LVars:
Lattice-based Data Structures
for Deterministic Parallel
and Distributed Programming

Lindsey Kuper

, New York, NY
March 24, 2014
Parallel systems

Distributed systems
Deterministic Parallel Programming
Parallel Programming (observably) Deterministic Parallel Programming
```
data Item = Book | Shoes | ...
```
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
      async (atomicModifyIORef cart
            (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
          (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
          (\m -> (insert Shoes 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
    wait res
landin:lvar-examples lkuper$ make map-io-ref-data-race
ghc -O2 map-io-ref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main
    ( map-io-ref-data-race.hs, map-io-ref-data-race.o )
Linking map-io-ref-data-race ...
while true; do ./map-io-ref-data-race +RTS -N2; done
landin:ivar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main
 (map-ioref-data-race.hs, map-ioref-data-race.o)
Linking map-ioref-data-race ...
while true: do /map-ioref-data-race +RTS -N2; done

((Book,1),(Shoes,1))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
  async (atomicModifyIORef cart
    (\m -> (insert Book 1 m, ())))
  async (atomicModifyIORef cart
    (\m -> (insert Shoes 1 m, ()))))
  res <- async (readIORef cart)
  wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
      (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
      (\m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
    wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
  al <- async (atomicModifyIORef cart
    (\m -> (insert Book 1 m, ())))
  async (atomicModifyIORef cart
    (\m -> (insert Shoes 1 m, ()))))
  res <- async (readIORef cart)
  wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)

p = do cart <- newIORef empty
    a1 <- async (atomicModifyIORef cart
                (\m -> (insert Book 1 m, ())))
    a2 <- async (atomicModifyIORef cart
                (\m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
    wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    a1 <- async (atomicModifyIORef cart
                     (\m -> (insert Book 1 m, ())))
    a2 <- async (atomicModifyIORef cart
                     (\m -> (insert Shoes 1 m, ()))))
    res <- async (do waitBoth a1 a2
                      wait res  —— readIORef cart)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    a1 <- async (atomicModifyIORef cart
                (\m -> (insert Book 1 m, ())))
    a2 <- async (atomicModifyIORef cart
                (\m -> (insert Shoes 1 m, ()))))
    res <- async (do waitBoth a1 a2
                   readIORef cart)
    wait res
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
                (\m -> (insert Book 1 m, ())))
  a2 <- async (atomicModifyIORef cart
                (\m -> (insert Shoes 1 m, ()))))
  res <- async (do waitBoth a1 a2
                  readIORef cart)
  wait res

main = do v <- p
         print v
p :: IO (Map Item Int)

p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart \m -> (insert Book 1 m, ()))
  a2 <- async (atomicModifyIORef cart \m -> (insert Shoes 1 m, ()))
  res <- async (do waitBoth a1 a2
                readIORef cart)

  wait res

main = do v <- p
          print v

deterministic...now
p :: IO (Map Item Int)
p = do
  cart <- newIORef empty
  a1 <- async (atomicModifyIORef cart
              (\m -> (insert Book 1 m, ()))))
  a2 <- async (atomicModifyIORef cart
              (\m -> (insert Shoes 1 m, ()))))
  res <- async (do waitBoth a1 a2
                readIORef cart)
  wait res

main = do v <- p
         print v

deterministic...now...we hope
p :: IO (Map Item Int)

\[
p = \text{do}
\]
\[
cart \leftarrow \text{newIORef} \text{ empty}
\]
\[
al \leftarrow \text{async} (\text{atomicModifyIORef} \ cart
\quad (\text{\textbackslash m} \rightarrow (\text{insert} \ \text{Book} \ 1 \ m, () )))
\]
\[
a2 \leftarrow \text{async} (\text{atomicModifyIORef} \ cart
\quad (\text{\textbackslash m} \rightarrow (\text{insert} \ \text{Shoes} \ 1 \ m, () )))
\]
\[
\text{res} \leftarrow \text{async} (\text{do} \ \text{waitBoth} \ a1 \ a2
\quad \text{readIORef} \ cart)
\]
\[
\text{wait} \ \text{res}
\]
\[
\text{main} = \text{do} \ v \leftarrow p
\quad \text{print} \ v
\]

\[
p :: \text{Par Det} (\text{IMap} \ \text{Item} \ \text{Int})
\]

\[
p = \text{do}
\]
\[
cart \leftarrow \text{newEmptyMap}
\]
\[
\text{fork} (\text{insert} \ \text{Book} \ 1 \ \text{cart})
\]
\[
\text{fork} (\text{insert} \ \text{Shoes} \ 1 \ \text{cart})
\]
\[
\text{return} \ cart
\]
\[
\text{main} = \text{print} (\text{runParThenFreeze} \ p)
\]

