Exploiting the Commutativity Lattice

Milind Kulkarni
Donald Nguyen, Dimitrios Prountzos, Xin Sui
and Keshav Pingali
Commutativity for speculative parallelization

- Speculative parallelization is a hot topic
- Most prior work has used memory level conflict detection (TM, TLS)
- Recent efforts have argued that we should use semantic conflict detection

- [Carlstrom et al. PPoPP07] [Ni et al. PPoPP07] [Kulkarni et al. PLDI07] [Herlihy & Koskinen PPoPP08, POPL2010]

- Key idea: exploit commutativity of method invocations
Commutativity example: Union find

```java
atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}
```
Commutativity example: Union find

```java
atomic { //TXN A
  ...
  uf.find(x);
  ...
}

atomic { //TXN B
  ...
  uf.find(y);
  ...
}
```
Commutativity example: Union find

```java
atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}
```
Commutativity example:
Union find

```
atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}
```

Both transactions complete!
Commutativity example: Union find

atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}

What about path compression?
Commutativity example: Union find

atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}

What about path compression?
Commutativity example: Union find

```
atomic { //TXN A
    ...
    uf.find(x);
    ...
}

atomic { //TXN B
    ...
    uf.find(y);
    ...
}
```

Transactions cannot execute concurrently!
Commutativity example: Union find

- Data structure semantics remain the same with or without path compression
- Executing find does not change semantic state
- But path compression triggers conflicts!
- Alternate approach: use commutativity
- Data structure operations are atomic
- Transactions that perform commutative operations can execute concurrently
- Semantics matter: find always commutes with find
Prior work on commutativity

- Most work focused on how to exploit commutativity
- Using commutativity in databases [Weihl 88]
- Using commutativity in optimistic parallelization [Kulkarni et al. 07]
- Using commutativity to improve TM performance [Carlstrom et al. 07] [Ni et al. 07] [Herlihy & Koskinen 08]

Open question: how do we check commutativity?
Checking commutativity

- Prior work has used ad hoc commutativity checkers
  - Log-based (Kulkarni 07)
  - Abstract lock-based (Ni 07)
  - Arbitrary code (Herlihy 08)
- Prior work has used different implementations for the same data structure!
  - R/W locks on set keys (Ni 07)
  - Exclusive locks on set keys (Herlihy 08)
  - Coarse locks on groups of keys (Kulkarni 08)
Implementing commutativity

- How can we implement a commutativity checker?
- How do we know our implementation is correct?
- How do we choose an implementation?
- What if our implementation has too much overhead? Does not allow enough parallelism?
- What we need is a systematic way to implement commutativity and reason about these questions
Implementing commutativity

- How can we implement a commutativity checker?
- How do we know our implementation is correct?
- How do we choose an implementation?
- What if our implementation has too much overhead? Does not allow enough parallelism?
- What we need is a systematic way to implement commutativity and reason about these questions
Outline

- Background
- Commutativity conditions and lattice
- Procedures for implementing commutativity
- Exploiting the commutativity lattice
- Evaluation
Commutativity conditions

- A predicate that evaluates to true only if two methods commute

\[ \phi_{m_1;m_2}(\sigma_1, v_1, r_1, \sigma_2, v_2, r_2) \equiv m_1(v_1)/r_1 \text{ commutes with } m_2(v_2)/r_2 \text{ if } \phi \]

- Commutativity based on semantics of methods
- Commutativity specification: set of conditions for all methods of an ADT
Using commutativity conditions

- Same basic strategy in all prior work
- When a transaction wants to invoke a method:
  - Runtime evaluates commutativity condition against *currently active transactions*
  - If all commutativity checks succeed, invocation proceeds
    - *i.e.*, method must commute with all other active methods
  - If a check fails, roll back
- Execution strategy guarantees *serializability*
**Commutativity specification for sets**

