Concurrency Kit

Towards accessible scalable synchronization primitives for C

Samy Al Bahra / @0xF390

About Me

Co-founder of Backtrace (http://backtrace.io)

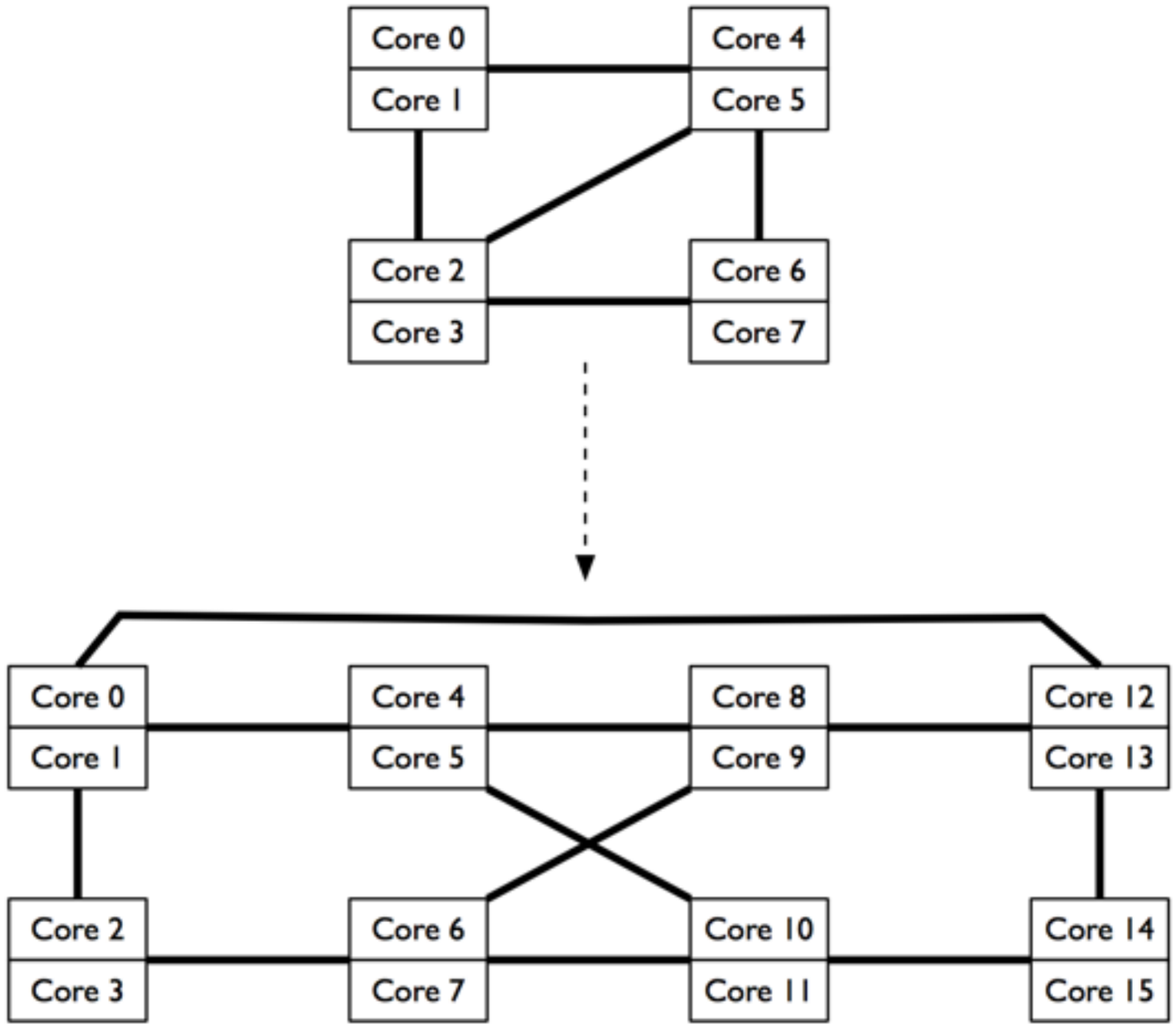
Building a better debugging platform for native applications.

Started Concurrency Kit

A concurrent memory model for C99 and arsenal of tools for high performance synchronization.

