Freeze After Writing:
Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper
Aaron Turon
Neelakantan R. Krishnaswami
Ryan R. Newton

POPL ’14, January 23, 2014
Freeze After Writing

- LVars
- Quasi-Deterministic Parallel Programming
data Item = Book | Shoes | ...
deriving (Show, Ord, Eq)
```haskell
data Item = Book | Shoes | ...
deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
```
data Item = Book | Shoes | ...
   deriving (Show, Ord, Eq)

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

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```haskell
data Item = Book | Shoes | ...
deriving (Show, Ord, Eq)

p :: IO (Map Item Int)
p = do cart <- newIORef empty
```
data Item = Book | Shoes | ...
deriving (Show, Ord, Eq)

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

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

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

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 | ...

deriving (Show, Ord, Eq)

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 | ...

deriving (Show, Ord, Eq)

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-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:liter-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 | ...

 deriving (Show, Ord, Eq)

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 | ...
  deriving (Show, Ord, Eq)

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 | ...
deriving (Show, Ord, Eq)

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 | ...
     deriving (Show, Ord, Eq)

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 | ...
  deriving (Show, Ord, Eq)

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
```haskell
data Item = Book | Shoes | ...
           deriving (Show, Ord, Eq)

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
```
Deterministic, but only because we put the \texttt{waits} in the right places.
Deterministic, but only because we put the **waits** in the right places
Freeze After Writing

하겠습니다.

---

Quasi-Deterministic Parallel Programming
data Item = Book | Shoes | ...
  deriving (Show, Ord, Eq)

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 | ...
  deriving (Show, Ord, Eq)

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
```haskell
data Item = Book | Shoes | ... 
  deriving (Show, Ord, Eq)

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 | ... 

deriving (Show, Ord, Eq)

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]
\[
data \mathrm{Item} = \mathrm{Book} \mid \mathrm{Shoes} \mid \ldots\]
\[
deriving \; (\mathrm{Show}, \; \mathrm{Ord}, \; \mathrm{Eq})\]
\[
p :: IO \; (\mathrm{Map} \; \mathrm{Item} \; \mathrm{Int})\]
\[
p = \begin{align*}
do & \text{cart} \leftarrow \text{newIORef empty} \\
\text{async} & \$ \; \text{atomicModifyIORef cart} \\
& (\{m \mapsto (\text{insert Book 1 m, ()})\}) \\
\text{async} & \$ \; \text{atomicModifyIORef cart} \\
& (\{m \mapsto (\text{insert Shoes 1 m, ()})\}) \\
res & \leftarrow \text{async} \; \$ \; \text{readIORef cart} \\
\text{wait} & \; \text{res}
\end{align*}\]

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

\textit{LVars: multiple least-upper-bound writes, blocking threshold reads}
data Item = Book | Shoes | ...
deriving (Show, Ord, Eq)

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

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

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

```
do
  fork (put num 4)
  fork (put num 4)
```
data Item = Book | Shoes

deriving (Show, Ord, Eq)

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

deriving (Show, Ord, Eq)

\[ p = \text{do} \]
\[ \text{cart} \leftarrow \text{newEmptyMap} \]
\[ \text{fork} \$ \text{insert} \text{Shoes} 1 \text{cart} \]
\[ \text{fork} \$ \text{insert} \text{Book} 2 \text{cart} \]
\[ \text{getKey} \text{Book} \text{cart} \quad \text{-- returns 2} \]
data Item = Book | Shoes
deriving (Show, Ord, Eq)

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

 deriving (Show, Ord, Eq)

p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  getKey Book cart -- returns 2
data Item = Book | Shoes
  deriving (Show, Ord, Eq)

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

deriving (Show, Ord, Eq)

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

deriving (Show, Ord, Eq)

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

deriving (Show, Ord, Eq)

p = do
    cart <- newEmptyMap
    fork $ insert Shoes 1 cart
    fork $ insert Book 2 cart
    getKey Book cart -- returns 2
```haskell
import Data.Map (Map)
import Data.Ord (comparing)

data Item = Book | Shoes
    deriving (Show, Ord, Eq)

p = do
    cart <- newEmptyMap
    fork $ insert Shoes 1 cart
    fork $ insert Book 2 cart
    getKey Book cart -- returns 2
```

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

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

data Item = Book | Shoes
  deriving (Show, Ord, Eq)

\[
p = \text{do}
  \text{cart} \leftarrow \text{newEmptyMap}
  \text{fork} \ {\text{\$ insert Shoes 1 cart}}
  \text{fork} \ {\text{\$ insert Book 2 cart}}
  \text{getKey Book cart} \quad \text{-- returns 2}
\]
The threshold set must be pairwise incompatible.
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
Freeze After Writing

*: LVars

*: Quasi-Deterministic Parallel Programming
seen nodes
seen nodes

0  1  3
4  5  6
7  9  10
seen nodes

already seen

already seen

already seen
already seen

already seen

already seen

already seen

already seen

already seen

already seen

already seen

seen nodes

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

0  1  3
4  5  6
7  9  10
11
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

```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 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
  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

\[
\text{traverse } g \text{ startNode} = \text{do}
\]

\[
\text{seen} \leftarrow \text{newEmptySet}
\]

\[
h \leftarrow \text{newHandler seen} \quad (\node \rightarrow \text{do})
\]

\[
\text{mapM } (\node \rightarrow \text{insert } \node \text{ seen})
\]

\[
\text{(neighbors } g \text{ node})
\]

\[
\text{return } ()
\]

\[
\text{insert } \text{startNode seen}
\]

\[
\text{quiesce } h
\]

\[
\ldots
\]

...
**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

```
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 $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, 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 \( \sigma \xrightarrow{}^* \sigma'' \), do not execute 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 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 $\sigma'' = \text{error}$. 

