The Ordering Requirements of Relativistic and Reader-Writer Locking Approaches to Shared Data Access

Philip W. Howard

with

Jonathan Walpole, Josh Triplett, Paul E. McKenney
Outline

• The Problem
• The RWL Solution
• The RP Solution
• Other problems and their solutions
• Multiple Writers
• Performance
The Problem

A → B → C

A → C

11/2/2011
The Problem

Reader 1
- obtain ref
- drop ref

Writer
- unlink
- reclaim

Reader 2
- obtain ref
- drop ref
void init(stuff *node) {
    node->a = COMPUTE_A;
    node->b = COMPUTE_B;
    node->initd = TRUE;
}

while (!node->initd) {
}

if (node->a) drop_nuke();
if (node->b) drop_napalm();
The Problem

```c
void init(stuff *node) {
    int value;
    value = some_computation;
    node->a = COMPUTE_A;
    node->b = COMPUTE_B;
    node->initd = value;
}
```
The Problem

```c
void init(stuff *node) {
    node->a = COMPUTE_A;
    node->b = COMPUTE_B;
    node->initd = TRUE;
}
```

```c
while (!node->initd) {}
   if (node->a) drop_nuke();
   if (node->b) drop_napalm();
```
The Problem

• Compiler reordering
• CPU reordering
• Memory System reordering
How are these dependencies maintained?

Reader 1

obtain ref → drop ref

Writer

unlink → reclaim

Reader 2

obtain ref → drop ref
How does \( \rightarrow \) work?

Thread 1

\textbf{A}: \ a=1;

Thread 2

\textbf{B}: \text{if (a)}

\textbf{A} \rightarrow \textbf{B}

\textbf{else}

\textbf{B} \rightarrow \textbf{A}
How does $\Rightarrow$ work?

Thread 1

A: a=1;
mb();
C: c=1;

Thread 2

while (!a)
B: A $\Rightarrow$ B

Thread 3

if (c)
D: A $\Rightarrow$ D
With Reader-Writer Locks

read-lock → obtain ref → drop ref → read-unlock

write-lock → unlink → reclaim → write-unlock
With Reader-Writer Locks

read-lock → obtain ref → drop ref → read-unlock

write-lock

unlink → reclaim → write-unlock
With Reader-Writer Locks

write-lock → unlink → reclaim → write-unlock

obtain ref → drop ref → read-unlock

read-lock
Locking primitives must

- Impose the semantics of the lock
- Prevent compiler, CPU, or Memory system from reordering operations across them
With Relativistic Programming

start-read → obtain ref → drop ref → end-read

unlink → start-wait → end-wait → reclaim
With Relativistic Programming

start-read ➔ obtain ref ➔ drop ref ➔ end-read

unlink ➔ start-wait ➔ end-wait ➔ reclaim
RP primitives must

- Impose the semantics of wait-for-readers

- Prevent compiler, CPU, or Memory system from reordering operations across them
Outline

• The Problem
• The RWL Solution
• The RP Solution
• Other problems and their solutions
  • Insert
  • Move Down
  • Move Up
  • General Case
• Multiple Writers
• Performance
Insert

C → E

↓

C → D → E
Insert

Reader 1
obtain ref → deref

Writer
init → link

Reader 2
obtain ref → deref
How does ⇒ work?

Thread 1

A: a = 1;
   mb();
C: c = 1;

Thread 2

while (!a)
B: A ⇒ B

Thread 3

if (c)
D: A ⇒ D
Insert

Use `rp-publish` to perform link operation
Move Down
Move Down

1. init F’
2. link F’
3. unlink F
Move Down

- init(F')
- link(F')
- unlink(F)
- reclaim(F)

- deref(H)
- deref(F)
- deref(D)
- deref(E)
Move Down

init(F') → link(F') → unlink(F) → reclaim(F)

deref(H) → deref(F) → deref(D) → deref(F') → deref(E)

11/2/2011
Move Down

init(F') → link(F') → unlink(F) → reclaim(F)

deref(H) → deref(D) → deref(F') → deref(E)
RBTree Delete with Swap (Move Up)
Move Up

init(C') \rightarrow \text{link}(C') \rightarrow \text{unlink}(C) \rightarrow \text{reclaim}(C)

deref(F) \rightarrow deref(C')
Move Up

init(C') → link(C') → unlink(C) → reclaim(C)

deref(F) → deref(B) → deref(E) → deref(C)

