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

Lindsey Kuper
Indiana University

Microsoft Research Silicon Valley
January 27, 2014
LVars:
Lattice-based Data Structures
for Deterministic Parallel
and Distributed Programming

Lindsey Kuper
Indiana University

Microsoft Research Silicon Valley
January 27, 2014
LVars:
Lattice-based Data Structures
for Deterministic Parallel
and Distributed Programming

Lindsey Kuper
Indiana University

Microsoft Research Silicon Valley
January 27, 2014
Parallel systems

Distributed systems
Deterministic Parallel Programming
Deterministic Parallel Programming
<table>
<thead>
<tr>
<th>Item</th>
<th>Book</th>
<th>Shoes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
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, ())))
```haskell
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: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
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

```
[(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, ()�))地段)
     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
  putStrLn (show (toList v))
\[
p :: \text{IO (Map Item Int)}
\]
\[
p = \textbf{do}
\]
\[
cart \leftarrow \text{newIORef empty}
\]
\[
a1 \leftarrow \text{async (atomicModifyIORef cart (m \mapsto (insert Book 1 m, ()))))}
\]
\[
a2 \leftarrow \text{async (atomicModifyIORef cart (m \mapsto (insert Shoes 1 m, ()))))}
\]
\[
\text{res} \leftarrow \text{async (do waitBoth a1 a2 readIORef cart)}
\]
\[
\text{wait res}
\]
\[
\text{main} = \textbf{do} \ v \leftarrow p
\]
\[
\quad \text{putStr (show (toList v))}
\]

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
          putStrLn (show (toList v))
Deterministic...now...we hope

Deterministic by construction

[Kuper and Newton, FHPC ’13]
[Kuper et al., POPL ’14]
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
\textbf{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

[Kuper and Newton, FHPF ’13]
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
[Kuper and Newton, FHP’13]

* actually a bounded join-semilattice
Raise 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)
```
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
\textbf{cart}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $1 \rightarrow 2$
  \item $2 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item \ldots
\end{itemize}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item \ldots
\end{itemize}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item \ldots
\end{itemize}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item \ldots
\end{itemize}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
  \item \ldots
\end{itemize}

\begin{itemize}
  \item $1 \rightarrow 1$
  \item $2 \rightarrow 2$
\end{itemize}

\textbf{data Item =} Book \mid Shoes \mid \ldots

\textbf{p = do}
\begin{align*}
\text{cart } & \leftarrow \text{newEmptyMap} \\
\text{fork (insert Shoes 1 cart)} \\
\text{fork (insert Book 2 cart)} \\
\text{getKey Book cart} \quad \text{-- returns } 2
\end{align*}
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 | ...

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

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

\[
\begin{align*}
\text{data Item} &= \text{Book} \mid \text{Shoes} \mid \ldots \\
\end{align*}
\]

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

0
seen nodes

0 1 6
7
seen nodes

0 1 3
4 5 6
7 9 10
11
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
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.

![Diagram with labeled nodes: 0, 1, 4, 6, 7, 9, 10, 11, 2. Nodes 0, 4, 6, 7, 9, 10, 11 are marked as "already seen." The diagram shows a traversal function `traverse g startNode = do`.](#)
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
```
Events are updates that change an LVar's state
Event handlers listen for events and launch callbacks in response

\[
\text{traverse } g \text{ startNode } = \text{ do }
\]
\[
\text{seen } \leftarrow \text{newEmptySet}
\]
\[
\text{h } \leftarrow \text{newHandler seen } \left( \text{\node } \rightarrow \text{ do }
\]
\[
\text{mapM } (\text{\node } \rightarrow \text{ insert } \text{\node } \text{seen})
\]
\[
\text{(neighbors } g \text{ \node})
\]
\[
\text{return } ()
\]
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 ()
    )
  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.

```plaintext
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
...
```
freeze: exact non-blocking read

```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

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

\[ \text{[Kuper et al., POPL '14]} \]

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

insert v seen
(inserting 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 \( \sigma'' = \text{error} \).

[Kuper et al., 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 $\sigma'' = \text{error}$.

[Kuper et al., 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 $\sigma'' = \text{error}$.

[Kuper et al., 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 \( \sigma'' = \text{error} \).

[Kuper et al., POPL ’14]

---

