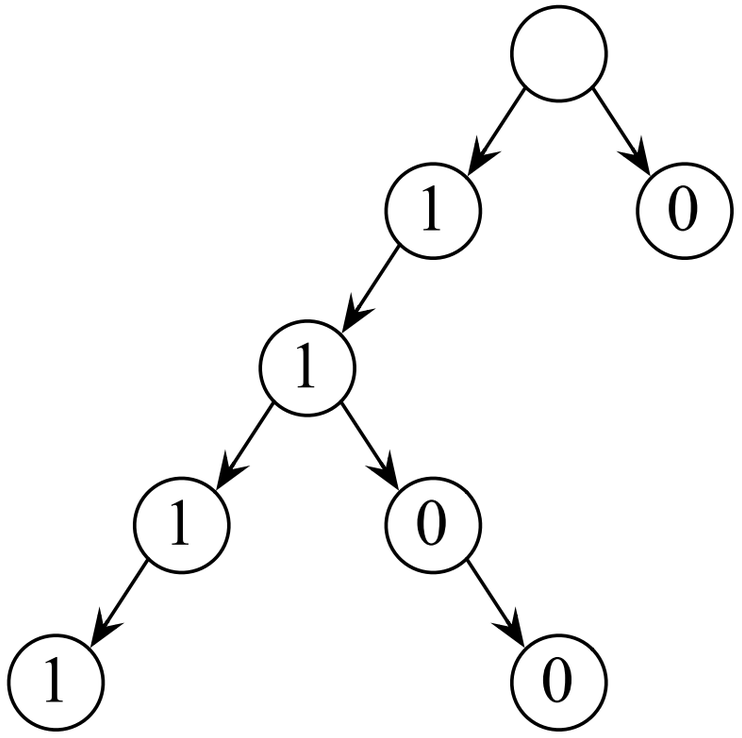
```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```



Deterministic programs that single-assignment forbids

```
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
```

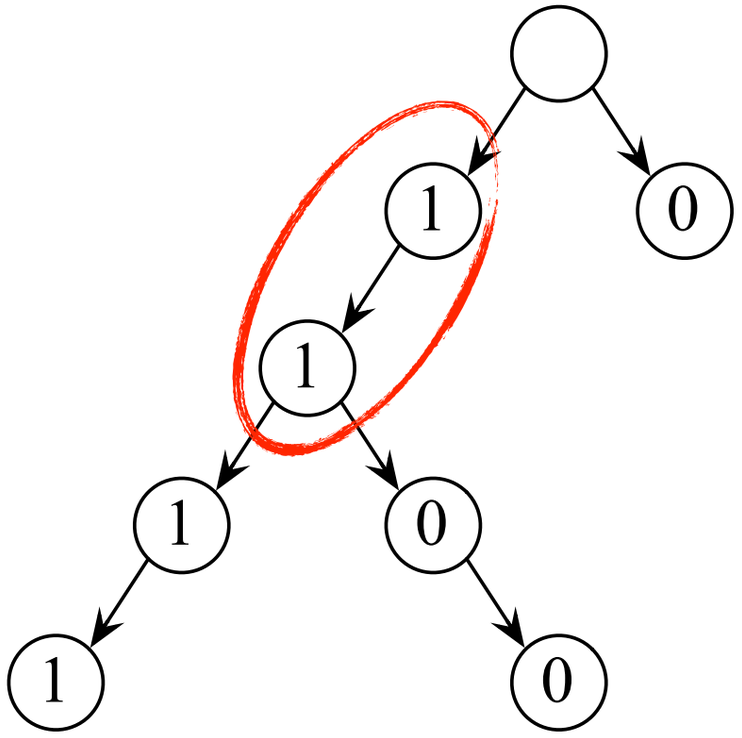



```
do
   fork (insert t "0")
   fork (insert t "1100")
   fork (insert t "1111")
   v <- get t
   return v
```

Deterministic programs that single-assignment forbids

```
p :: 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
```

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

```
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

```
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

```
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

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

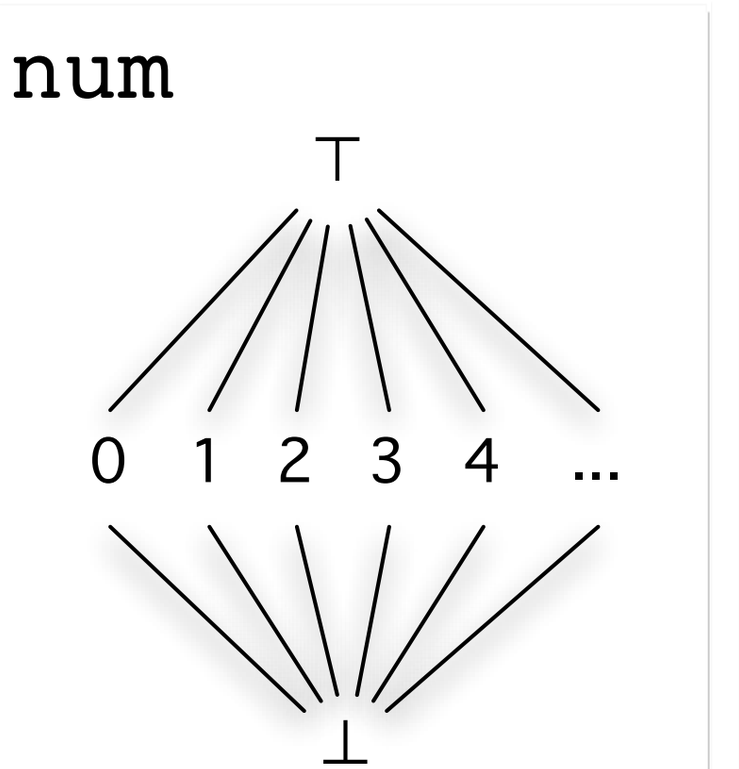
LVars: Multiple *monotonic* writes

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

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

- `cabal install lvish` today!

LVars: Multiple *monotonic* writes



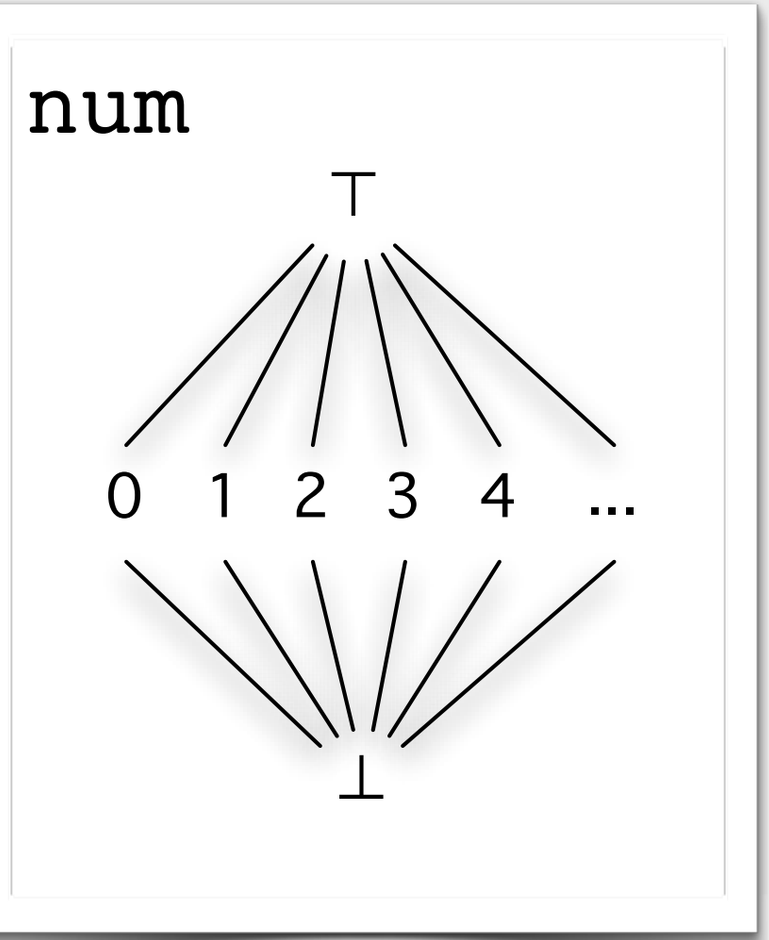
Raises an error, since $3 \sqcup 4 = \top$

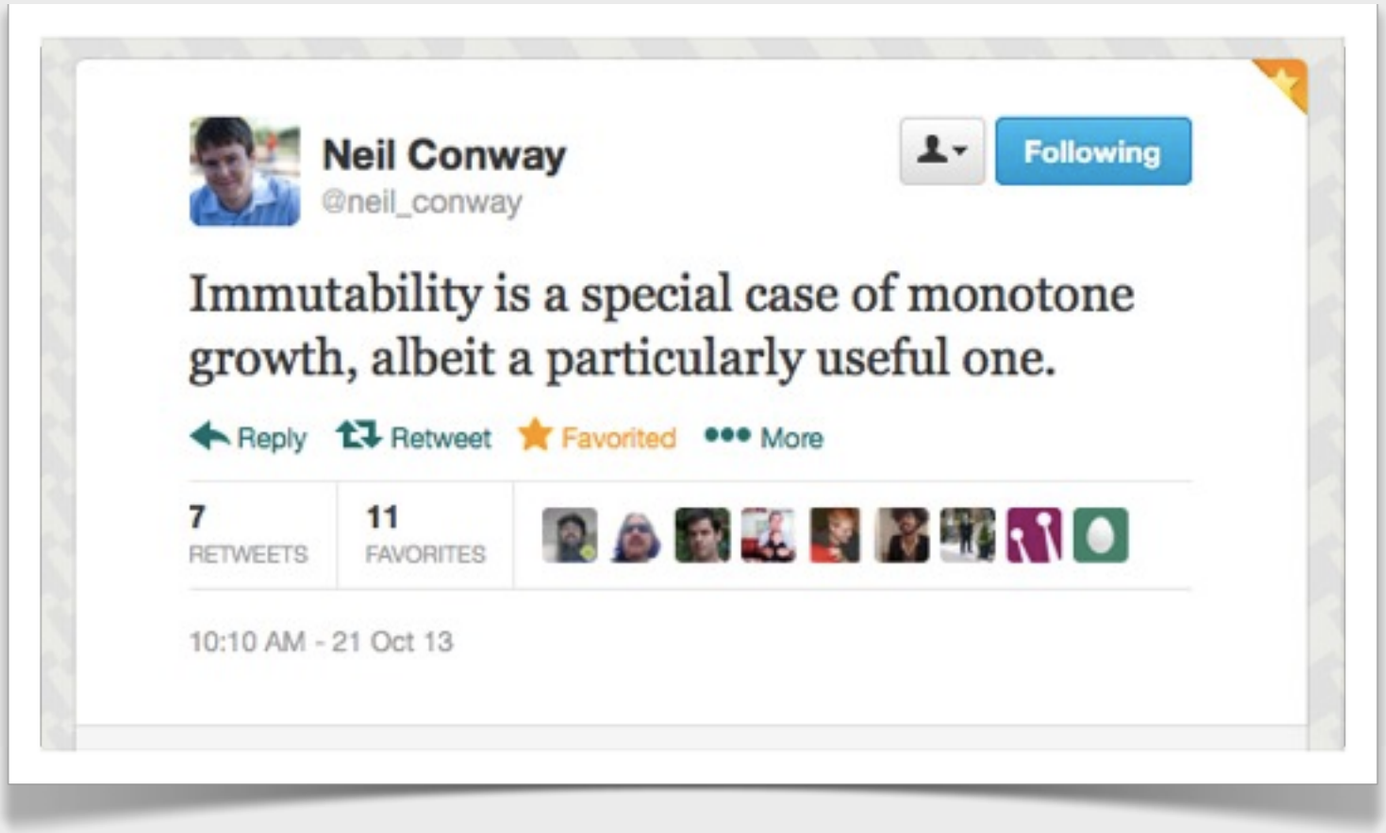
```
do
   fork (put num 3)
   fork (put num 4)
```

