Multicore Synchronization

The Lesser-Known Primitives

Who am I?

Samy Al Bahra or **@0xF390.**

Co-founder of Backtrace. Building a modern debugging platform for natively compiled software.

AppNexus, Message Systems, GWU HPCL. Scalability, reliability, multicore, heterogeneous architectures and PGAS.

Created Concurrency Kit. Advanced synchronization primitives for the research, design and implementation of high-performance concurrent systems.

And what are you talking about?

A pragmatic tour of advancements in concurrent and parallel synchronization over the last 27 years.

And what are you talking about?

Concurrency Primitives
- Hardware Support
- Transactional Memory

Blocking Synchronization
- Mutual Exclusion
- Asymmetric Synchronization

Non-Blocking Synchronization
- Asymmetric Synchronization
- Relativistic Software Transactional Memory
- Data Structures

Concurrency Primitives

Prior to the ubiquity of std::atomic, the design and implementation of optimal concurrent was a painstaking process involving inline assembly.

Which features require the developer to supply the abstraction?

Concurrency Primitives :: Atomic Operations

The Intel architecture provides a rich set of atomic operations, several operations are still not implemented in their native wait-free form.

Add With Carry
Subtract with Borrow

Bit Test and Set
Bit Test and Complement
Bit Test and Reset

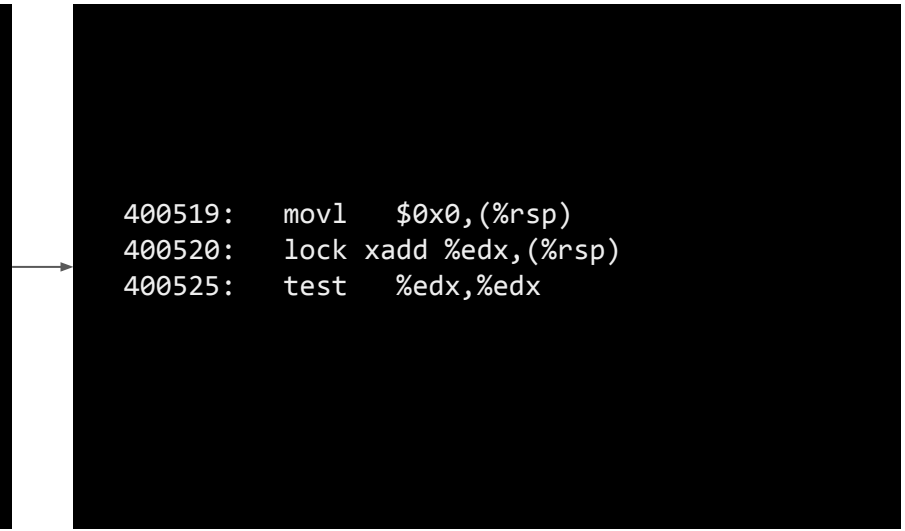
Negation
Not

Increment with Zero Flag*
Decrement with Zero Flag*

Concurrency Primitives :: Atomic Operations

The Intel architecture provides a rich set of atomic operations, several operations are still not implemented in their native wait-free form.

```
{
      std::atomic<unsigned int> value;

      value.store(0, std::memory_order_release);
      if (value.fetch_add(1) == 0)
            return 0;

      value.store(0, std::memory_order_release);
      if (value++ == 0)
            return 0;

      return 1;
}
```



Concurrency Primitives :: Atomic Operations

LL/SC is more powerful than CAS in for many sophisticated concurrent algorithms, but cannot be directly utilized by C++. This includes ARM and Power.

```
1: head = load_linked(&stack->head);
2: if (store_conditional(head, node) == false)
3:   goto 1;
4: return true;
```

Concurrency Primitives :: Pipeline Control

No interfaces exist for architecture-specific pause instructions for busy-wait primitives.

PPC

```
__asm__ __volatile__("or 1, 1, 1;"
                     "or 2, 2, 2;" ::: "memory");
```

x86-64

```
__asm__ __volatile__("pause" ::: “memory”);
```

Concurrency Primitives :: Cache Control

Intel now supports the ability to explicitly read for ownership or load a value into cache with intent to write.

This can eliminate unnecessary cache coherence traffic.

---

1. Supply C_0 data for reading and transition cache line to shared state.
2. Invalidate all cache lines and transition C_0 cache line to modified.

1. Supply C_0 data and invalidate all cache lines.

Concurrency Primitives :: Transactional Memory

Later Intel x86 and Power 8 processors support restricted transactional memory (RTM). The typical use-case is lock elision.

Thread 0

```
lock(&lock);
array[0] = 1;
unlock(&lock);
```

Thread 1

```
lock(&lock);
array[1291] = 1;
unlock(&lock);
```

Concurrency Primitives :: Transactional Memory

Later Intel x86 and Power 8 processors support restricted transactional memory (RTM). The typical use-case is lock elision.