Previously at AppNexus and Message Systems doing work centralized around architecture, performance and reliability. Worked on PGAS languages, multicore synchronization and parallel I/O at GWU HPCL.

Motivation



Motivation

Design and implement high performance synchronization primitives for high performance parallel software.

- A concurrent memory model for C.
- A unified interface for hardware intrinsics.

Design and implement high performance data structures.

- Safe memory reclamation mechanisms.
- Specialized concurrent data structures.

All this awesome technology has been out there for over 20 years.

Introduction

Concurrency Kit provides a concurrent memory model for C99 and arsenal of tools for high performance synchronization.

- Support for FreeBSD and Linux kernels.
- Tested with GCC, MingW, ICC, clang and SunCC. Used with several research compilers.
- Support for x86, ARM, SPARCv9+, Power.
- BSD licensed.

Likely that a text message, e-mail, image or advertisement you saw today involved Concurrency Kit.

Overview

Concurrency Primitives

- Compiler, memory ordering and primitive-operation abstraction.

Spinlocks

- Pluggable into generic cohort and elision framework.

Execution Barriers

- Primarily superseded by phasers.

Read-Write Synchronization

- Pluggable into cohort and elision framework.

Safe Memory Reclamation

- One passive and one active implementation.

Data Structures

- Array, ring buffer, bitmap, hash tables, queue, stack, etc…

Concurrency Primitives

Provides a complete partial ordering interface for interaction between stores, loads and atomic operations. Memory ordering exceptions expressed with strict fence interface.

Memory Barrier Interface

```
ck_pr_fence_atomic
ck_pr_fence_atomic_load
ck_pr_fence_atomic_store
ck_pr_fence_store_atomic
ck_pr_fence_load
ck_pr_fence_load_atomic
ck_pr_fence_load_store
ck_pr_fence_store_load
ck_pr_fence_store
ck_pr_fence_memory
ck_pr_fence_acquire
ck_pr_fence_release
ck_pr_fence_lock
ck_pr_fence_unlock
```

```
ck_pr_fence_strict_atomic
ck_pr_fence_strict_atomic_load
ck_pr_fence_strict_atomic_store
ck_pr_fence_strict_store_atomic
ck_pr_fence_strict_load_atomic
ck_pr_fence_strict_load_store
ck_pr_fence_strict_store_load
ck_pr_fence_strict_memory
ck_pr_fence_strict_acquire
ck_pr_fence_strict_release
ck_pr_fence_strict_lock
ck_pr_fence_strict_unlock
```

These are the lowest common denominator and can serve as building blocks for FreeBSD/C11-style acquire-release semantics.

Concurrency Primitives

The memory model and instruction set are decoupled. Porters transcribe a handful of operations from architecture manuals and rarely have to concern themselves with memory model minutiae and additional boilerplate.

include/ck_pr.h

Generated operations

Common-case barriers

include/arch/ck_pr.h

x86 | x86-64 | ARM | SPARCv9+ | Power

Native load, store, atomic operations and strict fences.

Concurrency Primitives

The memory model and instruction set are decoupled. Porters transcribe a handful of operations from architecture manuals and rarely have to concern themselves with memory model minutiae and additional boilerplate.

```
CK_F_PR_CAS_64
CK_F_PR_CAS_64_VALUE
CK_F_PR_CAS_PTR
CK_F_PR_CAS_PTR_VALUE
CK_F_PR_CAS_32
CK_F_PR_CAS_32_VALUE
CK_F_PR_CAS_UINT
CK_F_PR_CAS_INT
CK_F_PR_CAS_UINT_VALUE
CK_F_PR_CAS_INT_VALUE
CK_F_PR_STORE_64
CK_F_PR_STORE_32
CK_F_PR_STORE_DOUBLE
CK_F_PR_STORE_UINT
CK_F_PR_STORE_INT
CK_F_PR_STORE_PTR
CK_F_PR_LOAD_64
CK_F_PR_LOAD_32
CK_F_PR_LOAD_DOUBLE
CK_F_PR_LOAD_UINT
CK_F_PR_LOAD_INT
CK_F_PR_LOAD_PTR
```

```
CK_F_PR_ADD_INT
CK_F_PR_SUB_INT
CK_F_PR_AND_INT
CK_F_PR_XOR_INT
CK_F_PR_OR_INT
CK_F_PR_ADD_DOUBLE
CK_F_PR_SUB_DOUBLE
CK_F_PR_ADD_UINT
CK_F_PR_SUB_UINT
CK_F_PR_AND_UINT
CK_F_PR_XOR_UINT
CK_F_PR_OR_UINT
CK_F_PR_ADD_PTR
CK_F_PR_SUB_PTR
CK_F_PR_AND_PTR
CK_F_PR_XOR_PTR
CK_F_PR_OR_PTR
CK_F_PR_ADD_64
CK_F_PR_SUB_64
CK_F_PR_AND_64
CK_F_PR_XOR_64
CK_F_PR_OR_64
CK_F_PR_ADD_32
CK_F_PR_SUB_32
CK_F_PR_AND_32
CK_F_PR_XOR_32
CK_F_PR_OR_32
```