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

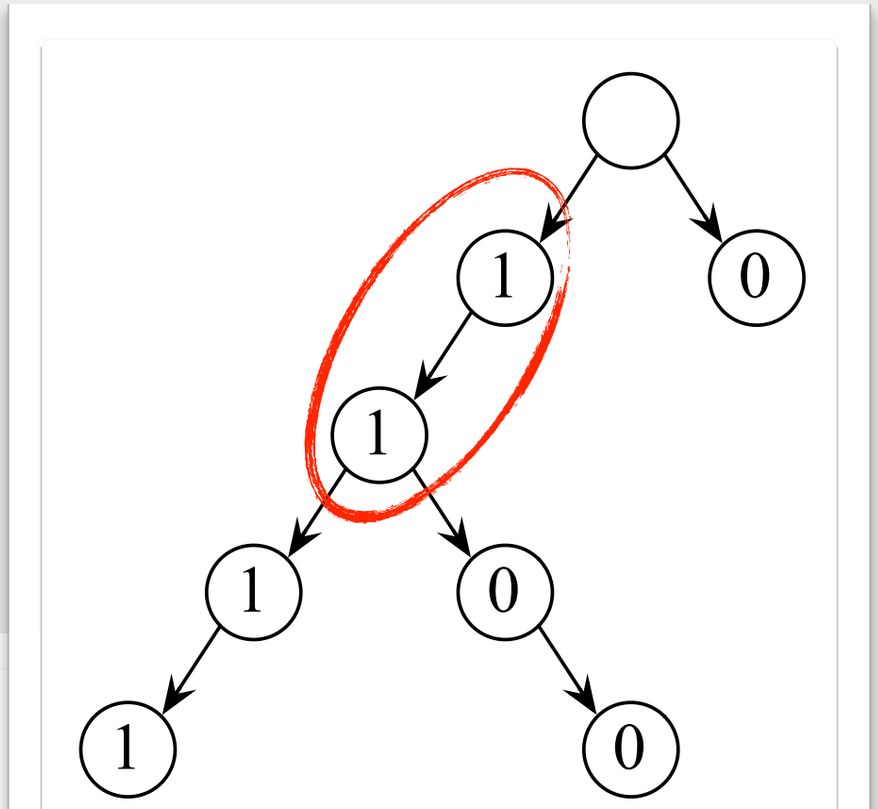
```
do
   fork (put num 4)
   fork (put num 4)
```

LVars: Multiple *monotonic* writes



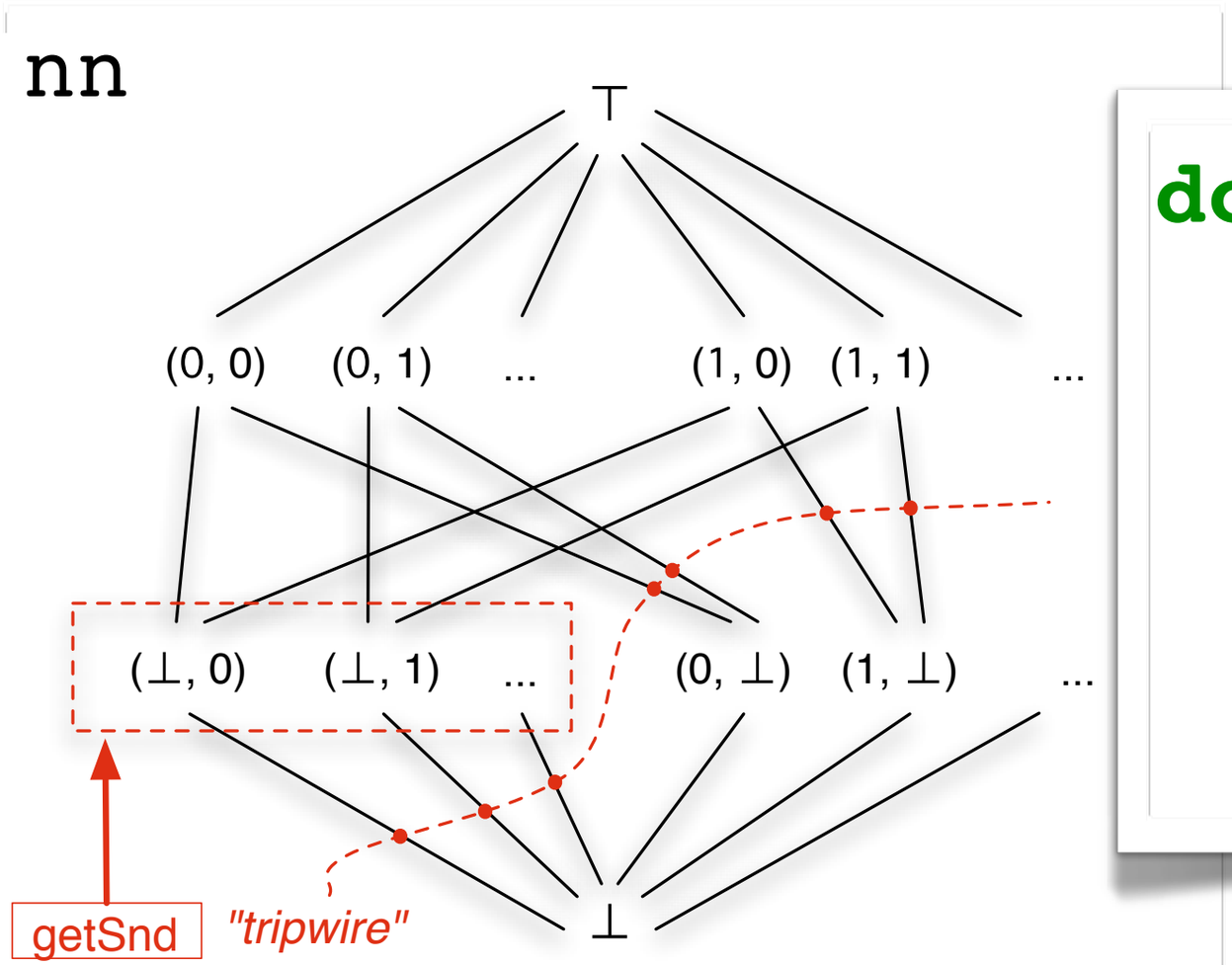


Overlapping writes are no problem



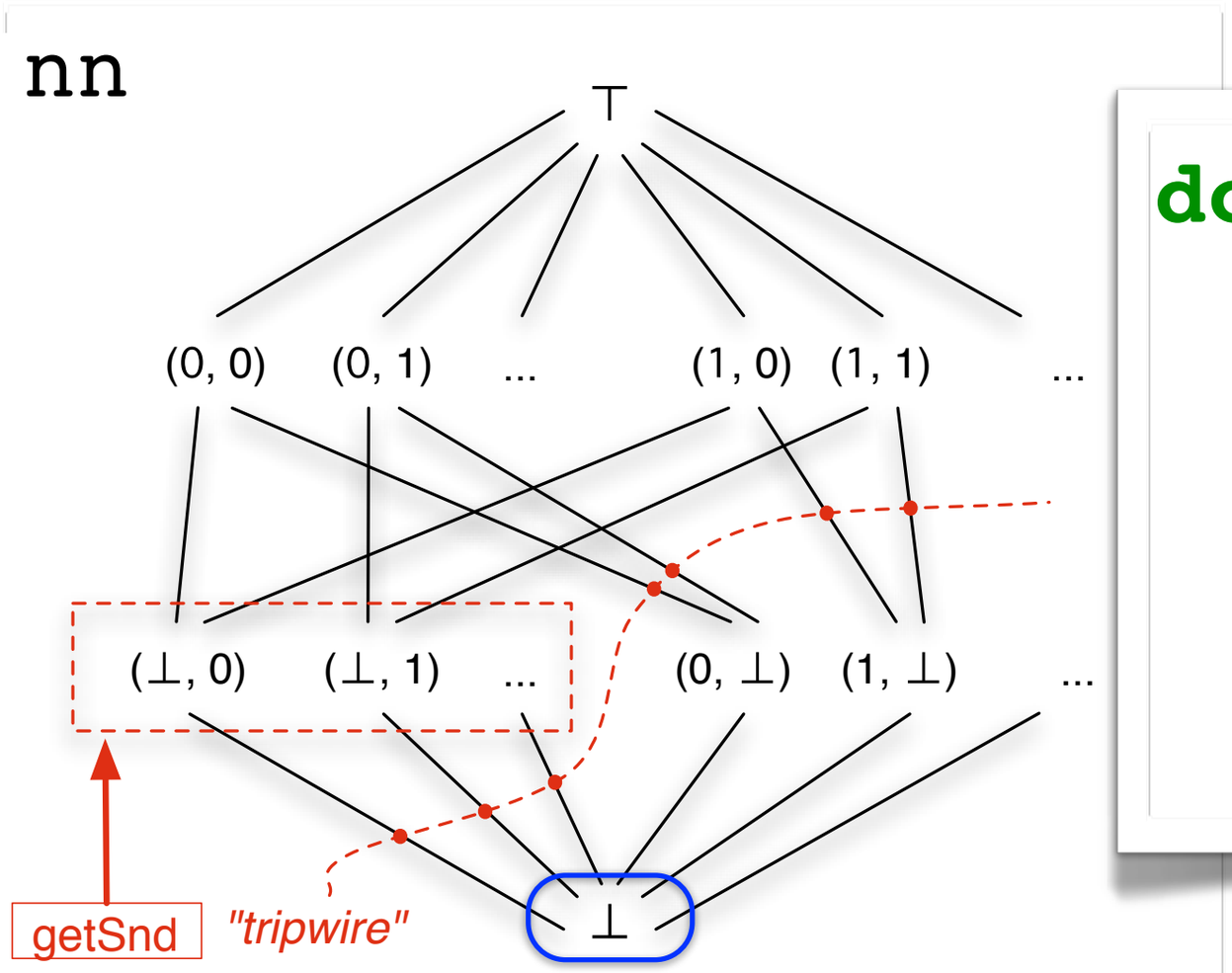

```
do
   fork (insert t "0")
   fork (insert t "1100")
   fork (insert t "1111")
   v <- get t
   return v
```

