Simplified synchronization through optimistic linearizability

Maurice Herlihy
Brown University

Eric Koskinen
University of Cambridge
Concurrent Data-structures

- Are a pain to build!
- Require fine-grained locks or atomic CAS to gain more concurrency -- Type Specific
- Must be designed with cache in mind. Cache-aware layouts are type specific
What’s an alternative to designing these kinds of concurrent algorithms?
Multicore as a distributed system
Multicore as a distributed system

Architecture is Shared Memory, but treated as distributed
Multicore as a distributed system

Architecture is Shared Memory, but treated as distributed

• Each thread’s memory is private
Multicore as a distributed system

Architecture is Shared Memory, but treated as distributed

- Each thread’s memory is private
- Objects are **replicated**
Multicore as a distributed system

- Each thread’s memory is private
- Objects are **replicated**
- Replicas kept up-to-date by a synchronization manager

Architecture is Shared Memory, but treated as distributed
Multicore as a distributed system

replica manager

hashtable
queue1
queue2

hashtable
queue1
queue2

hashtable
queue1
queue2

hashtable
queue1
queue2
We can use this approach to provide one-step transactions
Optimistic Linearizability

- Threads perform some operation `o.m()` (i.e. a one-step transaction) locally
- Atomic broadcast of the name “`o.m()`”
- Replicas lazily apply `o.m()` when they next need to access `o`
Optimistic Linearizability

Good News!
Optimistic Linearizability

Good News!

★ Sequential implementations!
Optimistic Linearizability

Good News!

★ Sequential implementations!
★ Programmer needn’t design concurrent algorithms!
Optimistic Linearizability

Good News!

★ Sequential implementations!
★ Programmer needn’t design concurrent algorithms!
★ Linearizability ensured internally!
Optimistic Linearizability

Good News!

★ Sequential implementations!
★ Programmer needn’t design concurrent algorithms!
★ Linearizability ensured internally!
★ Good cache locality!
Note: we are not abandoning shared-memory.

Instead, using shared-memory in a disciplined way to avoid communication complexity.
Example

Queue queue;
Hashtable ht;

A
if (foo())
  ht.set(5 => 'Red')
else
  ht.set(5 => 'Green')

B
q.enq('diesel');
q.enq('hybrid');

C
Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();
if (foo())
   ht.set(5 => 'Red')
else
   ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');

Car c = new Car();
c.color = ht.get(5);
c.type  = q.deq();
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');

Car c = new Car();
c.color = ht.get(5);
c.type  = q.deq();

-> q.enq('diesel');
q.enq('diesel');
-> q.enq('diesel');
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

-> q.enq('diesel');

if (foo())
    ...
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green')}
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');
q.enq('diesel');
if (foo())
    ...  
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green')}

Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();
-> q.enq('diesel');
-> q.enq('diesel');
-> q.enq('diesel');
-> ht.set(5 => 'Green');
-> ht.set(5 => 'Green');
-> ht.set(5 => 'Green');
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');

Car c = new Car();
c.type = q.deq();
c.color = ht.get(5);

c.color = ht.get(5);
c.type = q.deq();

-> q.enq('diesel');
if (foo())
    ...
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green') }
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

A

q.enq('diesel');
q.enq('hybrid');

B

Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();

C

Applies events as needed

q.enq('diesel');
{ do q.enq('diesel') } -> q.enq('diesel');
-> ht.set(5 => 'Green');
{ do ht.set(5 => 'Green') } -> ht.set(5 => 'Green');
Car c = new Car();
A

if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

-> q.enq('diesel');
if (foo())
    ...
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green') }

B

q.enq('diesel');
q.enq('hybrid');

C

Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();

Applies events as needed

read-only ops not replicated
```java
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');
q.enq('diesel');

Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();
c.color = ht.get(5);
```

