

Exploiting the Commutativity Lattice

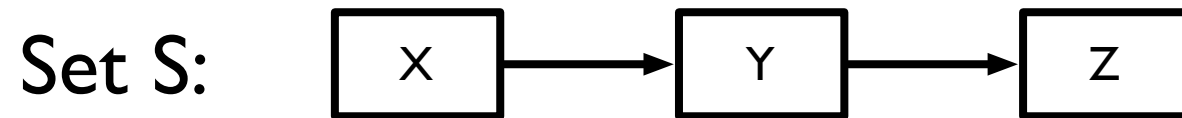
Milind Kulkarni

Donald Nguyen, Dimitrios
Prountzos, Xin Sui and
Keshav Pingali

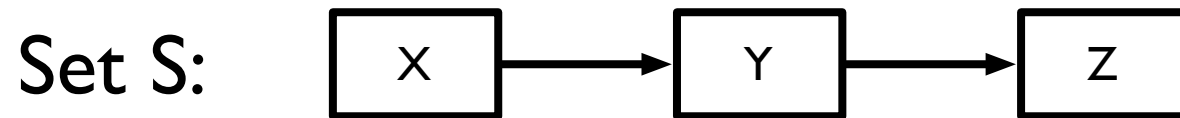
PURDUE
UNIVERSITY

THE UNIVERSITY OF
TEXAS
AT AUSTIN

Exploiting semantics in transactional execution



Exploiting semantics in transactional execution



```
atomic {  
  ...  
  S.add(a)  
  ...  
}
```

```
atomic {  
  ...  
  S.contains(b)  
  ...  
}
```

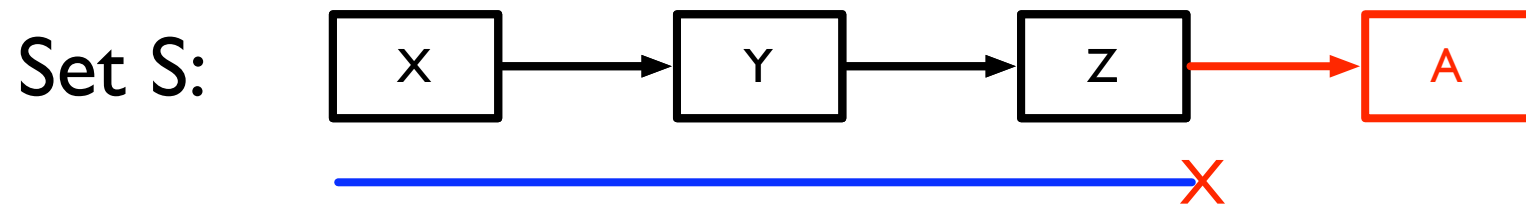
Exploiting semantics in transactional execution



```
atomic {  
  ...  
  S.add(a)  
  ...  
}
```

```
atomic {  
  ...  
  S.contains(b)  
  ...  
}
```

Exploiting semantics in transactional execution



```
atomic {  
  ...  
  S.add(a)  
  ...  
}
```

```
atomic {  
  ...  
  S.contains(b)  
  ...  
}
```

Exploiting semantics in transactional execution





```
atomic {  
  ...  
  S.add(a)  
  ...  
}
```

```
atomic {  
  ...  
  S.contains(b)  
  ...  
}
```

Exploiting semantics in transactional execution





```
atomic {  
  ...  
   S.add(a)  
  ...  
}
```

```
atomic {  
  ...  
   S.contains(b)  
  ...  
}
```

Exploiting semantics in transactional execution




```
atomic {  
  ...  
   S.add(a)  
  ...  
}
```


```
atomic {  
  ...  
   S.contains(b)  
  ...  
}
```


Exploiting semantics in transactional execution

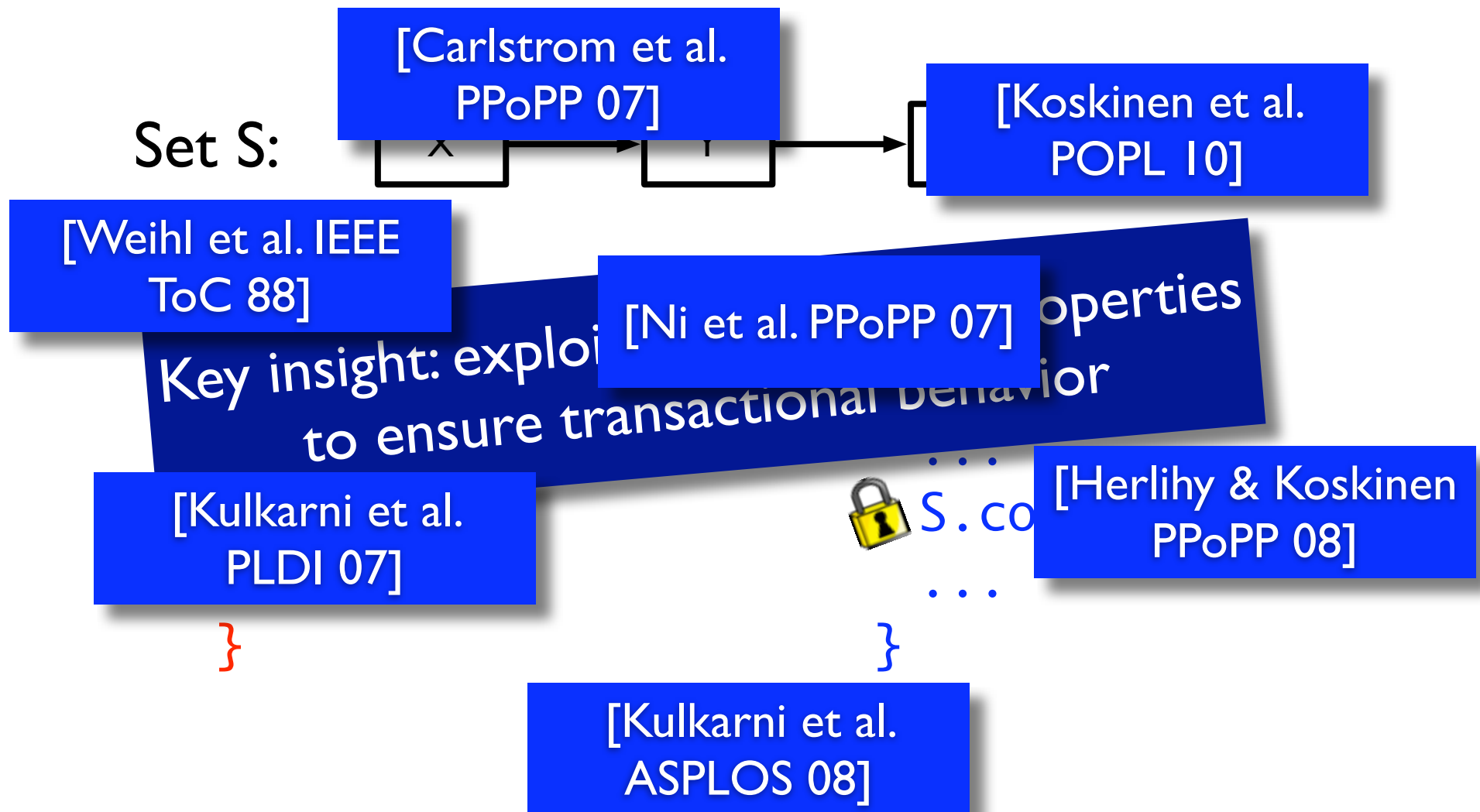


Key insight: exploit commutativity properties to ensure transactional behavior

 `S.add(a)`
...
}

 `S.contains(b)`
...
}

Exploiting semantics in transactional execution



How do we check commutativity?