\text{deterministic by construction}

\text{deterministic...now...we hope}
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

- Kahn process networks
- single-assignment
- imperative disjoint
- \( \lambda \)-calculus

\[
f \left( g \left( x \right) \right) \\
\left( h \left( y \right) \right)
\]
The deterministic by construction parallel programming landscape:

- Kahn process networks
- single-assignment
- imperative disjoint
- \( \lambda \)-calculus

\[
f (g x) (h y)
\]
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

- Kahn process networks
- Single-assignment
- Imperative disjoint
- \( \lambda \)-calculus

\[
(f \ (g \ x)) \ (h \ y)
\]
The deterministic by construction parallel programming landscape:

- Kahn process networks
- Single-assignment
- Imperative disjoint
- \( \lambda \)-calculus
- \((g \ x)\) (h y)
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

$\lambda$-calculus

$\lambda$-calculus

($\&$ array langs, ...)

$f (g \ x) (h \ y)$
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment (IVars)

imperative disjoint

λ-calculus (& array langs, ...)

G → H

g(left) h(right)

f (g x) (h y)
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

- **Kahn process networks**
- **single-assignment (IVars)**
- **imperative disjoint**
- **λ-calculus (& array langs, ...)**

\[
\lambda\text{-calculus}:
\begin{align*}
f (g x) & = (h y) \\
g\text{(left)} & \rightarrow h\text{(right)}
\end{align*}
\]
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment
(IVars)

imperative disjoint

λ-calculus
(& array langs, ...)

Can we generalize and unify these points on the map?
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment (IVars)

imperative disjoint

\( \lambda \)-calculus

\( f (g x) (h y) \)

Can we generalize and unify these points on the map? Yes!
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
            (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
            (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
   async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ())))
   async (atomicModifyIORef cart
         (\m -> (insert Shoes 1 m, ()))))
   res <- async (readIORef cart)
   wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
          (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
          (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]

LVars: multiple least-upper-bound writes, blocking threshold reads
```hsaskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
          (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
          (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res
```

**IVars:** single writes, blocking (but exact) reads

[Arvind et al., 1989]

**LVars:** multiple least-upper-bound writes, blocking threshold reads

* actually a bounded join-semilattice
\[
\begin{align*}
\text{num} & \\
\begin{array}{c}
\top \\
0 & 1 & 2 & 3 & 4 & \ldots \\
\bot
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{do} \\
\text{fork (put num 3)} \\
\text{fork (put num 4)} \\
\end{align*}
\]

Raises an error, since \(3 \sqcup 4 = \top\)

\[
\begin{align*}
\text{do} \\
\text{fork (put num 4)} \\
\text{fork (put num 4)} \\
\end{align*}
\]

Works fine, since \(4 \sqcup 4 = 4\)
data Item = Book | Shoes | ... 

p = do 
cart <- newEmptyMap 
fork (insert Shoes 1 cart) 
fork (insert Book 2 cart) 
getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
    cart <- newEmptyMap
    fork (insert Shoes 1 cart)
    fork (insert Book 2 cart)
    getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
    cart <- newEmptyMap
    fork (insert Shoes 1 cart)
    fork (insert Book 2 cart)
    getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
data Item = Book | Shoes | ...

p = do
  cart <- newEmptyMap
  fork (insert Shoes 1 cart)
  fork (insert Book 2 cart)
  getKey Book cart -- returns 2
\[ \text{data Item = Book | Shoes | \ldots} \]

\[ \{(\text{Book},1), (\text{Book},2), \ldots\} \]

\[ p = \text{do} \]
\begin{align*}
& \text{cart <- newEmptyMap} \\
& \text{fork (insert Shoes 1 cart)} \\
& \text{fork (insert Book 2 cart)} \\
& \text{getKey Book cart -- returns 2}
\end{align*}
The threshold set must be pairwise incompatible.
The threshold set must be pairwise incompatible

\{ (Book, 1), (Book, 2), ... \}

data Item = Book | Shoes | ... 

p = do 
cart <- newEmptyMap 
fork (insert Shoes 1 cart) 
fork (insert Book 2 cart) 
getKey Book cart -- returns 2
The threshold set must be pairwise incompatible.
seen nodes
seen nodes
seen nodes

0 1 3 4 5 6 7 9 10

11
seen nodes

0 1 3
4 5 6
7 9 10
11

already seen
0
1
3
4
5
6
7
9
10
11

already seen
6
7
9
10
11

already seen
0
1
3
4
5
6
7
9
10
11

already seen

already seen

already seen

already seen
already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen
Events are updates that change an LVar's state.
Events are updates that change an LVar's state
Event handlers listen for events and launch callbacks in response
Events are updates that change an LVar's state
Event handlers listen for events and launch callbacks in response

```haskell
traverse g startNode = do
```
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

```
traverse g startNode = do
  seen <- newEmptySet
```
Events are updates that change an LVar's state.