A
  /  
B   C
  /   
A    E
  /    
null  D

F

B
  /  
A   E
  /   
C
    / 
null D
The General Case

• Mutable vs. Immutable data
The General Case for Writers

- Copy nodes to update immutable data or to make a collection of updates appear atomic
- Use rp-publish to update mutable data
- Use wait-for-readers when updates are in traversal order

A → B → C
The General Case for Readers

• Use rp-read for reading mutable data
• Only dereference mutable data once

```c
if (node->next->key == key) {
    return node->next->value;
}
```
Outline

• The Problem
• The RWL Solution
• The RP Solution
• Other problems and their solutions
• Multiple Writers
• Performance
Multiple Writers
Multiple Writers

W1

W2

Reader
The phone company

Can’t wait to get my new phone

Can’t wait to ditch this phone

Where’s my phone book?

CUSTOMER SERVICE
Multiple Writers

W1

wait-for-readers

W2

Reader

W1

wait-for-readers

W2

Reader
RP vs RWL delays

RP

W1 W2
Reader

TORP

W1 W2
Reader

reader pref

W1 W2
Reader

writer pref

W1 W2
Reader
Trigger Events

<table>
<thead>
<tr>
<th>RP</th>
<th>RWLR</th>
<th>RWLW</th>
</tr>
</thead>
</table>

11/2/2011
Outline

• The Problem
• The RWL Solution
• The RP Solution
• Other problems and their solutions
• Multiple Writers
• Performance
How does $\Rightarrow$ work?

Thread 1

A: a=1;
mb();
C: c=1;

Thread 2

while (!a)
B: A $\Rightarrow$ B

Thread 3

if (c)
D: A $\Rightarrow$ D
Reader-Writer Locks

read-lock → read-unlock

write-lock → write-unlock

read-lock → read-unlock
Reader-Writer Locks

Reader

while (writing) {
    reading = 1;
}

Writer

while (reading) {
    writing = 1;
}
Relativistic Programming

start-read \rightarrow end-read

start-wait \rightarrow end-wait

start-read \rightarrow end-read
Relativistic Programming

start-read(i)
{
    reader[i]=1;
}

end-read(i)
{
    reader[i]=0;
}

start-wait()
{
}

end-wait()
{
    for (i) {
        while (reader[i]) {}
    }
}
start-read(i)
{
    reader[i]=Epoch;
}

end-read(i)
{
    reader[i]=0;
}

start-wait()
{
    Epoch++; my_epoch = Epoch;
}

end-wait()
{
    for (i) {
        while (reader[i] &&
               reader[i] < my_epoch)
        {}
    }
}
Performance Benefits to RP

• Less expensive read-side primitives
• Less serialization / more concurrency
• Less waiting
Benchmarked Methods

nolock  No synchronization
rp      Relativistic Programming
torp    Totally Ordered RP
rwlwr   Reader-Writer Lock Read preference
rwlw    Reader-Writer Lock Write preference
Read Performance (size=1)

The graph depicts the read performance in operations per second (operations/sec) in millions as a function of the number of threads. The performance metrics are categorized into four types:

- **NOLOCK** represented by red diamonds
- **RP** represented by black stars
- **RWLW** represented by cyan crosses
- **RWLR** represented by green dots

As the number of threads increases, the performance for all categories generally improves, with NOLOCK showing the highest performance, followed by RP, RWLW, and RWLR. The performance curves are linear, indicating a direct proportionality with the number of threads.
Read Performance (size=1000)

The graph shows the read performance in millions of operations per second as a function of the number of threads. The graph includes four lines, each representing a different locking strategy:

- **NOLOCK**: Red line
- **RP**: Purple line
- **RWLW**: Cyan line
- **RWLR**: Blue line

The x-axis represents the number of threads, ranging from 0 to 20. The y-axis represents the number of operations per second, ranging from 0 to 3.5. The data points for each strategy show a linear trend, indicating proportional performance improvements with increasing thread counts.
Update Scalability

Operations/sec vs. List Size

- RP writes
- TORP writes
Update Scalability part 2

![Graph showing the scalability of operations per second (Operations/sec) versus list size. The graph compares two methods, RP and TORP, with lines indicating the performance over different list sizes.](chart.png)
Update Scalability (part 3)

![Graph showing the relationship between list size and operations per second for TORP, RWLW, TORP r, and RWLW r.](image-url)
Conclusions

• Correctness can be preserved by limiting allowable orderings
• RP read primitives are less expensive than RWL primitives
• RP allows more concurrency and less waiting
Conclusions

• RP can preserve both correctness and scalability or reads