A

```java
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green')}

-> q.enq('diesel');
-> q.enq('diesel');
-> q.enq('diesel');

if (foo())
    ...
    ht.set(5 => 'Green')
{ do q.enq('diesel') }

-> q.enq('diesel');
-> ht.set(5 => 'Green');
-> ht.set(5 => 'Green');

-> q.enq('hybrid');
-> q.enq('hybrid');
-> q.enq('hybrid');
```

B

```java
q.enq('diesel');
q.enq('hybrid');

c.color = ht.get(5);
c.type = q.deq();
```

C
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');
Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');

Car c = new Car();
c.color = ht.get(5);
c.type  = q.deq();
if (foo())
    ht.set(5 => 'Red')
else
    ht.set(5 => 'Green')

q.enq('diesel');
q.enq('hybrid');

Car c = new Car();
c.color = ht.get(5);
c.type = q.deq();

-> q.enq('diesel');
if (foo())
    ... 
    ht.set(5 => 'Green')
{ do ht.set(5 => 'Green') } 

q.enq('hybrid');

-> q.enq('hybrid');
{ do q.enq('hybrid') }
Optimistic Linearizability

Caveats
Operations must be deterministic. Since operations are replayed, outcomes must agree. A non-deterministic specification is acceptable.
Operations must be **deterministic**.
Since operations are replayed, outcomes must agree.
A **non**deterministic specification is acceptable.
Optimistic Linearizability

Caveats

- Operations must be **deterministic**.
  Since operations are replayed, outcomes must agree.
  A **non**-deterministic specification is acceptable.

- Operations must be **total**.
  Must return a value (or return an exception); cannot block.
Formal Model

Thread $\tau \in T$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$
Formal Model

Thread $\tau \in T$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$
Formal Model

Thread $\tau \in T$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$

program local state
Formal Model

Thread $\tau \in T$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$

- program
- local state
- map objects to method list
Formal Model

Thread $\tau \in T$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$

Language $s ::= c; s | o.m; s | \text{skip}$
Formal Model

Thread $\tau \in \mathcal{T}$ is $\langle s, \sigma_\tau, A_\tau, \ell \rangle$

Semantics: Configuration $\mathcal{C} \in \mathcal{T} \times \mathcal{A}$

A step in the semantics involves choosing an element of $\mathcal{T}$ and taking one of the following steps:
Formal Model
Formal Model

\( \langle \text{skip}, \sigma_T, A_T, \text{nil} \rangle, T, A \rightarrow T, A \)
Formal Model

\[ \langle \text{skip}, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \rightarrow T, A \]

\[ \langle \text{cmd}; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \rightarrow \langle s, \sigma'_\tau, A_\tau, \text{nil} \rangle, T, A \quad \text{if} \quad \sigma'_\tau \in [\text{cmd}] \sigma_\tau \]
Formal Model

\[
\langle \text{skip}, \sigma_T, A_T, \text{nil} \rangle, T, A \rightarrow T, A
\]

\[
\langle \text{cmd}; s, \sigma_T, A_T, \text{nil} \rangle, T, A \rightarrow \langle s, \sigma'_T, A_T, \text{nil} \rangle, T, A \quad \text{if } \sigma'_T \in \llbracket \text{cmd} \rrbracket \sigma_T
\]

\[
\langle o.m; s, \sigma_T, A_T, \text{nil} \rangle, T, A \rightarrow \langle s, \sigma_T, A_T, \text{diff}(A(o), A_T(o)) @ [o.m] \rangle,
\]
\[
T, A[o \mapsto A(o) :: m]
\]
Formal Model

\[ \langle \text{skip}, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \rightarrow T, A \]

\[ \langle \text{cmd}; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \rightarrow \langle s, \sigma'_\tau, A_\tau, \text{nil} \rangle, T, A \text{ if } \sigma'_\tau \in \llbracket \text{cmd} \rrbracket \sigma_\tau \]

\[ \langle o.m; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \rightarrow \langle s, \sigma_\tau, A_\tau, \text{diff}(A(o), A_\tau(o))\@[o.m] \rangle, T, A[o \mapsto A(o) :: m] \]