LVars: Threshold reads



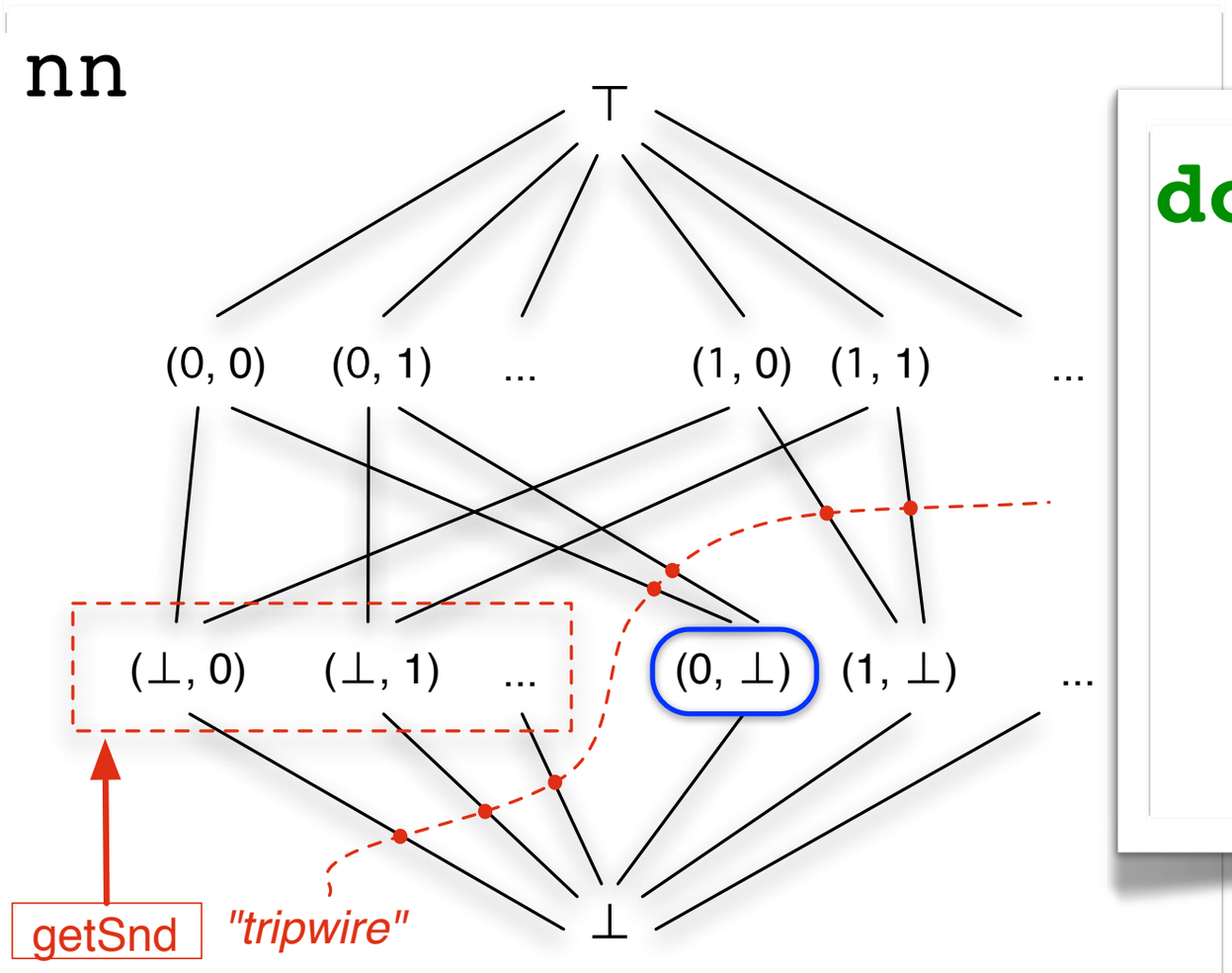

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



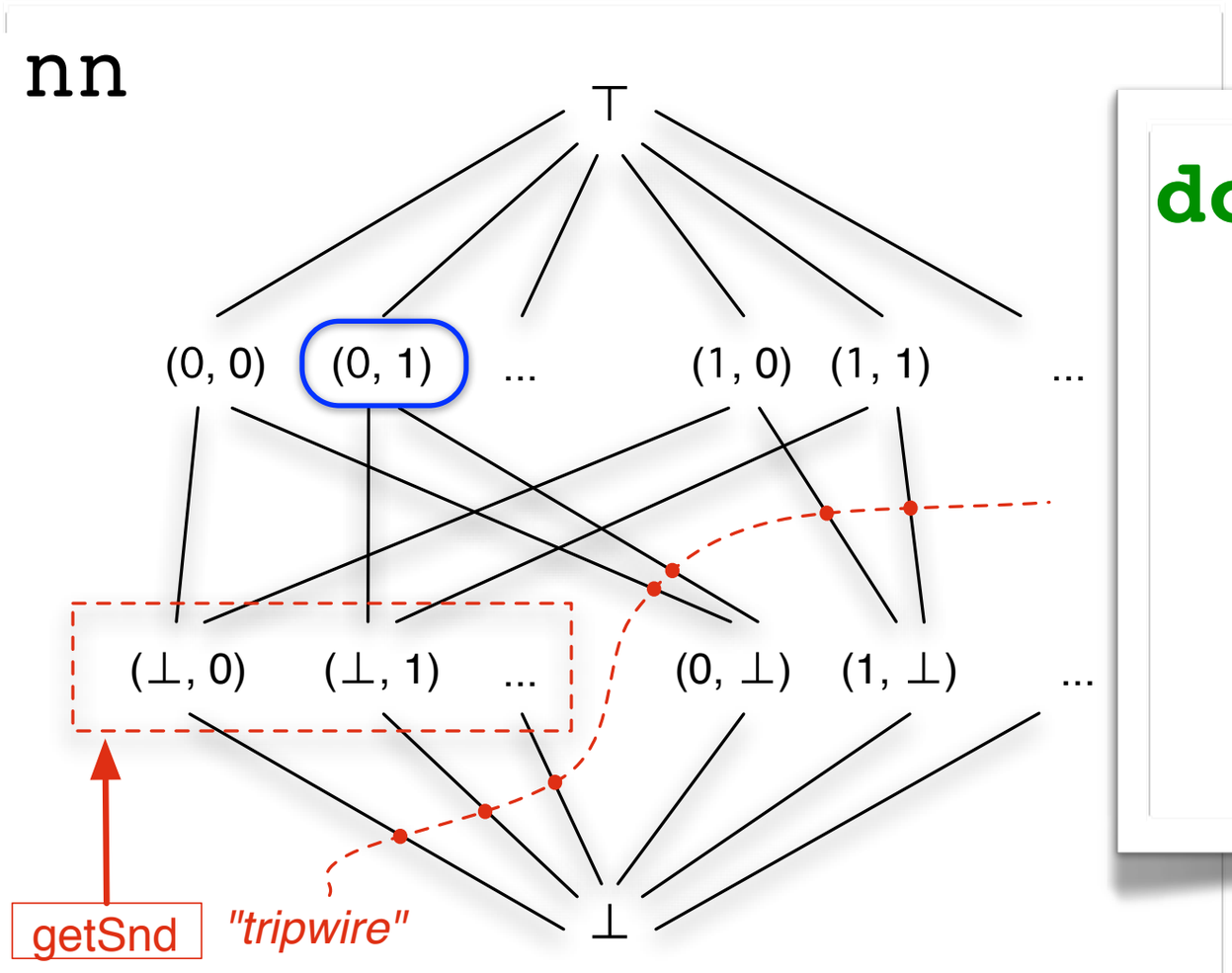

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



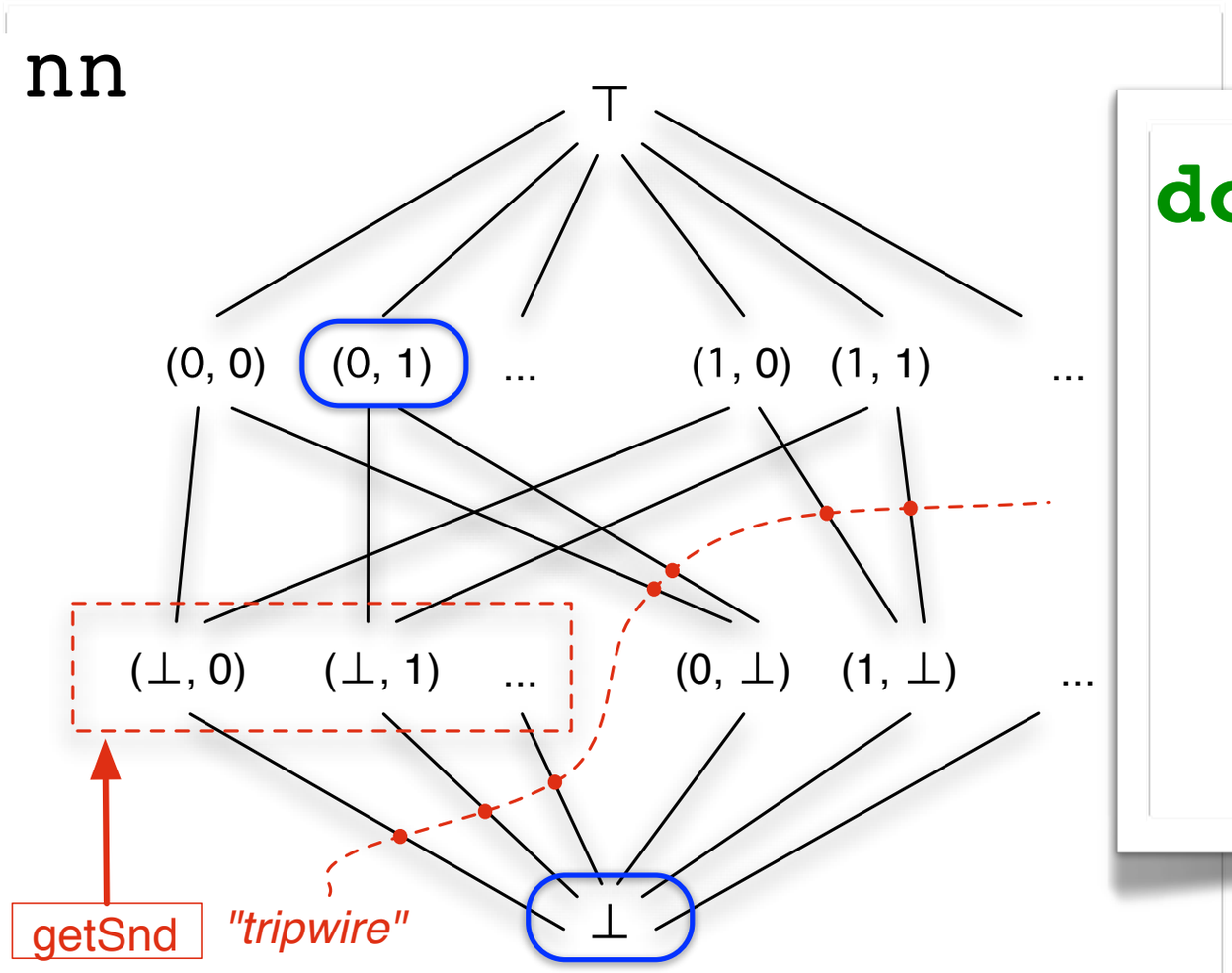

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



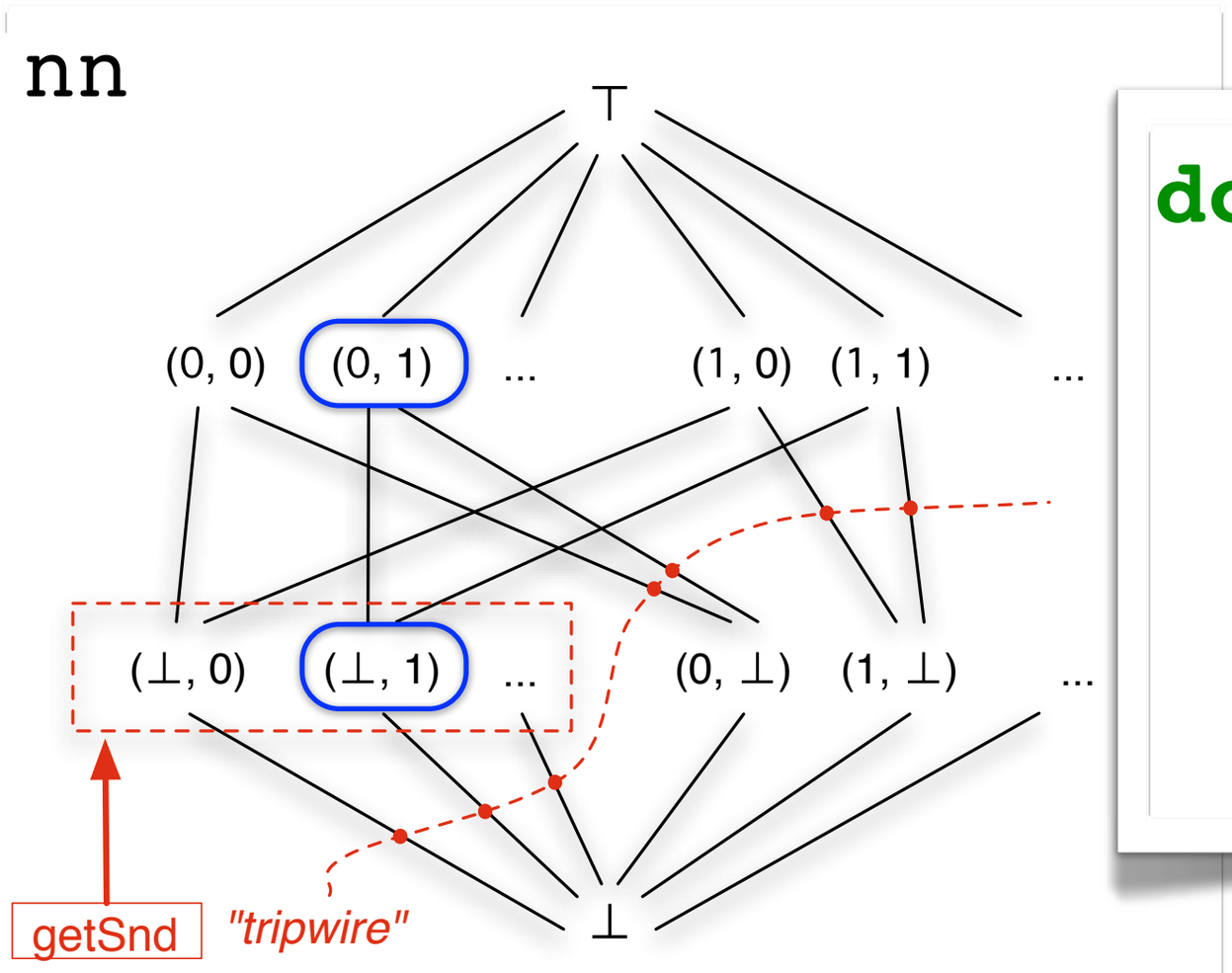

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



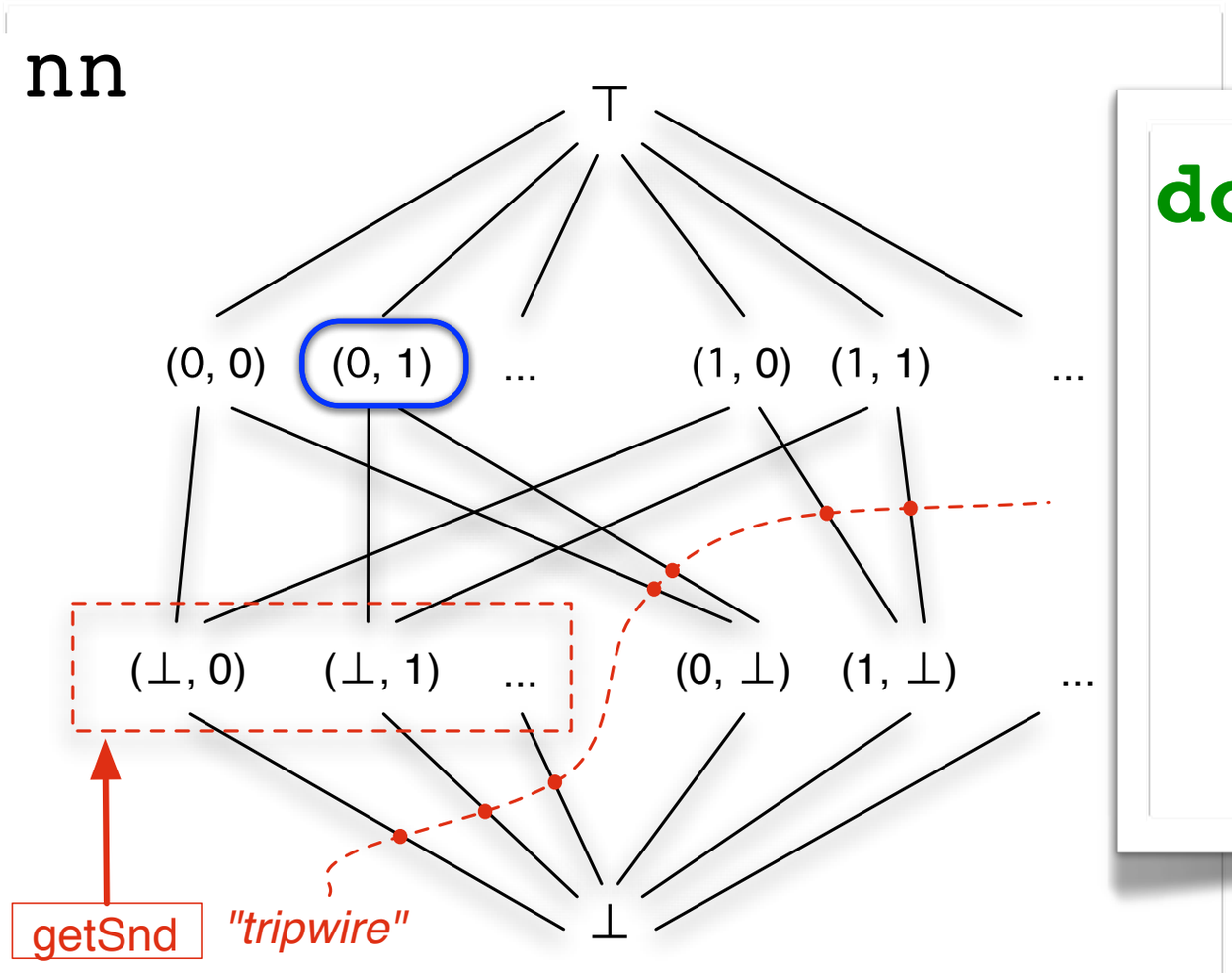

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



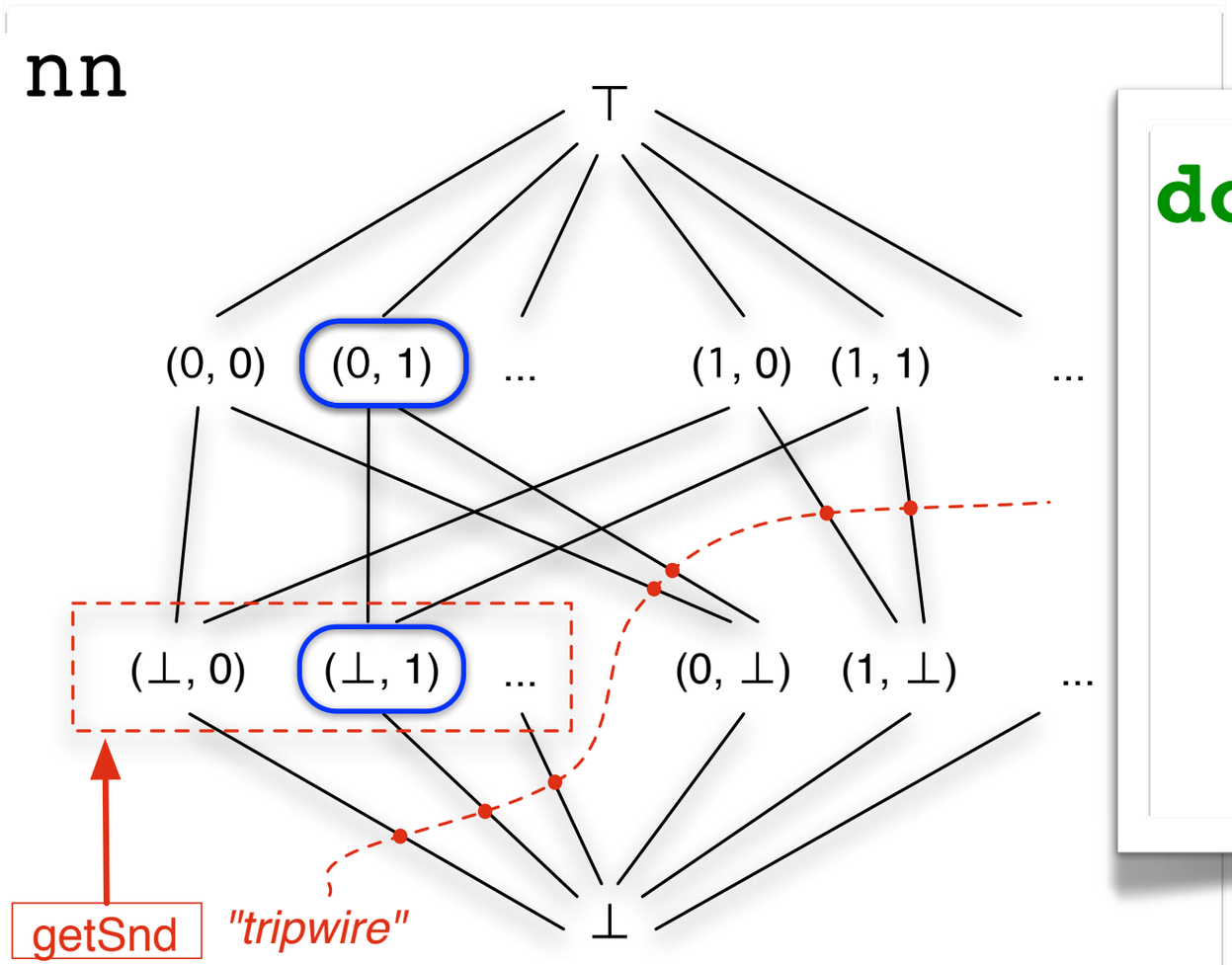

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads



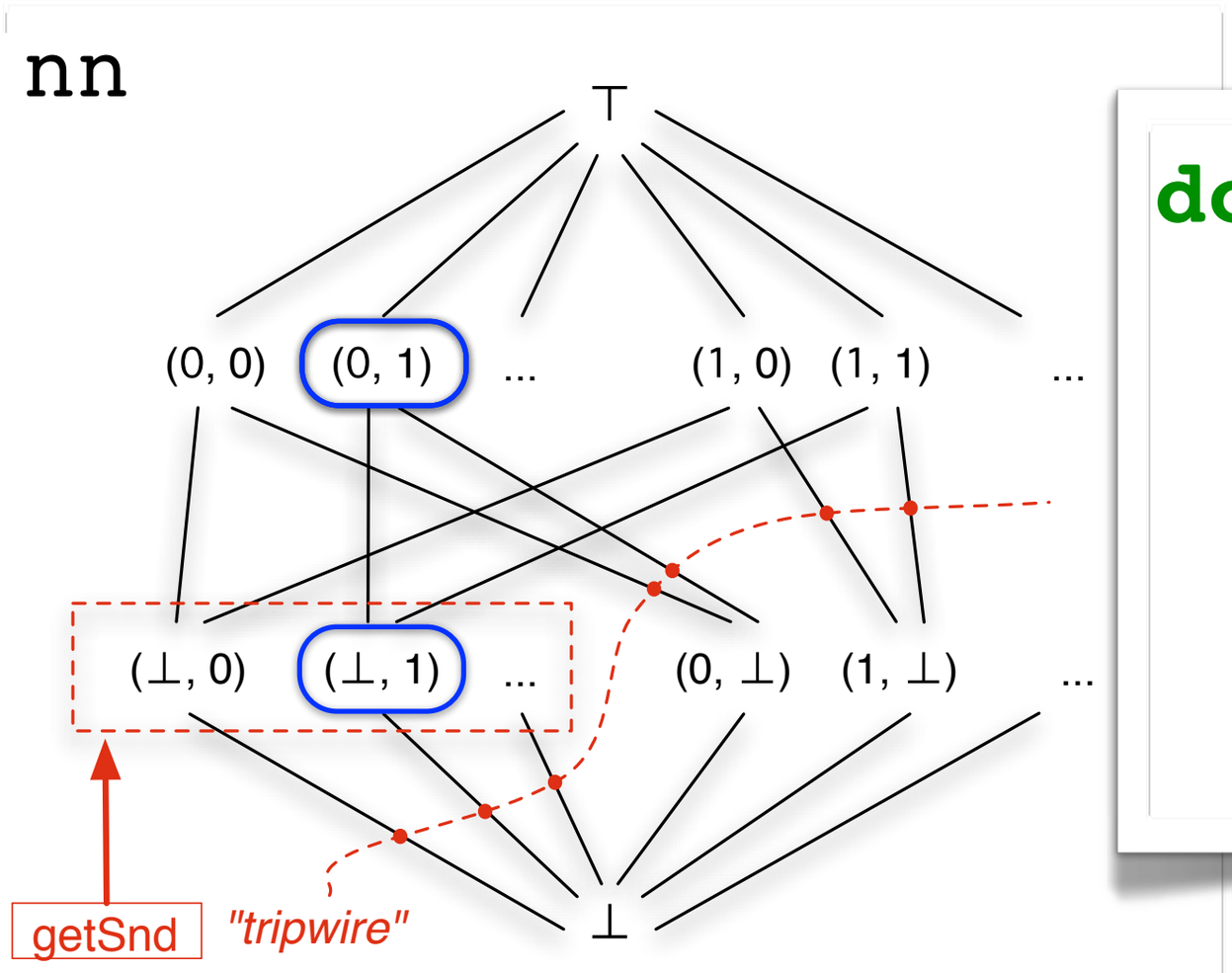

```
do
   nn <- newPair
   fork (putFst nn 0)
   fork (putSnd nn 1)
   v <- getSnd nn
   return v -- returns 1
```

LVars: Threshold reads




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

LVars: Threshold reads




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

The threshold set must be *pairwise incompatible*

Monotonicity enables deterministic parallelism

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type σ (boolean, real, etc...) we could have declared a σ *channel*. Then processes f, g and h are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes : at line (2) f is stated to communicate via two input lines that can carry integers, and one similar output line.
The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable *on* a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes.
Such a style of programming is close to may systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.
What sort of things would we like to prove on a program like S ? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.
The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; logical B ;
          B := true ;
          Repeat Begin
(4)         I := if B then wait(U) else wait(V) ;
(7)         print (I) ;
(5)         send I on W ;
            B := ¬B ;
            end ;
    End ;