Concurrency Primitives

All concurrent **accesses** on actively mutable state are annotated with ck_pr operations that expand to the necessary hardware instructions and also serve as a compiler barrier.

- **volatile** does not provide any atomicity guarantees and ordering is only defined with respect to other volatile accesses.
- **volatile** unnecessarily disables a lot of optimizations.

Concurrency Primitives

Some notable primitives that are applicable to FreeBSD.

- Type-safe interaction with pointers in master (no need to play casting games with long) using only C99, true assignment semantics.*
  - `struct a *a; struct b *b; ck_pr_store_ptr(&a, b) is invalid.`
- **ck_pr_rfo**
  - Read-for-ownership primitive was once AMD only, now available in Intel Broadwell+ (compatible).
  - Drastic reduction in cache-coherency traffic for prefetchable workloads.
- **ck_pr_rtm**
  - Restricted transactional memory now available on Intel (Haswell+ for broken version, recent Broadwell for fixed version) and Power 8+.
  - Scalability for qualifying workloads (fast path cost slightly more expensive than atomic and high abort cost).

* Thanks to pkhuong and jwittrock for the store and load coverage.

Concurrency Primitives

Future work is to provide MD-agnostic acquire-release interface similar to C11/FreeBSD interface.

Questions?

Spinlocks

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)

Fast path is tuple nAkF where n is number of atomics and k is number of fences (ignoring necessary lock fence).

* Thanks to cognet@ for ck_hclh work.

Spinlocks

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

```
pm:benchmark sbahra$ ./ck_cas.THROUGHPUT 2 2 0
Creating threads (fairness)...done
Waiting for threads to finish acquisition regression...done
0       17349179
1       42371582

# total       :        59720761
# throughput  :         2986038 a/s
# average     :        29860380
# deviation   : 12511201.50 (41.90%)
```

