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

Lindsey Kuper
Compose :: Conference
January 31, 2015

Joint work with Ryan Newton, Neel Krishnaswami, Aaron Turon, Sam Tobin-Hochstadt
(with illustrations by Jason Reed)
Parallel systems

Distributed systems
Deterministic Parallel Programming
(observably)
Deterministic Parallel Programming
<table>
<thead>
<tr>
<th>data Item</th>
<th>Book</th>
<th>Shoes</th>
<th>...</th>
</tr>
</thead>
</table>

= Shopping cart icon
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, ()�)))
\[ \text{data Item = Book | Shoes | ...} \]

\[ p :: \text{IO (Map Item Int)} \]

\[ p = \text{do cart <- newIORef empty} \]
\[ \quad \text{async (atomicModifyIORef cart} \]
\[ \quad \quad (\text{m -> (insert Book 1 m, ())}) \]
\[ \quad \quad \text{async (atomicModifyIORef cart} \]
\[ \quad \quad (\text{m -> (insert Shoes 1 m, ())}) \]
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:ivar-examples lkuper$ make map-ioref-data-race
ghc -O2 map-ioref-data-race.hs -rtsopts -threaded
[1 of 1] Compiling Main
(linking map-ioref-data-race ...)
while true; do /map-ioref-data-race +RTS -N2; done

[[Book,1],(Shoes,1)]
```bash
landin:lvar-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
```

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

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

```bash
[(Shoes,1)]
[(Shoes,1)]
[(Shoes,1)]
[(Shoes,1)]
[(Shoes,1)]
```
if we want determinism, we have to learn to share nicely
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

deterministic
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)

```haskell
def 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 by construction

[FHPC '13, POPL '14]
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

- Kahn process networks
- single-assignment
- imperative disjoint
- $\lambda$-calculus
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:
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment

imperative disjoint

$\lambda$-calculus

$G \rightarrow H$

$\lambda$ (g x)

(h y)

$\{ \text{array langs, ...} \}$
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:
The deterministic by construction parallel programming landscape:
The deterministic by construction parallel programming landscape:

Kahn process networks

single-assignment (IVars, CnC, ...)

imperative disjoint

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

\[ g\ \text{(left)} \ \\
\text{h\ (right)} \]

\[ &\text{array langs, ...} \]

\[ \lambda\text{-calculus} \]
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 (IVars, CnC, ...)

imperative disjoint (DPJ, ...)

λ-calculus (& array langs, ...)

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

Can we generalize and unify these points in the space? 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]
\begin{verbatim}
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
\end{verbatim}

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

LVars: multiple commutative and inflationary writes, blocking threshold reads
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 commutative and inflationary writes, blocking threshold reads

* actually a bounded join-semilattice
Raised 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)
Raise an error, since $3 \sqcup 4 = \top$

\begin{verbatim}
  do
  fork (put num 3)
  fork (put num 4)
\end{verbatim}

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

\begin{verbatim}
  do
  fork (put num 4)
  fork (put num 4)
\end{verbatim}

get blocks until threshold is reached

\begin{verbatim}
  do
  fork (put num 4)
  get num
\end{verbatim}
num

\[ \begin{array}{c}
\top \\
\downarrow \\
0 & 1 & 2 & 3 & 4 & \ldots \\
\downarrow \\
\bot 
\end{array} \]

\textbf{ Raises an error, since } 3 \sqcup 4 = T \\
