Exploiting the Commutativity Lattice

Milind Kulkarni

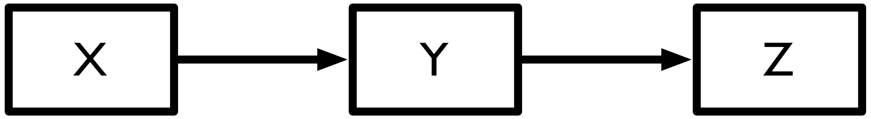
Donald Nguyen, Dimitrios Prountzos, Xin Sui and Keshav Pingali

PURDUE UNIVERSITY

THE UNIVERSITY OF TEXAS AT AUSTIN

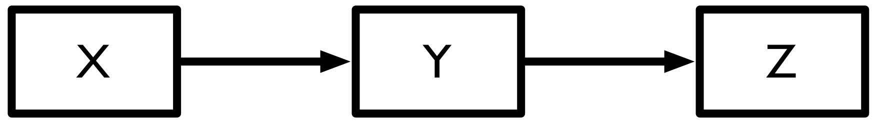
Exploiting semantics in transactional execution

Set S:



Exploiting semantics in transactional execution

Set S:

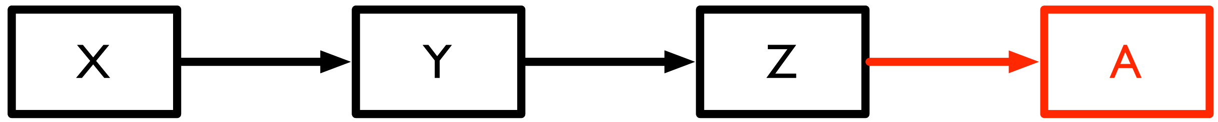



```
atomic {
  ...
  S.add(a)
  ...
}
```

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

Exploiting semantics in transactional execution

Set S:

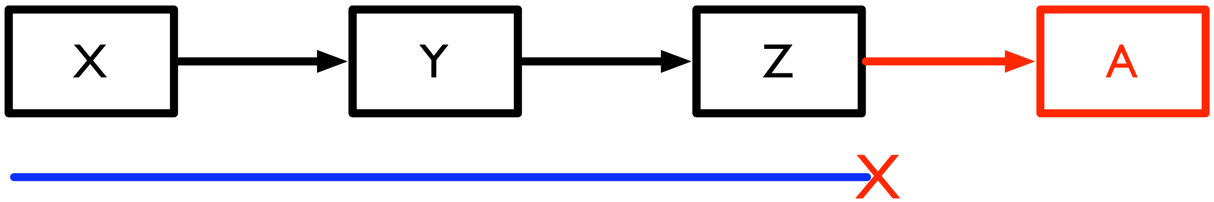



```
atomic {
  ...
  S.add(a)
  ...
}
```

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

Exploiting semantics in transactional execution

Set S:

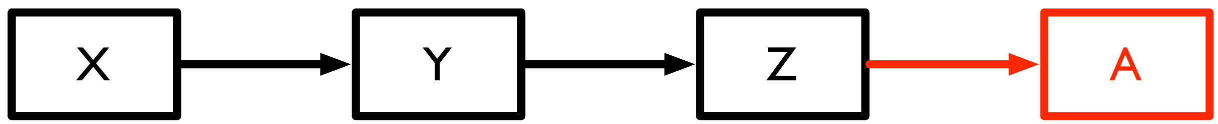



```
atomic {
  ...
  S.add(a)
  ...
}
```

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

Exploiting semantics in transactional execution

Set S:

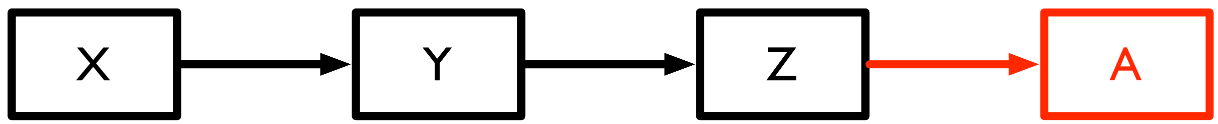



```
atomic {
  ...
  S.add(a)
  ...
}
```

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

Exploiting semantics in transactional execution

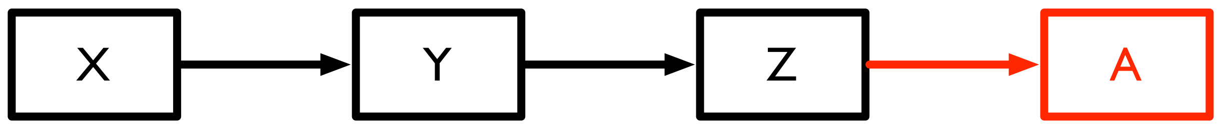
Set S:




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

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

Exploiting semantics in transactional execution

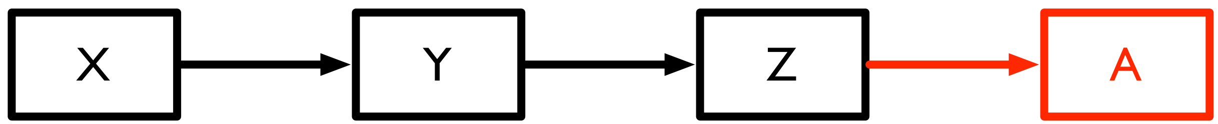
Set S: 


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

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

Exploiting semantics in transactional execution

Set S:



Key insight: exploit *commutativity* properties to ensure transactional behavior

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

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

Exploiting semantics in transactional execution

[Carlstrom et al. PPoPP 07]

Set S:

X → Y →

[Koskinen et al. POPL 10]

[Weihl et al. IEEE ToC 88]

Key insight: exploi [Ni et al. PPoPP 07] operties to ensure transactional behavior

...

S.co

[Herlihy & Koskinen PPoPP 08]

[Kulkarni et al. PLDI 07]

...

}

}

[Kulkarni et al. ASPLOS 08]

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?