---

```haskell
let (Book, 1), (Shoes, 1) 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 \( \sigma \xrightarrow{*} \sigma'' \), \( \sigma' \neq \sigma'' \), 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 \( \sigma'' = \text{error} \).

```haskell
define-LVish-language nat downset-op max natural

1. (Book, 1)
2. (Shoes, 1)
3. (Shoes, 1)

or error:
```

```haskell
insert v seen
insert startNode seen
quiesce h
freeze seen
```
LVish

a Haskell library for programming with LVars
LVish

a Haskell library for programming with LVars

LVar operations run inside a Par monad
LVish
a Haskell library for programming with LVars
LVar operations run inside a Par monad
Library-level threads
LVish
a Haskell library for programming with LVars

LVars operations run inside a `Par` monad

Library-level threads
`Par` computations indexed by effect level

```haskell
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart
```
LVish
a Haskell library for programming with LVars

LVar operations run inside a \texttt{Par} monad
Library-level threads
\texttt{Par} computations indexed by effect level
\texttt{runParThenFreeze} captures the freeze-after-writing idiom

\begin{verbatim}
   p :: Par Det s (IMap Item s Int)
   p = do
     cart <- newEmptyMap
     fork $ insert Shoes 1 cart
     fork $ insert Book 2 cart
     return cart
\end{verbatim}
LVish
a Haskell library for programming with LVars

LVars operations run inside a Par monad
Library-level threads
Par computations indexed by effect level
runParThenFreeze captures the freeze-after-writing idiom

```
p :: Par Det s (IMap Item s Int)
p = do
    cart <- newEmptyMap
    fork $ insert Shoes 1 cart
    fork $ insert Book 2 cart
    return cart

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

LVar operations run inside a Par monad
Library-level threads
Par computations indexed by effect level
runParThenFreeze captures the freeze-after-writing idiom
LVar data structures: sets, maps, etc.

```haskell
p :: Par Det s (IMap Item s Int)
p = do
cart <- newEmptyMap
fork $ insert Shoes 1 cart
fork $ insert Book 2 cart
return cart

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

LVars operations run inside a `Par` monad

Library-level threads

Par computations indexed by effect level

runParThenFreeze captures the freeze-after-writing idiom

LVars data structures: sets, maps, etc.

Implement your own LVars, too

```
p :: Par Det s (IMap Item s Int)
p = do
cart <- newEmptyMap
fork $ insert Shoes 1 cart
fork $ insert Book 2 cart
return cart

main = do
putStr $ show $ toList $ fromIMap $ runParThenFreeze p
```
LVish
a Haskell library for programming with LVars

LVar operations run inside a Par monad
Library-level threads
Par computations indexed by effect level
runParThenFreeze captures the freeze-after-writing idiom
LVar data structures: sets, maps, etc.
Implement your own LVars, too

cabal install lvish today!

```
import Data.Map.Strict (Map)
import Data.Set (Set)
import Data.Ord (comparing)

class LVar a where
  (<=<) :: Par Det s (a -> a -> a -> a)

instance LVar (Pair a a) where
  (<=<) = (<=<) (Pair (<=<))

instance LVar a => LVar (Pair a b) where
  (<=<) = (<=<) (Pair (<=<) id)

instance LVar a => LVar (Map a b) where
  (<=<) = (<=<) (Map (<=<)

instance LVar a => LVar (Set a) where
  (<=<) = (<=<) (Set (<=<)

runParThenFreeze :: Par Det s (a -> a -> a -> a) -> Par Det s a
runParThenFreeze p = do
  cart <- newEmptyMap
  insert shoes 1 cart
  insert book 2 cart
  return cart
```

```
p :: Par Det s (IMap Item s Int)
p = do
  cart <- newEmptyMap
  fork $ insert Shoes 1 cart
  fork $ insert Book 2 cart
  return cart

main = do
  putStrLn $ show $ toList $ fromIMap $ runParThenFreeze p
```
More in the paper and TR
More in the paper and TR
The LVish calculus and its quasi-determinism proof
More in the paper and TR
The LVish calculus and its quasi-determinism proof
Gory details of the LVish scheduler implementation
More in the paper and TR
The LVish calculus and its quasi-determinism proof
Gory details of the LVish scheduler implementation
Case study: parallel $k$-CFA written with the LVish library
More in the paper and TR
The LVish calculus and its quasi-determinism proof
Gory details of the LVish scheduler implementation
Case study: parallel k-CFA written with the LVish library

Parallel Speedup

<table>
<thead>
<tr>
<th>Threads</th>
<th>linear speedup</th>
<th>blur/lockfree</th>
<th>blur</th>
<th>notChain/lockfree</th>
<th>notChain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank you!

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