Event handlers listen for events and launch callbacks in response.

```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
    (\node -> do
       mapM (\v -> insert v seen)
        (neighbors g node)
      return ()
    )
```
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

```hs
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
    (\node -> do
      mapM (\v -> insert v seen) (neighbors g node)
      return ()
    )
  insert startNode seen
```
Events are updates that change an LVar's state.

Event handlers listen for events and launch callbacks in response.

**quiesce** blocks until all callbacks launched by a given handler are done running.

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
    (\node -> do
      mapM (\v -> insert v seen)
      (neighbors g node)
      return ()
    )
  insert startNode seen
```
Events are updates that change an LVar's state

Event handlers listen for events and launch callbacks in response

**quiesce** blocks until all callbacks launched by a given handler are done running

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
    (\node -> do
      mapM (\v -> insert v seen)
        (neighbors g node)
      return ()
    )
  insert startNode seen
  quiesce h
```
Events are updates that change an LVar's state
Event handlers listen for events and launch callbacks in response
quiesce blocks until all callbacks launched by a given handler are done running

```
traverse g startNode = do
    seen <- newEmptySet
    h <- newHandler seen
        (\node -> do
          mapM (\v -> insert v seen)
            (neighbors g node)
            return ()
        )
    insert startNode seen
    quiesce h
...
```
**freeze**: exact non-blocking read

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
    (neighbors g node)
    return ()
  )
  insert startNode seen
  quiesce h
...
```
**freeze**: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-`freeze` exception

```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
      (neighbors g node)
      return ()
  )
  insert startNode seen
  quiesce h
...
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception

```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
    (neighbors g node)
    return ()
  )
  insert startNode seen
  quiesce h
  freeze seen
```
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception
Two possible outcomes: either the same final value or an exception

```haskell
traverse g startNode = do
    seen <- newEmptySet
    h <- newHandler seen
    (\node -> do
        mapM (\v -> insert v seen) (neighbors g node)
        return ()
    )
    insert startNode seen
    quiesce h
    freeze seen
```
**freeze**: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**freeze** exception

Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). If \( \sigma \xrightarrow{*} \sigma' \) and \( \sigma \xrightarrow{*} \sigma'' \), no \( \sigma' \) and neither \( \sigma' \) nor \( \sigma'' \) can take a step, then either:

1. \( \sigma' = \sigma'' \) up to a permutation on locations \( \pi \), or
2. \( \sigma' = \text{error} \) or \( \sigma'' = \text{error} \).

```scheme
(define-LVish-language nat downset-op max natural)
```

## 5. Quasi-Determinism for LVish

- A **redex** is a macro to \( \text{Racket proper} \), but because

```scheme
(define runParIO (runParThenFreeze (insert v seen) (node g node) (insert v seen)))
return ()
insert startNode seen
quiesce h
freeze seen
```
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception
Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). If \( \sigma \xrightarrow{}* \sigma' \) and \( \sigma \xrightarrow{}* \sigma'' \), and neither \( \sigma' \) nor \( \sigma'' \) can take a step, then either:

1. \( \sigma' = \sigma'' \) up to a permutation on locations \( \pi \), or
2. \( \sigma' = \text{error} \) or \( \sigma'' = \text{error} \).
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception
Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). If $\sigma \xrightarrow{}^* \sigma'$ and $\sigma \xrightarrow{}^* \sigma''$, and neither $\sigma'$ nor $\sigma''$ can take a step, then either:
1. $\sigma' = \sigma''$ up to a permutation on locations $\pi$, or
2. $\sigma' = \text{error}$ or $\sigma'' = \text{error}$.  

```
[(Book,1),(Shoes,1)]
[(Book,1)]
[(Shoes,1)]
```

```
[(Book,1),(Shoes,1)]
```

```
insert v seen
insert startNode seen
quiesce h
freeze seen
```
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception
Two possible outcomes: either the same final value or an exception

Theorem 1 (Quasi-Determinism). If $\sigma \xrightarrow{*} \sigma'$ and $\sigma \xrightarrow{*} \sigma''$, where neither $\sigma'$ nor $\sigma''$ can take a step, then either:
1. $\sigma' = \sigma''$ up to a permutation on locations $\pi$, or
2. $\sigma' = \text{error}$ or $\sigma'' = \text{error}$.
**freeze**: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**freeze** exception

Two possible outcomes: either the same final value or an exception

---

**Theorem 1** (Quasi-Determinism). If $\sigma \xrightarrow{}^* \sigma'$ and $\sigma \xrightarrow{}^* \sigma''$, do and neither $\sigma'$ nor $\sigma''$ can take a step, then either:

1. $\sigma' = \sigma''$ up to a permutation on locations $\pi$, or
2. $\sigma' = \text{error}$ or $\sigma'' = \text{error}$.
**freeze**: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**freeze** exception.