\textbf{ do} \\
\textbf{ fork (put num 3)} \\
\textbf{ fork (put num 4)} \\

\textbf{ Works fine, since } 4 \sqcup 4 = 4 \\
\textbf{ do} \\
\textbf{ fork (put num 4)} \\
\textbf{ fork (put num 4)} \\

\textbf{ get blocks until threshold is reached} \\
\textbf{ do} \\
\textbf{ fork (put num 4)} \\
\textbf{ get num}
threshold set; elements must be pairwise incompatible

does
fork (put num 3)
fork (put num 4)

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

does
fork (put num 4)
fork (put num 4)

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

gets blocks until threshold is reached
do
fork (put num 4)
get num
Data structure author's obligation:

threshold set; elements must be \textit{pairwise incompatible}

\begin{align*}
\text{num} & \quad \top \\
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad \ldots \\
\bot &
\end{align*}

\begin{align*}
\text{Raises an error, since } 3 \sqcup 4 &= \top \\
\text{do} & \\
& \quad \text{fork (put num 3)} \\
& \quad \text{fork (put num 4)}
\end{align*}

\begin{align*}
\text{Works fine, since } 4 \sqcup 4 &= 4 \\
\text{do} & \\
& \quad \text{fork (put num 4)} \\
& \quad \text{fork (put num 4)}
\end{align*}

\begin{align*}
\text{get blocks until threshold is reached} \\
\text{do} & \\
& \quad \text{fork (put num 4)} \\
& \quad \text{get num}
\end{align*}
Works fine, since \texttt{incrs commute}

do

\begin{align*}
&\text{fork (incr1 counter)} \\
&\text{fork (incr42 counter)}
\end{align*}
Works fine, since \textit{incrs commute}

\begin{itemize}
\item \textbf{do}
\begin{itemize}
\item fork (incr\textsubscript{1} counter)
\item fork (incr\textsubscript{42} counter)
\end{itemize}
\end{itemize}

\begin{itemize}
\item \textbf{get blocks until threshold is reached}
\begin{itemize}
\item \textbf{do}
\begin{itemize}
\item fork (incr\textsubscript{1} counter)
\item fork (incr\textsubscript{42} counter)
\end{itemize}
\end{itemize}
\end{itemize}

get counter 2
Works fine, since \texttt{incrs} commute

\begin{align*}
\textbf{do} & \\
& \text{fork (incr1 counter)} \\
& \text{fork (incr42 counter)}
\end{align*}

\begin{align*}
\textbf{get} & \text{ blocks until threshold is reached} \\
\textbf{do} & \\
& \text{fork (incr1 counter)} \\
& \text{fork (incr42 counter)} \\
& \text{get counter 2}
\end{align*}
Works fine, since \texttt{incrs} commute
\begin{verbatim}
do
  fork (incr1 counter)
  fork (incr42 counter)
\end{verbatim}

get blocks until threshold is reached
\begin{verbatim}
do
  fork (incr1 counter)
  fork (incr42 counter)
  get counter 2
\end{verbatim}

unblocks when \texttt{counter} is at least 2
exact contents of \texttt{counter} not observable
Can't see the exact, complete contents of an LVar
Can't see the exact, complete contents of an LVar
Can't iterate over the contents of an LVar
Can’t see the exact, complete contents of an LVar
Can’t iterate over the contents of an LVar
Can’t determine if something isn’t in the LVar
Can’t see the exact, complete contents of an LVar
Can’t iterate over the contents of an LVar
Can’t determine if something isn’t in the LVar
Can’t react to writes that we weren’t expecting
- Can see the exact, complete contents of an LVar
- Can iterate over the contents of an LVar
- Can determine if something *isn’t* in the LVar
- Can react to writes that we weren’t expecting
Can see the exact, complete contents of an LVar
Can iterate over the contents of an LVar
Can determine if something isn’t in the LVar
Can react to writes that we weren’t expecting

{ handlers, quiescence, freezing }
freeze after writing
(or before reading)
seen nodes
seen nodes

0
seen nodes

already seen

already seen
seen nodes

0 1 3
4 5 6
7 9 10
11

already seen nodes

already seen

...
seen nodes

0 1 3

4 5 6

7 9 10

11

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