<table>
<thead>
<tr>
<th></th>
<th>Operation 1</th>
<th>Commutes with</th>
<th>Operation 2</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td><code>add(a)/r_1</code></td>
<td>if <code>a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false})</code></td>
<td><code>add(b)/r_2</code></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td><code>add(a)/r_1</code></td>
<td>if <code>a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false})</code></td>
<td><code>remove(b)/r_2</code></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td><code>add(a)/r_1</code></td>
<td>if <code>a \neq b \lor r_1 = \text{false}</code></td>
<td><code>contains(b)/r_2</code></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td><code>remove(a)/r_1</code></td>
<td>if <code>a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false})</code></td>
<td><code>remove(b)/r_2</code></td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td><code>remove(a)/r_1</code></td>
<td>if <code>a \neq b \lor r_1 = \text{false}</code></td>
<td><code>contains(b)/r_2</code></td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td><code>contains(a)/r_1</code></td>
<td>if <code>a \neq b \lor r_1 = \text{false}</code></td>
<td><code>contains(b)/r_2</code></td>
<td></td>
</tr>
</tbody>
</table>
Commutativity specification for sets

(1) \( \text{add}(a)/r_1 \) commutes with \( \text{add}(b)/r_2 \) if \( a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \)

(2) \( \text{add}(a)/r_1 \) commutes with \( \text{remove}(b)/r_2 \) if \( a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \)

(3) \( \text{add}(a)/r_1 \) commutes with \( \text{contains}(b)/r_2 \) if \( a \neq b \lor r_1 = \text{false} \)

(4) \( \text{remove}(a)/r_1 \) commutes with \( \text{remove}(b)/r_2 \) if \( a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \)

(5) \( \text{remove}(a)/r_1 \) commutes with \( \text{contains}(b)/r_2 \) if \( a \neq b \lor r_1 = \text{false} \)

(6) \( \text{contains}(a)/r_1 \) commutes with \( \text{contains}(b)/r_2 \)
A commutativity lattice

- Order commutativity conditions by strength
  \[ \phi_1 \sqsubseteq \phi_2 \iff \phi_1 \Rightarrow \phi_2 \]
  \[ \phi_1 \sqcup \phi_2 \equiv \phi_1 \lor \phi_2 \]
  \[ \phi_1 \sqcap \phi_2 \equiv \phi_1 \land \phi_2 \]

- What about top and bottom?
  \[ \bot = \text{false} \]
  \[ \top = \phi^* \]

- Generalize to full commutativity specifications
Example:
add(a)/r₁ vs. contains(b)/r₂

\[(a \neq b) \lor (r₁ = \text{false})\]
What does the lattice mean?

- The lower you go in the lattice, the stronger the commutativity condition
- The *less likely* the condition is to prove that two methods commute
- The *more likely* a transaction is to conflict with other transactions
- If all commutativity conditions are “false” then no methods commute and all transactions are serialized
- Exact effect on parallelism is application dependent
Outline

- Background
- Commutativity conditions and lattice
- Procedures for implementing commutativity
- Exploiting the commutativity lattice
- Evaluation
Implementing commutativity checks

- Commutativity lattice lets us generate a variety of commutativity specifications for a given ADT
- How do we implement commutativity check?
  - Run-time needs to check commutativity to ensure serializability
    - Check returns true → runtime assumes methods commute
  - Check should be low overhead
  - Check should be correct
- Goal: systematic approach to building checker
Correctness of commutativity implementation

- Define correctness relative to a particular commutativity condition

- **Soundness:**
  - If check claims that methods commute, then commutativity condition is true

- **Completeness:**
  - If commutativity condition is true, then check claims that methods commute
Three schemes

- Abstract locking
- Forward gatekeeping
- General gatekeeping

Goals

- Systematic construction of sound and complete checkers using each scheme
- Characterization of when each scheme is applicable
Abstract locking

