CS510 Concurrent Systems

Jonathan Walpole

Portland State University

A Methodology for Implementing Highly Concurrent Data Objects

Concurrent Object

- A data structure shared by concurrent processes
- Traditionally implemented using locks
- Problem: in asynchronous systems slow/failed processes impede fast processes
- Alternative approaches include non-blocking and wait-free concurrent objects

  ... but these are hard to write!

  ... and hard to understand!

Goals

Make it easy to write concurrent objects (automatable method)

Make them as easy to reason about as sequential objects (linearizability)

Preserve as much performance as possible

The Plan

Write sequential object first

Transform it to a concurrent one (automatically)

Ensure it is linearizable

Use load-linked, store conditional for non-blocking

Transform non-blocking implementation to wait-free implementation when necessary

Non-Blocking Concurrency

- Non-blocking: after a finite number of steps at least one process must complete
- Wait-free: every process must complete after a finite number of steps
- A system that is merely non-blocking is prone to starvation
- A wait-free system protects against starvation

The Methodology

- Sequential objects must be total, i.e., well defined on all valid states of the data
- It must also have no side effects
- Why?

Load-Linked Store Conditional

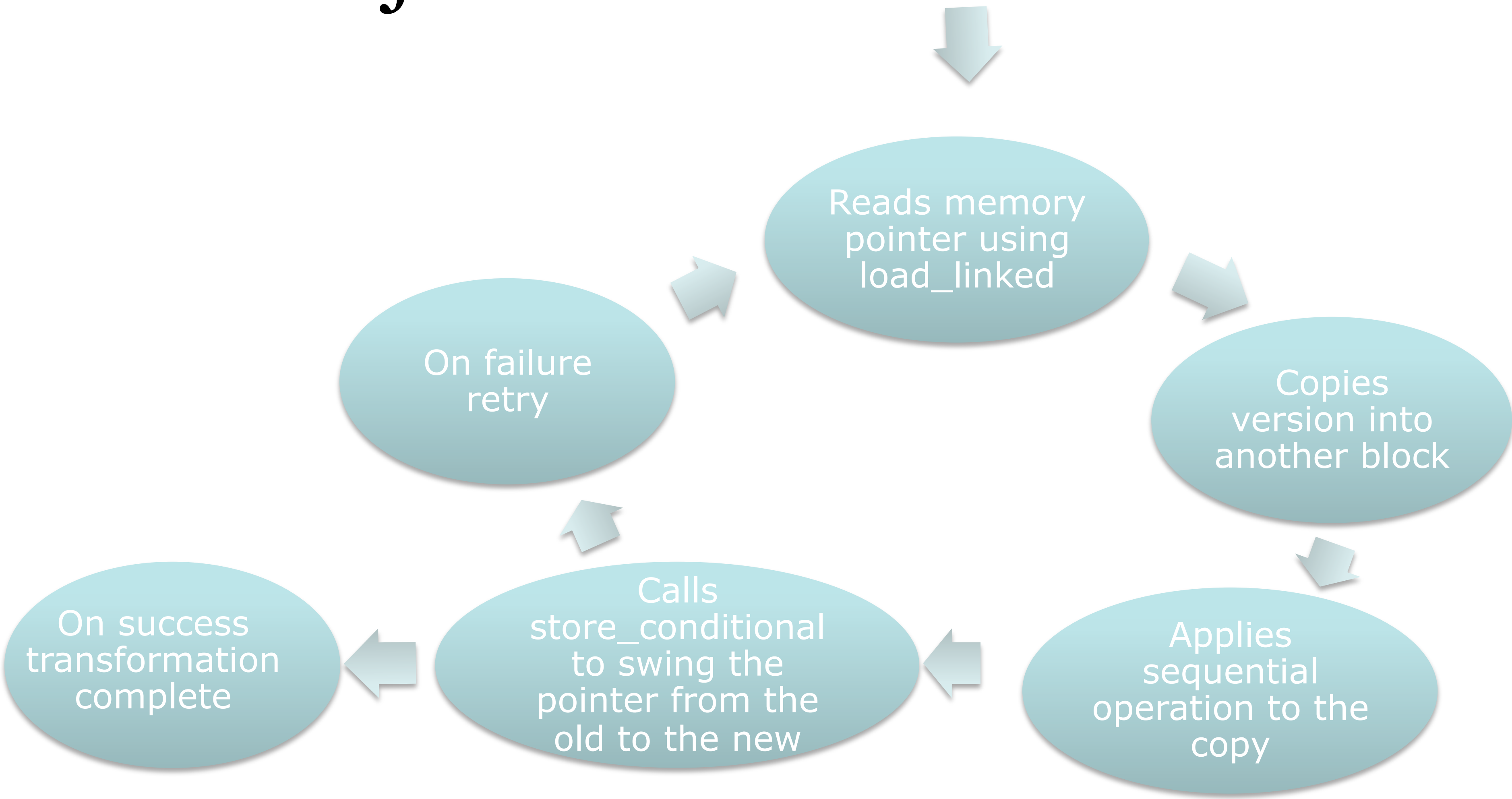
Load-linked copies a memory location