```haskell
insert v seen

return ()

insert startNode seen

quiesce h

freeze seen
```
Quasi-Determinism

\[ \sigma = \sigma' = \sigma'' \]
Quasi-Determinism

Strong Local Quasi-Confluence

\[ \sigma \]

\[ \sigma \]

\[ \sigma_a \quad \sigma_b \quad \sigma_c \]

\[ \leq 1 \]

\[ \leq 1 \]

\[ \sigma' = \sigma'' \]
Independence

\[
\langle S; e \rangle \leftrightarrow \langle S'; e' \rangle \\
\langle S \sqcup S' S''; e \rangle \leftrightarrow \langle S' \sqcup S S''; e' \rangle
\]

Strong Local Quasi-Confluence

Quasi-Determinism

\[
\sigma \\
\downarrow^* \\
\sigma' = \sigma''
\]
Frame rule

\[
\begin{align*}
\{p\} & \quad c \quad \{q\} \\
\{p \ast r\} & \quad c \quad \{q \ast r\}
\end{align*}
\]

[O’Hearn et al., 2001]

Independence

\[
\begin{align*}
\langle S; e \rangle & \quad \leftrightarrow \quad \langle S'; e' \rangle \\
\langle S \sqcup_S S''; e \rangle & \quad \leftrightarrow \quad \langle S' \sqcup_S S''; e' \rangle
\end{align*}
\]
Frame rule

\[
\begin{align*}
\{p\} & \quad c \quad \{q\} \\
\{p \ast r\} & \quad c \quad \{q \ast r\}
\end{align*}
\]

[O’Hearn et al., 2001]

Independence

\[
\begin{align*}
\langle S; e \rangle & \quad \leftarrow \quad \langle S'; e' \rangle \\
\langle S \sqcup_S S''; e \rangle & \quad \leftarrow \quad \langle S' \sqcup_S S''; e' \rangle
\end{align*}
\]
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-\texttt{freeze} exception
Two possible outcomes: either the same final value or an exception

\begin{verbatim}
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
\end{verbatim}
**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:
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

Let the system handle this for us:
runParThenFreeze

```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
```
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 Par computations
Lightweight threads
LVish
a Haskell library for parallel programming with LVars

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

```
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 \texttt{Par} computations
Lightweight threads
\texttt{Par} computations indexed by \textit{effect level}
\texttt{runParThenFreeze} captures the freeze-after-writing idiom