  Process g(integer in U ; integer out V, W) ;
    Begin integer I ; logical B ;
      B := true ;
      Repeat Begin
        I := wait (U) ;
        if B then send I on V else send I on  W ;
        B := ¬B ;
        End ;
    End ;
(3) Process h(integer in U;integer out V; integer INIT);
    Begin integer I ;
      send INIT on V ;
      Repeat Begin
        I := wait(U) ;
        send I on V ;
        End ;
    End ;
  Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
  End ;
```

Fig.1. Sample parallel program S.

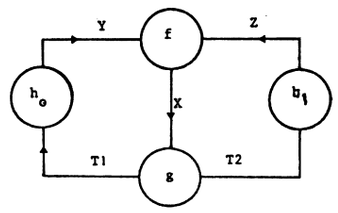


Fig.2. The schema P for the program S.

Kahn, 1974

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given $\leqslant$,

$$x \leqslant y \Longrightarrow f(x) \leqslant f(y)$$

Kahn, 1974

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type σ (boolean, real, etc...) we could have declared a σ *channel*. Then processes f, g and h are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes : at line (2) f is stated to communicate via two input lines that can carry integers, and one similar output line.
The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable *on* a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes.
Such a style of programming is close to may systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.
What sort of things would we like to prove on a program like S ? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.
The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; logical B ;
          B := true ;
          Repeat Begin
(4)         I := if B then wait(U) else wait(V) ;
(7)         print (I) ;
(5)         send I on W ;
            B := ¬B ;
            end ;
    End ;
  Process g(integer in U ; integer out V, W) ;
    Begin integer I ; logical B ;
      B := true ;
      Repeat Begin
        I := wait (U) ;
        if B then send I on V else send I on  W ;
        B := ¬B ;
        End ;
    End ;
(3) Process h(integer in U;integer out V; integer INIT);
    Begin integer I ;