- Can specify conditions for commutativity:

$add(a)/r$ commutes with $contains(b)/r$ if
 $a \neq b$

- How should a transactional run-time system check these?
- Prior work: *ad hoc* combinations of logging and locking

How do we check commutativity?

$\text{add}(a)/r$ commutes with $\text{contains}(b)/r$ if
 $a \neq b$

How do we check commutativity?

`add(a)/r` commutes with `contains(b)/r` if
`a` \neq `b` or `r` = **false**

How do we check commutativity?

- Commutativity can be more complex:

`add(a)/r` commutes with `contains(b)/r` if
`a` \neq `b` or `r` = **false**

- Prior work often did not fully check commutativity to reduce overhead
- How do we know this is correct?

Contributions

- Define a *commutativity lattice* for reasoning about commutativity specifications
- **How do we check commutativity?**
 - Provide *systematic approaches* for implementing commutativity checks
- **How do we implement low overhead checks?**
 - Show how to use commutativity lattice to correctly construct lower-overhead checkers

Commutativity

σ_1
|

Commutativity

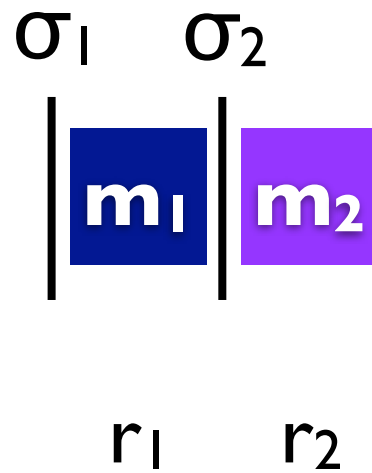
$$\sigma_1 \left| \begin{array}{c} m_1 \end{array} \right\rangle$$

r_1

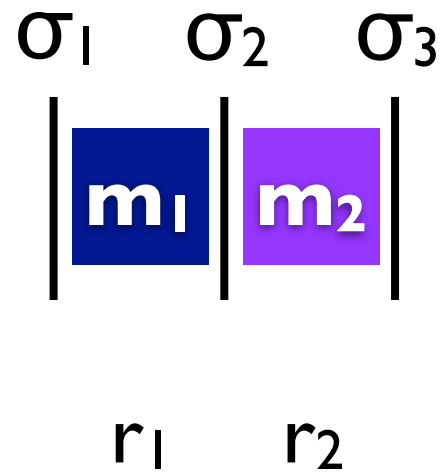
Commutativity

$$\begin{array}{cc} \sigma_1 & \sigma_2 \\ \left| \begin{array}{c} \mathbf{m}_1 \end{array} \right| \\ r_1 \end{array}$$

