Microsoft

Why lock-free synchronization?

A practitioner's perspective

Aleksandar Dragojević
Microsoft Research Cambridge

alekd@microsoft.com

Microsoft Research Cambridge

https://careers.microsoft.com/

FaRM

A distributed computing platform

A shared distributed address space with transactions

Strong consistency

Strictly serializable transactions with opacity

High performance

Millions of operations per second

A lot of work on efficient concurrency

Crucial to achieve good performance of the whole system

Honeycomb

Hardware acceleration of high-level abstractions

Build distributed systems in custom hardware

Work in progress

Promising preliminary results

Again a lot of work on synchronization

A very interesting problem of synchronizing hardware and software

EPFL

Software transactional memory

Simple abstraction with good performance

Users only define transactions in their code, runtime executes them

Fine-grained synchronization in the runtime

A very hard problem as every cycle counts

Synchronization is at the core of the system

Synchronization is easy

Just use one lock for all synchronization

```
the_lock.acquire()

do anything()

the_lock.release()
```

```
void synchronized do_anything() {
        // do the work
}
```

Fine-grained synchronization is hard

“Most code is mostly thread-safe”

“Using pointers read in a previous transaction is safe because it has worked fine so far”

What is the goal?

Performance

And scalability – performance with the number of threads

Correctness is not the goal

It is a pre-requisite

For simple correctness, use one lock

It is much simpler

What does performance depend on?

Waiting or retrying

On locks or because validation fails

Cache misses

Accessing data in other core’s cache is expensive (an L3 cache miss takes ~100ns)

Writes are more expensive than reads

Atomic operations

Compare-and-swap, fetch-and-add, memory ordering fences, …

CPU pipeline flushes

What does this mean?

Short critical sections

Fine-grained locking or lock-free sections

As little shared data as possible

Thread-local data as much as possible

Few atomic operations

A few locks and compare-and-swap operations

Where does lock-free synchronization fit?

Not a clear performance winner

Can increase the number of cache misses due to indirection

Reads are often faster

Avoid writing needed to acquire locks

Writes are often slower

More complex and more expensive

Really good to guarantee time bounds

No unpredictable blocking on locks held by other threads

Today’s plan – examples from practice

Unique identifier generator

MPSC messaging queue

Scalable IO

Atomic RDMA reads

Computation model

Many threads each running on its core
Sometimes called processes

Memory to store data
Consists of 64-bit words

Atomic operations
Compare-and-swap, fetch-and-add, read, write

Sequentially consistent
No re-ordering of operations on one thread

Problem 1: Unique Identifier Generator

Write a generate() function that returns a unique 64-bit identifier. The same identifier is not returned twice.

Sample execution

generate() → 0

generate() → 1

generate() → 2

generate() → 100

generate() → 99

~~generate() → 100~~

Can you propose a solution?

Use a counter.

Lock-based generator

```
Shared:
Lock the_lock
int64 next = 0
```

```
generate():
  the_lock.acquire()
  ret = next
  next = ret + 1
  the_lock.release()

  return ret
```

Are these identifiers really unique?

What about wrap-around?

In a computer, 0 comes after the maximum for the specified bit width

With 8 bits: 254, 255, 0, 1,…

64-bit numbers are practically infinite

Even smaller numbers might be sufficient, depending on the context

It takes 194 years to overflow a 64-bit number

With 3 billion updates per second, which is really fast

60,000 thousand years if we update once every 100 nanoseconds

Lock-free generator

```
Shared:                  generate():
int64 next = 0             return fetch_and_add(next, 1)
```

Which one is faster?

```
   generate():
AW the_lock.acquire()
 R ret = next
 W next = ret + 1
AW the_lock.release()

   return ret
```

```
   generate():
ARWreturn fetch_and_add(next, 1)
```

Lock-free is faster
It maps to hardware very well

Is this fast enough?

Thread identifiers at the program start

Any approach will be fast enough

Dynamic thread identifiers

100s thousand per second – still fast enough

Operation identifiers

Millions per second
Might become a scalability bottleneck

Problem 1a: Scalable Identifier Generator

Write a generate() function that returns a unique 64-bit identifier. The same identifier is not returned twice. *Generate() should scale to tens of millions of calls per second.*

Can you propose a solution?

Batching

When performing the fetch-and-add, generate() acquires a large number of identifiers e.g. 100 for the calling thread.

Generate() returns one of the identifiers in the batch, unless the batch is exhausted. Then it acquires a new batch using fetch-and-add.

Reduces contention on the shared counter significantly e.g. by a factor of 100.