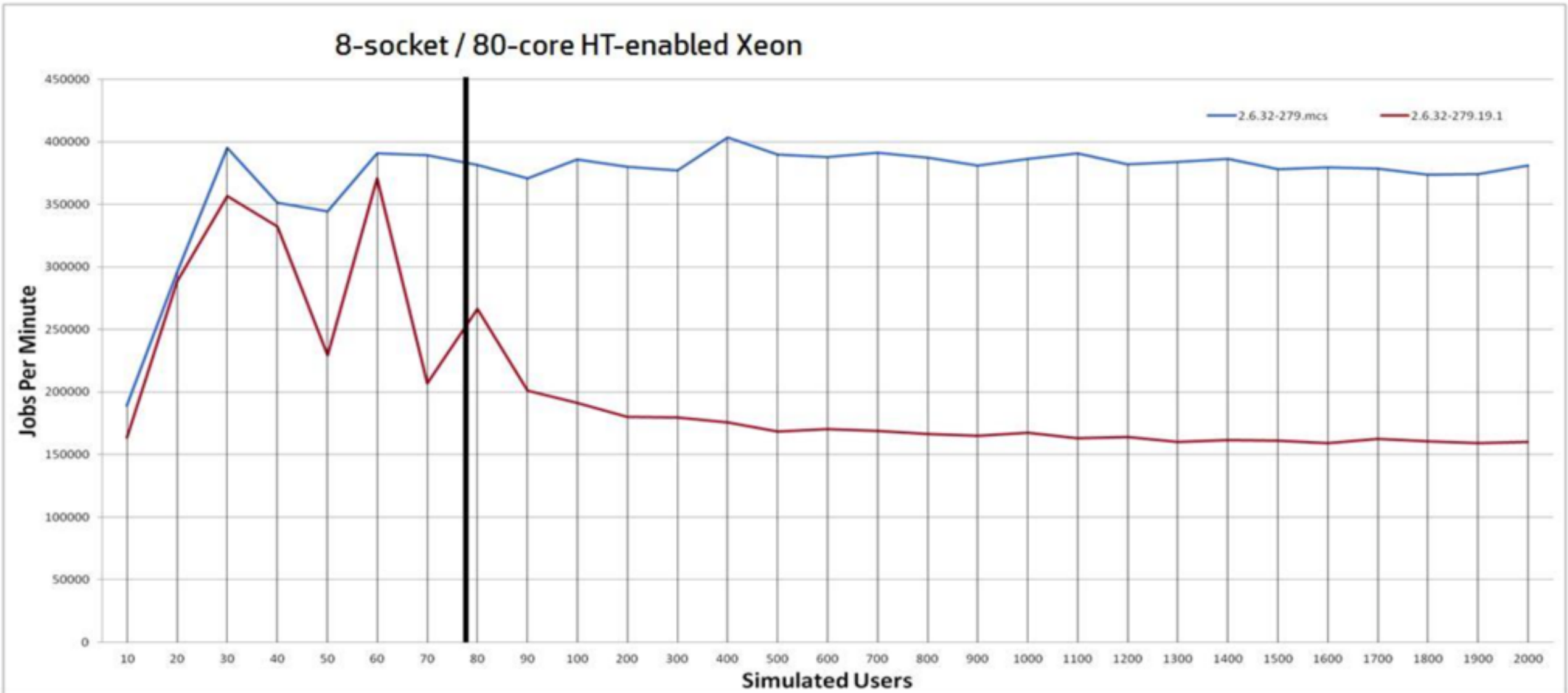
```
pm:benchmark sbahra$ ./ck_clh.THROUGHPUT 2 2 0
Creating threads (fairness)...done
Waiting for threads to finish acquisition regression...
0       16472093
1       16469477

# total       :        32941570
# throughput  :         1647078 a/s
# average     :        16470785
# deviation   : 1308.00 (0.01%)
```

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

Spinlocks

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

Spinlocks

These algorithms apply to adaptive locks, they optimize busy-wait stage while providing stronger fairness guarantees.

Questions?

Elision

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

Thread 0

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

Thread 1

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

Elision

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

Thread 0

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

Thread 1

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

Elision

Questions?

Cohorts

Turn any lock into a NUMA-aware lock without sacrificing throughput. At least 2 additional atomic operations on fast path, not competitive on commodity multicore systems.

Questions? Talk to Brendon, he's here.

Execution Barriers

Prevents execution at a barrier until all threads have paired*. A plethora of implementations from the literature. Scalability is usually not a concern.

Questions? Let's talk later.

* Thanks to djoseph for work on these.