      send INIT on V ;
      Repeat Begin
        I := wait(U) ;
        send I on V ;
        End ;
    End ;
  Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
  End ;
```

Fig.1. Sample parallel program S.

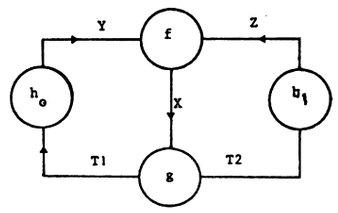


Fig.2. The schema P for the program S.

Monotonicity enables deterministic parallelism

f is *monotonic* iff, for a given $\leqslant$,

$$x \leqslant y \Longrightarrow f(x) \leqslant f(y)$$

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

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thor- ficiencies are exhibited by this theoret- ven. We hope in this way to make a case f languages for systems programming and

```
n
teger channel X, Y, Z, T1, T2 ;
ocess f(integer in U,V; integer out W) ;
in integer I ; logical B ;
   B := true ;
   Repeat Begin
     I := if B then wait(U) else wait(V) ;
     print (I) ;
     send I on W ;
     B := ¬B ;
     end ;
;
ss g(integer in U ; integer out V, W) ;
in integer I ; logical B ;
:= true ;
epeat Begin
 I := wait (U) ;
 if B then send I on V else send I on  W ;
 B := ¬B ;
 End ;
;
ss h(integer in U;integer out V; integer INIT);
n integer I ;
nd INIT on V ;
epeat Begin
 I := wait(U) ;
 send I on V ;
 End ;
End ;
Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
End ;
```

Fig.1. Sample parallel program S.

until something is being sent on this line by ano- ther process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par*

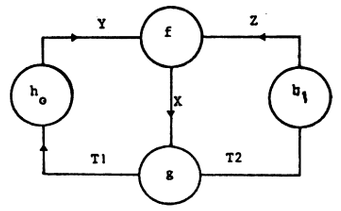


Fig.2. The schema P for the program S.

for the theoretical study of the next sections.

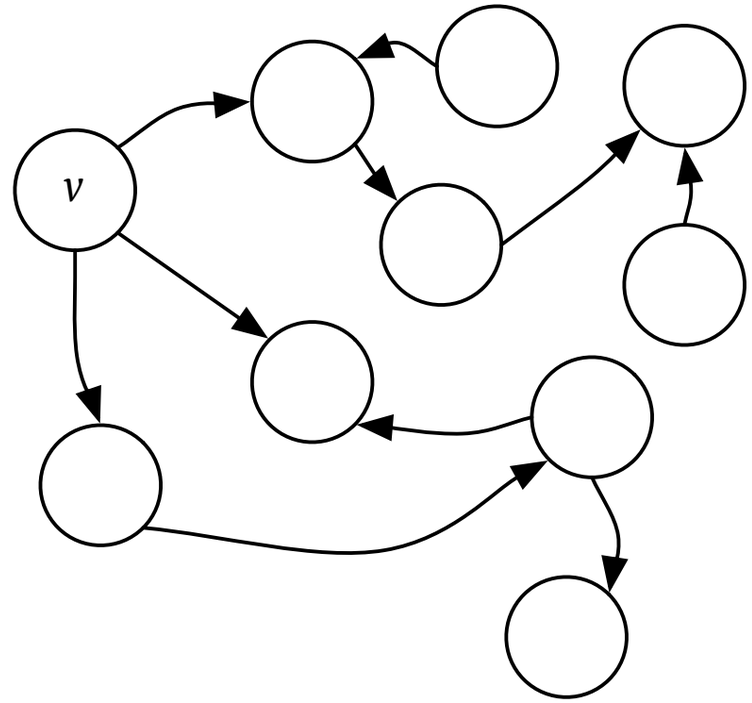
2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

Challenge problem

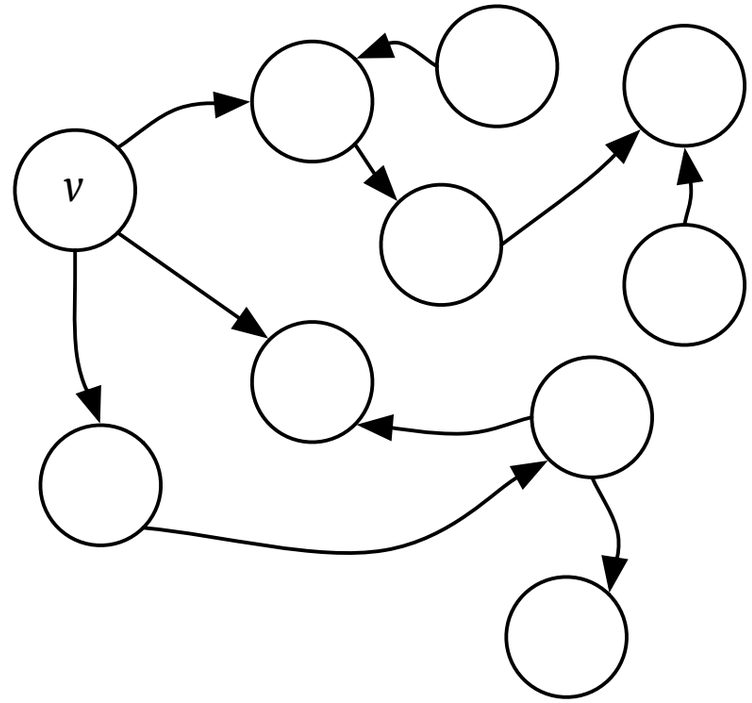
In a directed graph:

- find the connected component of all nodes within *k* hops of a vertex *v*
- and compute a function *analyze* over each vertex in that component
- **making the set of results available asynchronously to other computations**