Thread 0

```
CK_ELIDE_LOCK(&lock);
array[0] = 1;
CK_ELIDE_UNLOCK(&lock);
```

Thread 1

```
CK_ELIDE_LOCK(&lock);
array[1291] = 1;
CK_ELIDE_UNLOCK(&lock);
```

Concurrency Primitives :: Transactional Memory

ISO/IEC TS 19841:2015 proposes a transactional memory interface for C++ but the generation comes at a cost.

Restricted transactional memory on Intel and Power architectures are advisory, and have no guarantee of forward progress.

Abortions can fail due to architectural constraints (conflicts, capacity, etc…) and application-specific reasons.

```
The EAX register is updated to reflect an XABORT instruction caused the abort, and
the imm8 argument will be provided in bits 31:24 of EAX.
```

Blocking Synchronization

These are our beloved synchronization primitives such as mutexes, read-write locks and condition variables.

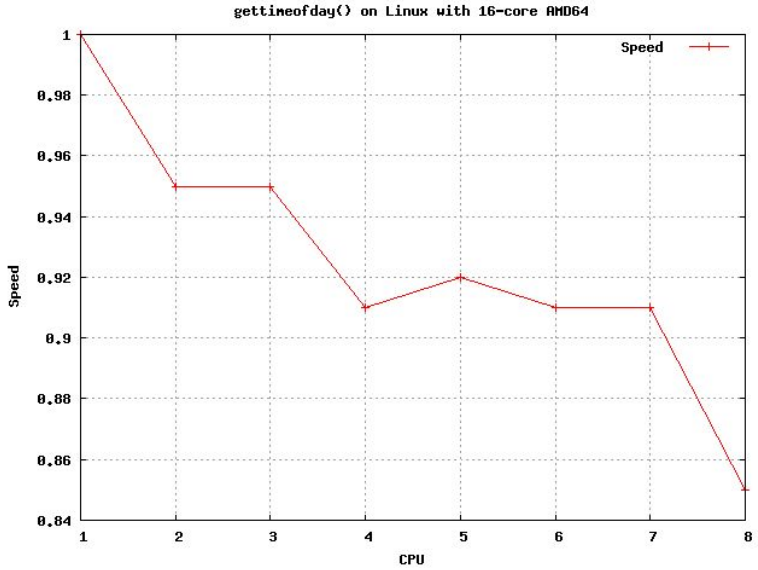
Blocking Synchronization :: The Mutex

There are a myriad of spinlock implementations with varying fairness and scalability guarantees.

	Fair	Scalable	Fast Path	Space
ck_anderson	✓	✓	1A1F	o(N)
ck_cas			1A0F	o(1)
ck_clh	✓	✓	1A1F	o(N)
ck_dec			1A0F	o(1)
ck_fas			1A0F	o(1)
ck_hclh*	✓	✓	1A2F	o(N)
ck_mcs	✓	✓	2A1F	o(N)
ck_ticket	✓		2A0F	o(1)
struct mtx			1A0F	o(1)

Blocking Synchronization :: The Mutex

Starvation-freedom and fairness are especially important on NUMA, and pretty much everything that isn't small-form is NUMA.



Blocking Synchronization :: The Mutex

Starvation-freedom and fairness are especially important on NUMA, and pretty much everything that isn't small-form is NUMA.

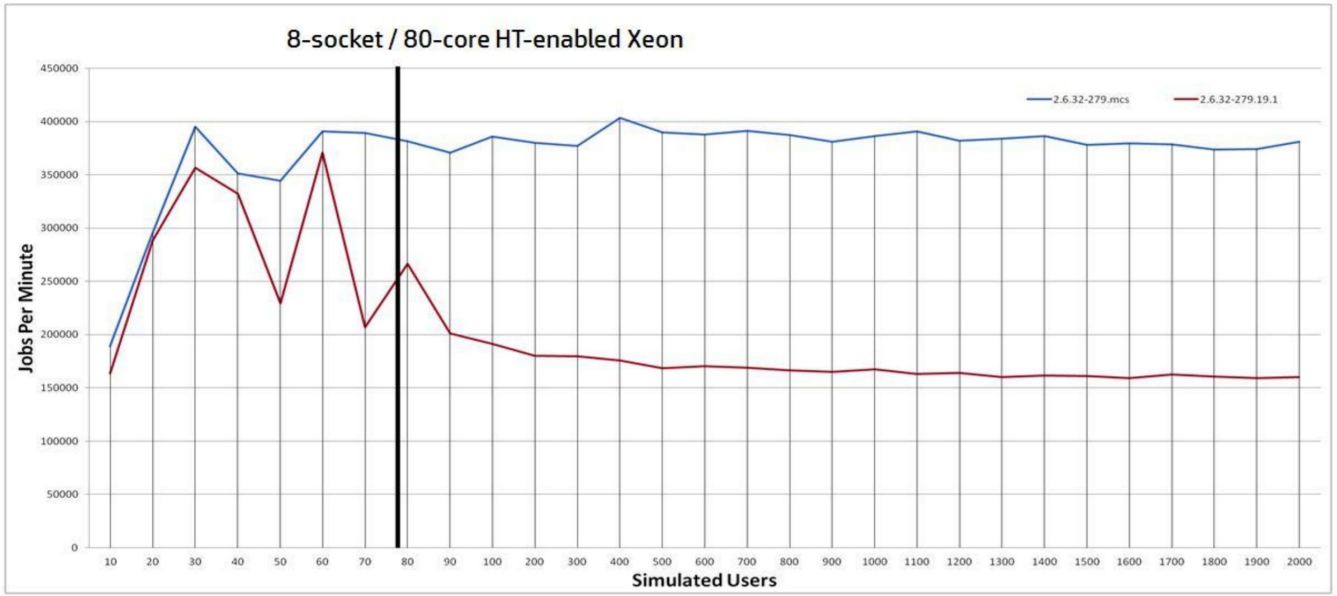
```
$ ./ck_cas.THROUGHPUT 2 3 0
Creating threads (fairness)...done
Waiting for threads to finish acquisition regression...done

0       551369361
1       913893136

# total       :      1465262497
# throughput  :        73263124 a/s
# average     :       732631248
# deviation   : 181261887.50 (24.74%)
```

```
$ ./ck_mcs.THROUGHPUT 2 3 0
Creating threads (fairness)...done
Waiting for threads to finish acquisition regression...done

0       57246076
1       57245137

# total       :       114491213
# throughput  :         5724560 a/s
# average     :        57245606
# deviation   : 469.50 (0.00%)
```