`add(a)/r` *commutes* with `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

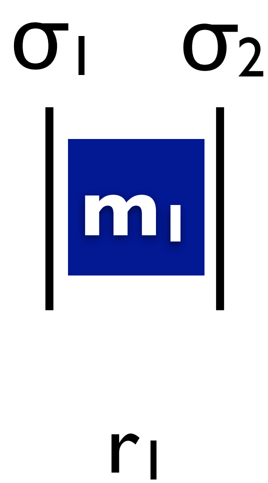
Commutativity

σ_1

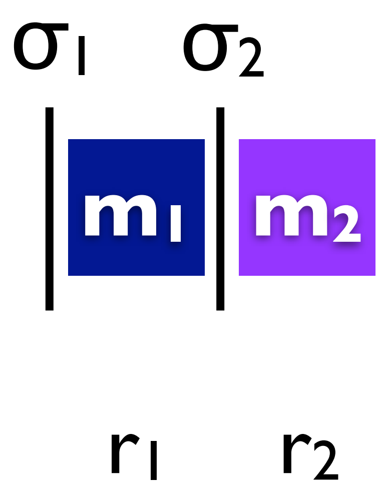
$\boxed{m_1}$

r_1

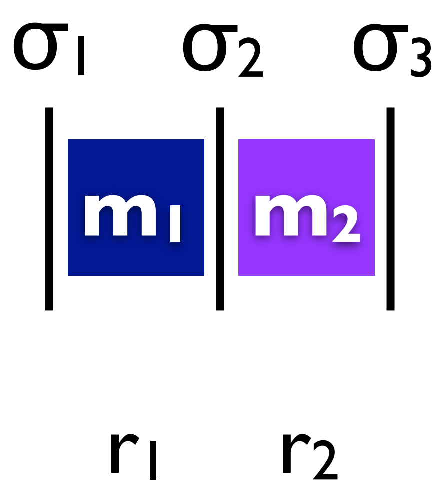
Commutativity



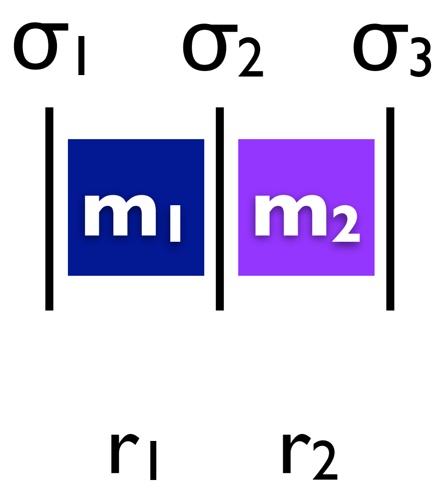
Commutativity



Commutativity

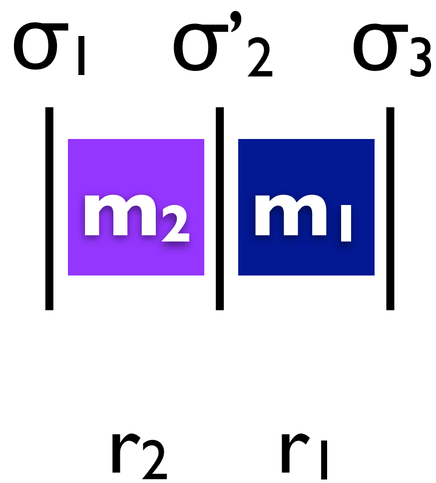


Commutativity



m_1, m_2 commute in σ_1

Commutativity



m_1, m_2 commute in σ_1

Using commutativity to guarantee serializability

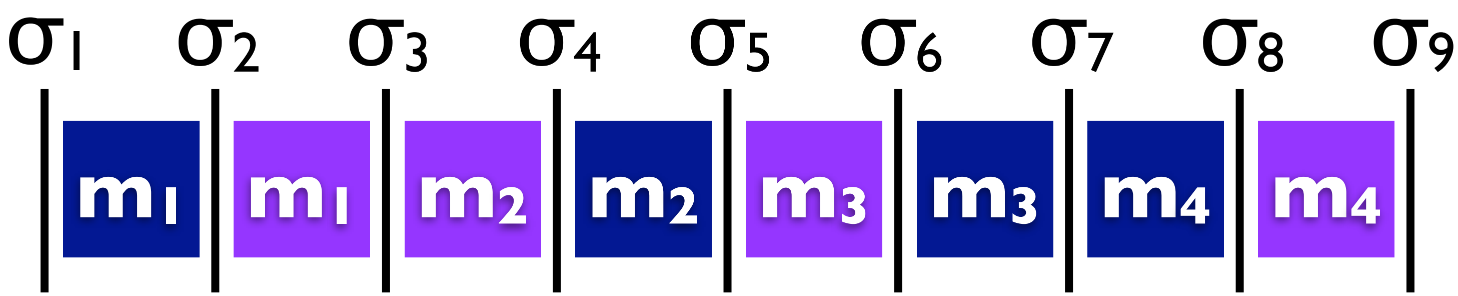
A:	m_1	m_2	m_3	m_4
B:	m_1	m_2	m_3	m_4

Using commutativity to guarantee serializability

History : m_1 m_1 m_2 m_2 m_3 m_3 m_4 m_4

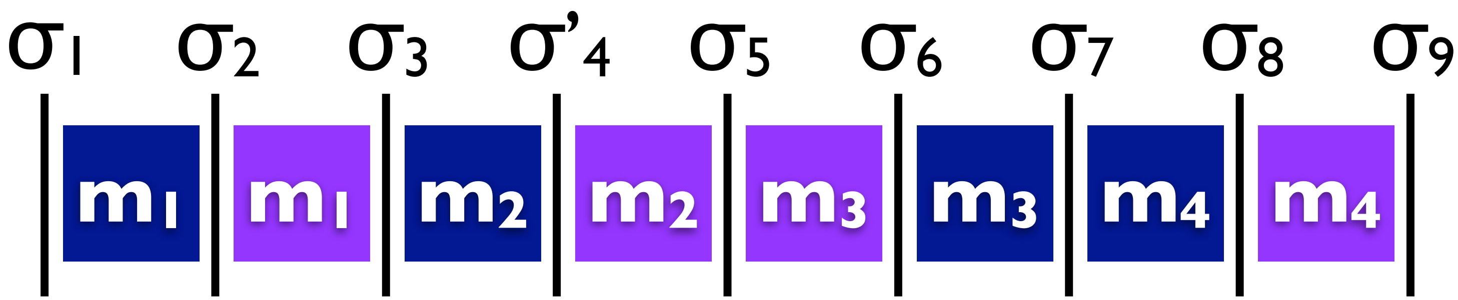
Using commutativity to guarantee serializability

History :