```
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

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

```haskell
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

```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
...
```
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

```
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
...
```
freezel: 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 \( \sigma \xrightarrow{*} \sigma'' \), 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 \( \sigma'' = \text{error} \).

---

```haskell
runParThenFreeze :: DeepFrz a => a -> a
runParIO :: Par lvl a => a
runPar :: Par Det a => a
```

[POPL '14]
**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)]
```

---

```
insert v seen)
        (inserting node)
    return ()
insert startNode seen
quiesce h
freeze seen
```
**freeze**: exact non-blocking read

Attempts to write to a frozen LVare 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 $\sigma'' = \text{error}$.

---

```plaintext
(Book,1), (Shoes,1)
```

```
(Book,1)
```

```
(Shoes,1)
```

Redex
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 \( \sigma'' = \text{error} \).

[POPL '14]

```haskell
runParIO :: Par lvl a
runPar :: Par Det a
```

```haskell
runST
```

```
runParThenFreeze
```

```
runIO
```

```
runSTIO
```

```
runParIO
```

```haskell
insert startNode seen
return ()
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 \( \sigma'' = \text{error} \).

[POPL '14]

---

```
runPar :: Par Det a
runParIO :: Par lvl a

runParIO (runPar (Par Det a)) = return ()

insert v seen
insert startNode seen
quiesce h
freeze seen
```
LVish
a Haskell library for parallel programming with LVars
LVish

a Haskell library for parallel programming with LVars

LVar operations run in Par computations
LVish

a Haskell library for parallel programming with LVars

LVar operations run in \texttt{Par} computations
Lightweight threads
a Haskell library for parallel programming with LVars

LVish

LVar operations run in \texttt{Par} computations
Lightweight threads
\texttt{Par} computations indexed by \textit{effect level}

\begin{code}
\begin{verbatim}
p :: HasPut e => Par es (IMap Item s Int)
p = do
cart <- newEmptyMap
fork (insert Book 1 cart)
fork (insert Shoes 1 cart)
\end{verbatim}
\end{code}
LVish

a Haskell library for parallel programming with LVars

LVar operations run in \texttt{Par} computations

Lightweight threads

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

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

\begin{verbatim}

p :: HasPut e =>
    Par e s (IMap Item s Int)

p = do
    cart <- newEmptyMap
    fork (insert Book 1 cart)
    fork (insert Shoes 1 cart)

\end{verbatim}
LVish

a Haskell library for parallel programming with LVars

LVar operations run in \textbf{Par} computations

Lightweight threads

\textbf{Par} computations indexed by \textit{effect level}

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

\begin{verbatim}
p :: HasPut e => Par e s (IMap Item s 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 \texttt{Par} computations

Lightweight threads

\texttt{Par} computations indexed by effect level

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

\begin{verbatim}
p :: HasPut e => Par e s (IMap Item s 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 expresses the freeze-after-writing idiom
Efficient lock-free sets, maps, etc.

```haskell
p :: HasPut e => Par e s (IMap Item s 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` expresses the freeze-after-writing idiom

Efficient lock-free sets, maps, etc.

Implement your own LVars, too

```haskell
p :: HasPut e => Par e s (IMap Item s 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 expresses the freeze-after-writing idiom
Efficient lock-free sets, maps, etc.
Implement your own LVars, too

hackage.haskell.org/package/lvish
LVish

a Haskell library for parallel programming with LVars

LVar operations run in `Par` computations
Lightweight threads

`Par` computations indexed by `effect level`

`runParThenFreeze` expresses the freeze-after-writing idiom
Efficient lock-free sets, maps, etc.
Implement your own LVars, too

hackage.haskell.org/package/lvish

github.com/iu-parfunc/lvars
Deterministic Parallel Programming
(observably)
Deterministic Parallel Programming
(observably)  (irregular)

Deterministic Parallel Programming
Case study: $k$-CFA static analysis parallelized with LVish [POPL ’14]

[Earl et al., ICFP ’12]
Case study: $k$-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!

[Earl et al., ICFP ’12]
Case study: $k$-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
Case study: k-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup

Parallel Speedup

- Linear speedup
- blur (lock-free)
- notChain (lock-free)
- blur (pure)
- notChain (pure)
Case study: $k$-CFA static analysis parallelized with LVish [POPL '14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help

### Parallel Speedup

<table>
<thead>
<tr>
<th>Processors</th>
<th>linear speedup</th>
<th>blur (lock-free)</th>
<th>blur (pure)</th>
<th>notChain (lock-free)</th>
<th>notChain (pure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

- 0-20x speedup, even on one core.
Case study: k-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help
Case study: k-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help
Case study: k-CFA static analysis parallelized with LVish [POPL '14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help
Case study: $k$-CFA static analysis parallelized with LVish [POPL ’14]

- up to 25x speedup, even on one core, from not having to copy data!
- 7-8x parallel speedup
- Lock-free structures help

**Parallel Speedup**

<table>
<thead>
<tr>
<th>Speedup over one processor</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear speedup</td>
<td>0</td>
</tr>
<tr>
<td>blur (lock-free)</td>
<td>2</td>
</tr>
<tr>
<td>blur (pure)</td>
<td>4</td>
</tr>
<tr>
<td>notChain (lock-free)</td>
<td>6</td>
</tr>
<tr>
<td>notChain (pure)</td>
<td>8</td>
</tr>
</tbody>
</table>

See also: Phylogenetic tree binning parallelized with LVish [PLDI ’14]
Distributed systems
Eventual consistency.

“if updates stop arriving, replicas will eventually agree”
Eventual consistency...but how?

“if updates stop arriving, replicas will eventually agree”
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 data centers 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 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
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.
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.
Convergent replicated data types (CvRDTs) [Shapiro et al., 2011]

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.
Our contribution: deterministic threshold queries of CvRDTs
LVars and LVish across the landscape:

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

\[ f (\text{g}(\text{x})) \]

\[ (\text{h}(\text{y})) \]
LVars and LVish across the landscape:

Kahn process networks

imperative disjoint

single-assignment

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)
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)

G H

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

- Kahn process networks
- single-assignment
- imperative disjoint
- λ-calculus
- quasi-det.

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

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

f (g x) (h y)

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

Kahn process networks

single-assignment

imperative disjoint

λ-calculus

f (g x) (h y)

g(left) h(right)

Thank you!
Twitter: @lindsey
LVars project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples
Papers: cs.indiana.edu/~lkuper
Research blog: composition.al