But if there is non-negligible jitter, fairness does cost system-wide throughput.

Blocking Synchronization :: The Mutex

A scalable lock may sound like a misnomer, but an unscalable lock will **degrade** performance under contention



Source: http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf

Blocking Synchronization :: Asymmetric Synchronization

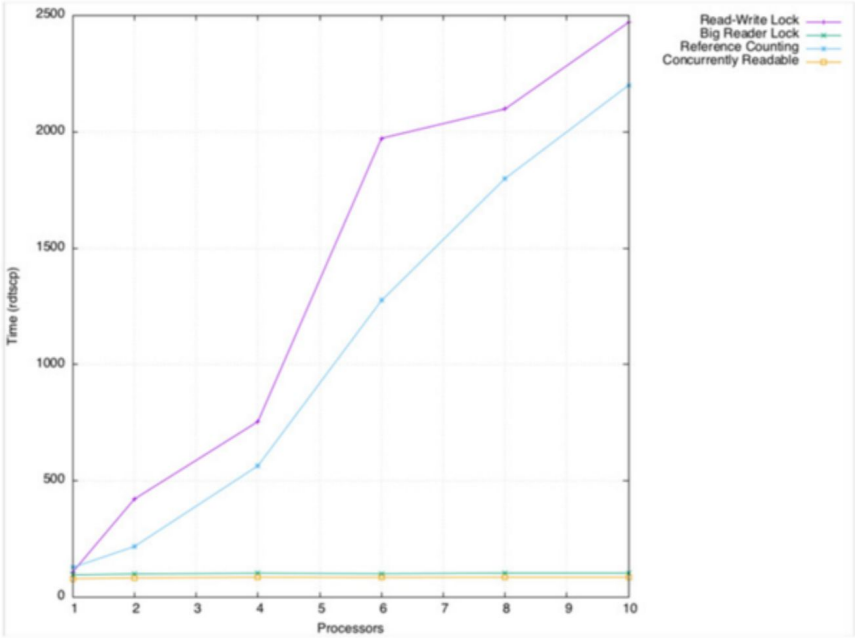
The most common form of asymmetric synchronization is the read-write lock.

	Bias	Fair	Scalable	Reader Fast Path	Space
rwlock	WB	No	No	1A0F	o(1)
rmlock	WB*	No	Yes	0	o(N)
ck_brlock_t	WB*	No	Yes	1A1F	o(1)
ck_rwlock_t	WB	No	No	1A0F	o(1)
ck_bytelock_t	WB	No	No	0A1F	o(1)
ck_pflock_t	None	Phase	No	1A0F	o(1)
ck_tflock_t*	None	Yes	No	1A1F	o(1)
ck_sequence_t	WB*	No	No	0A1F	o(1)
ck_swlock_t	WB	No	No	1A0F	o(1)

The concepts of scalability and fairness still apply.

Blocking Synchronization :: Asymmetric Synchronization

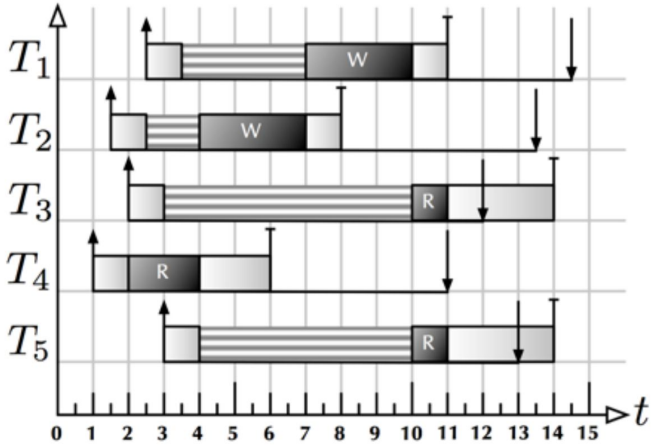
A **big-reader lock** is a distributed write-biased read-write lock with perfect scalability for readers at an increased cost for writers.