Store conditional stores to a memory location, only if the location load-linked has not been accessed

No ABA, but it can suffer from spurious failures!

- for example, due to cache line replacement
- or execution of a new load-linked on a different location

Small Object Method



Is This Safe?

What if one thread reads while another is modifying, can the copy be corrupted?

- to prevent access to incomplete state, two version counters are used (check[0] and check[1])
- Updates: a thread updates check[0], then does the modification, then updates check[1]
- Copies: a thread reads check[1], copies the version, then reads check[0]

Small Objects Cont. - Code

```
int Pqueue_deq(Pqueue_type **Q){
    ...
    while(1){
        old_pqueue = load_linked(Q);
        old_version = &old_pqueue->version;
        new_version = &new_pqueue->version;
        new_pqueue->check[0] = new_pqueue->check[1] + 1;
        first = old_pqueue->check[1];
        copy(old_version, new_version);
        last = old_pqueue->check[0];
        if (first == last){
            result = pqueue_deq(new_version);
            new_pqueue->check[1] = new_pqueue->check[0] + 1;
            if (store_conditional(Q, new_version))break;
        }
    }
    new_pqueue = old_pqueue;
    return result;
}
```

If the check values DO match, now we can perform our dequeue operation!

Try to publicize the new heap via **store_conditional**, which could fail and we loop back.

Preventing the race condition!

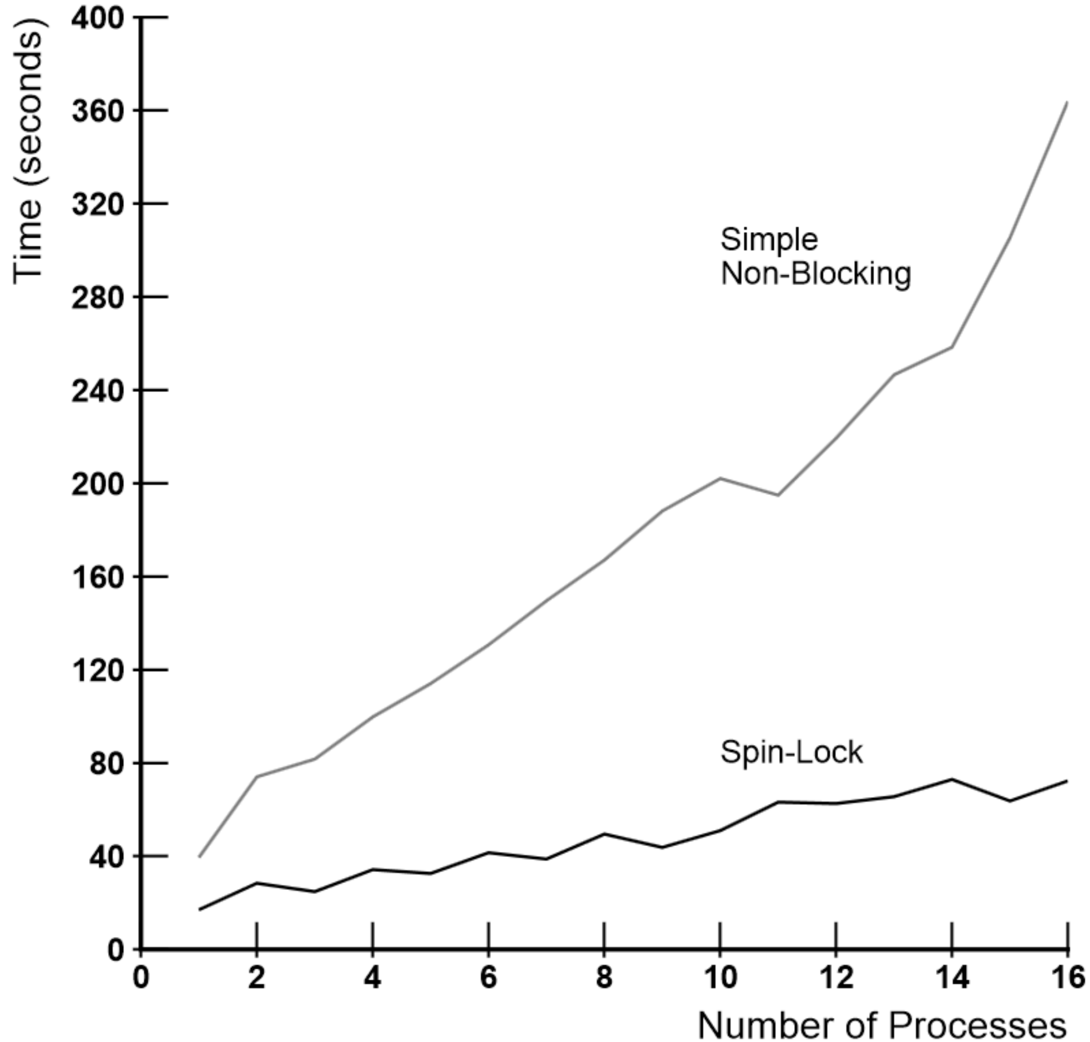
Copy the old, new data.