Lock-free generator with batching

```
Shared:
int64 next = 0

Thread:
int64 batch_next = 0
int64 batch_last = 0
```

```
generate():
  if batch_next == batch_last:
    batch_next = fetch_and_add(next, 100)
    batch_last = batch_next + 100
  ret = batch_next
  batch_next = batch_next + 1
  return ret
```

Much better. Can we improve further?

Other than using a bigger batch.

Thread-local generate()

Assume we will not use more than 1024 threads and that we only need to generate a million identifiers per second per thread.

At start, acquire a thread identifier. This will be a 10-bit number.

Identifiers are formed from a next value of a thread-local counter as the top 54 bits and the thread identifier as the bottom 10 bits.

This requires very little synchronization.

Lock-free generator with batching

```
Shared:
int64 next_lsb = 0

Thread:
int64 lsb
int64 next_msb = 0
```

```
init():
  lsb = fetch_and_add(next_lsb, 1)

generate():
  msb = next_msb
  next_msb = next_msb + 1
  return (msb << 10) | lsb
```

Problem 1: Summary

Different ways of generating identifiers

Depending on the requirements of the workload

Batching

A useful technique to amortize the cost of synchronization

Splitting the identifier space among threads

Almost no synchronization, but causes fragmentation

Food for thought 1

What changes if identifiers need to be consecutive? In some cases, this is desirable e.g. when generating transaction commit versions. Having consecutive identifiers helps with keeping track of completed transactions which is useful for garbage collection.

Food for thought 2

Limit the number of objects of type T created by threads in the system to 1000.

Real problem: Reading objects from a database is expensive. To reduce the cost, we cache most recently read objects in machine’s main memory. We would like to keep the size of the cache limited to e.g. 1 GB to reduce the impact of caching.

Problem 2: MPSC messaging queue

Write a multi-producer, single-consumer queue to use for sending messages between threads.

The queue supports enqueue(m) operation that adds a new message to the queue. Enqueue() can be executed by any of the threads in the system.

The queue also supports dequeue_all() call which removes all the messages from the queue and returns them to the caller. Only a designated consumer thread can invoke this call. The messages are returned in the order in which they were enqueued to the queue.

Sample execution

enqueue(1)

enqueue(2)

dequeue_all() → (1, 2)

enqueue(3)

enqueue(4)

~~dequeue_all() → (4, 3)~~

Can you propose a solution?

Lock-based queue

```
struct msg {
  int64 data
  msg *next
}

Shared:
Lock the_lock
msg *head = null
msg *tail = null
```

```
enqueue(m):
  m.next = null
  the_lock.acquire()
  if head == null:
    head = m
    tail = m
  else:
    tail.next = m
    tail = m
  the_lock.release()
```

```
dequeue_all():
  the_lock.acquire()
  ret = head
  head = null
  tail = null
  the_lock.release()
  return ret
```

Why not use a lock-free queue?

Well-known queues exist

See Michael-Scott queue for a classic example

Harder to implement

We needed to add uncommon dequeue_all()