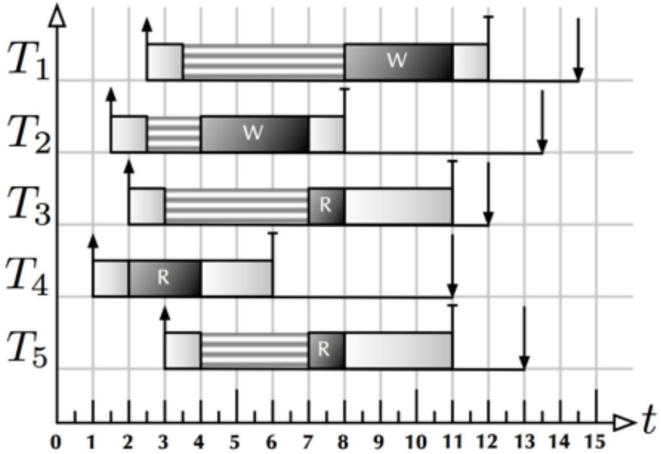
Blocking Synchronization :: Asymmetric Synchronization

Task-fair and phase-fair locks provide nicer scheduling guarantees between readers and writers.

Write-Biased



Phase Fair



Source: Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems by Brandenburg and Anderson

Blocking Synchronization :: Asymmetric Synchronization

Sequence locks allows for concurrent reads with no interference to writers. Readers may spin indefinitely.

```
void
reader(void)
{
  struct example copy;
  unsigned int version;

  CK_SEQUENCE_READ(&seqlock, &version) {
    copy = global;
  }

  return;
}
```

Non-Blocking Synchronization

How can we ensure reader progress?

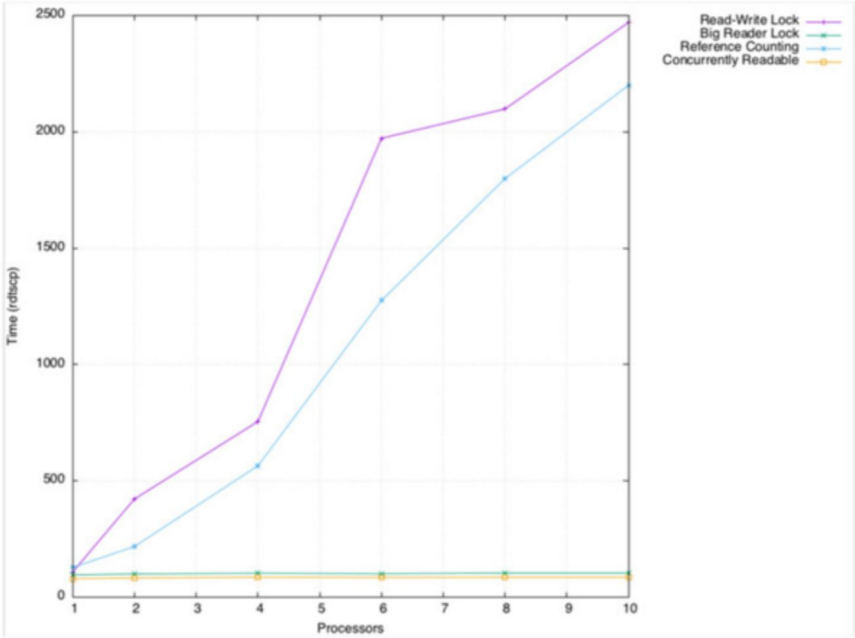
Non-Blocking Synchronization :: Memory Management

Any non-negligible amount of write workload leads to performance degradation of readers.

If liveness and reachability of object is decoupled with blocking synchronization, techniques like reference counting must be used. Reference counting with atomics is expensive.

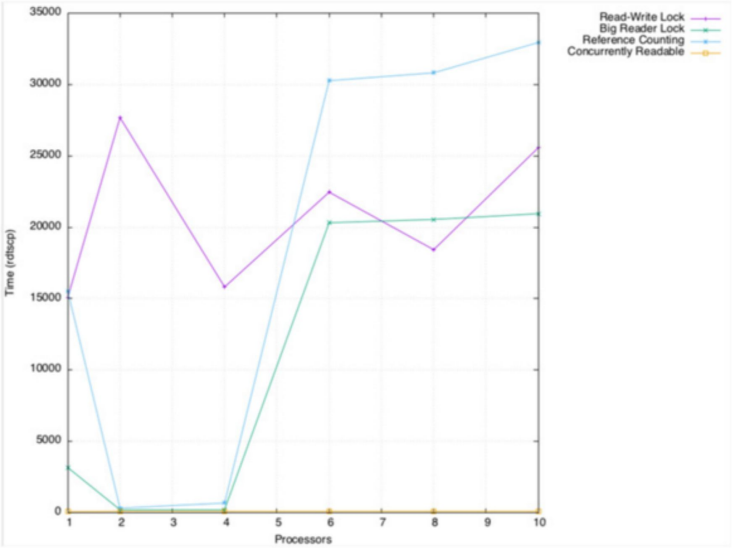
Non-Blocking Synchronization :: Memory Management

Safe memory reclamation protects against read-reclaim races with minimal to no interference to readers.