If the check values do not match, loop again. We failed.

Reclaim the memory!

Return our priority queue **result**.

Simple Non-Blocking Algorithm vs. Spin-Lock



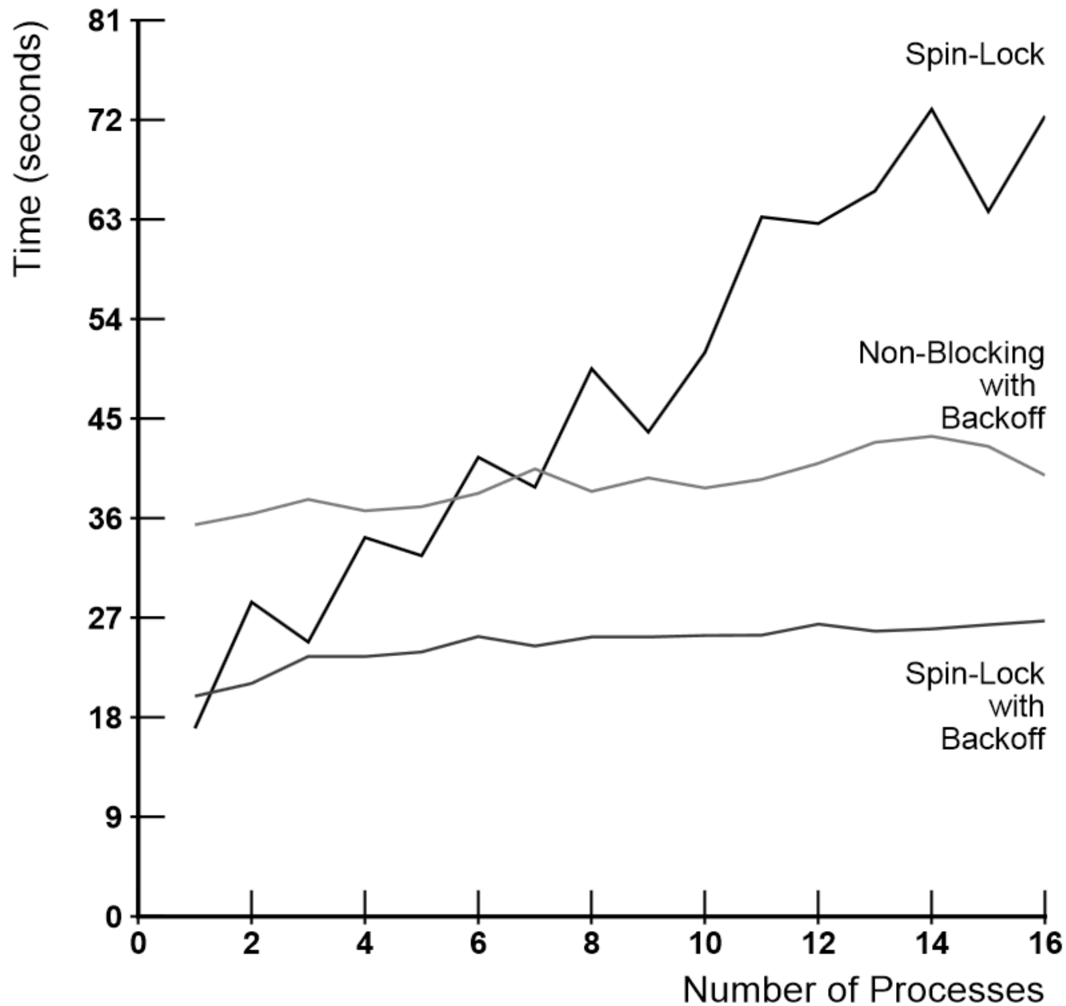
Why Does It Perform So Poorly?