Two possible outcomes: either the same final value or an exception

```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen) (neighbors g node)
    return ()
  )
  insert startNode seen
  quiesce h
  freeze seen
```
**freeze**: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-**freeze** exception
Two possible outcomes: either the same final value or an exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
      (neighbors g node)
      return ()
  )
  insert startNode seen
  quiesce h
  freeze seen
```
freeze: exact non-blocking read
Attempts to write to a frozen LVar raise a write-after-freeze exception
Two possible outcomes: either the same final value or an exception

```
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
    (neighbors g node)
    return ()
  )
  insert startNode seen
  quiesce h
  freeze seen
```

Let the system handle this for us:
**freeze**: exact non-blocking read

Attempts to write to a frozen LVar raise a write-after-**freeze** exception

Two possible outcomes: either the same final value or an exception

```haskell
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen)
    (neighbors g node)
    return ()
  )
  insert startNode seen
  quiesce h
  freeze seen
```

*Let the system handle this for us:* runParThenFreeze
LVish

a Haskell library for parallel programming with LVars
LVish

a Haskell library for parallel programming with LVars

LVar operations run in \texttt{Par} computations
LVish

a Haskell library for parallel programming with LVars

LVar operations run in Par computations

Lightweight threads
LVish

a Haskell library for parallel programming with LVars

LVars operations run in $\text{Par}$ computations

Lightweight threads

$\text{Par}$ computations indexed by effect level

\begin{verbatim}
 p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart
\end{verbatim}
LVish
a Haskell library for parallel programming with LVars

LVar operations run in **Par** computations
Lightweight threads
**Par** computations indexed by *effect level*
**runParThenFreeze** captures the freeze-after-writing idiom

```haskell
p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart
```
LVish

a Haskell library for parallel programming with LVars

LVar operations run in Par computations
Lightweight threads
Par computations indexed by effect level

runParThenFreeze captures the freeze-after-writing idiom

```hs
p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart

main = print (runParThenFreeze p)
```
LVish

a Haskell library for parallel programming with LVars

LVar operations run in Par computations

Lightweight threads

Par computations indexed by effect level

runParThenFreeze captures the freeze-after-writing idiom

Efficient lock-free sets, maps, etc.

```
p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart

main = print (runParThenFreeze p)
```
LVish

a Haskell library for parallel programming with LVars

LVar operations run in \texttt{Par} computations

Lightweight threads

\texttt{Par} computations indexed by \texttt{effect level}

\texttt{runParThenFreeze} captures the freeze-after-writing idiom

Efficient lock-free sets, maps, etc.

Implement your own LVars, too

\begin{verbatim}

p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart

main = print (runParThenFreeze p)

\end{verbatim}
LVish
a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations
Lightweight threads
`Par` computations indexed by effect level
`runParThenFreeze` captures the freeze-after-writing idiom
Efficient lock-free sets, maps, etc.
Implement your own LVars, too