- When method is invoked, can acquire a lock associated with arguments, return value or even whole data structure
- Locks can be acquired in various modes
  - Compatibility between modes defined by compatibility matrix
  - Basically, database mode locks
- Encompasses locking schemes used by Ni et al., Carlstrom et al., Herlihy & Koskinen
Constructing abstract locking scheme

- Systematic algorithm for building abstract locking schemes for a data structure
- Construction algorithm works for simple specifications
- Commutativity conditions are either true, false, or a conjunction of inequalities between arguments and return values
Construction algorithm example

- Will explain construction algorithm in terms of accumulator data structure
- Commutativity specification on right

\[
\begin{align*}
(1) & \quad \text{increment}(a) \text{ commutes with } \text{increment}(b) \\
(2) & \quad \text{increment}(a) \text{ commutes with } \text{read}() / r_2 \quad \text{if} \quad \text{false} \\
(3) & \quad \text{read}() / r_1 \text{ commutes with } \text{read}() / r_2
\end{align*}
\]
Construction algorithm example

- Step 1: create locks
  - One lock with the data structure
  - One lock with each object that could be an argument or return value to the data structure

\[
\begin{align*}
(1) & \quad \text{increment}(a) \quad \text{commutes with} \quad \text{increment}(b) \\
(2) & \quad \text{increment}(a) \quad \text{commutes with} \quad \text{read}()/r_2 \quad \text{if} \quad \text{false} \\
(3) & \quad \text{read}()/r_1 \quad \text{commutes with} \quad \text{read}()/r_2
\end{align*}
\]
Construction algorithm example

- Step 1: create locks
  - One lock with the data structure
  - One lock with each object that could be an argument or return value to the data structure

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>increment(a) commutes with increment(b)</td>
</tr>
<tr>
<td>2</td>
<td>increment(a) commutes with read()/r_2 if false</td>
</tr>
<tr>
<td>3</td>
<td>read()/r_1 commutes with read()/r_2</td>
</tr>
</tbody>
</table>

`acc` \[1, 2, 3 \ldots n\]
Construction algorithm example

- Step 2: create modes for each lock
  - One “global” mode per method
  - One mode per method argument
  - One mode per method return value
  - All locks support the same modes

<table>
<thead>
<tr>
<th>Step</th>
<th>Mode</th>
<th>Commutes with</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{increment}(a)</td>
<td>\text{increment}(b)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\text{increment}(a)</td>
<td>\text{read}() / r_2 &amp; \text{false}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\text{read}() / r_1</td>
<td>\text{read}() / r_2</td>
<td></td>
</tr>
</tbody>
</table>

\text{acc} \quad 1, 2, 3 \ldots n
Construction algorithm example

- Step 2: create modes for each lock
  - One “global” mode per method
  - One mode per method argument
  - One mode per method return value
  - All locks support the same modes

\[
\begin{align*}
(1) & \quad \text{increment}(a) \quad \text{commutes with} \quad \text{increment}(b) \\
(2) & \quad \text{increment}(a) \quad \text{commutes with} \quad \text{read}() / r_2 \\
(3) & \quad \text{read}() / r_1 \quad \text{commutes with} \quad \text{read}() / r_2
\end{align*}
\]

\[
\begin{align*}
\text{acc} & \quad 1, 2, 3 \ldots n \\
\text{inc:ds} & \quad \text{inc:x} \quad \text{read:ds} \quad \text{read:ret}
\end{align*}
\]

Exploiting the Commutativity Lattice
Construction algorithm example

- Step 3: define locking policy
- When a method is invoked, acquire the data structure lock in the appropriate mode
- Acquire locks on any arguments in the appropriate mode
- Acquire locks on any return values in the appropriate mode

