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

PSU CS 533
Philip Howard
March 14, 2012
Scalability

Operations/sec vs. Threads
Amdahl’s Law

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

Where:

- \( S(N) \): Speedup with \( N \) threads
- \( P \): parallel part
- \( (1 - P) \): serial part
Computation Components

- Serial part \( S(N) = 1 \)
- Parallelizable part \( S(N) = N \)
- Synchronization part \( S(N) = \frac{1}{N} \)
Synchronization Approaches

- Coarse Grained Locking
- Fine Grained Locking
- Reader-Writer Locking (RWL)
Linked List Performance

Reads/sec

Millions

Threads

small list

large list

3/14/2012
Reader-Writer Lock Implementation

read_lock()
{
    while (writer) {} readers++;
}

read_unlock()
{
    readers--;  
}

write_lock()
{
    while (readers>0) {}
    writer=TRUE;
}

write_unlock()
{
    writer = FALSE;
}
Solution

• Atomic read-modify-write instructions
  • Serialize access to lock variable
  • Orders of magnitude slower than “standard” instructions
• \( S(N) = \frac{1}{N} \)
Linked List Performance

Reads/sec

Threads

small list
large list

3/14/2012
Another way to slice the pie

Relativistic Programming
Relativistic Programming

• Writers don’t exclude readers

• Readers don’t require atomic RMW instructions

• Impose only minimum ordering constraints
Reordering

```c
a = A;
b = B;
if (b == B) {
    c = a;
} else {
    c = A;
}

// c == ?
```
The problem

• Programmers carefully construct orders
• Optimizing compilers
• Out Of Order Execution Units
• Memory Systems
• Can’t assume programs execute in program order
Important Ordering
if \( \text{link} \rightarrow \text{obtain ref} \) then \( \text{init} \Rightarrow \text{deref} \)
How do we implement Required-Before?

- Dependency Ordering
- Wait
required-before
dependency ordering

write(a)
barrier
write(b)

read(b)
barrier
read(a)
required-before
dependency ordering

write(a)
barrier
write(b)

node = read(b-next)
barrier
read(node->value)
required-before

wait

writer()
{
A: write(a);
}

waiter()
{
wait for (A);
barrier();
C:
}
Reader-Writer Lock Implementation

read_lock()
{
    while (writer) {}
    readers++;
}

read_unlock()
{
    readers--;
}

write_lock()
{
    while (readers>0)
    {
    }
    writer=TRUE;
}

write_unlock()
{
    writer = FALSE;
}
Required-before Ordering
Mechanisms

- Dependency Ordering
  - Waits for memory system
  - Inexpensive

- Wait
  - Waits on program logic
  - Waits on memory system
  - Expensive
Primitives

RWL
• read-lock
• read-unlock
• write-lock
• write-unlock

RP
• rp-publish
• rp-read
• wait-for-readers
• start-read
• end-read
• rp-free
wait-for-readers
wait-for-readers

start-read
end-read

start-wait
end-wait

start-read
end-read
RP Ordering

store  store  store  store  store  store
store  store  store  store  store  store
rp-publish  wait-for-readers
store  store  store  store  store  store
rp-publish  store  store  store  store  store
Case Studies

• Insert
• Delete
• Move Forward
• Move Back
Insert

 updater

 init(b);
 rp_publish(a->next, b);

 reader

 node = rp_read(a->next)
deref(node);
RWL Insert

read-lock
obtain-ref
deref
read-unlock

write-lock
init
link
write-unlock

read-lock
obtain-ref
deref
read-unlock
Delete
Delete

obtain ref

deref

unlink

reclaim

obtain ref
deref
wait-for-readers

start-read
end-read

start-wait
end-wait

start-read
end-read
Delete

**Updater**

```c
a->next = b->next;
wait-for-readers();
reclaim(b);
```

**Reader**

```c
start-read();
node = a->next;
deref(node);
end-read();
```
Delete

start-read
obtain ref
deref
end-read

unlink
start-wait
end-wait
reclaim

start-read
obtain ref
deref
end-read
Move Forward
Move Forward

Visible states

- old position
- new position
- neither position
- both positions

- one copy
- multiple copies
Move Forward
Move Forward

deref(A) deref(C) deref(B') deref(D)

init(B’) link(B’) unlink(B) reclaim(B)

deref(A) deref(C) deref(B’) deref(D)
When is “both” OK?

• Map vs. Multimap
  • Directory information
  • File system
Move Back

A → B' → C → D

B

3/14/2012
Move Back

deref(A) → B′ → C → D

deref(A)
deref(C)
deref(B)
deref(D)

init(B′)
link(B′)
wait-for-readers()
unlink(B)
reclaim(B)
Traversing Order Rule

- if changes are in traversal order: *wait*
- if changes in reverse traversal order: *dependency ordering*
Case Studies

- Insert dependency ordering
- Delete wait (async)
- Move Forward dependency ordering
- Move Back wait
RP Ordering

• Insert, Delete, Move Forward, Move Back can be used to build a wide range of data structures
  • Red-Black Tree
  • AVL Tree
  • Hash Table Move
  • Hash Table Resize

• RP gives access to light weight *dependency order* mechanism as well as heavy weight *wait* mechanism
• RP *wait* mechanism allows low cost read implementation
• RP allows more concurrency than RWL
Relativistic RBTree

![Graph showing the performance of different lock protocols]

- Red line: no lock (nolock)
- Blue line: read lock (rp)
- Orange line: read lock (rw)
- Green line: read lock (rwlr)
- Pink line: lock-free (ccav1)

The graph plots the number of reads per second (in millions) against the number of concurrent readers. The x-axis represents the number of concurrent readers, while the y-axis represents the reads per second.
Relativistic RBTree

![Graph showing performance metrics for Relativistic RBTree with various concurrency levels and read/write operations. The graph plots reads/second (millions) against concurrent readers. The graph includes lines representing different scenarios labeled as 'uncontended rp', 'contended rp', 'uncontended ccavl', 'contended ccavl', 'uncontended rwlw', and 'contended rwlw'.]
Relativistic RBTree
Conclusion

• RP allows fast scalable reads
• RP allows more concurrency than RWL
• RP has simple rules
• RP runtimes in both kernel and user mode
Questions/Discussion

Phil W. Howard
phil.w.howard@gmail.com
http://sites.google.com/site/philwhoward/
Backup Slides
wait-for-readers implementation

start-read() {
    reader[me] = Global_Epoch;
}

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

wait-for-readers() {
    epoch = atomic_inc(Global_Epoch);
    forall (readers) {
        while (reader[ii] != 0 && reader[ii] <= epoch) {}
    }
}

3/14/2012
Case Studies: Delete

if $obtain\ ref \rightarrow unlink$ then $drop\ ref \Rightarrow reclaim$
Move Forward

deref(A) deref(B) deref(C) deref(D)
deref(A) deref(B) deref(C) deref(D')

A -> B -> C -> D

B'