Using commutativity to guarantee serializability

History :



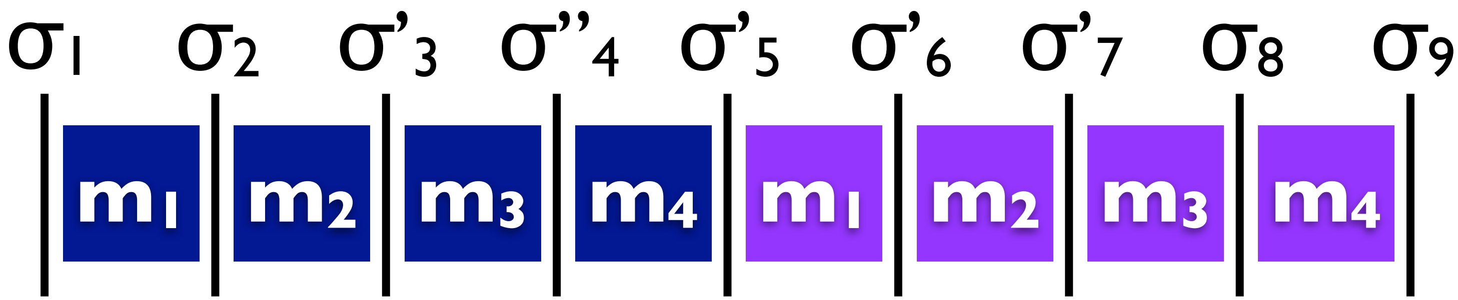
Using commutativity to guarantee serializability

$\sigma_1 \quad \sigma_2 \quad \sigma'_3 \quad \sigma'_4 \quad \sigma_5 \quad \sigma_6 \quad \sigma_7 \quad \sigma_8 \quad \sigma_9$

History : | m_1 | m_2 | m_1 | m_2 | m_3 | m_3 | m_4 | m_4 |

Using commutativity to guarantee serializability

History :



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: $\boldsymbol{\varphi}(\mathbf{m_a}, \mathbf{m_b})$

Commutativity conditions

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

true only if m_a and m_b commute

Commutativity conditions

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

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

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: $\boldsymbol{\varphi}\,(\mathbf{m_a}, \mathbf{m_b})$