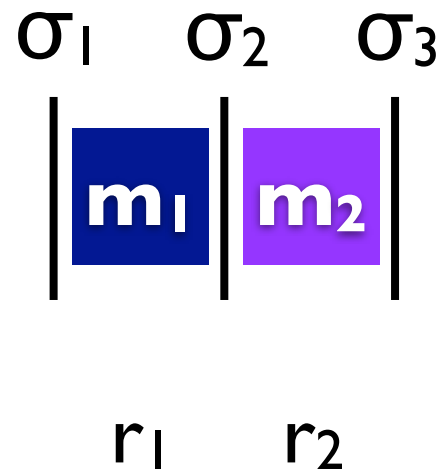
Commutativity



Commutativity

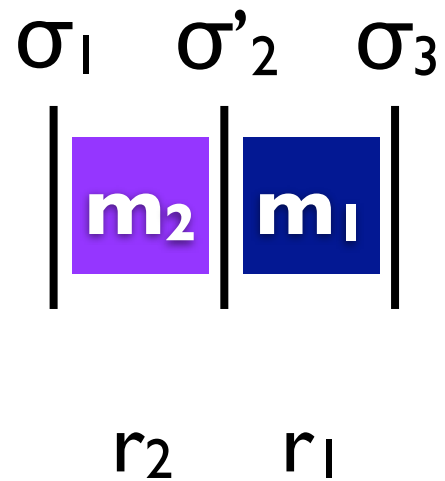


Commutativity



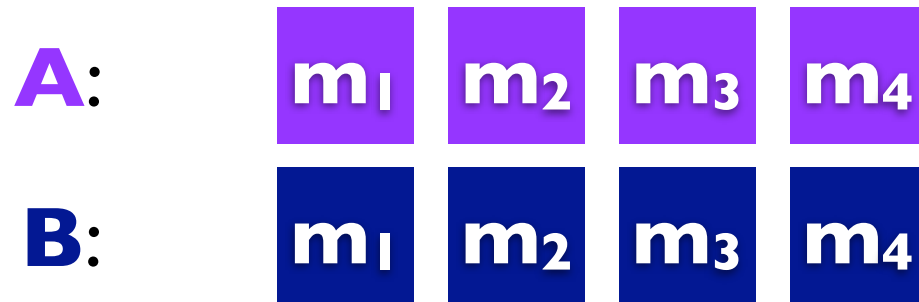
m_1, m_2 commute in σ_1

Commutativity



m_1, m_2 commute in σ_1

Using commutativity to guarantee serializability

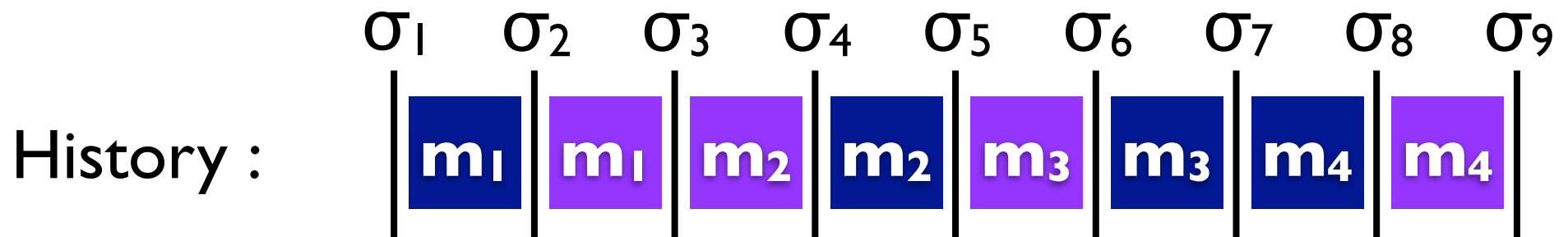


Using commutativity to guarantee serializability

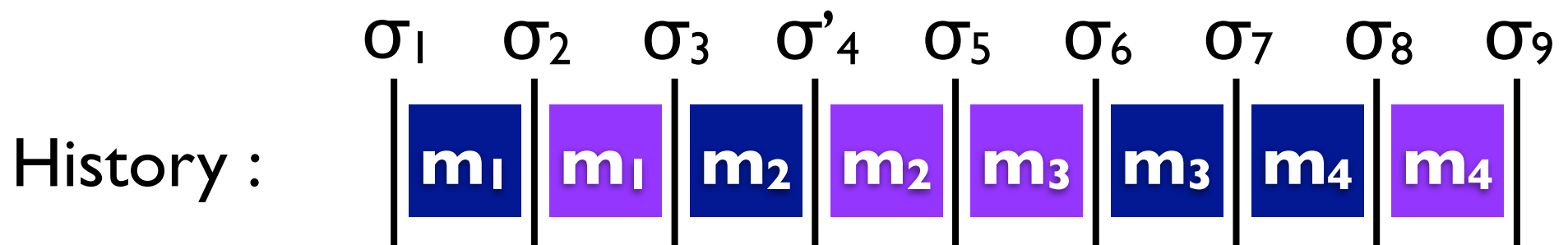
History :



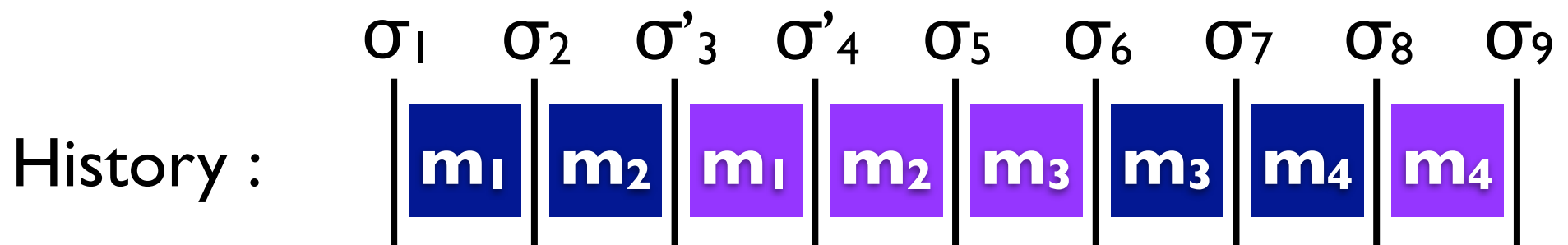
Using commutativity to guarantee serializability



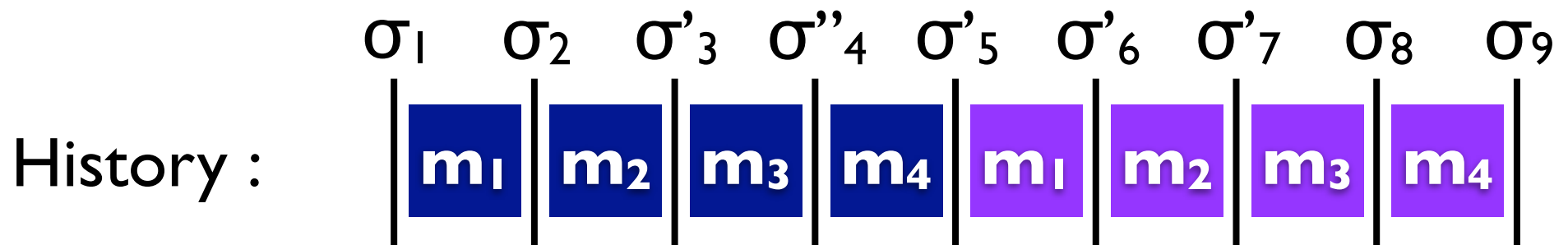
Using commutativity to guarantee serializability



Using commutativity to guarantee serializability



Using commutativity to guarantee serializability



Runtime commutativity checks

- For each method invocation by transaction B
 - Runtime checks commutativity with all methods invoked by transaction A
 - If all checks succeed, execution continues
 - If *any* commutativity check fails, one transaction rolled back