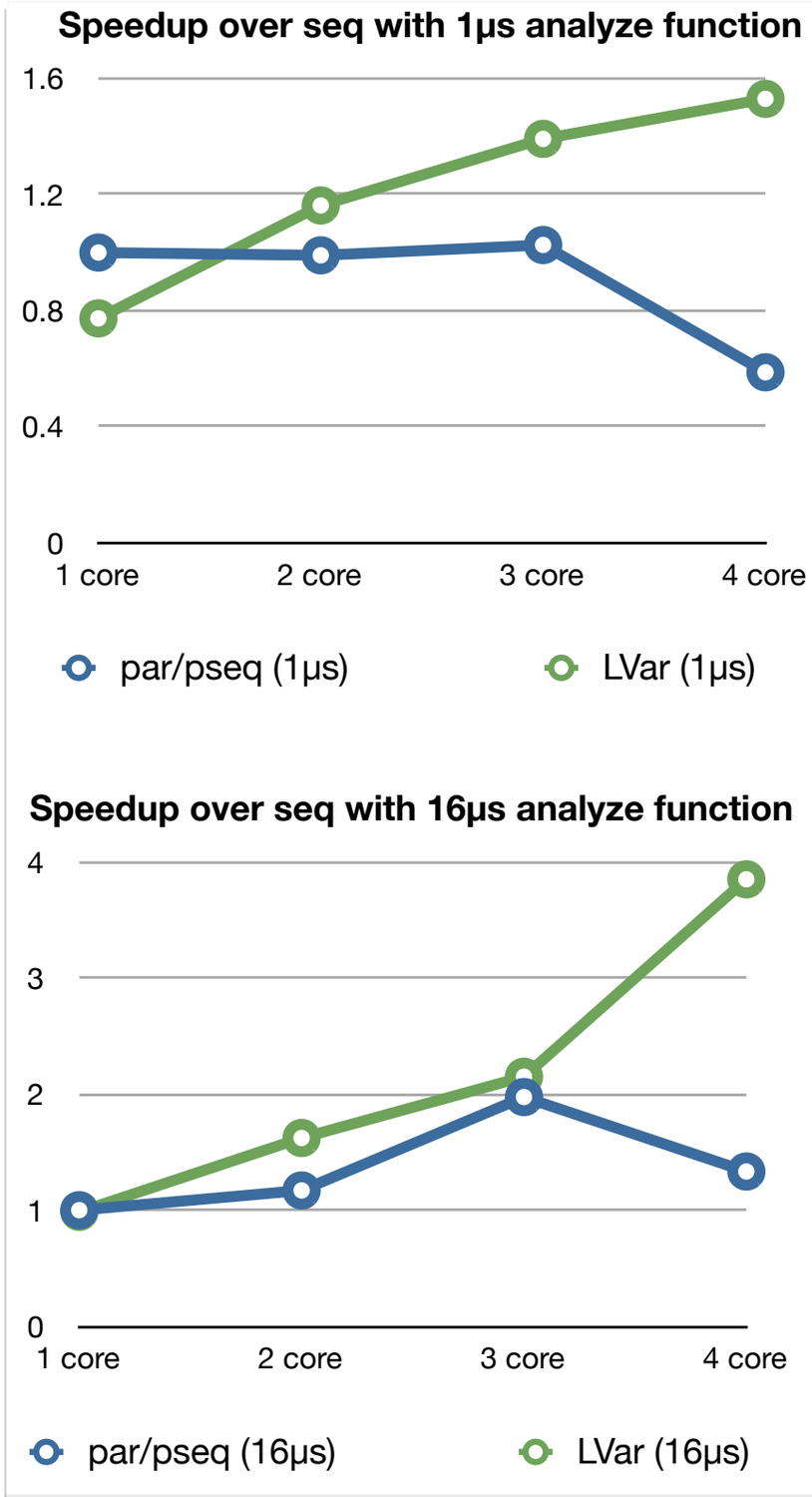


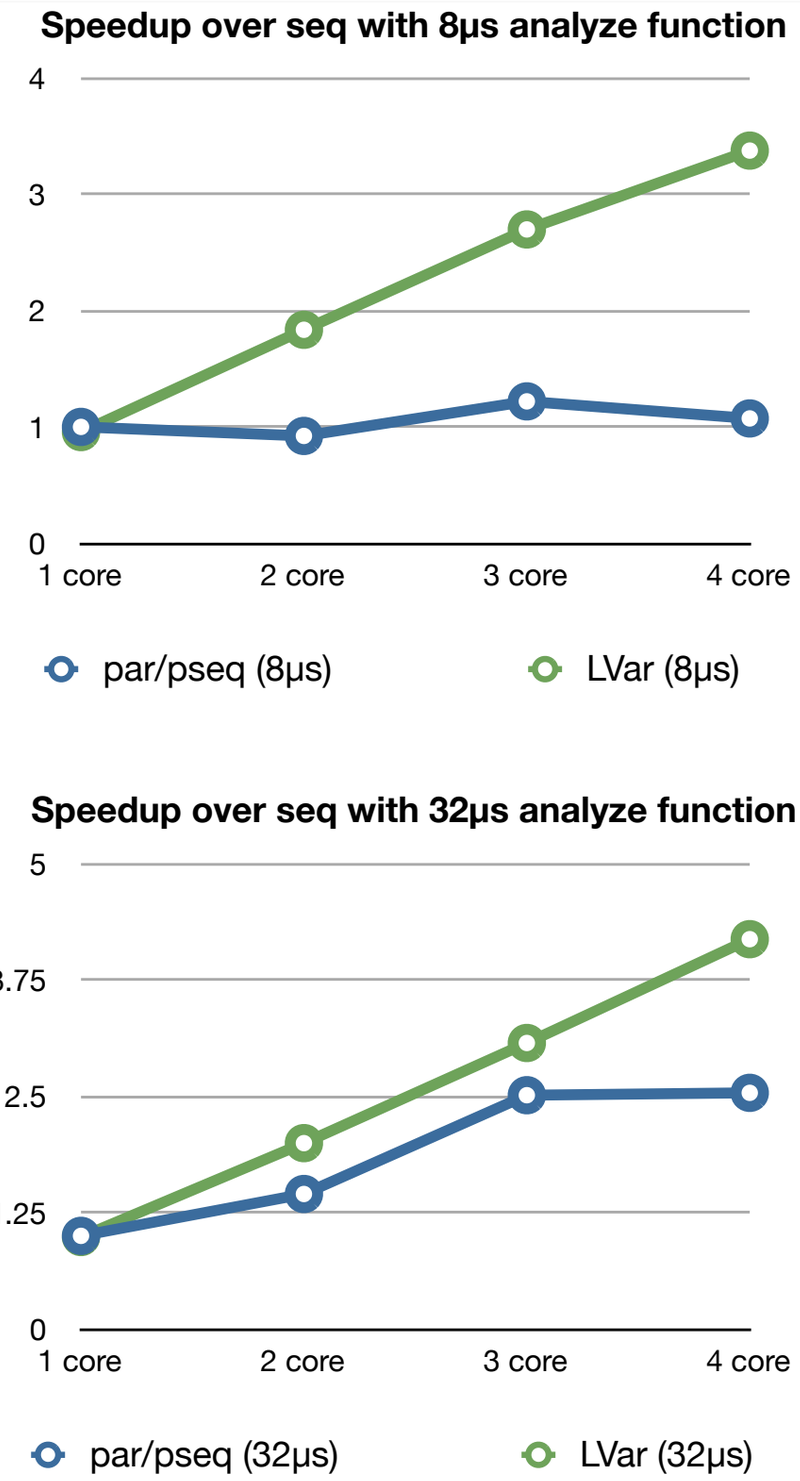
Challenge problem

- 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*

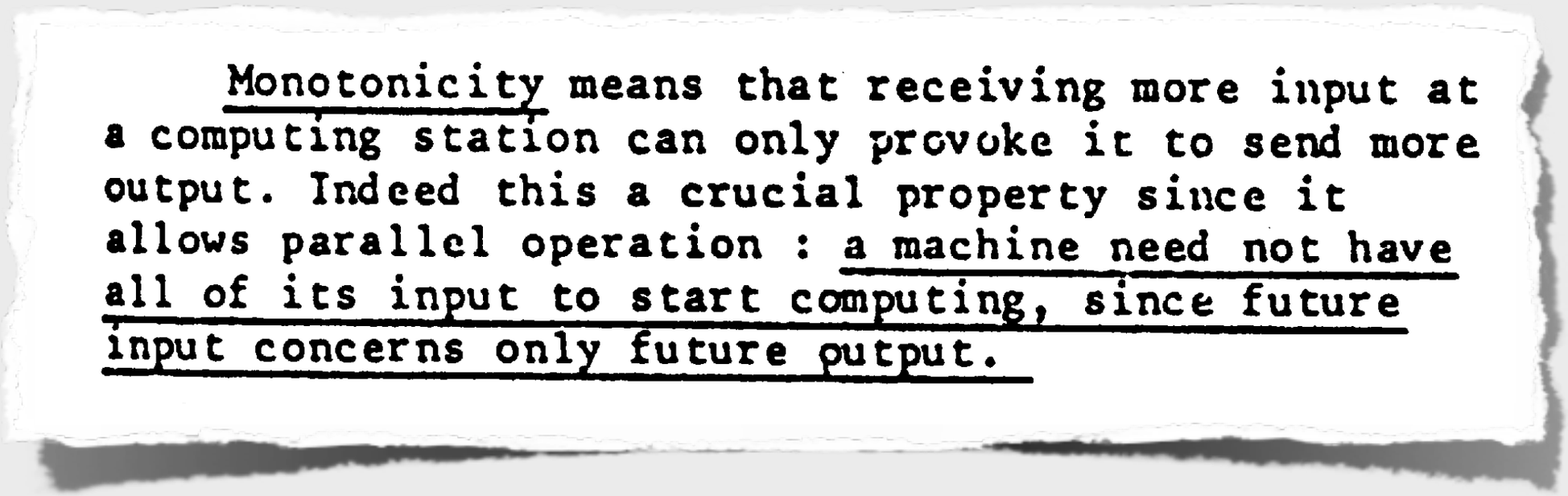


Challenge problem: Strategies vs. LVars





Challenge problem: Strategies vs. LVars

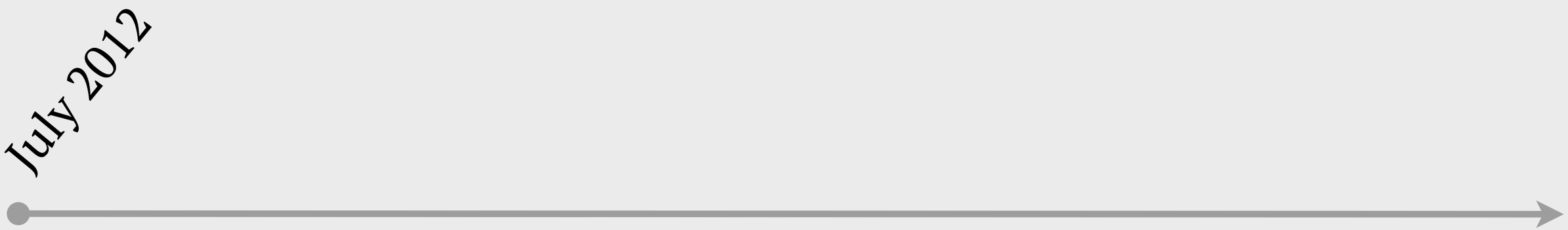


Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation : a machine need not have all of its input to start computing, since future input concerns only future output.

- Average time from start of program to first invocation of *analyze*:
  - Strategies version: 64.64 ms
  - LVar version: 0.18 ms

(observably)

Deterministic Parallelism via *monotonic writes* and *threshold reads*.


July 2012

July 2012

July 2012

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

Example 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

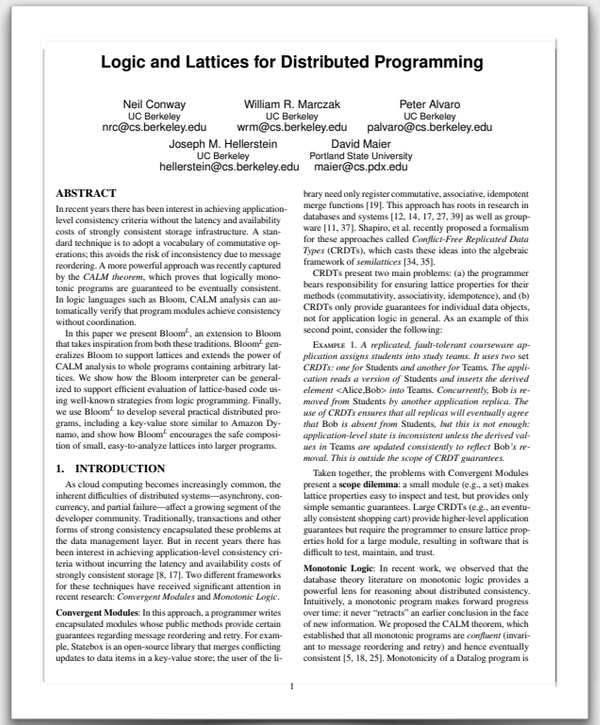
Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter



Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

"It's very interesting stuff."

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

June 2013

"It's very interesting stuff."

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

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..."

<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013

"It's very interesting stuff."