Precise condition: $\boldsymbol{\varphi}^*(\mathbf{m_a}, \mathbf{m_b})$

$$\boldsymbol{\varphi}^*(\texttt{add(a)/r}_1, \texttt{contains(b)/r}_2)$$

$$||$$

$$\boxed{a \neq b \text{ or } r_1 = \textbf{false}}$$

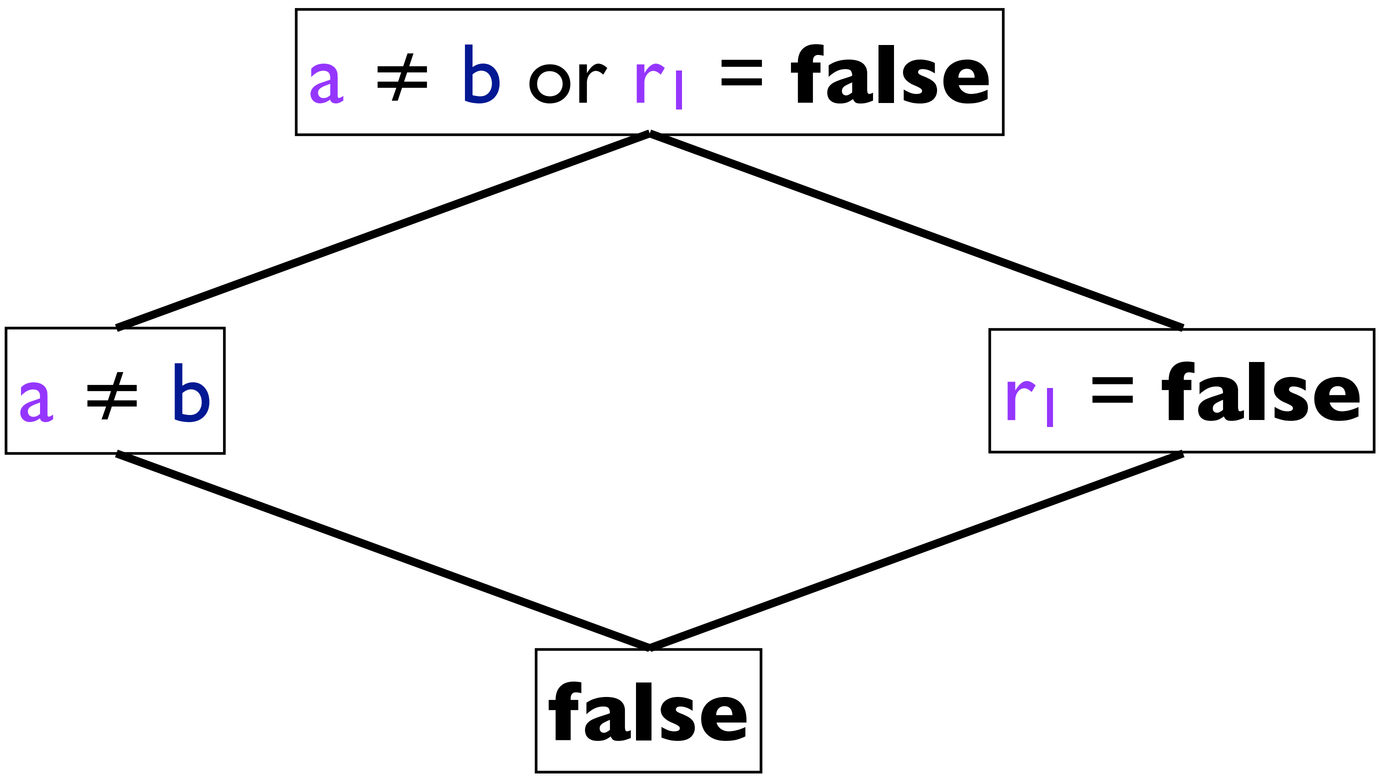
Commutativity lattice

(`add(a)/`r_1, `contains(b)/`r_2**)**

$\boxed{a \neq b \text{ or } r_1 = \textbf{false}}$

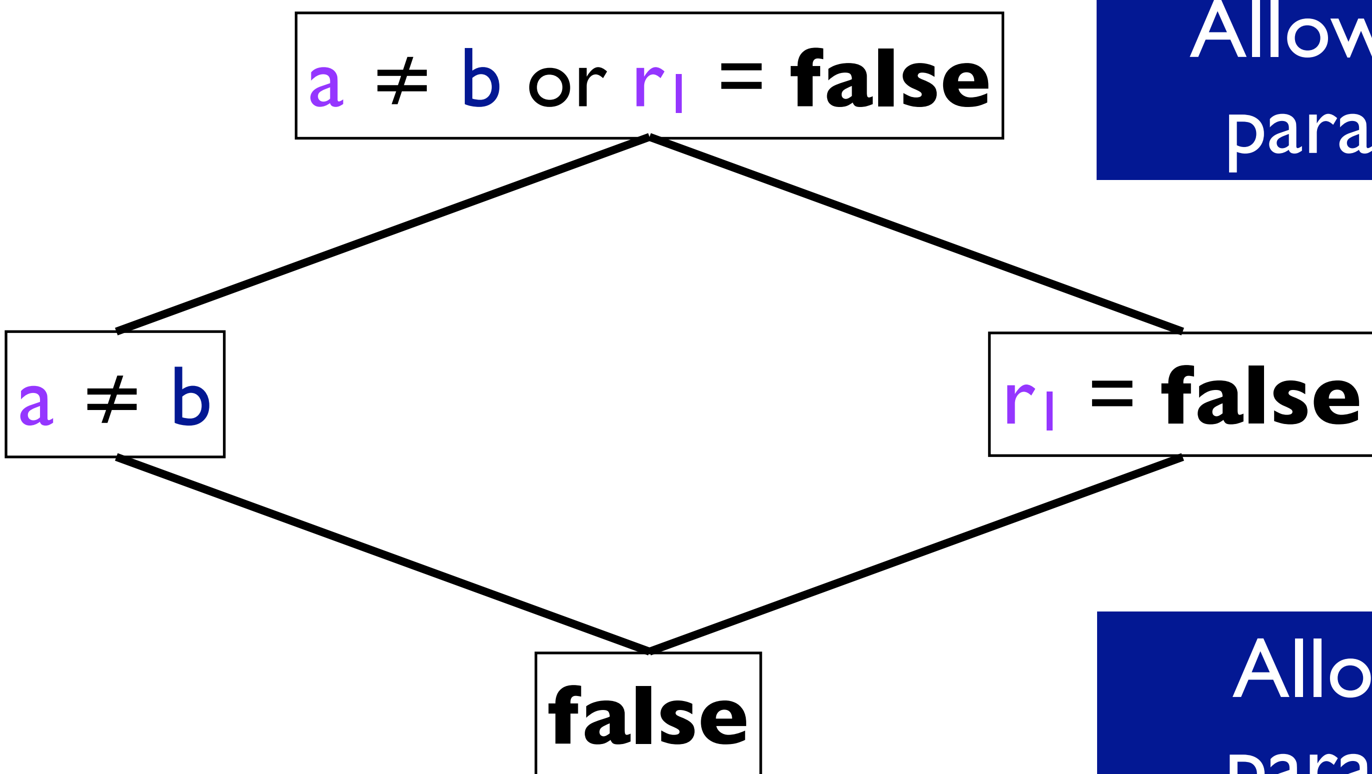
Commutativity lattice

(`add(a)/r`$_1$, `contains(b)/r`$_2$**)**



Commutativity lattice

(`add(a)/r1`, `contains(b)/r2`**)**



Allows most parallelism

Allows no parallelism

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*

(add(a)/r_1, contains(b)/r_2)

$\boxed{a \neq b \text{ or } r_1 = \textbf{false}}$

(add(a)/r_1, add(b)/r_2)

$\boxed{a \neq b \text{ or } (r_1 = \textbf{false} \text{ and } r_2 = \textbf{false})}$

(contains(a)/r_1, contains(b)/r_2)

$\boxed{\textbf{true}}$

Running example

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

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

$\boxed{a \neq b}$

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

$\boxed{a \neq b}$

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

$\boxed{\textbf{true}}$

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 "$x \neq y$"

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

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

Simple: $\boxed{a \neq b}$

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:1*

  `contains(b)` → *cont:1*

Building compatibility matrix

- One mode per argument of a method

`add(a)` → *add:I*

`contains(b)` → *cont:I*

	cont:I	*add:I*
cont:I		
add:I		

Building compatibility matrix

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

φ(add(a)/r$_1$, contains(b)/r$_2$**)** :

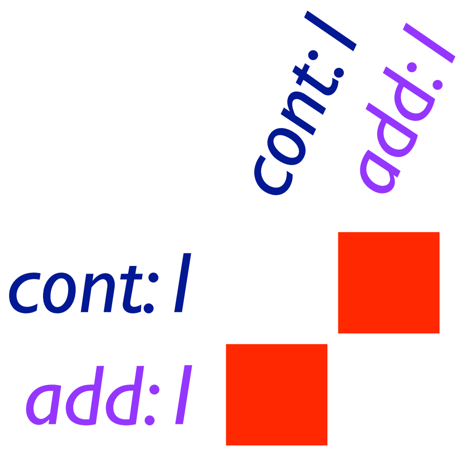
$\boxed{a \neq b}$

	cont: I	*add: I*
cont: I		
add: I		

Building compatibility matrix

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

$\boldsymbol{\varphi}(\texttt{add(a)/r}_1, \texttt{contains(b)/r}_2) :$ $\boxed{a \neq b}$