Commutativity conditions

Commutativity condition: $\varphi(m_a, m_b)$

Commutativity conditions

Commutativity condition: $\varphi(m_a, m_b)$

true only if m_a and m_b commute

Commutativity conditions

Commutativity condition: $\varphi(\mathbf{m}_a, \mathbf{m}_b)$

Precise condition: $\varphi^*(\mathbf{m}_a, \mathbf{m}_b)$

Commutativity conditions

Commutativity condition: $\varphi(m_a, m_b)$

Precise condition: $\varphi^*(m_a, m_b)$

true if and only if m_a and m_b commute

Commutativity conditions

Commutativity condition: $\varphi(\mathbf{m}_a, \mathbf{m}_b)$

Precise condition: $\varphi^*(\mathbf{m}_a, \mathbf{m}_b)$

$\varphi^*(\text{add}(a)/r_1, \text{contains}(b)/r_2)$

||

$a \neq b$ or $r_1 = \mathbf{false}$

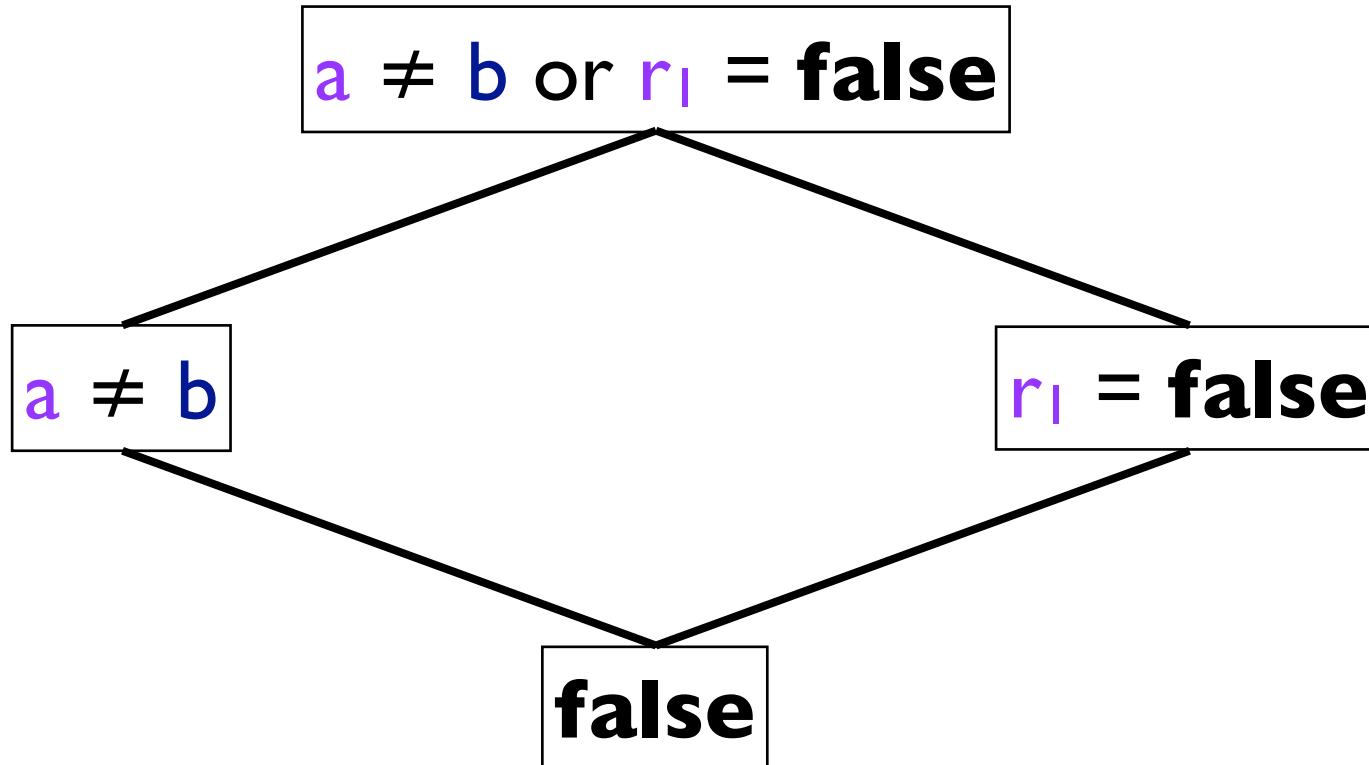
Commutativity lattice

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$

$a \neq b$ or $r_1 = \mathbf{false}$

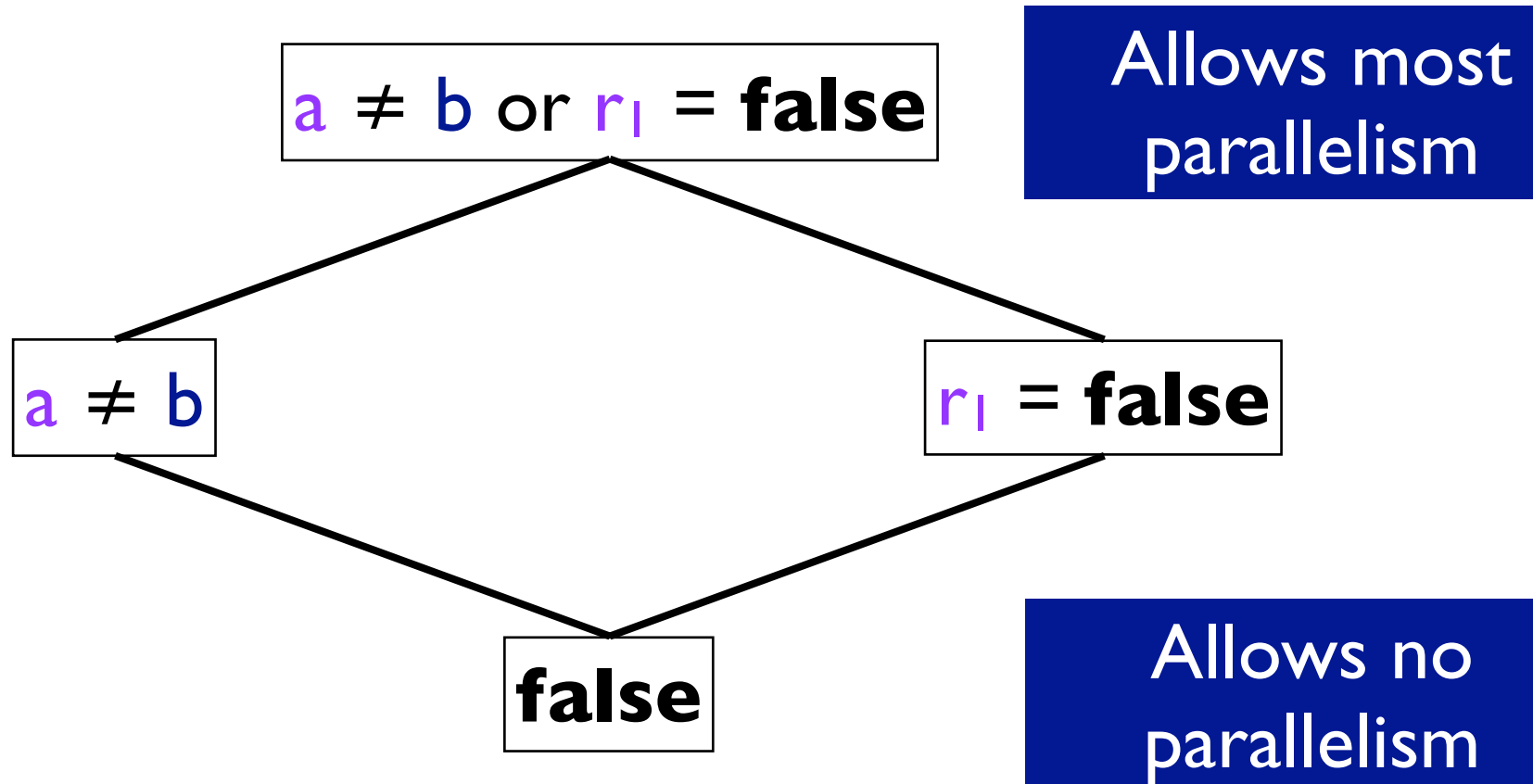
Commutativity lattice

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$



Commutativity lattice

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$



Implementing Commutativity

Soundness and completeness

- A conflict detection implementation is *sound* if it claims methods commute *only if* they actually do according to the conditions
- A conflict detection implementation is *complete* if it claims methods commute *if* they do according to the conditions

Running example

- Set-like data structure
 - Supports *add* and *contains*

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$

$a \neq b$ or $r_1 = \mathbf{false}$

$(\text{add}(a)/r_1, \text{add}(b)/r_2)$

$a \neq b$ or $(r_1 = \mathbf{false}$ and $r_2 = \mathbf{false})$

$(\text{contains}(a)/r_1, \text{contains}(b)/r_2)$

\mathbf{true}

Running example

- Set-like data structure
 - Supports *add* and *contains*

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$

$a \neq b$

$(\text{add}(a)/r_1, \text{add}(b)/r_2)$

$a \neq b$

$(\text{contains}(a)/r_1, \text{contains}(b)/r_2)$

true

Implementing commutativity

- Three schemes
 - Abstract locking
 - Forward gatekeeping
 - General gatekeeping

Abstract locking

- Sound and complete implementation when commutativity condition is *simple*
- Is either **true**, **false**, or a set of conjuncts of the form “ $x \neq y$ ”

$(\text{add}(a)/r_1, \text{contains}(b)/r_2)$

Not simple: $a \neq b$ or $r_1 = \text{false}$

Simple: $a \neq b$

Abstract locking

- Basic skeleton
 - Associate an *abstract lock* with each object that can be passed as an argument to a method
 - When a method is called, acquire locks on each argument in appropriate *mode*
 - Object already locked → commutativity violation
 - All locks released when transaction ends
- Key problem: building *compatibility matrix*