Non-Blocking Synchronization :: Memory Management

Safe memory reclamation protects against read-reclaim races with minimal to no interference to readers.



Non-Blocking Synchronization :: Hazard Pointers

Hazard pointers are a simple lock-free implementation of safe memory reclamation.

Reader marks used memory

```
read(i):
     do {
          target = array[i];
          hazard.push(target);
          memory_fence();
     while (array[i] != target);
```

Non-Blocking Synchronization :: Hazard Pointers

Hazard pointers are a simple lock-free implementation of safe memory reclamation.

Logical Delete

```
delete_object(i):
     target = array[i];
     array[i] = NULL;
     memory_fence();
     defer(target);
```

Physical Delete

```
reclaim:
     for (target in deferrals) {
          for (t in threads) {
               for (hazard in t) {
                    if (hazard == target) {
                         goto next;
                    }
               }
          }
          free(target);
next:
     }
```

Non-Blocking Synchronization :: Hazard Pointers

Despite the nice properties such as provable bounds on memory usage, Hazard Pointers require additional space for every hazardous reference and require a full memory barrier on the fast path.

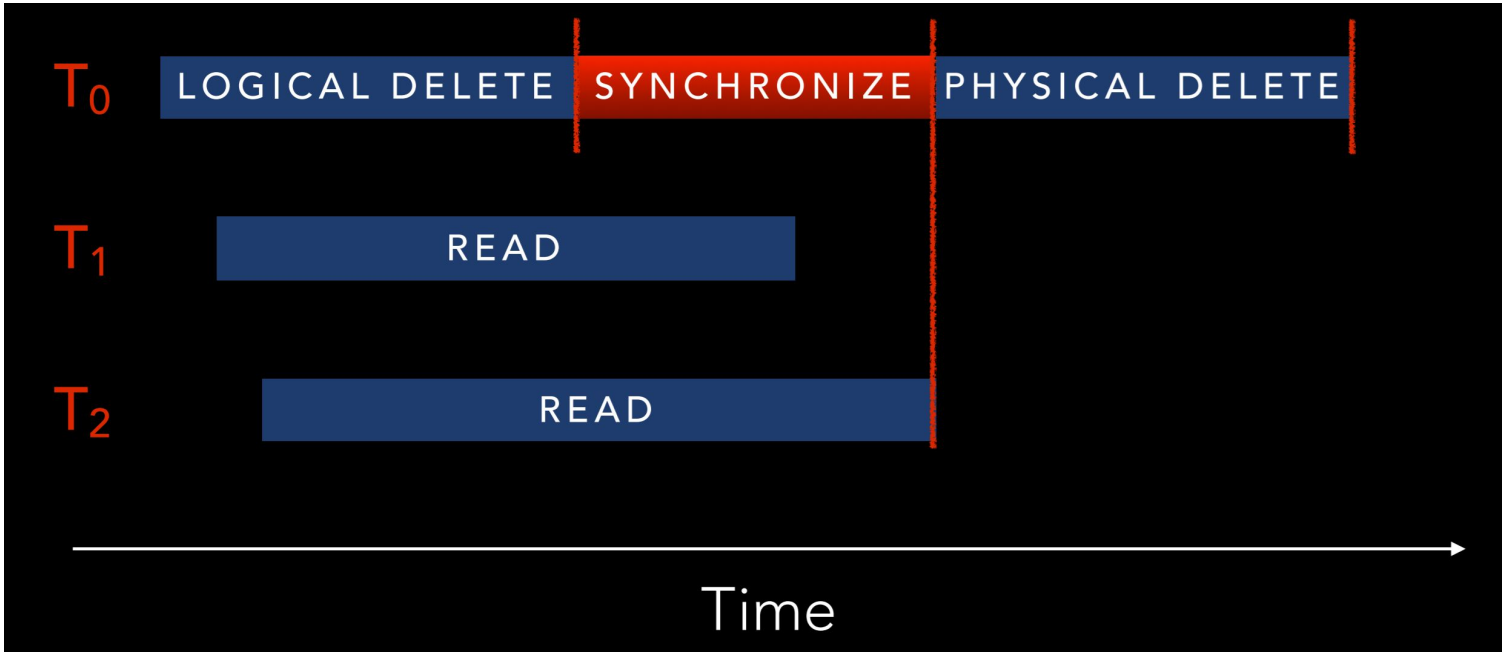
Alternatives exist, but they require write-side blocking.

Non-Blocking Synchronization :: Passive Mechanisms

Passive safe memory reclamation mechanisms ensure read-side progress at the cost of write-side progress.

Non-Blocking Synchronization :: Passive Mechanisms

Central to passive mechanisms is **grace period detection**.