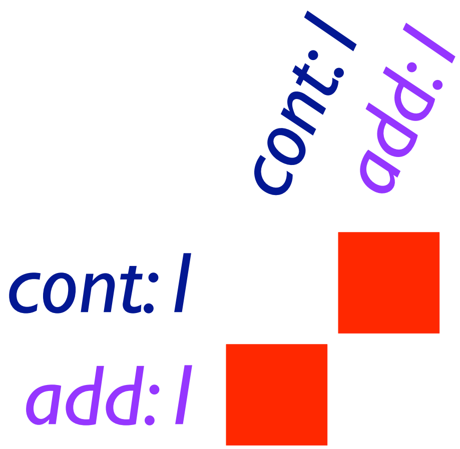


Building compatibility matrix

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

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

$\boxed{a \neq b}$

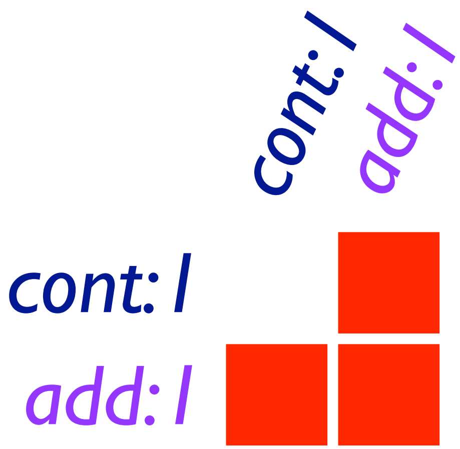


Building compatibility matrix

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

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

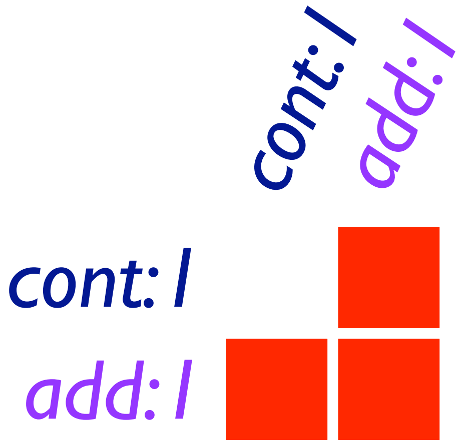
$\boxed{a \neq b}$



Building the compatibility matrix

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

$\boldsymbol{\varphi}($`contains(a)/`r_1, `contains(b)/`$r_2$$)$: $\boxed{\textbf{true}}$

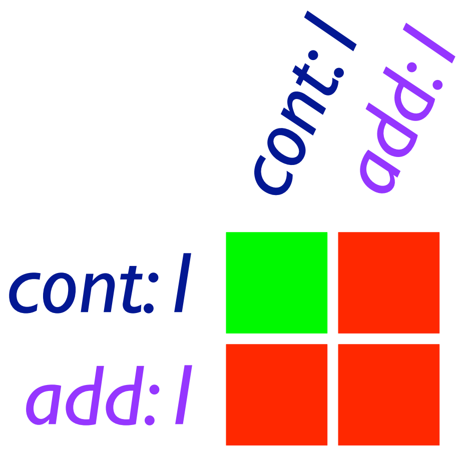


Wednesday, July 20, 2011

Building the compatibility matrix

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

$\boldsymbol{\varphi}($`contains(a)/r`$_1$, `contains(b)/r`$_2$$)$: $\boxed{\textbf{true}}$