\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

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

main = do
    putStrLn (show (toList (fromIMap (runParThenFreeze p))))
```
LVish
a Haskell library for parallel programming with LVars

LVars 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.

\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 = do
putStr (show (toList (fromIMap (runParThenFreeze p)))))
\end{verbatim}
LVish
a Haskell library for parallel programming with LVars

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

\begin{verbatim}
p :: \texttt{Par Det (IMap Item Int)}
p = \texttt{do}
  cart <- \texttt{newEmptyMap}
  fork (\texttt{insert Book 1 cart})
  fork (\texttt{insert Shoes 1 cart})
  return cart

main = \texttt{do}
  putStrLn (\texttt{show (toList (fromIMap (runParThenFreeze p))))}
\end{verbatim}
L Vish
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} captures the freeze-after-writing idiom
Efficient lock-free sets, maps, etc.
Implement your own LVars, too
\texttt{cabal install lvish} today!

\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 = do
putStr (show (toList (fromIMap (runParThenFreeze p))))
\end{verbatim}
Deterministic Parallel Programming
(observably)
Deterministic Parallel Programming
Deterministic Parallel Programming

(observably)  (irregular)
Case study:
k-CFA static analysis parallelized with LVish
Case study: k-CFA static analysis parallelized with LVish

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

Parallel Speedup

- linear speedup
- blur/lockfree
- notChain/lockfree
- blur
- notChain

Speedup over one processor

Processors
Parallel systems

Distributed systems
Distributed systems
getKey Book
getKey Book
getKey Book
getKey Book
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 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...
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

To meet the flexible and scaling needs, Amazon has developed one of which the Amazon Simple

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

Marc Shapiro1,5, Nuno Preguiça1,5, Carlos Baquerô3, and Marek Zagirski1,4
1 INRIA, Paris, France
2 CITI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 UPMC, 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 [3].

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 replicas.

Dynamo: Amazon's Highly Available Key-value Store

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

Amazon.com

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 performance and scaling needs, Amazon has developed Dynamo, another highly available key-value store built for Amazon’s platform. Amazon is aware of the data schema it exposes, and is designed such that the application that maintains the data which is close to “merge” the conflicting updates into a single storage layer.

To meet the performance and scaling needs, Amazon has developed Dynamo, another highly available key-value store built for Amazon’s platform. Amazon is aware of the data schema it exposes, and is designed such that the application that maintains the data which is close to “merge” the conflicting updates into a single storage layer.

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.

[Shapiro et al., SSS '11]
[DeCandia et al., SOSP '07]
Conflict-Free Replicated Data Types*

Marc Shapiro1,5, Nuno Preguiça1,2, Carlos Baquero3, and Marek Zawirski1,4
1 INRIA, Paris, France
2 CNTI, Universidade Nova de Lisboa, Portugal
3 Universidade do Minho, Portugal
4 LPMC, 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 detail 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 [13]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [3].

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 [1721]. 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 Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall 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 it’s 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 add items to their shopping cart even if disks are down, network routes are flapping, or data centers are being flooded by tomdromes. Therefore, the service responsible for storing shopping cart 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.

To meet the reliability and scaling needs, Amazon has developed Dynamo, another highly available key-value store built inside of Amazon and known as Amazon SDB. This paper presents the Dynamo architecture, which is a highly distributed database that has been engineered for high availability and scalability. Dynamo is aware of the data schema it stores. It provides a method that is best suited for each type of data, the application that maintains consistency. In particular, it allows for “merge” the conflicting updates from different 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]
Threshold reads of CRDTs
Consistency choices at the granularity of queries, not that of databases
Threshold reads of CRDTs
Consistency choices at the granularity of *queries*, not that of databases

General inflationary LVar updates
Non-idempotent, incrementable counters
Threshold reads of CRDTs
Consistency choices at the granularity of queries, not that of databases

General inflationary LVar updates
Non-idempotent, incrementable counters
Non-monotonic LVar updates
Encode using CRDT tombstones
Threshold reads of CRDTs
Consistency choices at the granularity of queries, not that of databases

General inflationary LVar updates
Non-idempotent, incrementable counters
Non-monotonic LVar updates
Encode using CRDT tombstones

Distributed LVish
Distributed work-stealing, distributed GC
Threshold reads of CRDTs
Consistency choices at the granularity of queries, not that of databases

General inflationary LVar updates
Non-idempotent, incrementable counters

Non-monotonic LVar updates
Encode using CRDT tombstones

Distributed LVish
Distributed work-stealing, distributed GC

Differential dataflow for LVars
Handling continuous input
Threshold reads of CRDTs
Consistency choices at the granularity of queries, not that of databases

General inflationary LVar updates
Non-idempotent, incrementable counters
Non-monotonic LVar updates
Encode using CRDT tombstones
Distributed work-stealing, distributed GC
Differential dataflow for LVars
Handling continuous input

Abstract