Blocking Asymmetric Synchronization

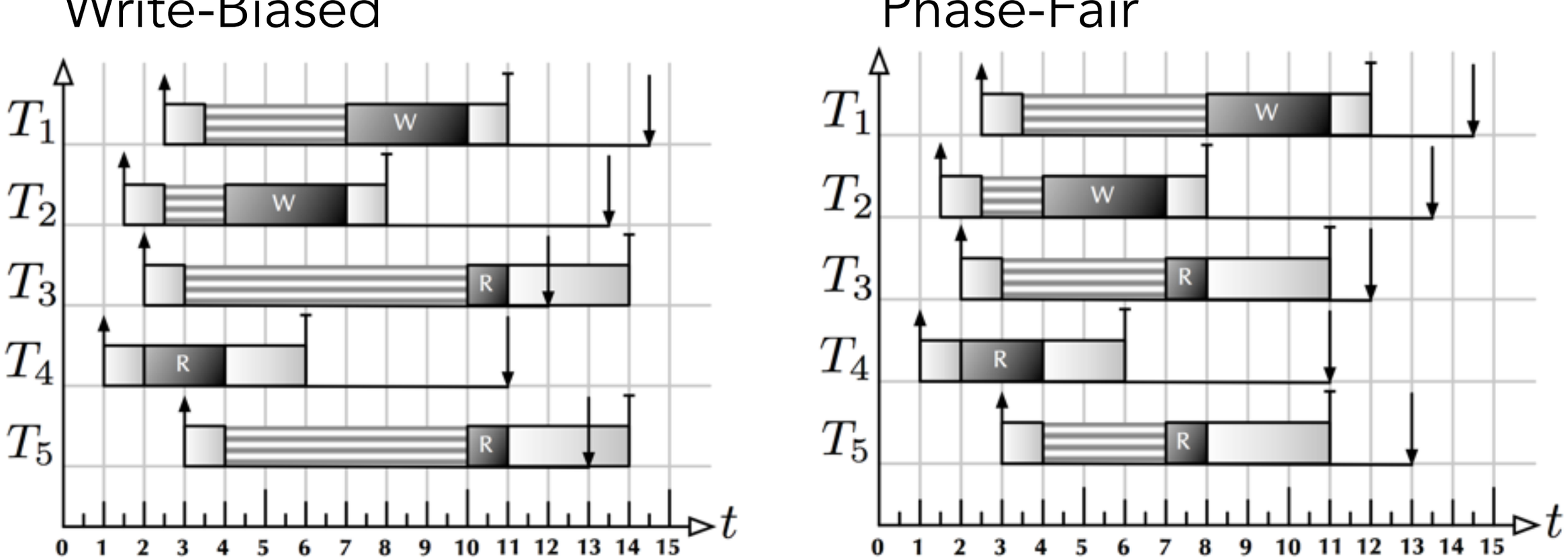
The most common form of asymmetric synchronization is the read-write lock.

	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)

* Thanks to jwittrock for ck_tflock work and jsridhar for ck_swlock_t work

Blocking Asymmetric Synchronization

Task-fair and phase-fair locks provide nicer scheduling guarantees between readers and writers. They can augment adaptive paths.



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

Blocking Asymmetric Synchronization

ck_sequence_t is a simple blocking primitive that allows for concurrently reading data without interfering with writers. Readers may spin indefinitely.