Small Objects Cont. – Back Off

```
        ...
        if (first == last)
        {
           result = pqueue_deq(new_version);
           if (store_conditional(Q, new_version )) break;
        }
        if (max_delay < DELAY_LIMIT) max_delay = 2 * max_delay;
        delay = random() % max_delay;
        for (i = 0; i < delay; i++);
   } /* end while*/
   new_pqueue = old_pqueue;
   return result;
}
```

When the consistency check or the store_conditional fails, introduce back-off for a random amount of time!

Effect of Adding Backoff



Why Does It Still Perform Poorly?

Other Problems

Long operations struggle to finish

They are repeatedly forced to retry by short operations

The approach is subject to starvation!

How can we fix this?

Small Objects – Wait Free

Each process declares its intended operations ahead of time, using an invocation structure

- name, arguments, and toggle bit to determine if invocation is old or new

The results of each operation are recorded in a response structure

- result value, toggle bit

Every process tries to do every other processes operations before doing its own!!!!

- but only one succeeds (exactly one succeeds!)

Small Objects – Wait Free

```
announce[P].op_name = DEQ_CODE;
new_toggle = announce[P].toggle = !announce[P].toggle;
if (max_delay> 1) max_delay = max_delay >> 1;

while(((*Q)->responses[P].toggle != new_toggle)
          || ((*Q)->responses[P].toggle != new_toggle)){
          old_pqueue = load_linked(Q);
          old_version = &old_pqueue->version;
          new_version = &new_pqueue->version;
          first = old_pqueue->check[1];
          memcopy(old_version, new_version, sizeof(pqueue_type));
          last = old_pqueue->check[0];
          if (first == last){
             result = pqueue_deq(new_version);
             apply(announce, Q);
             if (store_conditional(Q, new_version )) break;
          }
          if (max_delay < DELAY_LIMIT) max_delay = 2 * max_delay;
          delay = random() % max_delay;
          for (i = 0; i < delay; i++);
    }
    new_pqueue = old_pqueue;
    return result;
}
```

Record your operation

Flip the toggle bit

?