<table>
<thead>
<tr>
<th></th>
<th>increment(a) commutes with (a)</th>
<th>(a) commutes with increment(b)</th>
<th>increment(b) commutes with (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>increment(a) commutes with (a)</td>
<td>(a) commutes with increment(b)</td>
<td>increment(b) commutes with (b)</td>
</tr>
<tr>
<td>(2)</td>
<td>increment(a) commutes with (a)</td>
<td>(a) commutes with increment(b)</td>
<td>increment(b) commutes with (b)</td>
</tr>
<tr>
<td>(3)</td>
<td>increment(a) commutes with (a)</td>
<td>(a) commutes with increment(b)</td>
<td>increment(b) commutes with (b)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
\text{acc} & 1, 2, 3 \ldots n \\
\hline
inc:ds & inc:ds & inc:x & inc:x \\
read:ds & read:ds & read:ret & read:ret \\
\end{array}
\]
Construction algorithm example

- Step 3: define locking policy

- When a method is invoked, acquire the data structure lock in the appropriate mode

- Acquire locks on any arguments in the appropriate mode

- Acquire locks on any return values in the appropriate mode

\[
\begin{align*}
(1) \quad \text{increment}(a) & \quad \text{commutes with} & \quad \text{increment}(b) \\
(2) \quad \text{increment}(a) & \quad \text{commutes with} & \quad \text{read}() / r_2 \quad \text{if} \quad \text{false} \\
(3) \quad \text{read}() / r_1 & \quad \text{commutes with} & \quad \text{read}() / r_2 \\
\end{align*}
\]

\[\text{increment}(7) \rightarrow \text{acquire}(\text{acc, inc:ds}), \text{acquire}(7, \text{inc:x})\]
Construction algorithm example

- Step 4: define compatibility matrix

  - Default: lock modes are compatible

  - If two methods never commute ($\phi = \text{false}$), global lock modes conflict

  - If two methods commute if ($a \neq b$), appropriate argument modes conflict

<table>
<thead>
<tr>
<th>$\text{inc:ds}$</th>
<th>$\text{inc:x}$</th>
<th>$\text{read:ds}$</th>
<th>$\text{read:ret}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
<tr>
<td>$\text{inc:ds}$</td>
<td>$\text{inc:x}$</td>
<td>$\text{read:ds}$</td>
<td>$\text{read:ret}$</td>
</tr>
</tbody>
</table>

acc 1, 2, 3 ... n

Monday, January 31, 2011
Construction algorithm example

- Step 4: define compatibility matrix
- Default: lock modes are compatible
- If two methods never commute ($\phi = \text{false}$), global lock modes conflict
- If two methods commute if ($a \neq b$), appropriate argument modes conflict

<table>
<thead>
<tr>
<th></th>
<th>$inc:ds$</th>
<th>$inc:x$</th>
<th>$read:ds$</th>
<th>$read:\text{ret}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$inc:ds$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$inc:x$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$read:ds$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$read:\text{ret}$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>

(1) $\text{increment}(a)$ commutes with $\text{increment}(b)$
(2) $\text{increment}(a)$ commutes with $\text{read}() / r_2$ if $\phi = \text{false}$
(3) $\text{read}() / r_1$ commutes with $\text{read}() / r_2$
Construction algorithm example

- Step 5: simplify
  - Remove any lock modes that are always compatible
  - Remove locks on any arguments/return values that never conflict

<table>
<thead>
<tr>
<th></th>
<th>inc:ds</th>
<th>inc:x</th>
<th>read:ds</th>
<th>read:ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc:ds</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>inc:x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>read:ds</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>read:ret</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Construction algorithm example

- Step 5: simplify
- Remove any lock modes that are always compatible
- Remove locks on any arguments/return values that never conflict

\[
\begin{array}{ccc}
(1) & \text{increment}(a) & \text{commutes with} & \text{increment}(b) \\
(2) & \text{increment}(a) & \text{commutes with} & \text{read}() / r_2 \\
& & \text{if} & \text{false} \\
(3) & \text{read}() / r_1 & \text{commutes with} & \text{read}() / r_2 \\
\end{array}
\]

\[
\text{acc} \quad 1, 2, 3 \ldots n
\]

\[
\begin{array}{c|cc}
\text{inc:ds} & \text{read:ds} \\
\hline
\text{inc:ds} & \checkmark & \times \\
\text{read:ds} & \times & \checkmark \\
\end{array}
\]

Exploiting the Commutativity Lattice

Monday, January 31, 2011
Exploiting the Commutativity Lattice

Constructive algorithm produces sound & complete abstract locking scheme for any simple specification.