EVars—shared, infinite locations whose semantics are defined in terms of an application-specific lattice—offer a principled approach to deterministic-by-construction, shared-state parallel programming: write to an LVar takes the join 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. This semantics guarantees that programs have a deterministic outcome, despite parallel execution and schedule nondeterminism, while allowing non-monotonic LVar updates. Like convergent replicated data types (CvRDTs), which leverage lattice properties to ensure that all replicas of a distributed object (for instance, a distributed database) are eventually consistent. Unlike LVars, in which all updates are joins, CvRDTs allow updates that are inflationary with respect to the lattice but do not compute a join. Moreover, CvRDTs differ from LVars in that they allow monotonically increasing writes, so that the value of an LVar is never decreased even via remote updates. Encoded using CRDT tombstones, the semantics of LVars ensure determinism under parallel execution by leveraging the lattice properties that CvRDTs use to ensure eventual consistency. Therefore, a sensible next research question is—how can we take inspiration from CvRDTs to improve the LVars model?

Joining Forces

Toward a Unified Account of LVars and Convergent Replicated Data Types

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

1. Introduction

One of the challenges facing parallel-programming models is that all programs written using the model have the same observability behavior every time they are run, offering freedom from subtle, hard-to-reproduce nondeterministic bugs in parallel code. Ideally, a deterministic-by-construction parallel program will run faster where more parallel resources are available, and we do not want our models to require that exact scheduling behavior is deterministic. As we discussed in Section 3.1, LVars are a distributed system for shared state. Because they update the LVar to the minimum of the two replicas' contents for some other operation blocks until the LVar in question reaches a unique value in the threshold set, this solution and counter example that we can use to extend CRDTs with LVars-style threshold reads and how to extend LVars with CvRDT-style inflationary updates, and we advocate for the introduction of these extensions.

1.1. Convergence

In previous work [9, 10], we proposed EVars as a principled approach to shared-state parallel programming that guarantees observably deterministic outcomes. The EVar is a memory location that can be shared among multiple threads and accessed through the EVar interface. Like an LVar, an EVar has a minimum value and a threshold set, and an EVar's semantics depend on the minimum value in the threshold set, then unblocks and returns. Unlike an LVar, an EVar cannot take on or change elements of an application-specific lattice. This application-specific lattice determines the semantics of the put and get operations that constitute the interface to LVars.

• put operations can only change an EVar's state in a way that is monotonically increasing with respect to the application-specific lattice, because it updates the EVar to the join of the old and the new state.

• get operations allow only limited observations of the state of an EVar. A get operation requires the programmer to specify a threshold set of minimum values that can be read from the EVar, where every two elements in the threshold set must have a least upper bound (lub) in the lattice, but such a “last-write-wins” policy does not necessarily make sense in the context of a shared state. Indeed, we want our model to require that exact scheduling behavior is deterministic, and so we require that the EVar's exact contents.

• Operational semantics to put and threshold reads via put and threshold reads yield a deterministic-by-construction programming model. This is a program in which reads and puts on LVars are the only side effects will have the same observable result on every run, in spite of parallel execution and schedule nondeterminism.

Lattices for eventual consistency

The problem of ensuring deterministic parallel programs is closely related to the problem of ensuring the eventual consistency [13, 14] of replicated objects in a distributed system. Consider, for example, an object representing the contents of a shopping cart, replicated across a number of physical locations. If two replicas disagree on the contents of the cart—for instance, if one replica sees only that item a has been added to the cart, while another sees only item b—then we know what the “real” cart contents are? One option is to give every write a timestamp and allow the last-written replica to override the others, but such a “last-write-wins” policy also breaks the semantics of minimum values that can be read from the LVar, for some other operation blocks until the LVar in question reaches a unique value in the threshold set, and then unblocks and returns.

Lattices for eventual consistency

In the particular case of the shopping cart, replicated across a number of physical locations, which the states that replicas can take on can be viewed as elements of a join-semilattice. While at any given time, replicas may differ, as long as all replicas merge with one another periodically, eventual consistency is guaranteed.

Joining Forces

Although LVars and CRDTs were developed independently, both models leverage the mathematical properties of join-semilattices to ensure that a property of the model holds—

2014/1/10
Thank you!

Email: lkuper@cs.indiana.edu
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
Can’t see the exact, complete contents of the cart
Can’t see the exact, complete contents of the cart
Can’t iterate over the items in the cart
- Can’t see the exact, complete contents of the cart
- Can’t iterate over the items in the cart
- Can't determine if an item isn’t in the cart
Can’t see the exact, complete contents of the cart
Can’t iterate over the items in the cart
Can't determine if an item isn’t in the cart
Can’t react to writes that we weren’t expecting
Can see the exact, complete contents of the cart
Can iterate over the items in the cart
Can determine if an item isn’t in the cart
Can react to writes that we weren’t expecting
Can see the exact, complete contents of the cart
Can iterate over the items in the cart
Can determine if an item isn’t in the cart
Can react to writes that we weren’t expecting

handlers, quiescence, freezing