Building compatibility matrix

- One mode per argument of a method

add(a) → add:l

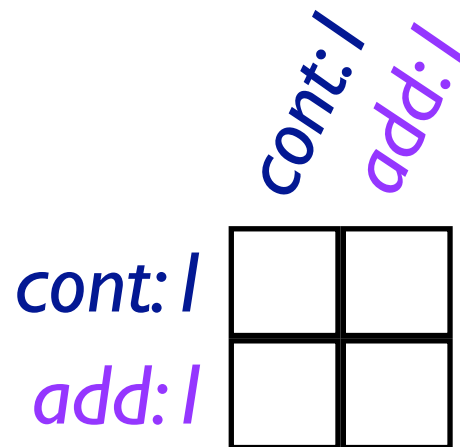
contains(b) → cont:l

Building compatibility matrix

- One mode per argument of a method

add(a) → add:l

contains(b) → cont:l



Building compatibility matrix

- Compatibility: If condition includes conjunct “a ≠ b” then modes for a and b incompatible

$\varphi(\text{add}(a)/r_1, \text{contains}(b)/r_2) :$

a ≠ b

cont:l
add:l

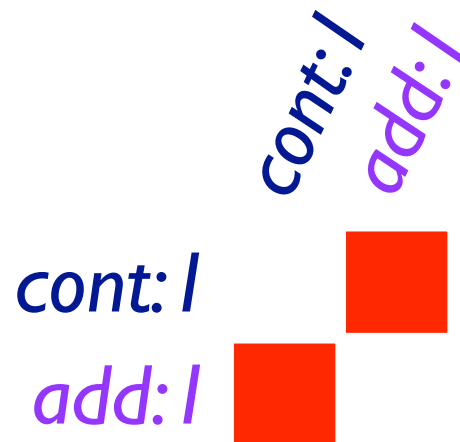
cont:l
add:l

Building compatibility matrix

- Compatibility: If condition includes conjunct “a ≠ b” then modes for a and b incompatible

$\varphi(\text{add}(a)/r_1, \text{contains}(b)/r_2)$:

a ≠ b

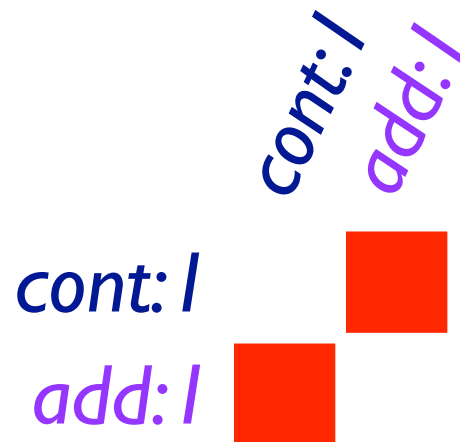


Building compatibility matrix

- Compatibility: modes for a and b incompatible if condition includes conjunct “a ≠ b”

$\varphi(\text{add}(a)/r_1, \text{add}(b)/r_2)$:

a ≠ b

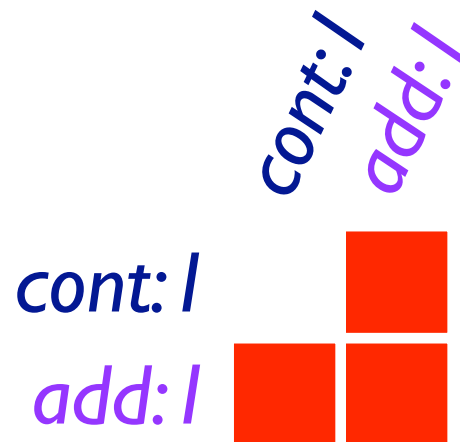


Building compatibility matrix

- Compatibility: modes for a and b incompatible if condition includes conjunct “a ≠ b”

$\varphi(\text{add}(a)/r_1, \text{add}(b)/r_2)$:

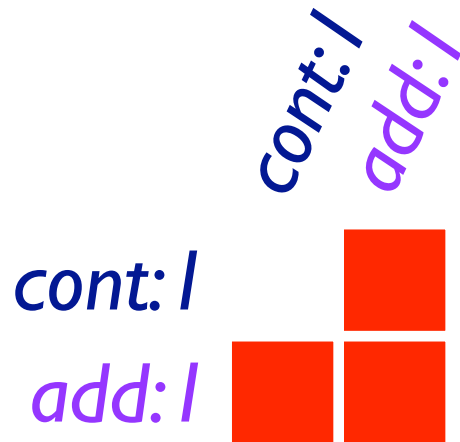
a ≠ b



Building the compatibility matrix

- Compatibility: modes for a and b incompatible if condition includes conjunct “a ≠ b”

$\varphi(\text{contains}(a)/r_1, \text{contains}(b)/r_2) : \mathbf{true}$



Building the compatibility matrix

- Compatibility: modes for a and b incompatible if condition includes conjunct “a ≠ b”

$\varphi(\text{contains}(a)/r_1, \text{contains}(b)/r_2) : \mathbf{true}$

	<i>cont:l</i>	<i>add:l</i>
<i>cont:l</i>	■	■
<i>add:l</i>	■	■

Other conflict detection techniques

- Forward gatekeeping: sound and complete for more complex conditions

$a \neq b$ or $r_1 = \mathbf{false}$

- General gatekeeping: allows most flexibility in commutativity conditions
- Basic tradeoff: Increasing complexity = more expressive, but more overhead

Trading off parallelism for overhead

Lowering overhead of conflict detection

- No prior work fully implemented commutativity for sets
- Used lower-overhead schemes instead