Other conflict detection techniques

- Forward gatekeeping: sound and complete for more complex conditions

$$\boxed{a \neq b \text{ or } r_l = \textbf{false}}$$

- 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

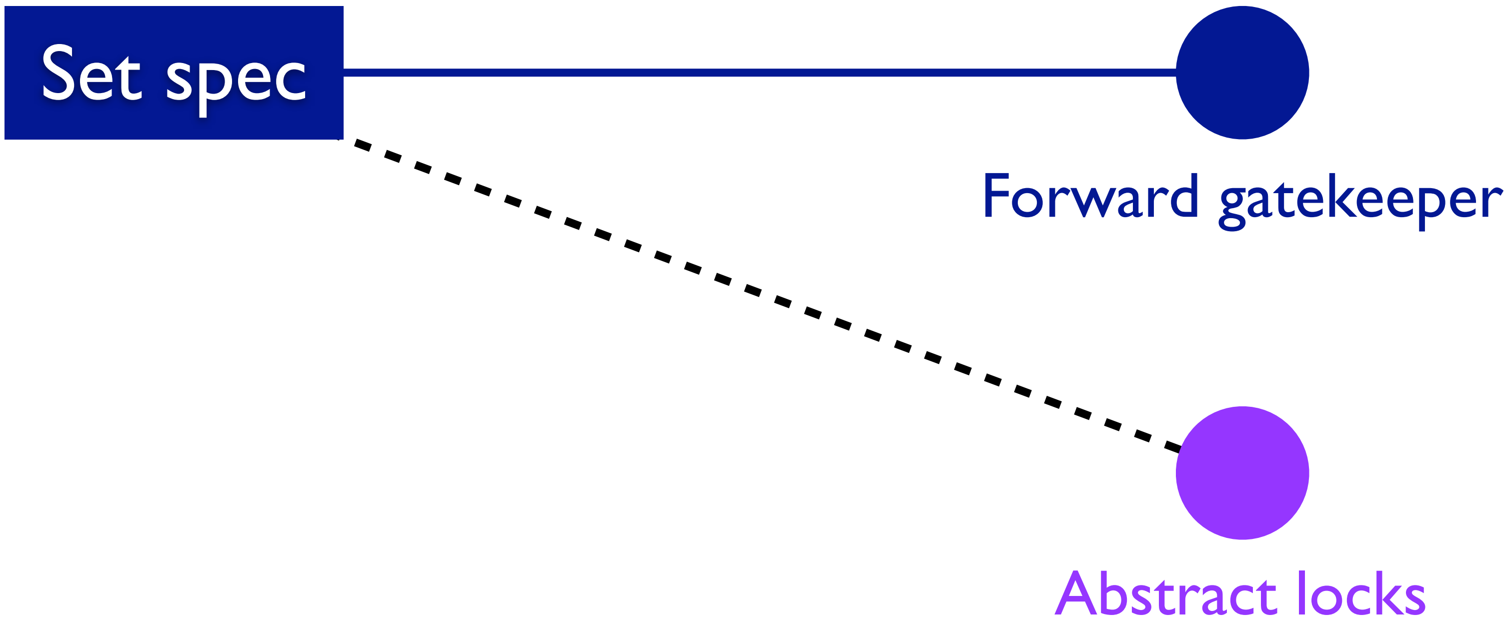
Set spec

Lowering overhead of conflict detection

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

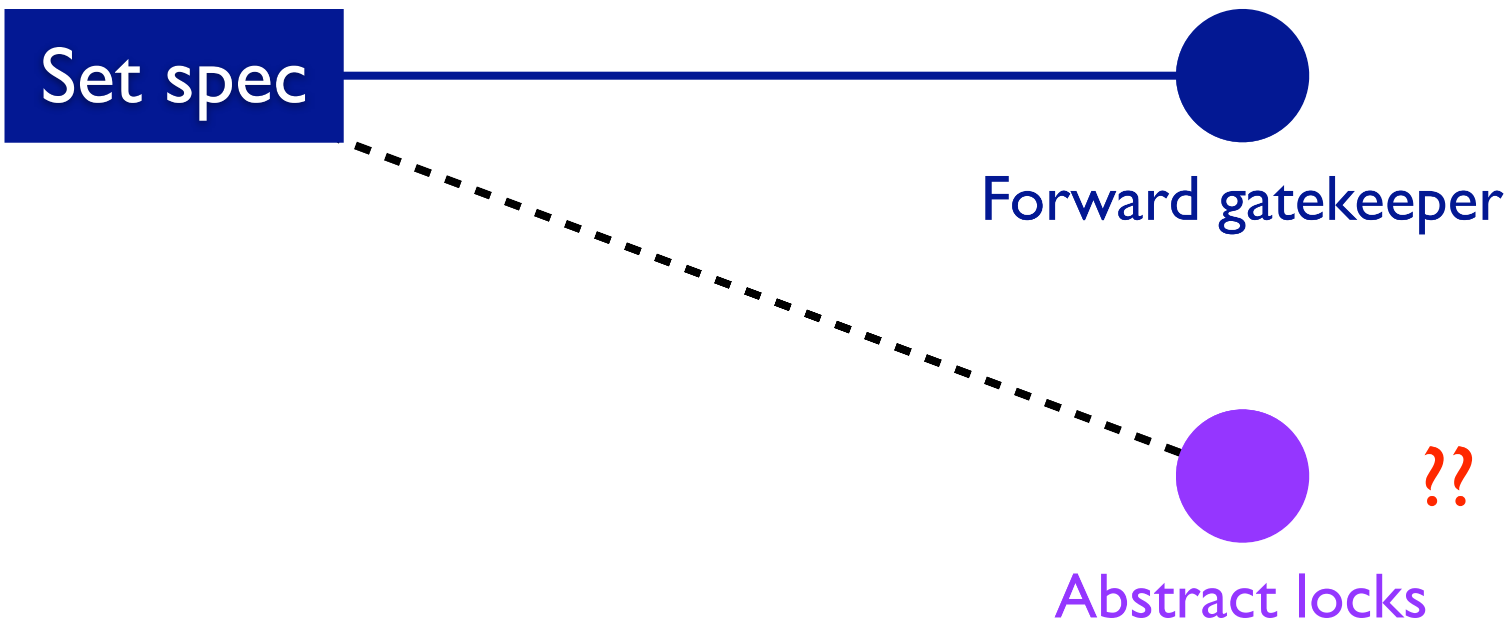
Lowering overhead of conflict detection

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



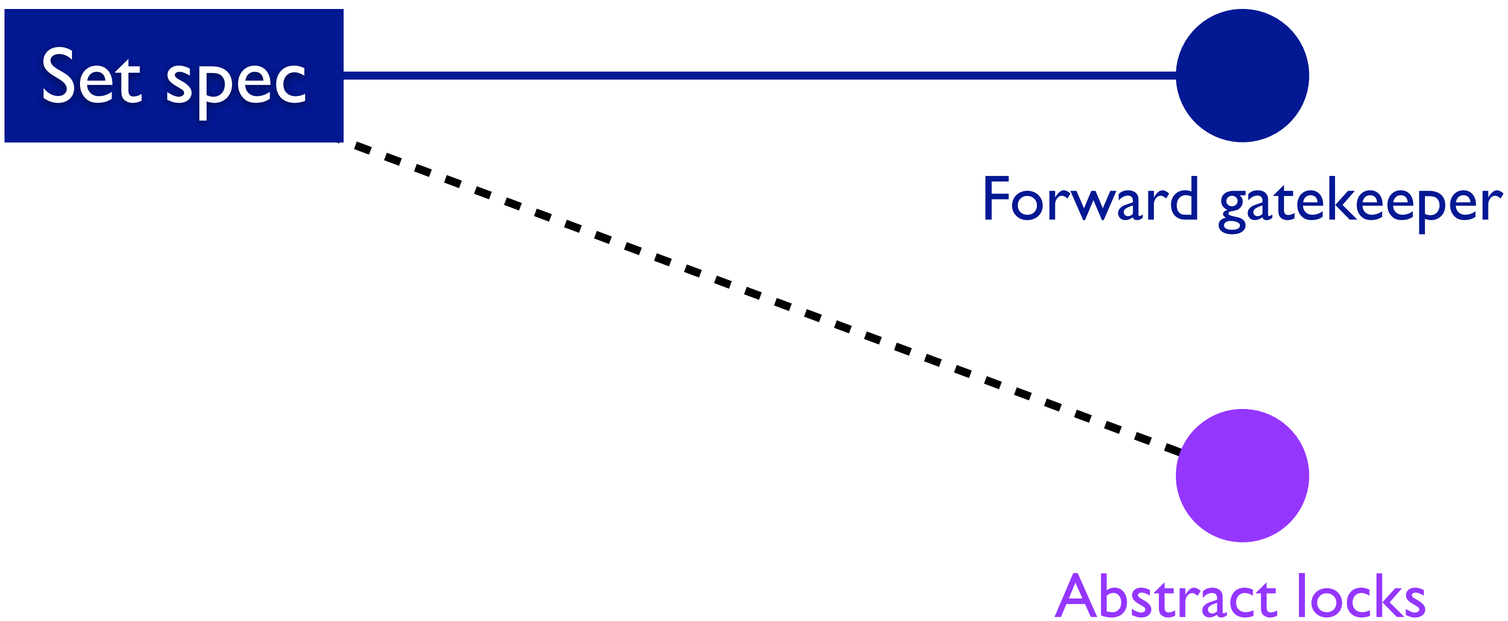
Lowering overhead of conflict detection

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



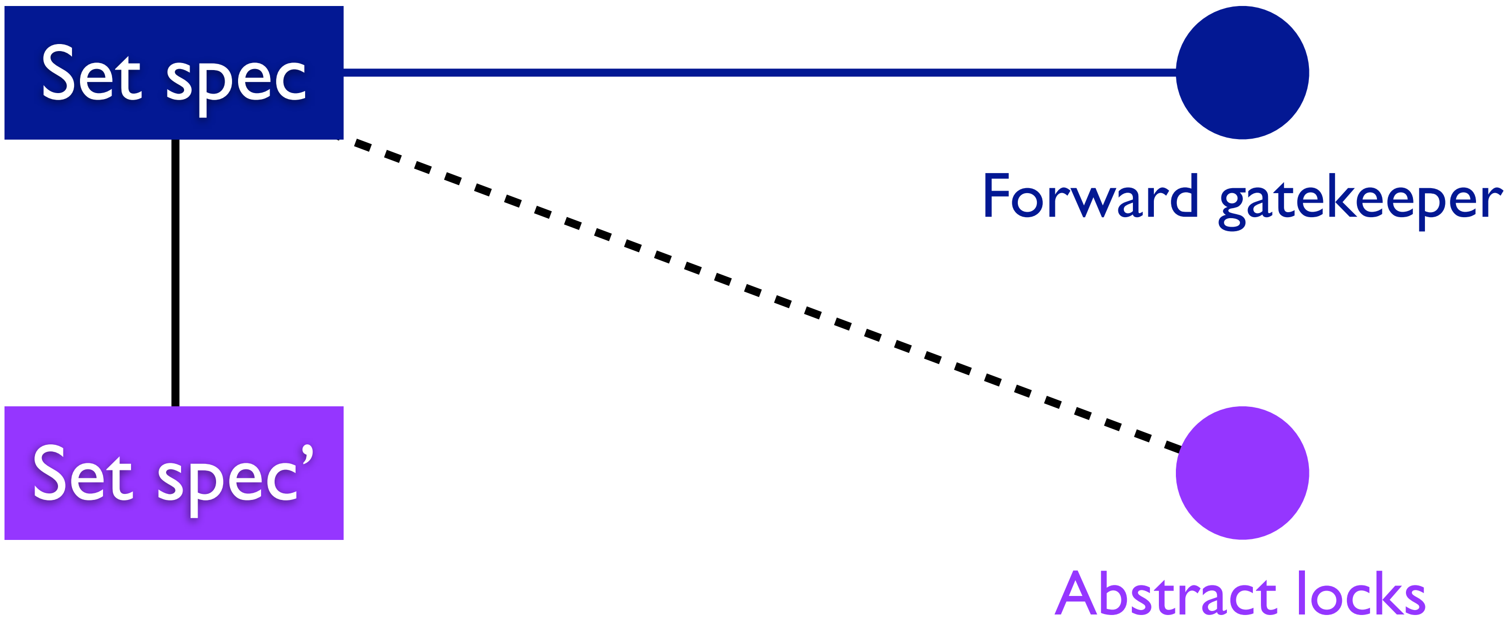