Performance not very important

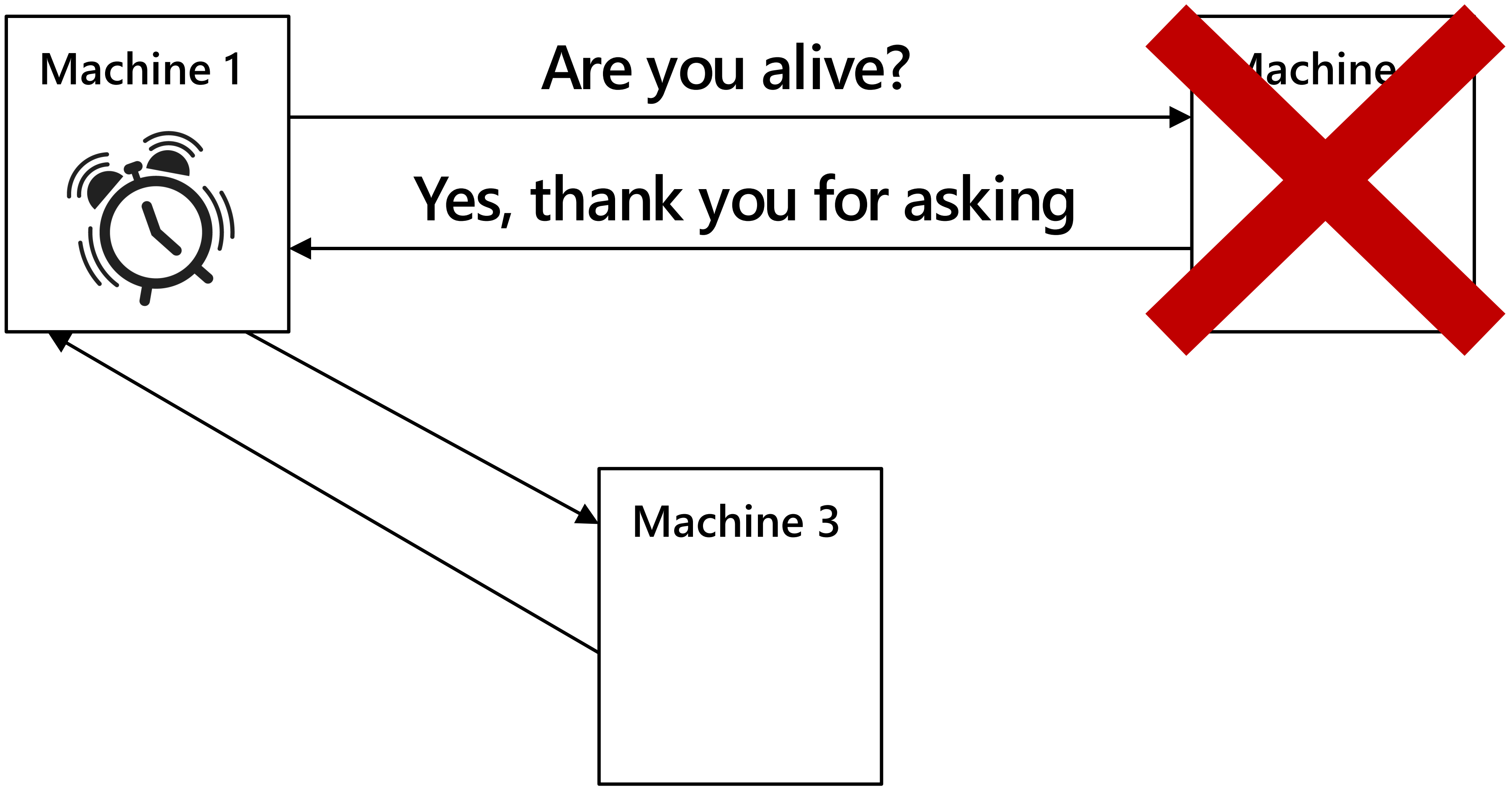
We expect tens of thousands of messages per second

We opted for the simple lock-based queue

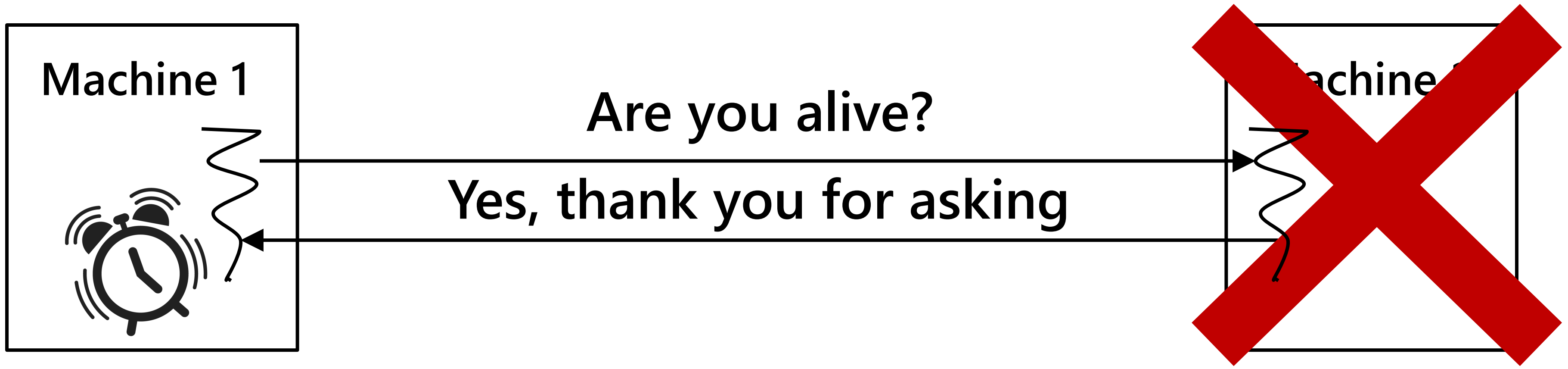
Until…

We decided to decrease failure-detection times in our system to 10 ms

Failure detection



Failure detection



Dedicated core

High-priority thread

If the failure detection thread gets blocked, we have a false (positive) failure

We need a lock-free messaging queue to avoid blocking on the queue

Lock-free dequeue_all()

We would like to avoid a complex queue and modifying an existing MPMC queue would not be trivial.

We use a stack instead of the queue. A lock-free stack is easy to implement. Items are added and removed by a compare-and-swap on the head of the stack.