Set spec

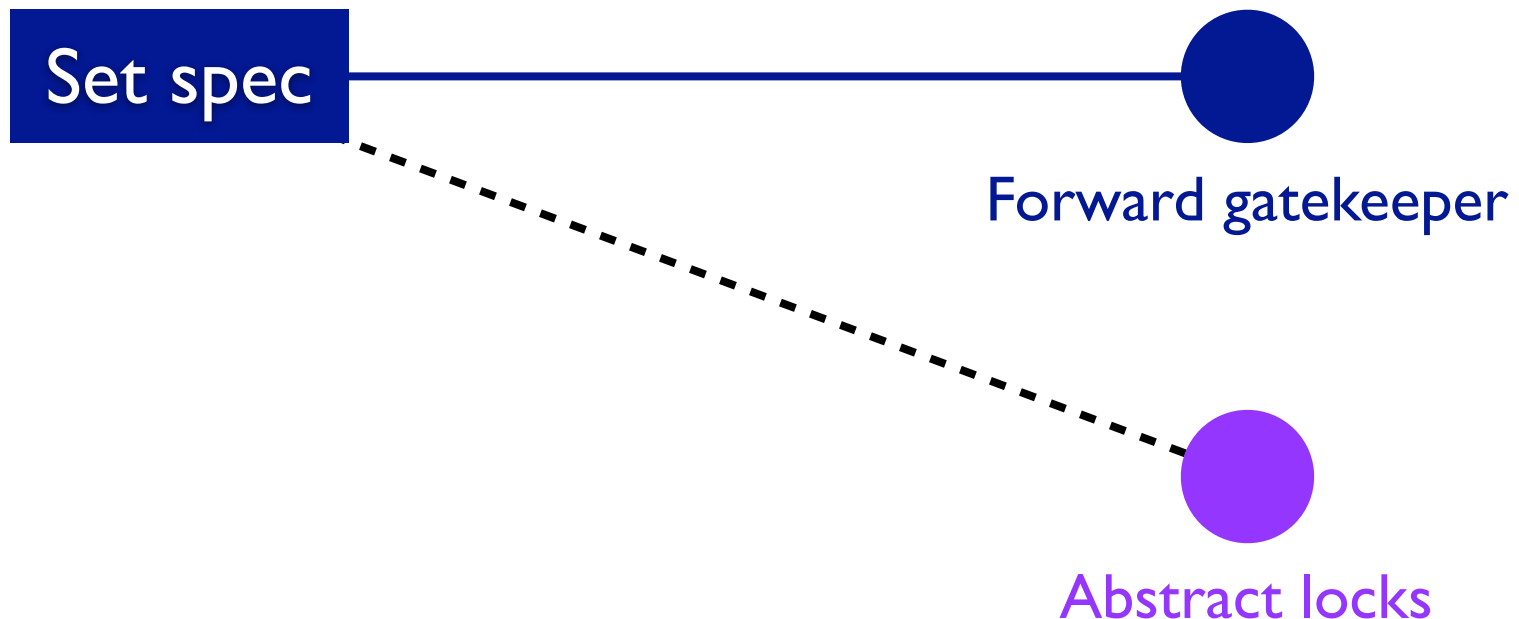
Lowering overhead of conflict detection

- No prior work fully implemented commutativity for sets
- Used lower-overhead schemes instead



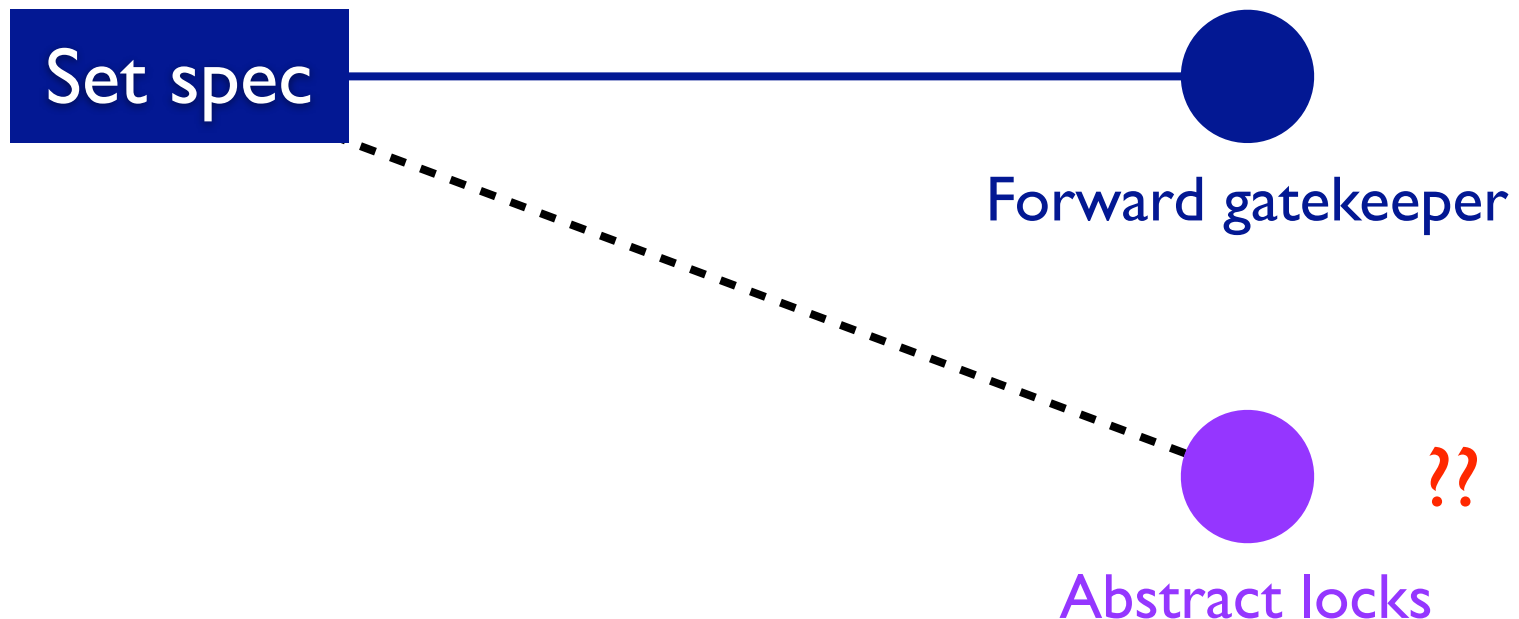
Lowering overhead of conflict detection

- No prior work fully implemented commutativity for sets
- Used lower-overhead schemes instead



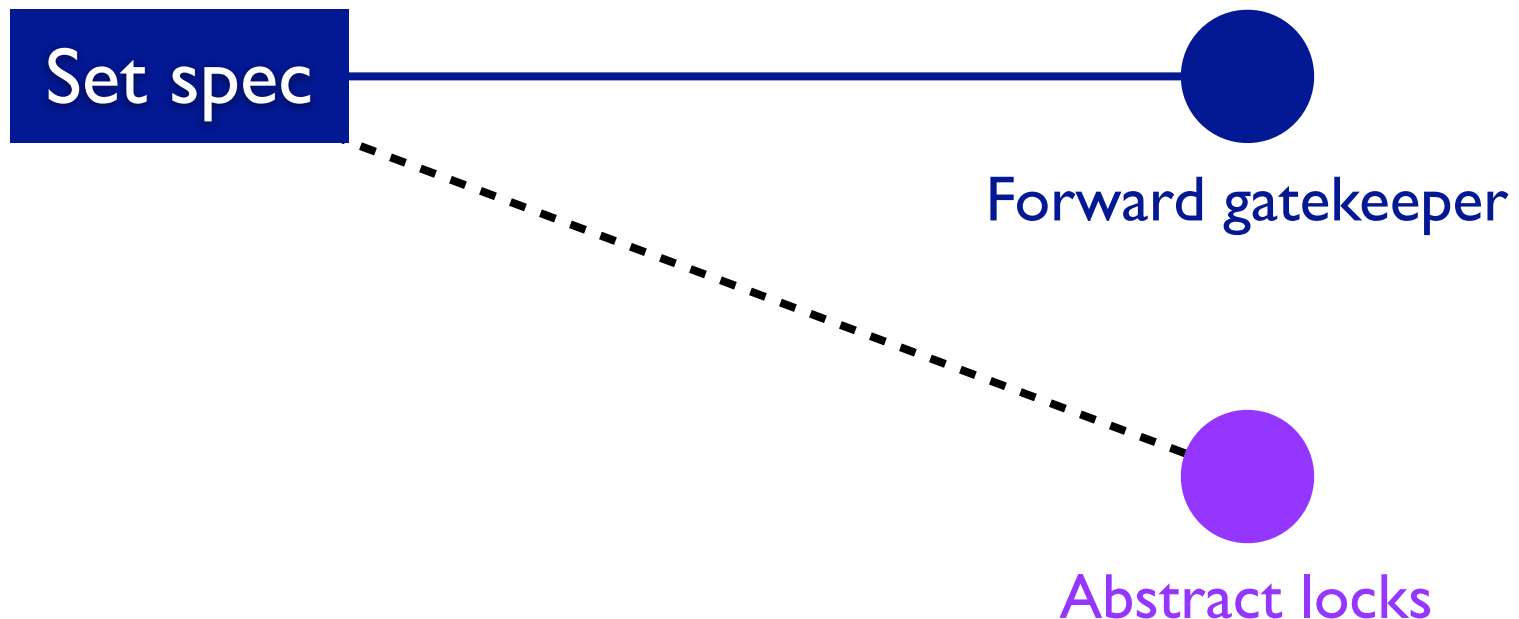
Lowering overhead of conflict detection

- No prior work fully implemented commutativity for sets
- Used lower-overhead schemes instead