Disciplined approach

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



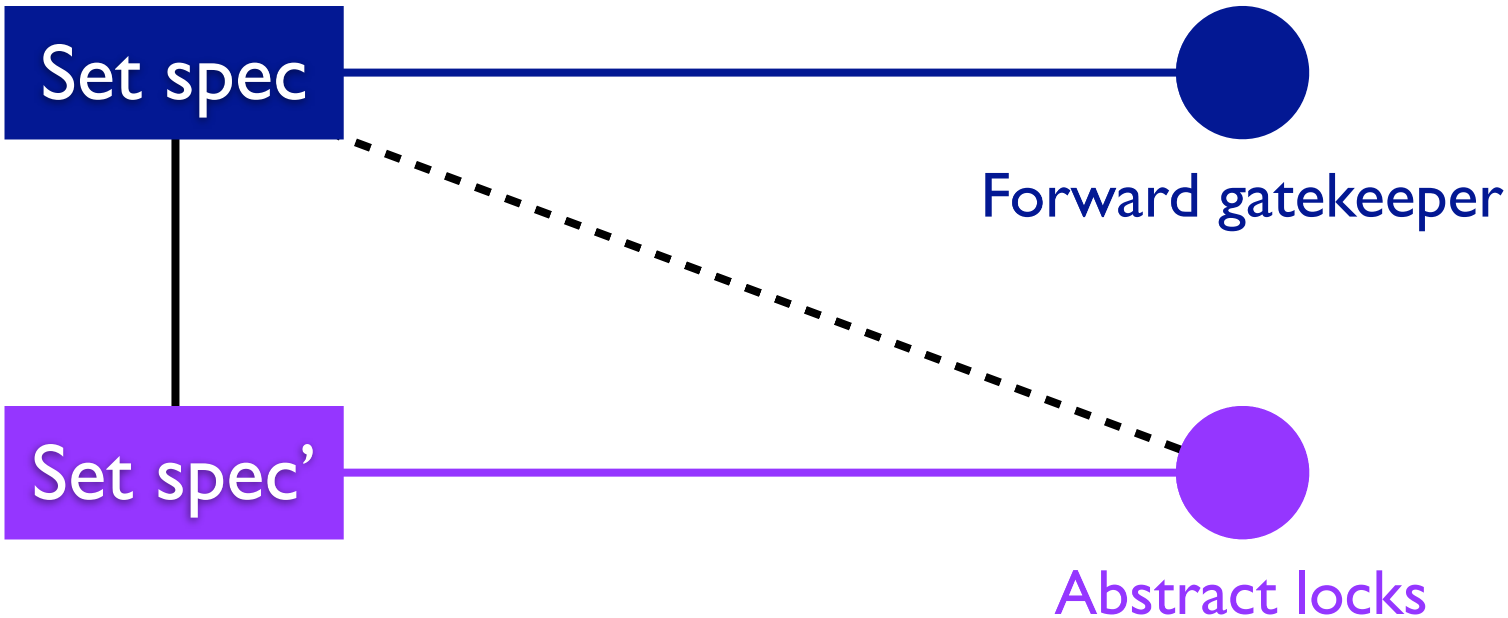
Disciplined approach

- 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

φ(contains(a)/r_1, contains(b)/r_2) : $\boxed{\textbf{true}}$

φ(add(a)/r_1, contains(b)/r_2) : $\boxed{a \neq b \vee r_1 = \textbf{false}}$

φ(add(a)/r_1, add(b)/r_2) $\boxed{a \neq b \vee (\ldots)}$

● Forward gatekeeper

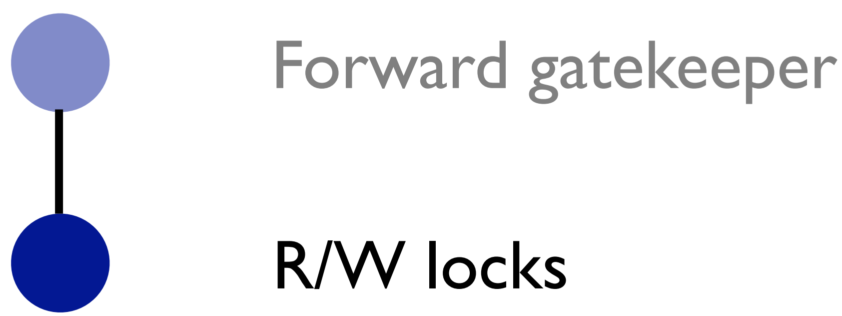
Exploiting the commutativity lattice

- Find simpler specifications from lower in the lattice

$\varphi(\mathtt{contains(a)}/r_1, \mathtt{contains(b)}/r_2)$: $\boxed{\textbf{true}}$