---

diff(m :: \ell_1, m :: \ell_2) = \text{diff}(\ell_1, \ell_2)

diff(nil, m :: \ell_2) = [m] :: \text{diff}(\text{nil}, \ell_2)

diff(nil, nil) = []
Formal Model

\[
\langle \text{skip}, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad T, A
\]

\[
\langle \text{cmd}; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad \langle s, \sigma'_\tau, A_\tau, \text{nil} \rangle, T, A \quad \text{if } \sigma'_\tau \in \llbracket \text{cmd} \rrbracket \sigma_\tau
\]

\[
\langle o.m; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad \langle s, \sigma_\tau, A_\tau, \text{diff}(A(o), A_\tau(o))@o.m \rangle, T, A[o \mapsto A(o) :: m]
\]

\[
\langle o.m; s, \sigma_\tau, A_\tau, \ell \rangle, T, A \quad \rightarrow \quad \langle s, \sigma'_\tau, A_\tau[o \mapsto A_\tau(o)@\ell], \text{nil} \rangle, T, A \quad \text{if } \sigma'_\tau = \text{apply}(\ell, \sigma_\tau)
\]

\[
\text{diff}(m :: \ell_1, m :: \ell_2) = \text{diff}(\ell_1, \ell_2)
\]
\[
\text{diff}(\text{nil}, m :: \ell_2) = [m] :: \text{diff}(\text{nil}, \ell_2)
\]
\[
\text{diff}(\text{nil}, \text{nil}) = []
\]
Formal Model

\[
\langle \text{skip}, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad T, A
\]

\[
\langle \text{cmd}; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad \langle s, \sigma'_\tau, A_\tau, \text{nil} \rangle, T, A \quad \text{if } \sigma'_\tau \in [\text{cmd}]\sigma_\tau
\]

\[
\langle o.m; s, \sigma_\tau, A_\tau, \text{nil} \rangle, T, A \quad \rightarrow \quad \langle s, \sigma_\tau, A_\tau, \text{diff}(A(o), A_\tau(o))@[o.m] \rangle,
\]
\[
T, A[o \mapsto A(o) :: m]
\]

\[
\langle o.m; s, \sigma_\tau, A_\tau, \ell \rangle, T, A \quad \rightarrow \quad \langle s, \sigma'_\tau, A_\tau[o \mapsto A_\tau(o)@\ell], \text{nil} \rangle, T, A
\]
\[
\text{if } \sigma'_\tau = \text{apply}(\ell, \sigma_\tau)
\]

\[
\text{diff}(m :: \ell_1, m :: \ell_2) = \text{diff}(\ell_1, \ell_2)
\]
\[
\text{diff}(\text{nil}, m :: \ell_2) = [m] :: \text{diff}(\text{nil}, \ell_2)
\]
\[
\text{diff}(\text{nil}, \text{nil}) = []
\]

\[
\text{apply}(m :: \ell, \sigma) = \text{apply}(\ell, [m]\sigma)
\]
\[
\text{apply}(\text{nil}, \sigma) = \sigma
\]
Formal Model

• Operations must be **deterministic**. Since operations are replayed, outcomes must agree. A **non**deterministic specification is acceptable.

• Operations must be **total**. Must return a value (or return an exception); cannot block.
Linearizability

• **Theorem:** All operations are linearizable.

• **Proof:** Trivial.
  Lin point is rule OI where intended operation is enqueued. When any other thread access the object in OC it will construct an equiv copy by applying all outstanding operations (all of which must complete b/c operations are deterministic and total).
Limitations

- Single-operation transactions
- Long-running operations have to be replayed
- Not yet evaluated experimentally.
  Though previous work (Baumann et al., Multikernel, SOSP 2009) shows replication better than sharing objects inside operating systems
Conclusion

• Avoid complex concurrent data structures
• Simple Theory
• Linearizability for free
• Generalized to multi-ops leads us back to semantics of Coarse Grained Transactions (optimistic semantics)
Related Work