To dequeue_all(), the thread atomically sets the head of the stack to null. It then flips the list it dequeued to get the correct order. The list flipping is thread-local, so it is fast.

Lock-free dequeue_all()

```
struct msg {
  int64 data
  msg *next
}

Shared:
msg *head = null
```

```
enqueue(m):
  while True:
    h = head
    m.next = h
    if cas(head, h, m) == h:
      return

dequeue_all():
  while True:
    h = head
    if cas(head, h, null) == h:
      return flip_list(h)
```

Lock-free dequeue_all() cont'd

```
flip_list(m):
  prev = null
  while m != null:
    next = m.next
    m.next = prev
    prev = m
    m = next
  return prev
```

Problem 2: Summary

Lock-freedom shines when there are deadlines

Failure detection, real-time processing

It is challenging

The first instinct is always to use locks, as this is simpler

Thread-local processing is scalable

No sharing

Food for thought 3

Wait-free MPSC queue would be even better in this case, as it would guarantee that operations would finish in constant time.

Can you propose a wait-free MPSC queue?

That is not based on a universal construction.

Food for thought 4

Modify the Michael and Scott queue to support a dequeue_all operation.

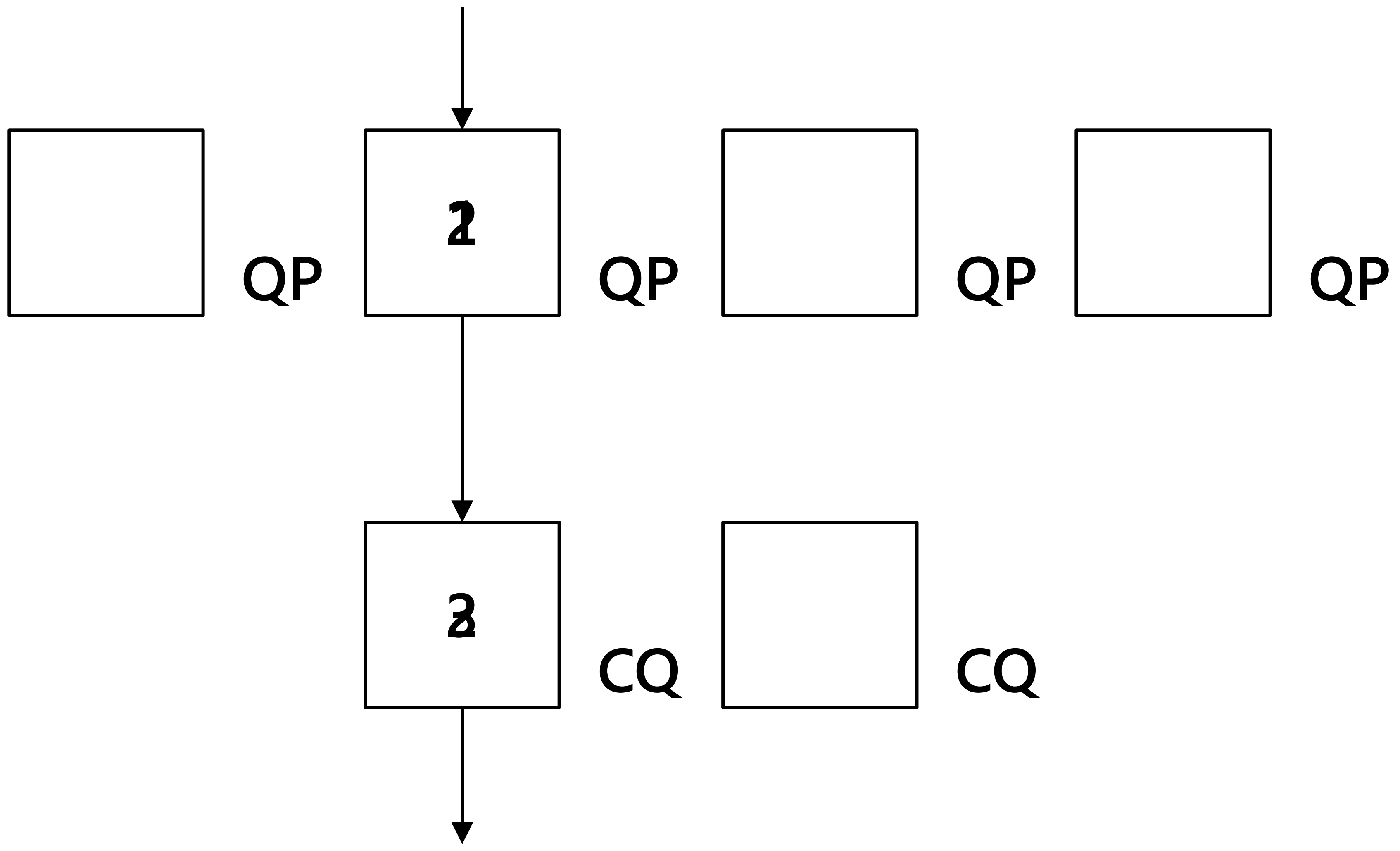
Problem 3: Scalable IO

Propose a scalable scheme for sharing connections on a network card (NIC) between threads.