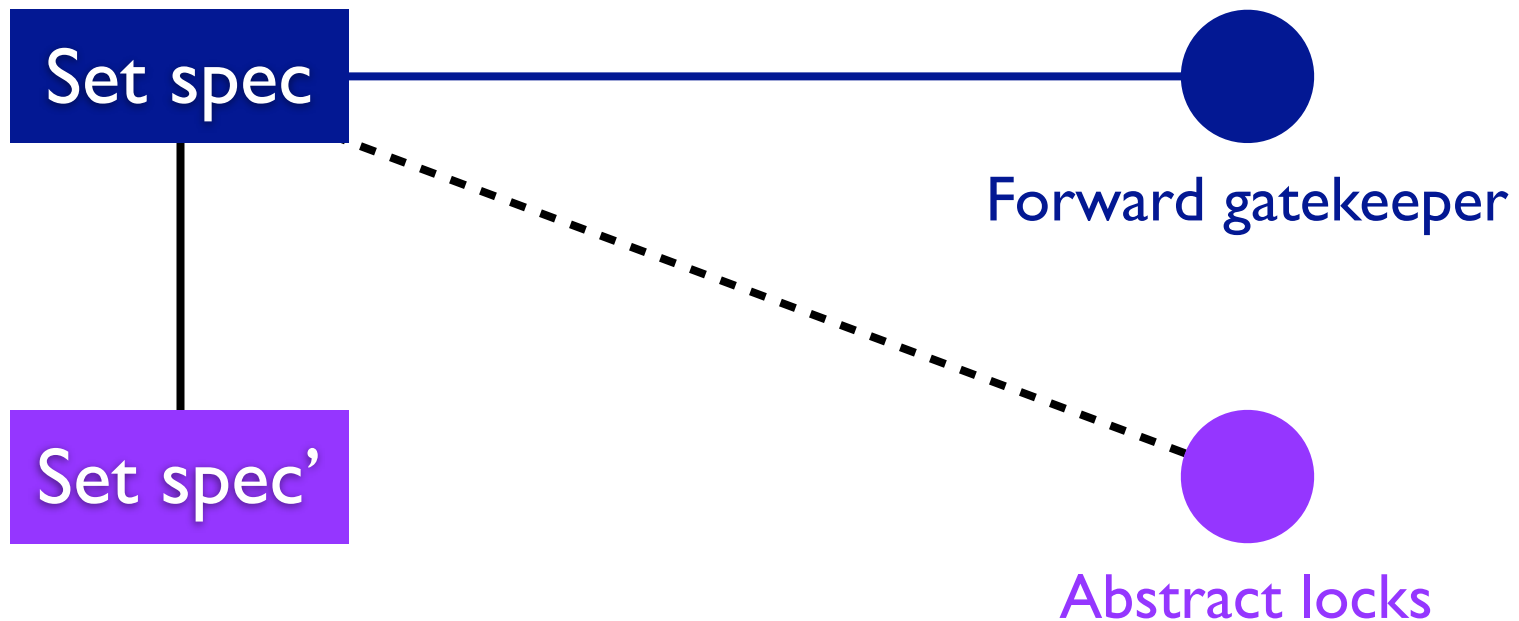
Disciplined approach

- To lower overhead, build sound and complete implementation of a different specification



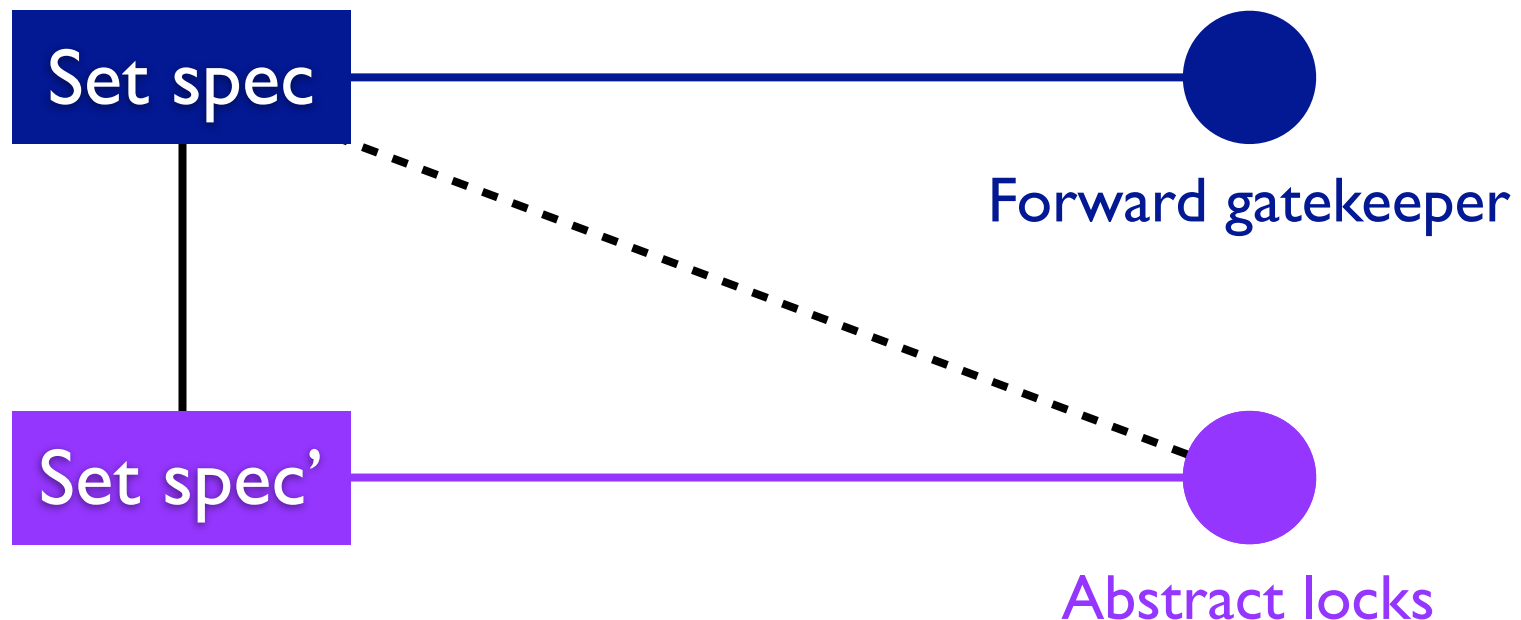
Disciplined approach

- To lower overhead, build sound and complete implementation of a different specification