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

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

        return;
}
```

Blocking Asymmetric Synchronization

The fairness and performance trade-offs are elaborated in manual pages.

Questions?

Legal Break

Nothing in this presentation constitutes legal advice. Consult with a lawyer, accountant, and insurance professional before making your decisions. The views and opinions in this presentation represent my own and not those of people, institutions or organizations I am affiliated with unless stated explicitly. This presentation does not represent the views, position or attitudes of my employer, their clients, or any of their affiliated companies.

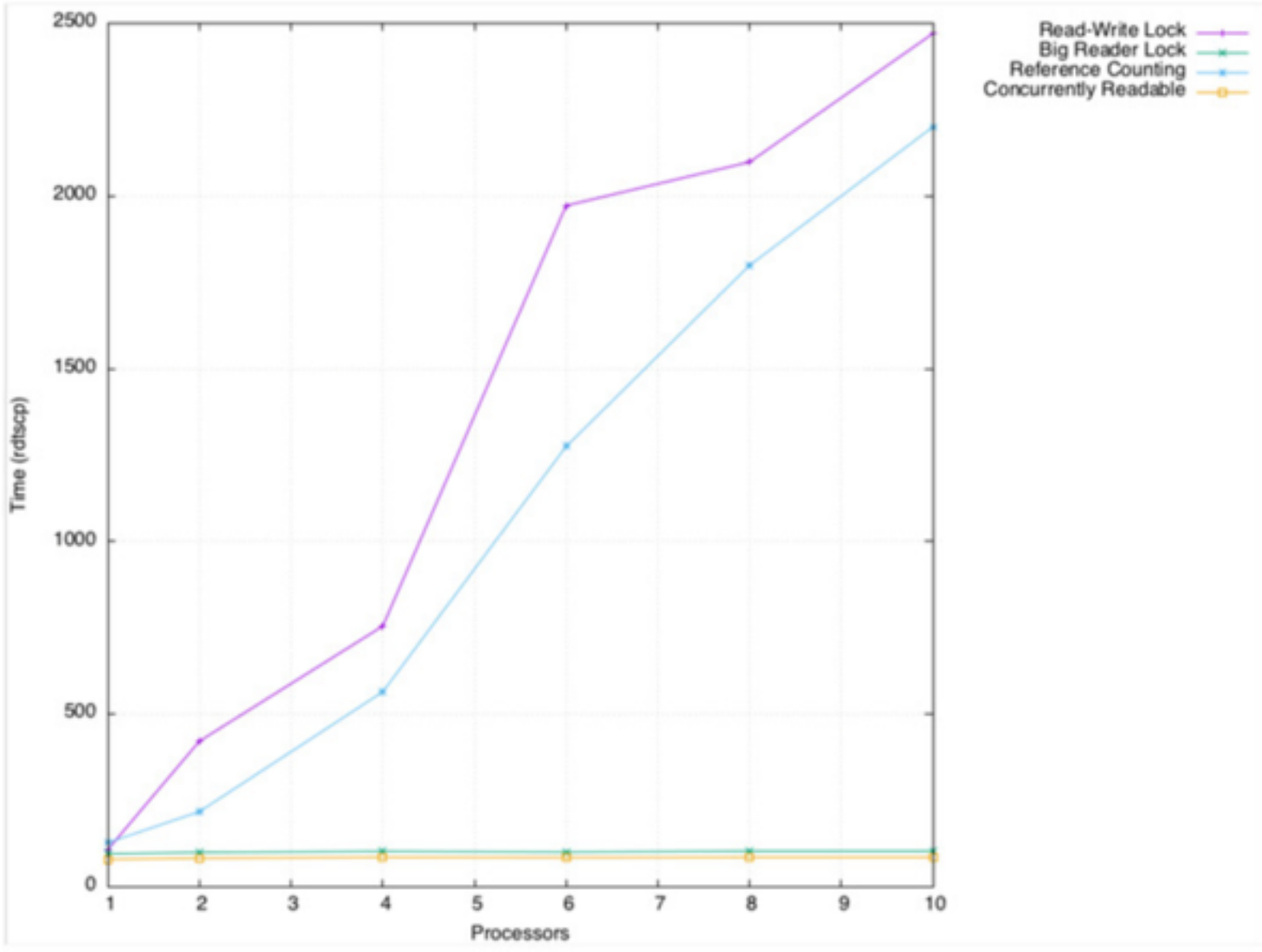
Blocking Asymmetric Synchronization

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.

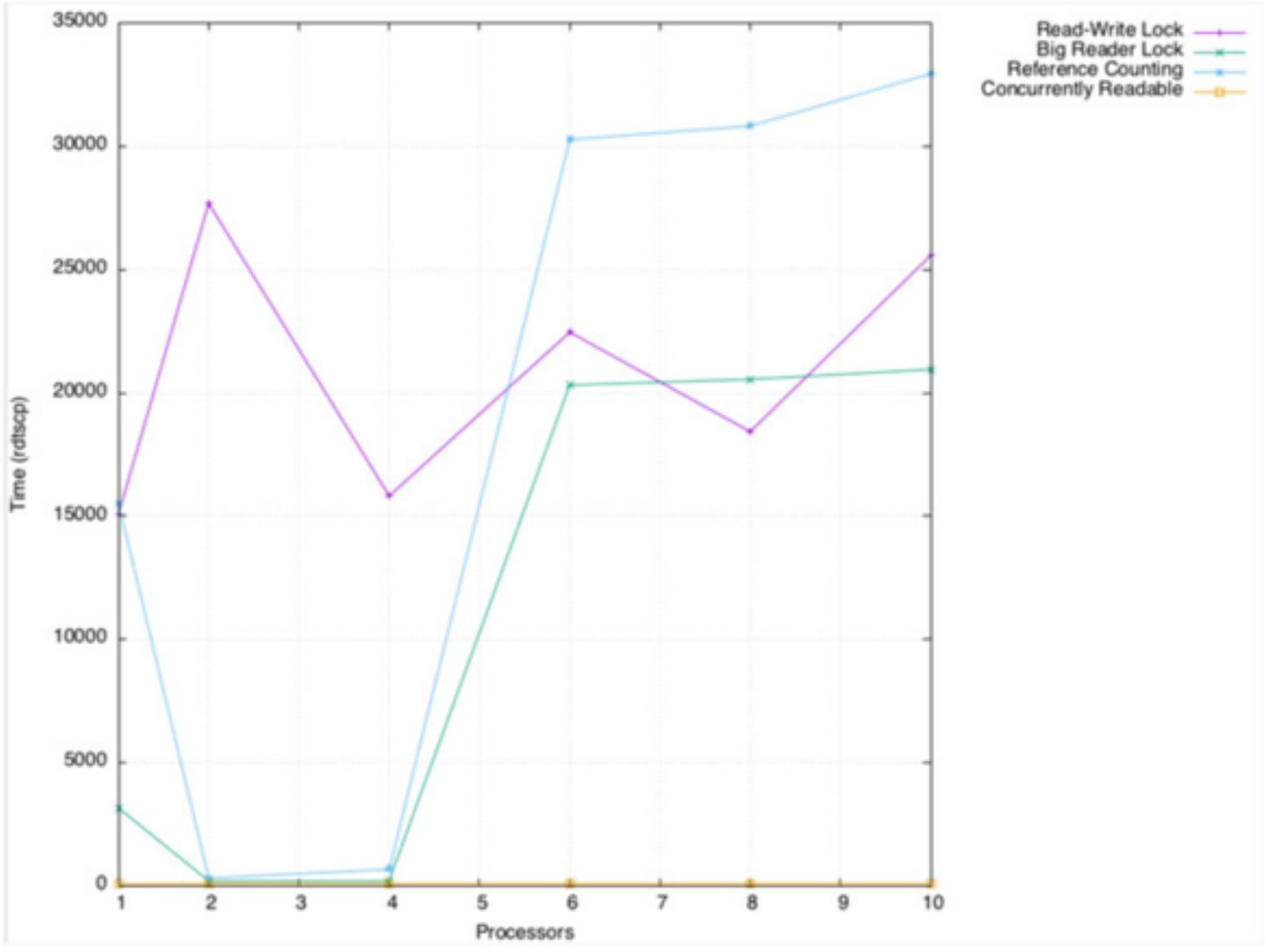
Safe Memory Reclamation

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



Safe Memory Reclamation

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



Safe Memory Reclamation

The most popular form of safe memory reclamation these days is RCU. Unfortunately, the patent exemption only applies to LGPL and GPL software.

	Fast Path	Traversal	Amortization
`ck_hp`	nA | nF	Expensive	Expensive
`ck_epoch`	1A | 1F	Fast	Amortized
`QSBR`	0	Fast	Fast

Safe Memory Reclamation