Each request needs to acquire a slot in a queue pair (QP) and a completion queue (CQ). When completed, the request will release the slots in the QP and CQ. If there are no available slots, the request cannot proceed.

Focus on the synchronization on the queues. Assume that the code for interfacing with the NIC is given.

Sample execution



Can you propose a solution?

A challenging problem

Solution 1: global locking

Poor performance and scalability

Solution 2: mid-grained locking

Better performance, but still a bottleneck in the system

Solution 3: fine-grained locking

Further improvements

Locking scalable IO

What should we do with a request that cannot acquire resources? We use helping, as in lock-free algorithms. The request is queued at the QP or CQ and the thread that is issuing it can proceed to do other work. When a request completes, it helps queued requests make progress.

Lock a queue with the counter with fine-grained locks.

We opted for locks for simplicity. Lock-free synchronization becomes hard when multiple words need to be updated atomically.

Locking scalable IO

```
struct req {
  resource *qp
  resource *cq
  req *next
}

struct resource {
  int64 available
  req *head = null
  req *tail = null
}
```

```
Shared:
resource qps[]
resource cqs[]
```

```
enqueue(req, res):
  if res.tail == null:
    res.head = req
    res.tail = req
  else:
    req.next = res.tail
    res.tail = req
```

```
is_empty(res):
  return res.head == null
```

```
dequeue(res):
  if res.head = null:
    return null
  ret = head
  head = head.next
  if head == null:
    tail = null
  return ret
```

Locking scalable IO cont’d

```
perform_io(r):
  if !reserve(r, r.qp)
    return
  if !reserve(r, r.cq)
    return
  do_perform_io(r)
```

```
reserve(r, q):
  q.lock.acquire()
  if q.available > 0:
    q.available--
    ret = True
  else:
    enqueue(r, q)
    ret = False
  q.lock.release()
  return ret
```

Locking scalable IO cont'd

```
complete_io(r):
  release_cq(r)
  release_qp(r)
```

```
release_cq(r):
  r.cq.lock.acquire()
  if is_empty(r.cq):
    h = null
    r.cq.available++
  else:
    h = dequeue(r.cq)
  r.cq.lock.release()
  do_perform_io(h)
```

```
release_qp(r):
  r.qp.lock.acquire()
  if is_empty(r.qp):
    h = null
    r.qp.available++
  else:
    h = dequeue(r.cp)
  r.qp.lock.release()
  if !reserve(h, h.cq)
    return
  do_perform_io(h)
```

We are done.   Or are we?

The NIC we are using guarantees in-order execution of requests on the same QP. Because different threads can help requests on the same QP, re-ordering can happen.

We allow requests to execute out-of-order, but complete them in order. These semantics were fine for our system. Each thread keeps a local queue of requests in the order in which they were issued. When the NIC reports request as done, it is completed only if it is at the head of the queue. Otherwise, it waits until it becomes gets to the head.