FHPC '13

"I'd love to link a few co-workers to your blog post..."

<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013

October 2013

"It's very interesting stuff."

FHPC '13 *POPL '14*

"I'd love to link a few co-workers to your blog post..."

<~ bloom

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

July 2012

August 2012

long, dark winter

May 2013

June 2013

July 2013

October 2013

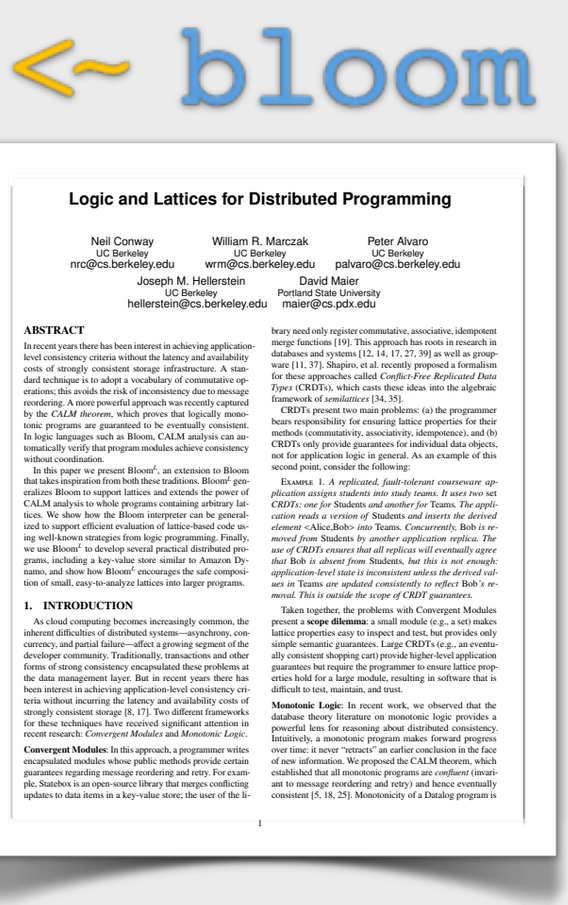
RICON

"It's very interesting stuff."

"I'd love to link a few co-workers to your blog post..."

FHPC '13

POPL '14



Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

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

Conflict-Free

Abstract. Replicating

1 Introduction

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

Example 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

1

LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper Ryan R. Newton
Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract

Programs wri
parallel comp
observable re
hard-to-reproc
parallel softwa
by-constructio
single-assignm
monotonically
LVars ensure
"threshold" re
give a proof o
language with
to support a li
but never wro

Categories an
and Features]
current Prog
Definitions an
cations]: Cons

Keywords D

1. Introd

Programs wri
parallel comp
observable re
hard-to-reproc
parallel softwa
guage extensi
programming
written using

The most
by-constructio
meaning mak
libraries and r
ming with fun
Haskell progr

FHPC '13, Sep

http://dx.doi.org/1

Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org

Ryan R. Newton
Indiana University
rrnewton@cs.indiana.edu

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by *LVars*: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited.

We extend LVars in two ways. First, we add the ability to "freeze" and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is *observable* within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.

While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.

The simplest strategy is to allow *no* communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and Streamlt [11], or shared write-only tables in Intel Concurrent Collections [6].

Big-tent deterministic parallelism Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

The *LVars* programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a *join* (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question "has a write occurred?" (*i.e.*, is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it

POPL '14, January 22-24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01. . . $15.00.
http://dx.doi.org/10.1145/2535838.2535842

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

Conflict-Free

Marc Shapiro[1,2], Nuno Preg
[1] CITI, Univ
[2] Univ

Abstract. Replicating da
any replica to accept upda
sures performance and ava
clouds). However, publishe
Under a formal Strong Eve
ficient conditions for conv
ditions is called a Conflict-
of any CRDT are guarant
despite any number of fai
proaches [state- and opera
ditions. We study a numb
semantics, supporting both
depth the more complex C
to develop large-scale distr
ceptual properties.

Keywords: Eventual Co
Scale Distributed Systems

1 Introduction

Replication and consistency are
such as the WWW, peer-to-pe
"strong consistency" approach
This constitutes a performanc
consistency conflicts with avail
When network delays are lar
networks, disconnected operati
consistency promises better av
erates at some replica, withou

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

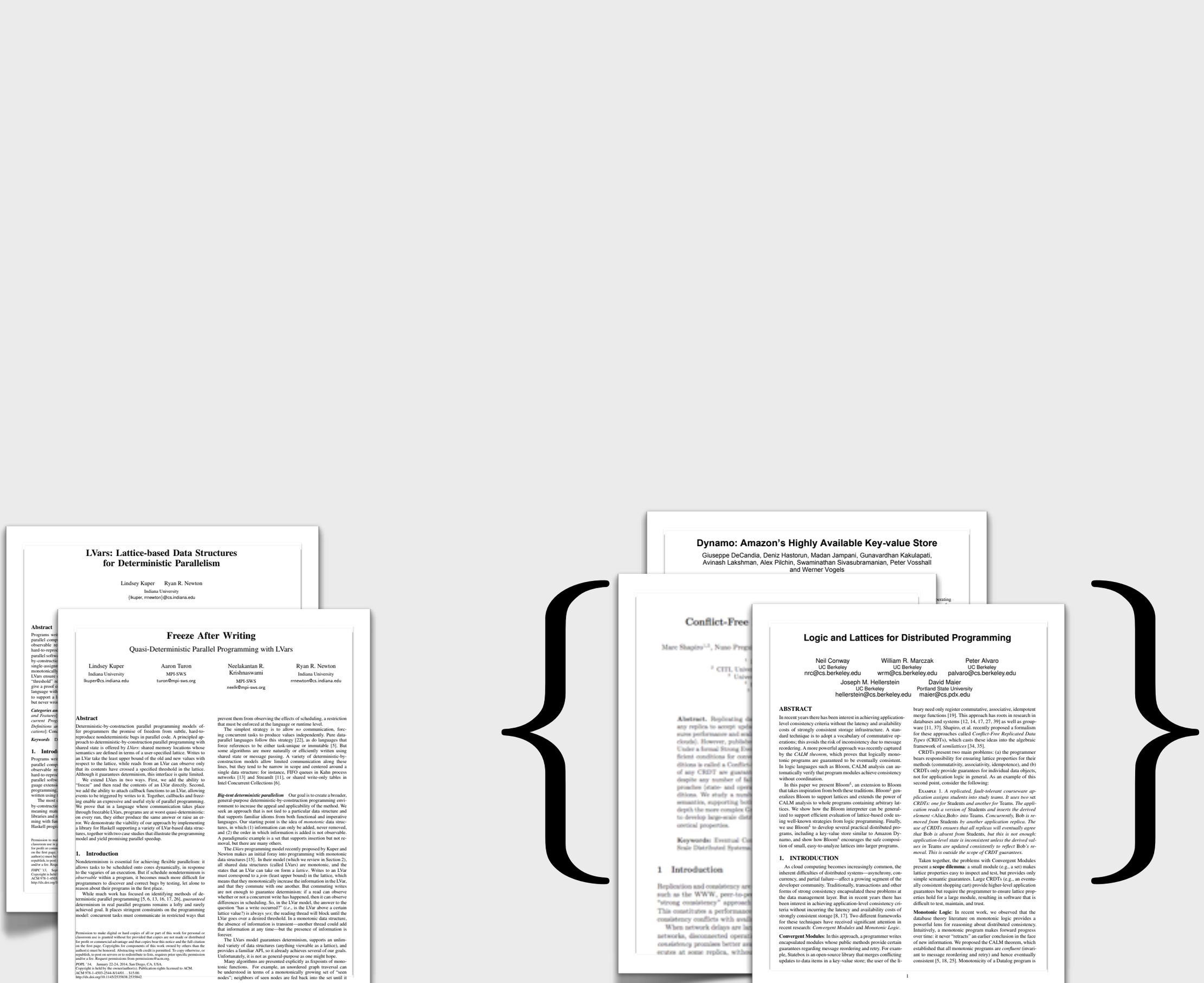
Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is



LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper Ryan R. Newton
Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract

Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org

Ryan R. Newton
Indiana University
rrnewton@cs.indiana.edu

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by *LVars*: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited.

We extend LVars in two ways. First, we add the ability to "freeze" and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is *observable* within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.

While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.

The simplest strategy is to allow *no* communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and Streamlt [11], or shared write-only tables in Intel Concurrent Collections [6].

Big-tent deterministic parallelism Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

The *LVars* programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a *join* (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question "has a write occurred?" (*i.e.*, is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FHPC '14, January 22-24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01 ... $15.00.
http://dx.doi.org/10.1145/2535838.2535842

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

Conflict-Free

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

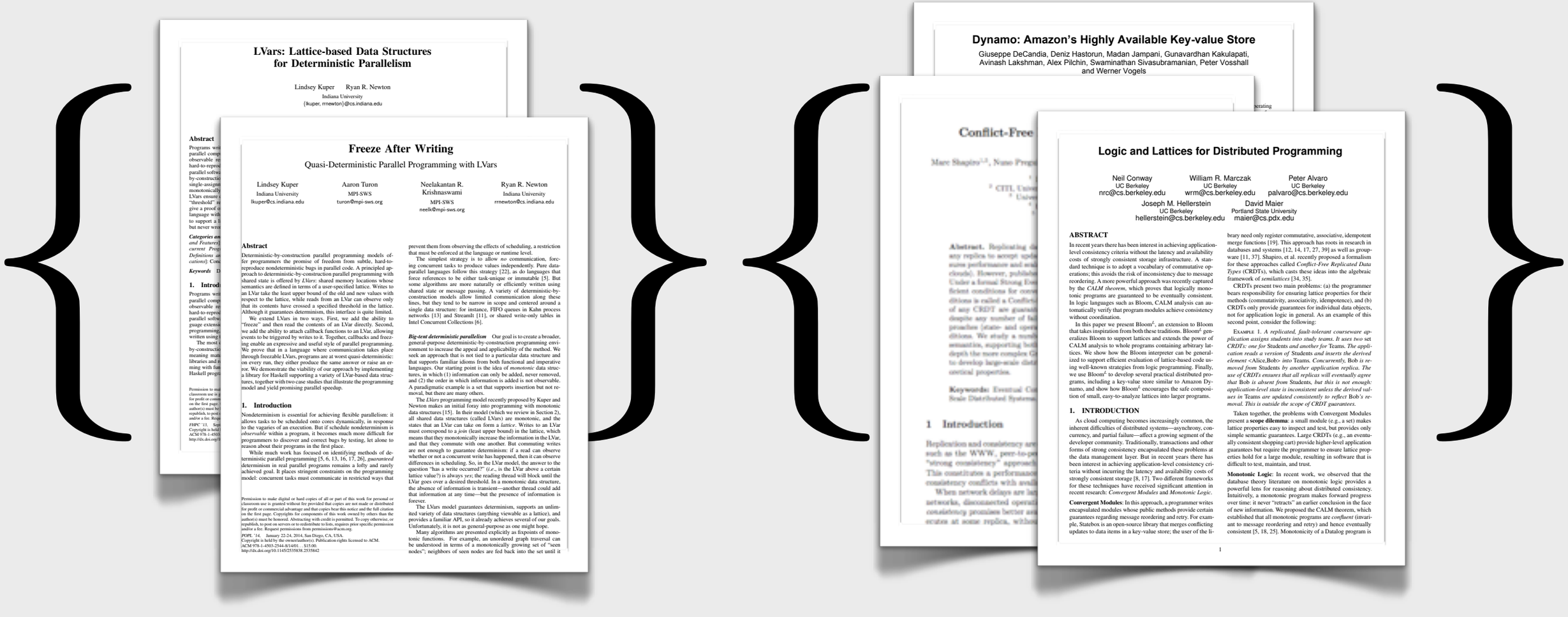
CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

1



LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper Ryan R. Newton
Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract

Programs wri
parallel comp
observable re
hard-to-reproc
parallel softwa
by-constructio
single-assignm
monotonically
LVars ensure
"threshold" re
give a proof o
language with
to support a li
but never wrot

Categories an
and Features
current Prog
Definitions an
cations]: Cons

Keywords D

1. Introd

Programs wri
parallel comp
observable re
hard-to-reproc
parallel softwa
guage extensi
programming
written using

The most
by-constructio
meaning mak
libraries and r
ming with fun
Haskell progr

Permission to ma
classroom use is
for profit or comm
on the first page.
author(s) must be
republish, to post
and/or a fee. Requ

FHPC '13, Sep
Copyright is held
ACM 978-1-4503
http://dx.doi.org/1

Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org

Ryan R. Newton
Indiana University
rrnewton@cs.indiana.edu

Abstract

Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by *LVars*: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited.