Disciplined approach

- To lower overhead, build sound and complete implementation of a different specification



Exploiting the commutativity lattice

- Find simpler specifications from lower in the lattice

$\varphi(\text{contains}(a)/r_1, \text{contains}(b)/r_2) : \mathbf{true}$

$\varphi(\text{add}(a)/r_1, \text{contains}(b)/r_2) : a \neq b \vee r_1 = \mathbf{false}$

$\varphi(\text{add}(a)/r_1, \text{add}(b)/r_2) : a \neq b \vee (\dots)$



Forward gatekeeper

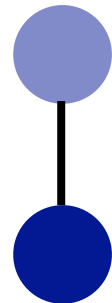
Exploiting the commutativity lattice

- Find simpler specifications from lower in the lattice

$\varphi(\text{contains}(a)/r_1, \text{contains}(b)/r_2) : \mathbf{true}$

$\varphi(\text{add}(a)/r_1, \text{contains}(b)/r_2) : a \neq b$

$\varphi(\text{add}(a)/r_1, \text{add}(b)/r_2) : a \neq b$



Forward gatekeeper

R/W locks

Exploiting the commutativity lattice

- Find simpler specifications from lower in the lattice

$\varphi(\text{contains}(a)/r_1, \text{contains}(b)/r_2) : a \neq b$

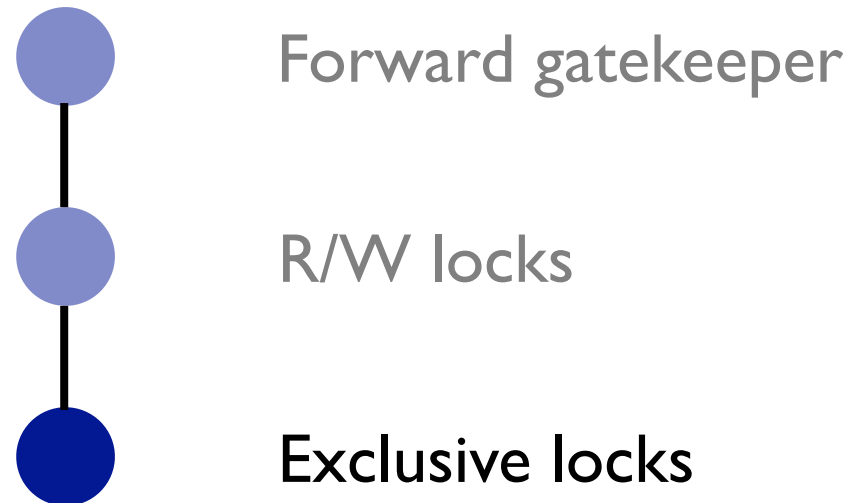
$\varphi(\text{add}(a)/r_1, \text{contains}(b)/r_2) :$

$\varphi(\text{add}(a)/r_1, \text{add}(b)/r_2)$

$a \neq b$

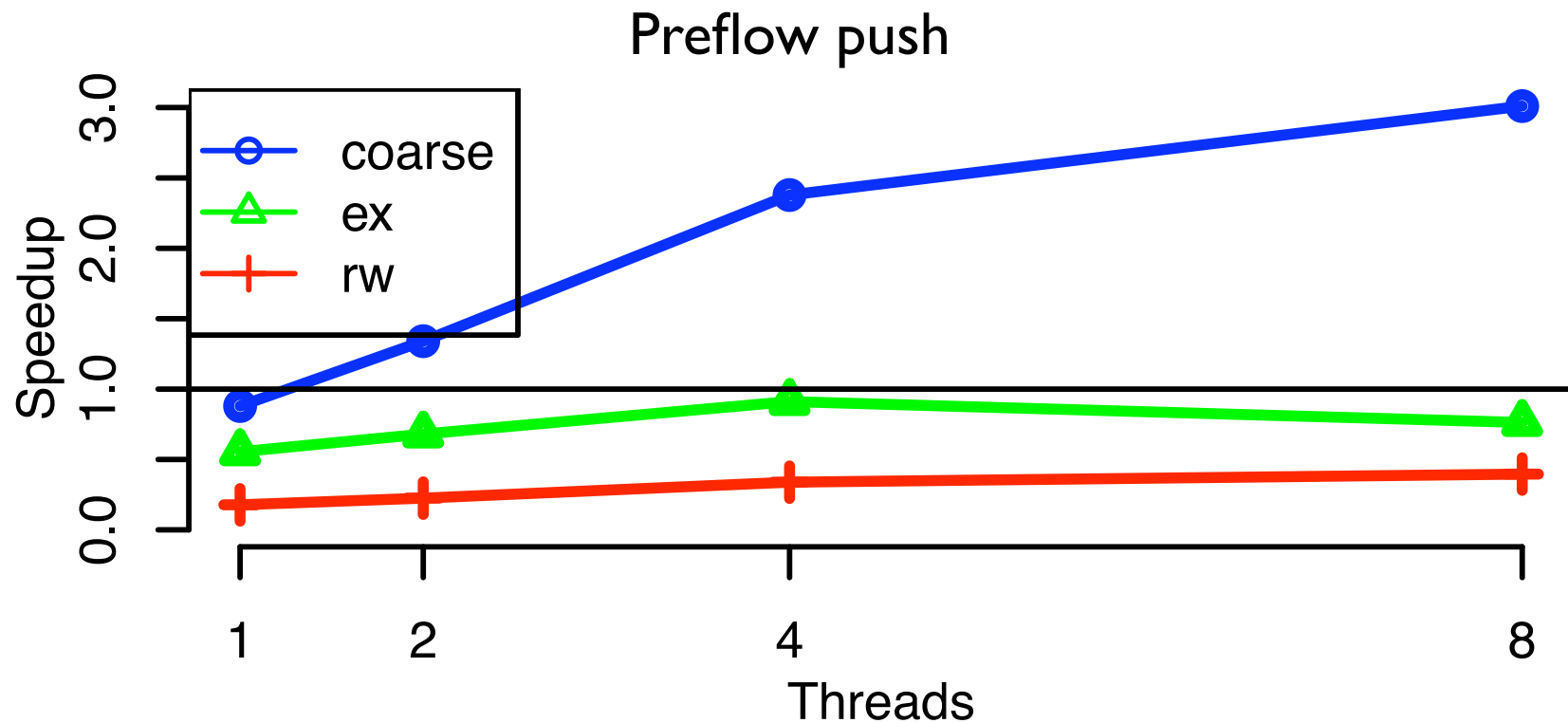
$a \neq b$

$a \neq b$



Evaluation

- Moving through commutativity lattice effectively trades off parallelism and overhead



Evaluation

- Showed that forward/general gatekeeping can provide more parallelism and better performance than memory-level locking (e.g., STM)
- Tradeoffs vary for different applications
 - ➔ Ability to generate and reason about different implementations critical

Conclusions

- Commutativity conditions are an attractive way to perform conflict detection for transactional execution
- Commutativity checkers can be systematically generated from specifications
- Commutativity lattice provides disciplined approach to producing checkers, reasoning about behavior