Problem 3: Summary

Fine-grained locking with helping

Using lessons learned from lock-free synchronization

Helped increase performance significantly

25% improvement in system performance compared to prior scheme

Hard problem for lock-free synchronization

IO is typically not lock-free

Food for thought 5

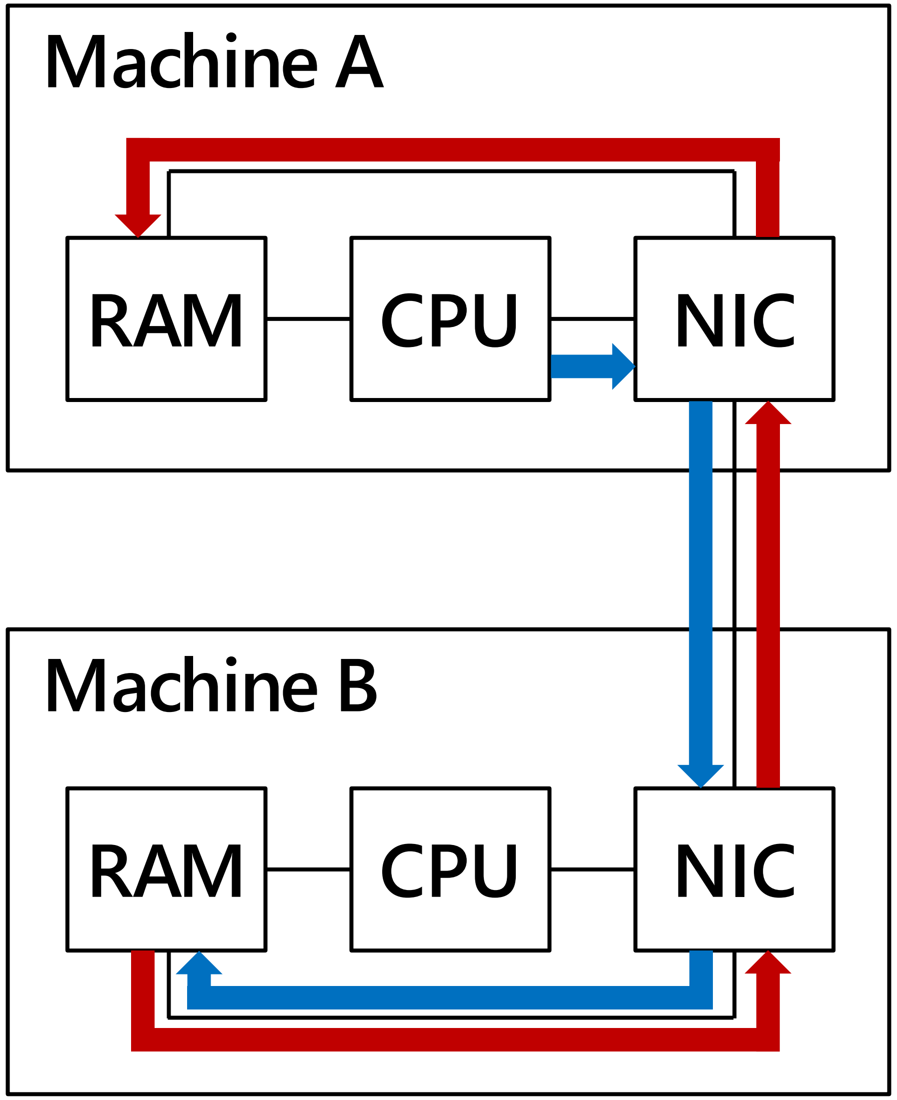
Can you propose a lock-free solution to the scalable IO problem?

Problem 4: Atomic RDMA reads

A machine can read objects from memory of remote machines by issuing remote read operations. The read is performed by the NIC on the remote machine.

NIC reading is not atomic with respect to updates on the CPU, because the object can consist of multiple consecutive words in memory . Propose a design that ensures read atomicity even in the presence of writes.

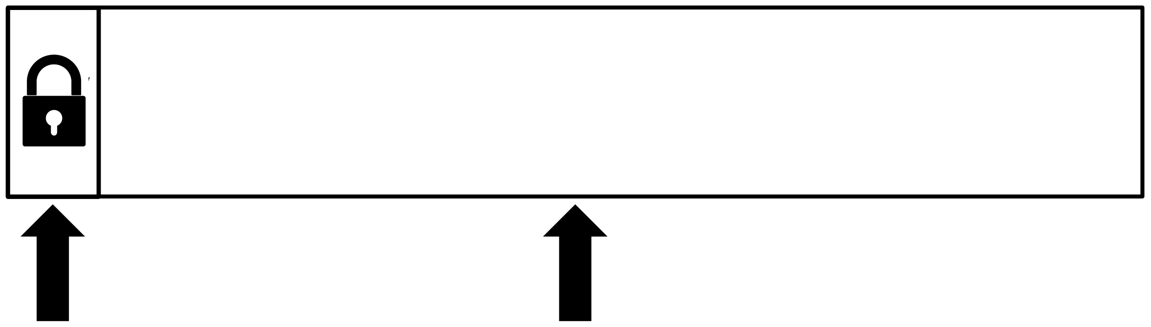
RDMA read



Different to local:

- High latency
- No cache coherence

How to do this on a single machine?



Consistent if version is unchanged and object is not locked

Lookup

Update

Can we just use this for remote reads?

Works very well on a local machine. With a successful read, the second header read hits in the CPU cache, so it is very fast. Also, latency to memory is short, so multiple reads are not a big overhead.

There is no cache coherence for data read over RDMA, so all reads are done over the network. Even consistent reads would do two network round-trips. Further, to order the first read of the header and data read after it, we need to issue two RDMA reads, bringing the total to three. This is very expensive.

We would like to perform a read with a single network round-trip.

Lock-free techniques are great here

Locking and unlocking would incur at least two network round-trips.

Here we didn’t even start with a locking scheme because of that.

Lock-free synchronization shines when synchronizing hardware and software because of high communication overheads.