Non-Blocking Synchronization :: Passive Mechanisms

It is guaranteed that the object logically deleted by the writer has no more incoming references after the call to synchronize.

Reader

```
for (;;) {
    smr_begin();
    object = get(index);
    do_stuff(object);
    smr_end();
}
```

Writer

```
for (;;) {
    object = get(index);
    put(index, object, NULL);
    smr_synchronize();
    free(object);
}
```

Non-Blocking Synchronization :: Passive Mechanisms

Fast path operations can be implemented as a no-op.

Reader

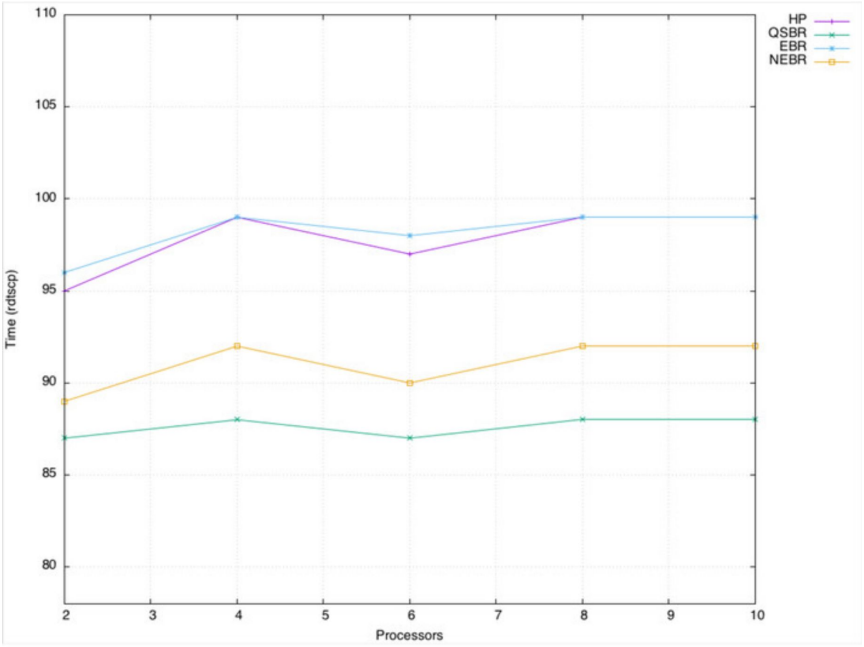
```
void smr_begin() {
     compiler_barrier();
}

void smr_end() {
     compiler_barrier();
     if (tls.end > N) {
          tls.time = time(NULL);
          memory_fence();
          tls.end = 0;
     }
```

Writer

```
void synchronize() {
    now = time(NULL);
    memory_fence();
    for (th in threads) {
        while (th.time <= now)
            yield();
    }
}
```

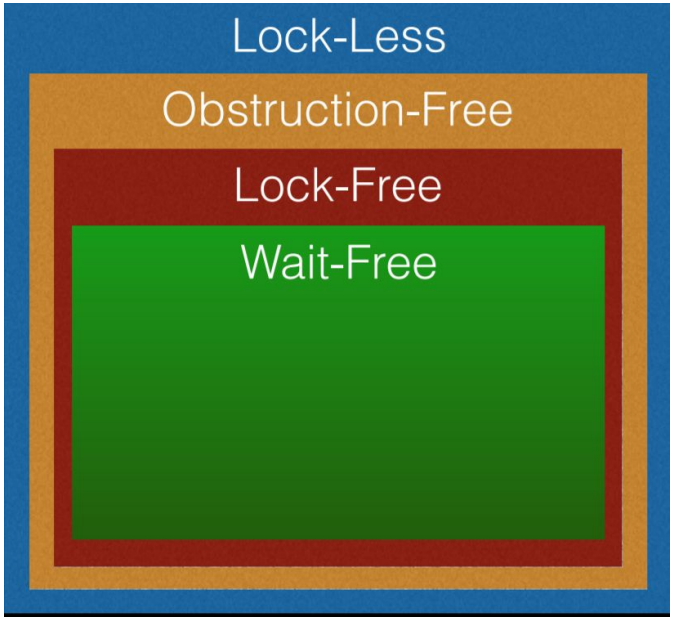
Non-Blocking Synchronization :: Passive Mechanisms

A myriad of implementations exist for safe memory reclamation.



Non-Blocking Synchronization :: Lock-Freedom

Non-blocking synchronization provides very specific progress guarantees and high levels of resilience at the cost of complexity on the fast path.



Lock-freedom provides system-wide progress guarantees.

Wait-freedom provides peroperation progress guarantees.

Non-Blocking Synchronization :: Lock-Freedom

Non-blocking synchronization is not a silver bullet.