apply all pending operations to the NEW version

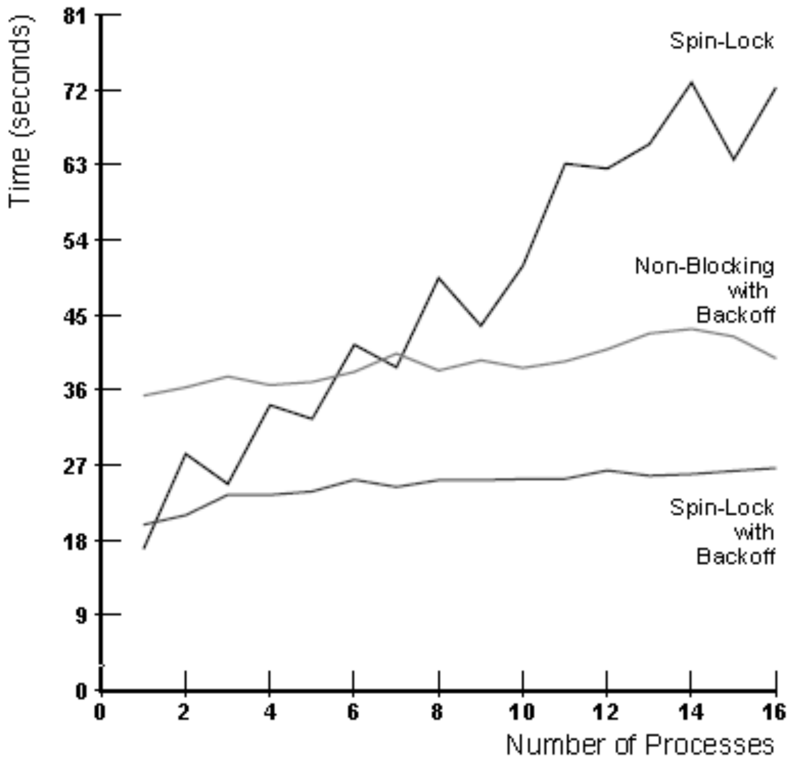
Commit the new version

Doing The Work of Others

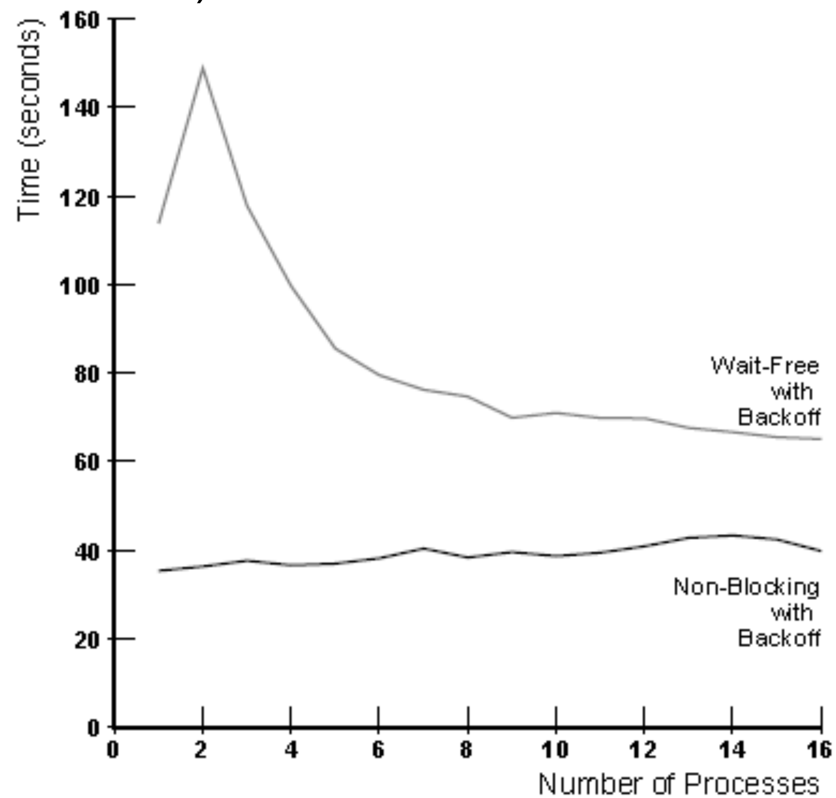
```
void apply (inv_type announce[MAX_PROCS], pqueue_type *object){
    int i;
    for (i = 0; i < MAX_PROCS; i++){
          if(announce[i].toggle != object->res_types[i].toggle){
            switch(announce[i].op_name){
                    case ENG_CODE:
                      object->res_type[i].value =
                        pqueue_enq(&ojbect->version, announce[i].arg);
                      break;
                    case DEQ_CODE:
                      object->res_type[i].value =
                        pqueue_deq(&ojbect->version, announce[i].arg);
                      break;
                    default:
                      fprintf(stderr, "Unknown operation code \n");
                      exit(1);
            };
          object->res_types[i].toggle = announce[i].toggle;
          }
    }
}
```

Small Objects Cont. – Wait Free

Small Object, Non-Blocking (back-off)



Small Object, Wait Free (back-off)



Wait-Free Performance Sucks!

Why?

Large Concurrent Objects

- Cannot be copied in a single block
- Represented by a set of blocks linked by pointers
- The programmer is responsible for determining which blocks of the object to copy
- The less copying the better the performance

Summary & Contributions

Foundation for transforming sequential implementations to concurrent ones

- uses LL and SC
- simplifies programming complexity
- could be performed by a compiler
- maintains a "reasonable" level of performance?