Memory model is different

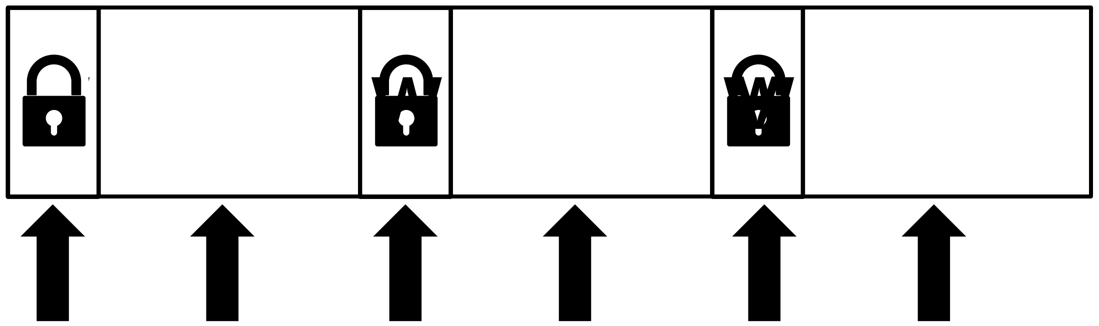
A read over the network will read a block of data. The size is specified in the request. For simplicity, let’s assume that each read is aligned to the cache line boundary and that it reads several cache lines. Cache line is 64 bytes on the machines we are using. So the read can be 64, 128, 192,… bytes in size.

Hardware effectively executes reads of different cache lines in the block in parallel.

This means that the NIC sees updates inside one cache line in the order in which they were performed, but it can observe updates in different cache line out of order.

Can you propose a solution?

We add versions to each cache line



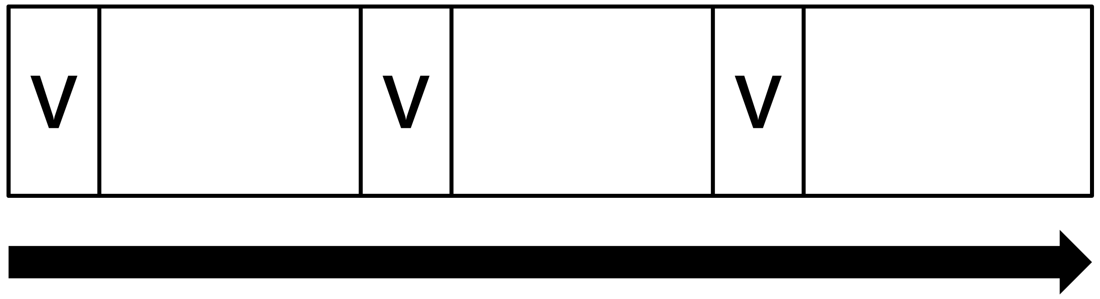
Increment version and unlock

Acquire all locks

Update data

Update

Read the whole object and check versions



Consistent if all versions are equal and nothing is locked

Remove the versions before passing the object to the user

Otherwise retry

This works with one read. Great!

But space overheads are very high.

We use 8 bytes of every 64-byte cache line to store a version. This is 12.5% space overhead just for consistency.

Introduce more assumptions

We assume that clock drift between machines is bounded.

Clocks on different machines tick at slightly different intervals. We assume that the difference between their rate of ticking is bounded.

Note that we don’t assume that clocks themselves are synchronized. Having synchronized clocks can be useful, but we don’t use it here.

This allows us to store 1-byte versions in all cache lines except the header.

Using the bounded clock drift

We make sure that every object update takes at least 200 ns. Object updates usually take longer than that, but we wait if necessary to ensure this is true.

When reading, we measure how long it takes for the read to complete. If it takes longer than it would to perform 256 updates, we retry the read. RDMA reads usually take less than 50 micro-seconds, so the retries for this reason are rare.

We account for the clock drift when measuring the RDMA read time.

Problem 4: Summary

Lock-free techniques work well

Optimistically reading reduces the read count in the common case

Several techniques for a complete solution

This is often the case in a complete system

Food for thought 6

Propose an alternative scheme for remote atomic reads that requires a single round-trip on the fast path.

Food for thought 7

What if NIC guaranteed it read from the lowest to the highest address? Can you propose a different approach to guarantee atomic reads using the in-order guarantee?

Summary

Lock-free techniques are very useful

Even in algorithms that are not completely lock-free

They can be hard to develop

Coarse-grain locking is often a good starting point

Synchronization is at the core of most systems

High-performance distributed systems, software-hardware synchronization,…

Extra slides

Food for thought 1

We cannot use batching or local identifier generation if identifiers need to be consecutive. In this scenario, the lock-free generate() is a good choice.