Theorem: sound & complete abstract locking scheme exists if and only if specification is simple.

Implication: more complex specifications (e.g., sets) might have to use more complex commutativity implementation.
Forward gatekeeping

- More expressive scheme
- Can support arbitrary logical clauses and boolean functions
- Works for online-checkable specifications
- No boolean functions that combine information from first method’s state with second method’s arguments/return values
Forward gatekeeping

- Intuition:
  - As methods execute, record information needed by any commutativity condition
  - To check commutativity, combine information from current method with information from previous methods

1. nearest(a)/r1 commutes with nearest(b)/r2
2. nearest(a)/r1 commutes with add(b)/r2 if r2 = false ∨ dist(a, b) > dist(a, r1)
3. nearest(a)/r1 commutes with remove(b)/r2 if (a ≠ b ∧ r1 ≠ b) ∨ r2 = false
Forward gatekeeping

- **Intuition:**
  - As methods execute, record information needed by any commutativity condition
  - To check commutativity, combine information from current method with information from previous methods

<table>
<thead>
<tr>
<th>(1)</th>
<th>nearest(a)/r₁ commutes with nearest(b)/r₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>nearest(a)/r₁ commutes with add(b)/r₂ if r₂ = false ∨ dist(a, b) &gt; dist(a, r₁)</td>
</tr>
<tr>
<td>(3)</td>
<td>nearest(a)/r₁ commutes with remove(b)/r₂ if (a ≠ b ∧ r₁ ≠ b) ∨ r₂ = false</td>
</tr>
</tbody>
</table>
General gatekeeping

- Most expressive scheme
- May require rollback during conflict detection to verify commutativity

(1) \( \text{find}(a)/r_1 \) commutes with \( \text{find}(b)/r_2 \)

(2) \( \text{union}(a, b) \) commutes with \( \text{find}(c)/r_2 \)
   if \( \text{rep}(\sigma_1, c) \neq \text{loser}(\sigma_1, a, b) \)

- finds commute
- union commutes with find if the find returns the same value it would have without executing union
General gatekeeping

- Most expressive scheme
- May require rollback during conflict detection to verify commutativity

\[
\begin{align*}
(1) & \quad \text{find}(a)/r_1 \text{ commutes with } \text{find}(b)/r_2 \\
(2) & \quad \text{union}(a, b) \text{ commutes with } \text{find}(c)/r_2 \\
& \quad \text{if } \text{rep}(\sigma_1, c) \neq \text{loser}(\sigma_1, a, b)
\end{align*}
\]

- finds commute
- union commutes with find if the find returns the same value it would have without executing union
Conflict detection hierarchy

- Schemes can be ranked by expressiveness and overhead
  - abstract locking < forward gatekeeping < general gatekeeping
- Would like to use the lowest overhead scheme possible
- How can we do this?
Trading performance for precision

- Recall: different systems use different conflict detectors for sets
  - R/W locks on keys vs. exclusive locks on keys

- Basic tradeoff
  - R/W locks: more parallelism, higher overhead
  - Exclusive locks: less parallelism, less overhead

- How do we know that these two schemes are both sound?
- How do we know which schemes expose more parallelism?
Tradeoff strategy

- Don’t use a single specification and build different implementations
  - Hard to prove correctness
  - Hard to reason about parallelism
- Change the specification according to lattice
  - Implementation sound and complete by construction
  - Easy to reason about parallelism
Disciplined tradeoffs