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

$\varphi(\mathtt{add(a)}/r_1, \mathtt{add(b)}/r_2)$ $\boxed{a \neq b}$



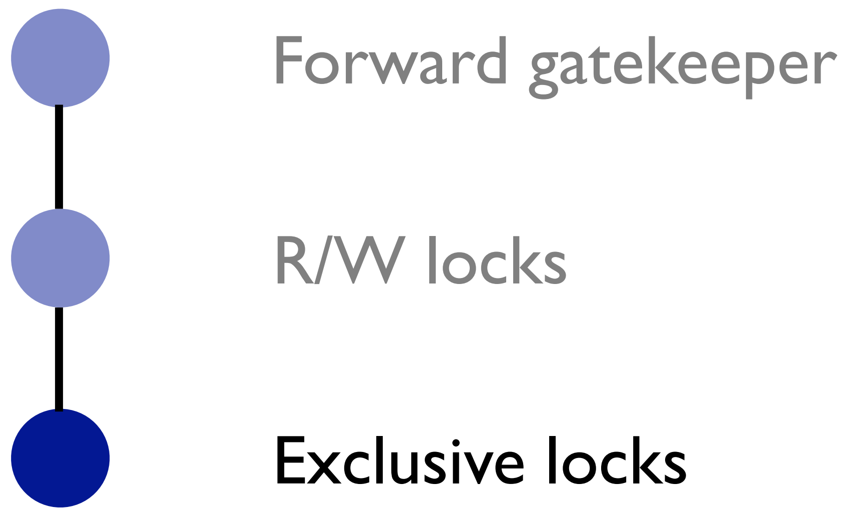
Exploiting the commutativity lattice

- Find simpler specifications from lower in the lattice

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

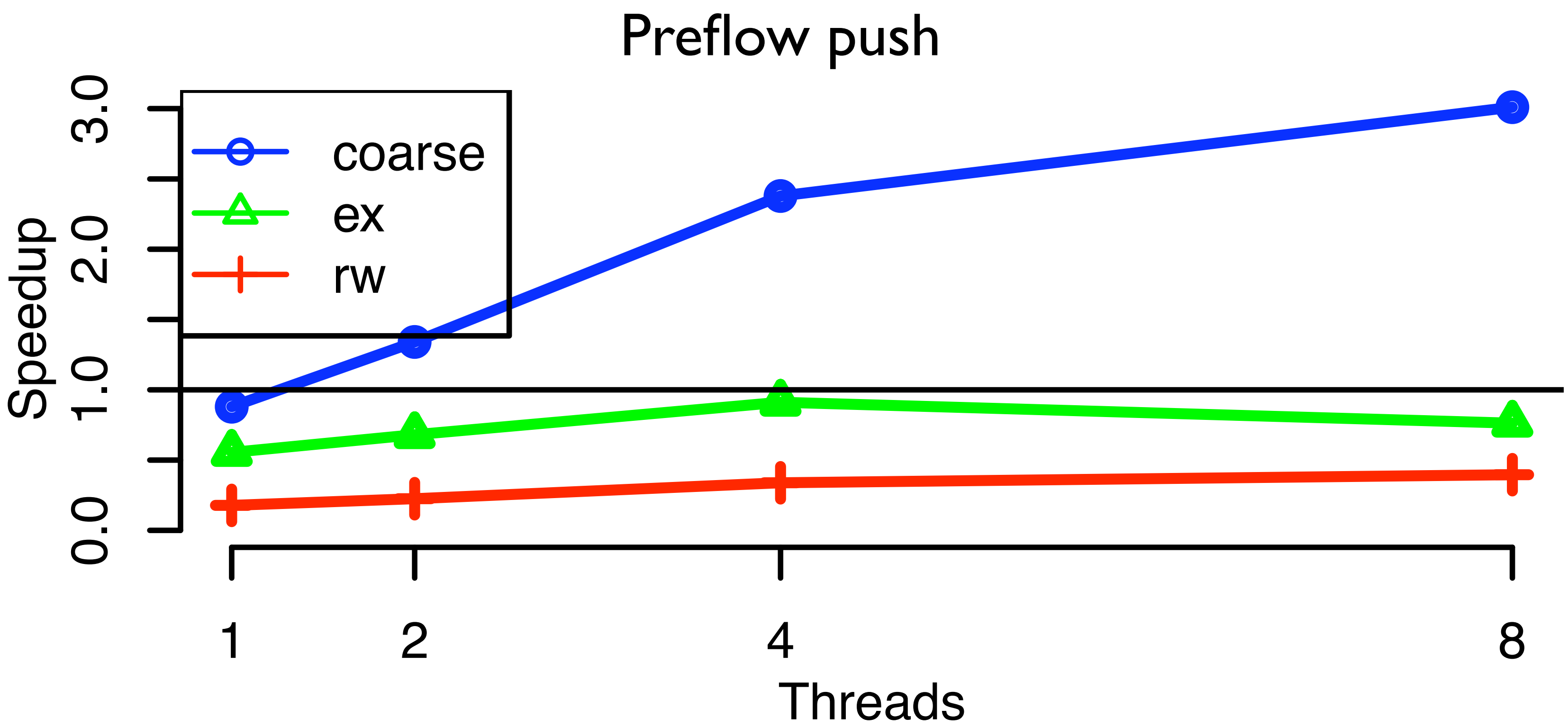
$$\varphi(\mathtt{add(a)}/r_1, \mathtt{contains(b)}/r_2) : \boxed{a \neq b}$$

$$\varphi(\mathtt{add(a)}/r_1, \mathtt{add(b)}/r_2) \quad \boxed{a \neq b}$$



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