The shared variable can become the point of contention. In a large multi-socket machines, we could reduce the pressure on it by using hierarchical counters.

Food for thought 2

We can use a lock-free counter or a batched counter to solve this problem.

Partitioning items across threads upfront can also work, but unless the workload is very symmetric, it will fragment memory and result in a sub-optimal cache size.

Other techniques can work. For instance, we could pre-assign the maximum number of objects to each thread, but allow threads to take some of the available capacity of another thread if needed.

Food for thought 3

A single-producer, single-consumer bounded queue can be implemented in a wait-free manner.

A system of wait-free queues can be used to implement MPSC queue with one SPSC queue between each producer and the consumer.

Note: this approach can result in message reordering. Importantly, the messages from the same sender are delivered in order, which is enough in many applications.

Note: bounded queues can cause issues if they are not large enough. If a thread waits for the queue to free up, they can cause blocking and cause issues similar to locking. We would typically ensure that the queue is big enough using knowledge of application semantics.

Food for thought 4

This is a great exercise, if you are interested in lock-free synchronization.

I don’t have a solution, as we solved the problem in a different way, but I would like to see yours if you do it.

Food for thought 5

This is a problem that is typically hard to do with lock-free synchronization

A possible approach is to use some of the spare bits in the pointer to store both the head of the queue and the capacity in the same word to allow to manipulate it atomically with compare-and-swap.

Food for thought 6

We can use object *signatures*. Signatures can be cryptographic or even non-cryptographic hashes. Object's signature is stored with the object in the header. The read is performed with a single RDMA read. The client then calculates the signature of the data and compares it to the signature in the header. If the signatures match, the object is consistent. Otherwise, the read is restarted.

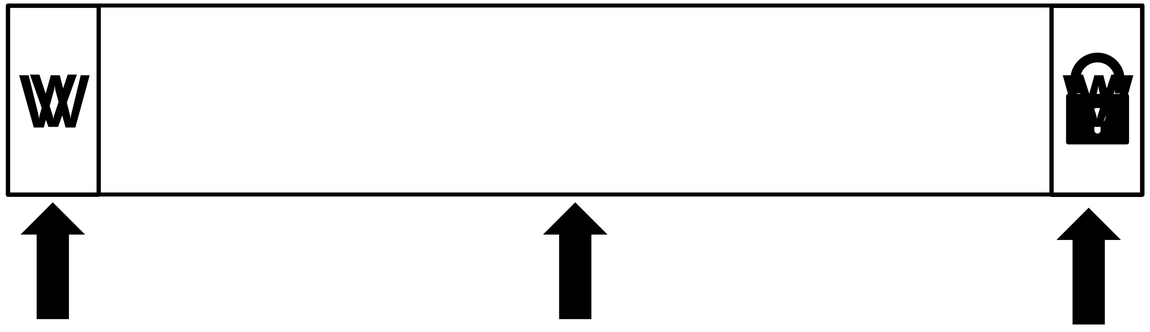
The main idea is that if a part of the object or the signature changes during the read, the signature stored with the object won't match the calculated signature.

Food for thought 7

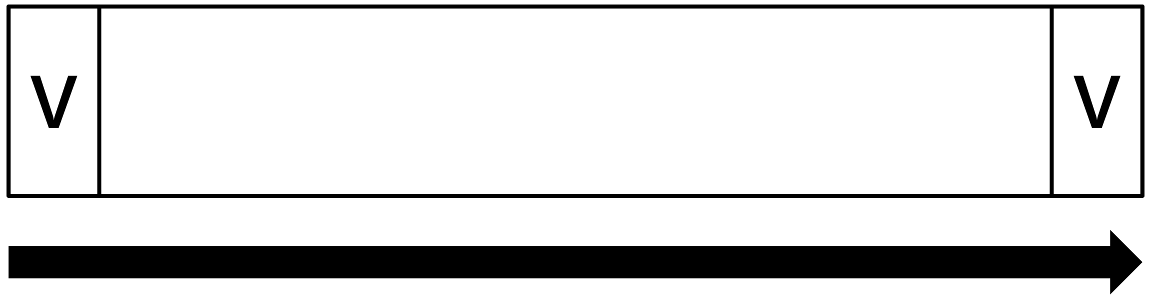
We can store a version in the header of the objects and also in the trailer. The read proceeds from the lowest to the highest address, so we need to update the two versions in the opposite direction – first the trailer and then the header.

Fun story: we first implemented this variant because of miscommunication with the hardware vendor and misunderstanding of the hardware behaviour. We found out later that the results were not consistent, but only through very focused testing.

Food for thought 7: update



Food for thought 7: read



Consistent if the header and the trailer versions are equal and object is not locked

Otherwise retry