We extend LVars in two ways. First, we add the ability to "freeze" and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.

1. Introduction

Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is *observable* within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.

While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], *guaranteed* determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.

The simplest strategy is to allow *no* communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and Streamlt [11], or shared write-only tables in Intel Concurrent Collections [6].

Big-tent deterministic parallelism Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of *monotonic* data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.

The *LVars* programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a *lattice*. Writes to an LVar must correspond to a *join* (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question "has a write occurred?" (*i.e.*, is the LVar above a certain lattice value?) is always *yes*; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.

The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.

Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it

POPL '14, January 22-24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01 ... $15.00.
http://dx.doi.org/10.1145/2535838.2535842

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

Conflict-Free

Marc Shapiro[1,2], Nuno Preg

[1] CITI, Univ
[2] Univ

Abstract. Replicating d
any replica to accept upd
sures performance and av
clouds). However, publish
Under a formal Strong Ev
ficient conditions for conv
ditions is called a Conflict
of any CRDT are guarant
despite any number of fa
proaches [state- and oper
ditions. We study a numb
semantics, supporting both
depth the more complex C
to develop large-scale distr
ceptual properties.

Keywords: Eventual Co
Scale Distributed Systems

1 Introduction

Replication and consistency are
such as the WWW, peer-to-pe
"strong consistency" approach
This constitutes a performanc
consistency conflicts with avail

When network delays are lar
networks, disconnected operati
consistency promises better an
ecutes at some replica, withou

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

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

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level 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 *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

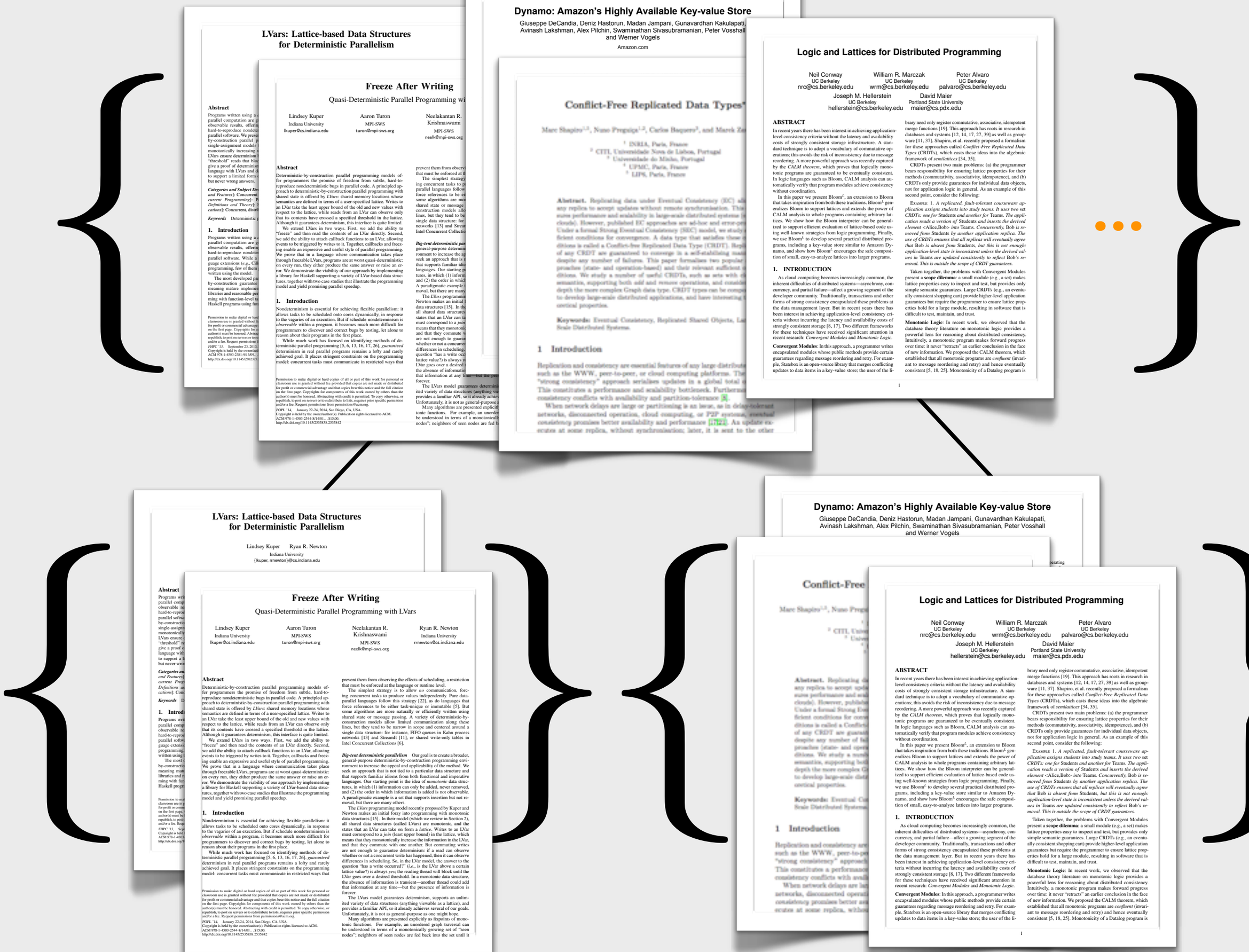
Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

EXAMPLE 1. *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CRDTs: one for* Students *and another for* Teams. *The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is


LVars: Lattice-based Data Structures for Deterministic Parallelism
Freeze After Writing
Quasi-Deterministic Parallel Programming with LVars
Lindsey Kuper
Indiana University
lkuper@cs.indiana.edu
Aaron Turon
MPI-SWS
turon@mpi-sws.org
Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org
Ryan R. Newton
Indiana University
rrnewton@cs.indiana.edu
Abstract
Deterministic-by-construction parallel programming models offer programmers the promise of freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. A principled approach to deterministic-by-construction parallel programming with shared state is offered by LVars: shared memory locations whose semantics are defined in terms of a user-specified lattice. Writes to an LVar take the least upper bound of the old and new values with respect to the lattice, while reads from an LVar can observe only that its contents have crossed a specified threshold in the lattice. Although it guarantees determinism, this interface is quite limited.
We extend LVars in two ways. First, we add the ability to "freeze" and then read the contents of an LVar directly. Second, we add the ability to attach callback functions to an LVar, allowing events to be triggered by writes to it. Together, callbacks and freezing enable an expressive and useful style of parallel programming. We prove that in a language where communication takes place through freezable LVars, programs are at worst quasi-deterministic: on every run, they either produce the same answer or raise an error. We demonstrate the viability of our approach by implementing a library for Haskell supporting a variety of LVar-based data structures, together with two case studies that illustrate the programming model and yield promising parallel speedup.
1. Introduction
Nondeterminism is essential for achieving flexible parallelism: it allows tasks to be scheduled onto cores dynamically, in response to the vagaries of an execution. But if schedule nondeterminism is observable within a program, it becomes much more difficult for programmers to discover and correct bugs by testing, let alone to reason about their programs in the first place.
While much work has focused on identifying methods of deterministic parallel programming [5, 6, 13, 16, 17, 26], guaranteed determinism in real parallel programs remains a lofty and rarely achieved goal. It places stringent constraints on the programming model: concurrent tasks must communicate in restricted ways that prevent them from observing the effects of scheduling, a restriction that must be enforced at the language or runtime level.
The simplest strategy is to allow no communication, forcing concurrent tasks to produce values independently. Pure data-parallel languages follow this strategy [22], as do languages that force references to be either task-unique or immutable [5]. But some algorithms are more naturally or efficiently written using shared state or message passing. A variety of deterministic-by-construction models allow limited communication along these lines, but they tend to be narrow in scope and centered around a single data structure: for instance, FIFO queues in Kahn process networks [13] and StreamIt [11], or shared write-only tables in Intel Concurrent Collections [6].
Big-tent deterministic parallelism Our goal is to create a broader, general-purpose deterministic-by-construction programming environment to increase the appeal and applicability of the method. We seek an approach that is not tied to a particular data structure and that supports familiar idioms from both functional and imperative languages. Our starting point is the idea of monotonic data structures, in which (1) information can only be added, never removed, and (2) the order in which information is added is not observable. A paradigmatic example is a set that supports insertion but not removal, but there are many others.
The LVars programming model recently proposed by Kuper and Newton makes an initial foray into programming with monotonic data structures [15]. In their model (which we review in Section 2), all shared data structures (called LVars) are monotonic, and the states that an LVar can take on form a lattice. Writes to an LVar must correspond to a join (least upper bound) in the lattice, which means that they monotonically increase the information in the LVar, and that they commute with one another. But commuting writes are not enough to guarantee determinism: if a read can observe whether or not a concurrent write has happened, then it can observe differences in scheduling. So, in the LVar model, the answer to the question "has a write occurred?" (i.e., is the LVar above a certain lattice value?) is always yes; the reading thread will block until the LVar goes over a desired threshold. In a monotonic data structure, the absence of information is transient—another thread could add that information at any time—but the presence of information is forever.
The LVars model guarantees determinism, supports an unlimited variety of data structures (anything viewable as a lattice), and provides a familiar API, so it already achieves several of our goals. Unfortunately, it is not as general-purpose as one might hope.
Many algorithms are presented explicitly as fixpoints of monotonic functions. For example, an unordered graph traversal can be understood in terms of a monotonically growing set of "seen nodes"; neighbors of seen nodes are fed back into the set until it
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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FHPC '13, September 23, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2381-9/13/09...$15.00.
http://dx.doi.org/10.1145/2502323.2502326
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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
POPL '14, January 22-24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535842
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
Conflict-Free Replicated Data Types*
Logic and Lattices for Distributed Programming
Neil Conway
UC Berkeley
nrc@cs.berkeley.edu
William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu
Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu
Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu
David Maier
Portland State University
maier@cs.pdx.edu
ABSTRACT
In recent years there has been interest in achieving application-level 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 CALM theorem, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.
In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.
1. INTRODUCTION
As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: Convergent Modules and Monotonic Logic.
Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called Conflict-Free Replicated Data Types (CRDTs), which casts these ideas into the algebraic framework of semilattices [34, 35].
CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:
Example 1. A replicated, fault-tolerant courseware application assigns students into study teams. It uses two set CRDTs: one for Students and another for Teams. The application reads a version of Students and inserts the derived element <Alice,Bob> into Teams. Concurrently, Bob is removed from Students by another application replica. The use of CRDTs ensures that all replicas will eventually agree that Bob is absent from Students, but this is not enough: application-level state is inconsistent unless the derived values in Teams are updated consistently to reflect Bob's removal. This is outside the scope of CRDT guarantees.
Taken together, the problems with Convergent Modules present a scope dilemma: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a large module, resulting in software that is difficult to test, maintain, and trust.
Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are confluent (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is
1

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.