```
p :: Par Det (IMap Item Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
return cart
```
```
main = print (runParThenFreeze p)
```

hackage.haskell.org/package/lvish
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

\[ f \ (g \ x) \ (h \ y) \]

g(left)  h(right)
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

G → H

\( f(g(x), h(y)) \)

\( g(left), h(right) \)
LVars and LVish across the landscape:
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

\( f (g \ x) \)

\( (h \ y) \)
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

f (g x) (h y)

g(left) h(right)
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

\[ f (g \ x) \ (h \ y) \]

\[ g(left) \ h(right) \]
Parallel systems

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

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

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

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

General Terms

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

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

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

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

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

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

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

[DeCandia et al., SOSP '07]
Conflicts-Free Replicated Data Types*

Marc Shapiro1,5, Nuno Preguiça2, Carlos Baquero3, and Marek Zawirski1,4
1 INRIA, Paris, France
2 CIITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPDC, Paris, France
5 LIPI, 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]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas.

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hasto, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, 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 system.

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

C.2.1 [Computer Systems Organization]: Special-Purpose and High-Performance Systems—Highly available systems

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

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

To meet the high-latency and scaling needs, Amazon has developed Dynamo, another highly available key-value store built for Amazon’s platform.

Dynamo is aware of the data schema it manages. It is a method that is best suited for managing the application that maintains the shopping cart. The application that maintains the shopping cart needs to “merge” the conflicting replicas.

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

[Shapiro et al., SSS ’11]

[DeCandia et al., SOSP ’07]
Conflict-Free Replicated Data Types

Marc Shapiro$^{1,5}$, Nuno Preguiça$^{1,2}$, Carlos Baqueró$^3$, 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 those conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both add and remove operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

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

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard “strong consistency” approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8]. When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, eventual consistency promises better availability and performance [12]. An update executes at some replica, without synchronisation; later, it is sent to the other replicas or is propagated directly. However, this approach is not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ©2007 ACM 978-1-59593-591-5/07/0010...$5.00.

Dynamo: Amazon’s Highly Available Key-value Store

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

Amazon.com

Abstract

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 attacked by tomahawks. 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 to multiple data centers.

Dynamo is aware of the data schema it is managing, and is designed to be fault-tolerant, highly scalable, and always available. Dynamo is built on the assumption that the data store is naturally fault-tolerant, so as to meet the reliability and scaling needs, Amazon has developed Dynamo, another highly available key-value store built on top of Amazon Simple Storage Service, and known as Amazon’s Dynamo. This paper presents the Dynamo architecture, its design, and the lessons learned from its implementation. Dynamo has been operational since its launch in 2007.

To maintain the reliability and scalability of the application, Dynamo is designed to be both fault-tolerant and highly available. Dynamo uses a combination of techniques to achieve these goals. One of the key design decisions is to use a distributed data store architecture that is highly scalable and fault-tolerant. Dynamo uses a specially designed data store that is designed to be fault-tolerant and highly available. Dynamo uses a combination of techniques to achieve these goals. Dynamo is designed to be both fault-tolerant and highly available. Dynamo uses a combination of techniques to achieve these goals.

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 using a quorum-based technique and a decentralized replica synchronization protocol. Dynamo employs

[Shapiro et al., SSS ’11]

[DeCandia et al., SOSP ’07]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent”
CvRDTs
“state-based”

“Commutative”
CmRDTs
“op-based”

\[
\text{put} \cdot \text{put} = \text{put} \cdot \text{put}
\]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent”
CvRDTs
“state-based”

“Commutative”
CmRDTs
“op-based”

\[\text{put} \cdot \text{put} = \text{put} \cdot \text{put}\]

[Shapiro et al., SSS '11]
Two “styles” of Conflict-Free Replicated Data Types:

“Convergent”
CvRDTs
“state-based”

≡

“Commutative”
CmRDTs
“op-based”

[Shapiro et al., SSS ’11]
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads</th>
<th>Ordinary reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>(deterministic)</td>
<td>(nondeterministic)</td>
</tr>
<tr>
<td>Least-upper-bound writes</td>
<td>General inflationary writes</td>
</tr>
<tr>
<td>(every write computes a join)</td>
<td>(only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes</td>
<td>General inflationary writes</td>
</tr>
<tr>
<td>(every write computes a join)</td>
<td>(only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes</td>
<td>General inflationary writes</td>
</tr>
<tr>
<td>(every write computes a join)</td>
<td>(only merges must be joins)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Replicated!</td>
</tr>
</tbody>
</table>
LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (nondeterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes (every write computes a join)</td>
<td>General inflationary writes (only merges must be joins)</td>
</tr>
</tbody>
</table>

Shared memory | Replicated!

so we’re proposing
Adding threshold reads to CvRDTs

LVars vs. CvRDTs

- **Threshold reads (deterministic)**
  - Least-upper-bound writes (every write computes a join)
  - Shared memory

- **Ordinary reads (nondeterministic)**
  - General inflationary writes (only merges must be joins)
  - Replicated!

- Adding threshold reads to CvRDTs
  - one framework for reasoning about both eventual and strong consistency

---

so we're proposing
Adding threshold reads to CvRDTs

so we're proposing

Adding general inflationary writes to LVars

e.g., non-idempotent, incrementable counters
Parallel systems

Distributed systems
LVars and LVish across the landscape:

- Kahn process networks
- single-assignment
- imperative disjoint
- λ-calculus

\[ f (g \ x) \quad (h \ y) \]
LVars and LVish across the landscape:

Kahn process networks

imperative disjoint

λ-calculus

f (g x) (h y)

g(left)  h(right)
Thank you!

LVars project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples
Research blog: composition.al