	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Epoch	72	76	76	204	236	53350
RCU	136	144	144	375	572	264600
QSBR	128	136	136	248	224	54330
Signal	44	44	44	62.35	48	63930

Safe Memory Reclamation

	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Epoch	72	100	484	619	940	551300
RCU	136	144	296	414	636	412800
QSBR	128	136	220	287	380	43050

Safe Memory Reclamation

	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Epoch	188	772	1408	1781	2348	1888000
RCU	720	9260	26610	44080	60670	2181000
QSBR	400	10130	25420	38250	53590	4526000

Safe Memory Reclamation

Questions?

Ring Buffer

ck_ring_t is a lock-free ring buffer in Concurrency Kit. It is wait-free for single-producer / single-consumer and lock-free for single-producer / multi-consumer.

	buf_ring_t	ck_ring_t
Correct	No (but most fixes in review)	Yes
SPMC	Serialized	Wait-Free / Lock-Free
MPSC	Serialized	N/A
SPSC	Wait-Free	Wait-Free
MPMC	Serialized	N/A (Serialized / Lock-Free)
Multibyte Storage	No	Yes

Serialized transformation with **buf_ring_t** fast path cost is easy to add, ck_ring_t counter representation would not require serializing consumer.

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.

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

* Thanks to cognet@ for ck_rhs

Hash Table

A ck_ht replacement is in the works.

Questions?

The End

http://concurrencykit.org

http://backtrace.io

@0xF390 / sbahra@backtrace.io