TABLE 4. Uncontested latency of various operations

Operation	Intel Core i7-3615QM	IBM Power 730 Express
spinlock_push	17	29
lockfree_push	25	12
spinlock_pop	18	29
lockfree_pop	27	12

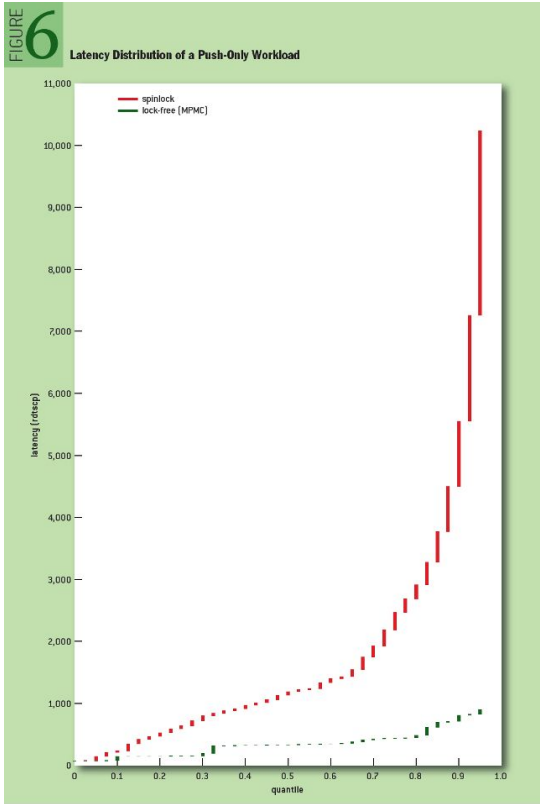
Source “Nonblocking Algorithms and Scalable Multicore Programming”

Non-Blocking Synchronization :: Lock-Freedom

Non-blocking synchronization is not a silver bullet.

The cost of complexity on the fast path will outweigh the benefits until sufficient levels of contention are reached

Non-Blocking Synchronization :: Lock-Freedom



Non-blocking synchronization is not a silver bullet.

The cost of complexity on the fast path will outweigh the benefits until sufficient levels of contention are reached

Non-Blocking Synchronization :: Lock-Freedom

Lock-free dynamic data structures require a run-time component to manage memory safety.

For write-heavy workloads, the cost of managing memory safety can be prohibitive.

Non-Blocking Synchronization :: Lock-Freedom

Workload specialization allows for drastic reductions in complexity at 0-cost to fast path.

Non-Blocking Synchronization :: Ring Buffer

A data structure that doesn't require dynamic memory allocation. Single-writer principle allows for wait-free operations with no expensive instructions on TSO architectures.

	Enqueue	**Dequeue**
SPMC	Wait-Free	Lock-Free
MPSC	Serialized (Lock-Less)	Wait-Free
SPSC	Wait-Free	Wait-Free
MPMC	Serialized (Lock-Less)	Lock-Free

Non-Blocking Synchronization :: Hash Table

State-of-the-art entailed significant trade-off in complexity and performance.

Chaining

- Memory management.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

Open Addressing

- Complex object re-use constraints in absence of key duplication.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

What does specialization get us?

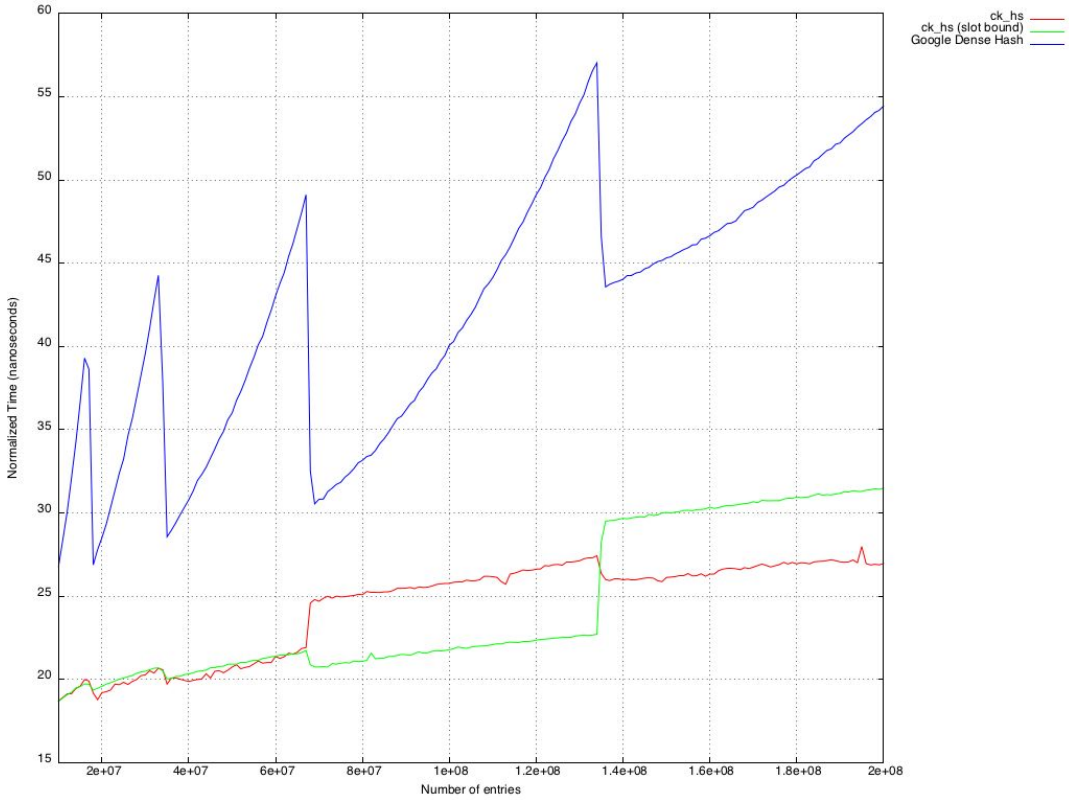
Non-Blocking Synchronization :: Hash Table