- Use commutativity lattice to derive higher/lower parallelism schemes, and construction algorithms to build conflict checkers

\begin{align*}
(1) \quad \text{add}(a)/r_1 & \text{ commutes with } \text{add}(b)/r_2 \\
& \text{if } a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \\
(2) \quad \text{add}(a)/r_1 & \text{ commutes with } \text{remove}(b)/r_2 \\
& \text{if } a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \\
(3) \quad \text{add}(a)/r_1 & \text{ commutes with } \text{contains}(b)/r_2 \\
& \text{if } a \neq b \lor r_1 = \text{false} \\
(4) \quad \text{remove}(a)/r_1 & \text{ commutes with } \text{remove}(b)/r_2 \\
& \text{if } a \neq b \lor (r_1 = \text{false} \land r_2 = \text{false}) \\
(5) \quad \text{remove}(a)/r_1 & \text{ commutes with } \text{contains}(b)/r_2 \\
& \text{if } a \neq b \lor r_1 = \text{false} \\
(6) \quad \text{contains}(a)/r_1 & \text{ commutes with } \text{contains}(b)/r_2
\end{align*}

Requires forward gatekeeping
Disciplined tradeoffs

- Use commutativity lattice to derive higher/lower parallelism schemes, and construction algorithms to build conflict checkers

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add($a$)/$r_1$ commutes with add($b$)/$r_2$ if $a \neq b$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add($a$)/$r_1$ commutes with remove($b$)/$r_2$ if $a \neq b$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add($a$)/$r_1$ commutes with contains($b$)/$r_2$ if $a \neq b$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>remove($a$)/$r_1$ commutes with remove($b$)/$r_2$ if $a \neq b$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>remove($a$)/$r_1$ commutes with contains($b$)/$r_2$ if $a \neq b$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>contains($a$)/$r_1$ commutes with contains($b$)/$r_2$</td>
<td></td>
</tr>
</tbody>
</table>

Can use abstract locks
Need R/W locks
Disciplined tradeoffs

- Use commutativity lattice to derive higher/lower parallelism schemes, and construction algorithms to build conflict checkers

<table>
<thead>
<tr>
<th></th>
<th>(\text{add}(a)/r_1) commutes with if (\text{add}(b)/r_2) (a \neq b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\text{add}(a)/r_1) commutes with if (\text{remove}(b)/r_2) (a \neq b)</td>
</tr>
<tr>
<td>2</td>
<td>(\text{add}(a)/r_1) commutes with if (\text{contains}(b)/r_2) (a \neq b)</td>
</tr>
<tr>
<td>3</td>
<td>(\text{remove}(a)/r_1) commutes with if (\text{remove}(b)/r_2) (a \neq b)</td>
</tr>
<tr>
<td>4</td>
<td>(\text{remove}(a)/r_1) commutes with if (\text{contains}(b)/r_2) (a \neq b)</td>
</tr>
<tr>
<td>5</td>
<td>(\text{contains}(a)/r_1) commutes with if (\text{contains}(b)/r_2) (a \neq b)</td>
</tr>
<tr>
<td>6</td>
<td>(\text{contains}(a)/r_1) commutes with if (\text{contains}(b)/r_2) (a \neq b)</td>
</tr>
</tbody>
</table>

Can use abstract locks
Can use exclusive locks
Outline

• Background
• Commutativity conditions and lattice
• Procedures for implementing commutativity
• Exploiting the commutativity lattice
• Evaluation
Effects of strengthening commutativity

- Implement preflow-push using a graph
- Try three different graph conflict checkers
- Conclusion: as long as checker exposes enough parallelism, should use lower overhead checker
Overhead of more complex schemes

- Use forward gatekeeping for kd-trees in agglomerative clustering

- Use general gatekeeping for union-find in Boruvka’s algorithm

![Graph showing speedup over threads for different algorithms](image)
Conclusions

- Commutativity lattice provides a framework to reason about commutativity specifications.
- Construction algorithms provide a systematic way to build conflict checkers given a specification.
- Can combine lattice and algorithms to provide a disciplined approach to building and selecting conflict checkers for an application.