- Termination-safety is not a requirement.
- SWMR allows us to rely on less complex instructions.
- Primarily executing on x86 processors.
- Load to load ordering.
- Store to store ordering.

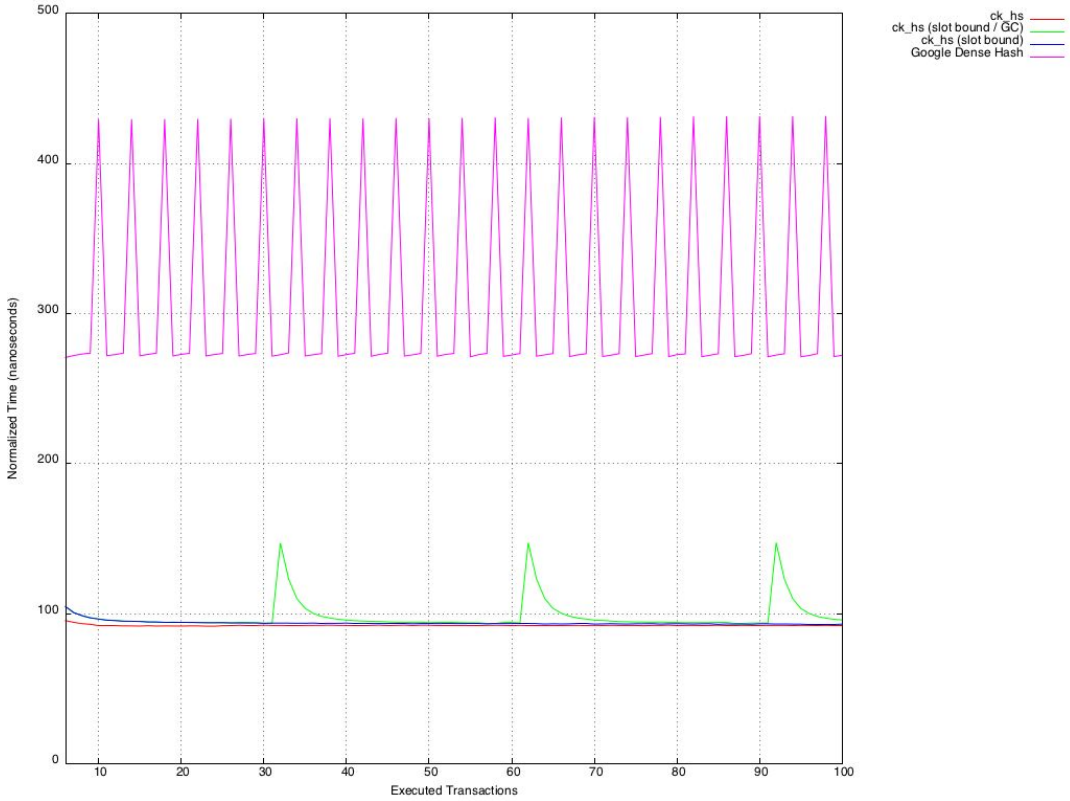
Non-Blocking Synchronization :: Hash Table

General technique for achieving lock-free / wait-free single-writer / many-reader open-addressed hash table with practically 0 cost for TSO and sometimes low-cost for RMO. Applicable even in absence of safe memory reclamation techniques.

Non-Blocking Synchronization :: Hash Table



Non-Blocking Synchronization :: Hash Table



Backtrace

Non-Blocking Synchronization :: Hash Table

Data Structure	Description
ck_hs	Hash set (cache line to double hash)
ck_ht	Hash table (cache line to double hash)
ck_rhs	Hash set (robin hood hash)

The End

There is a lot of material to cover, and unfortunately, a 45 minute slot does nowhere near justice. Happy to follow-up in greater detail offline.

Follow me at **@0xF390**.

See other talks for more in depth coverage of particular topics.

Resources:

- https://backtrace.io/blog (we're hiring!)
- http://concurrencykit.org
- http://liburcu.org
- https://github.com/facebook/folly
- https://backtrace.io/blog/blog/2015/03/13/workload-specialization/index.html
- http://queue.acm.org/detail.cfm?id=2492433
- https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

Interesting Developments:
A particularly interesting area to me recently is using relativistic techniques and bounded concurrency to implement transactional memory systems.

http://sigops.org/sosp/sosp15/current/2015-Monterey/077-matveev-online.pdf
http://delivery.acm.org/10.1145/3040000/3037721/p207-calciu.pdf