From the roots to the sky:
end-to-end concurrency verification at DresdenRC

Jonas Oberhauser

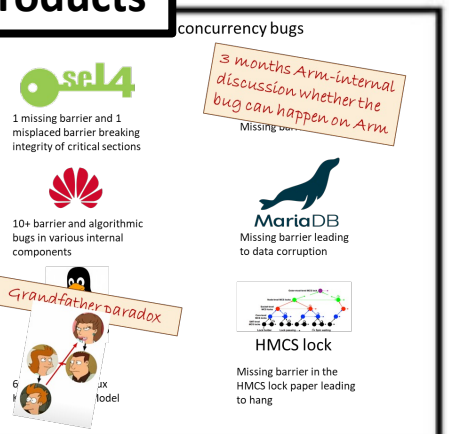
Dresden Research Center

Selected achievements of DresdenRC in the recent years

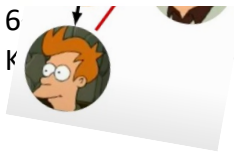
non-safety-critical products

Fix bugs

concurrency bugs

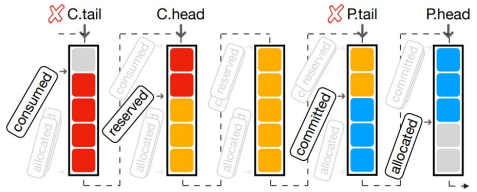


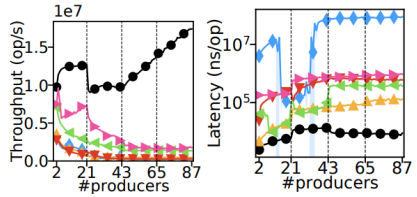
leave bug
that increases
throughput by 5%
& causes 5 minutes
downtime each month



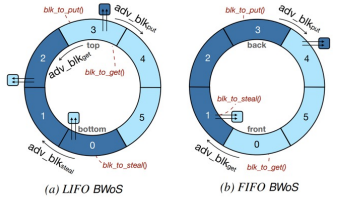
HMCS lock paper leading to hang

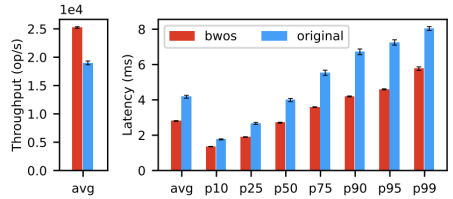
New concurrency algorithms



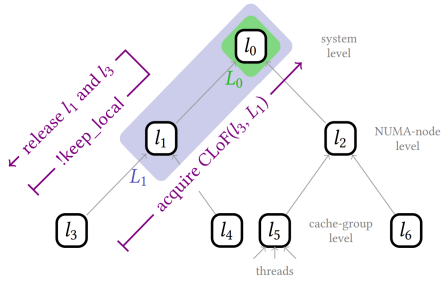


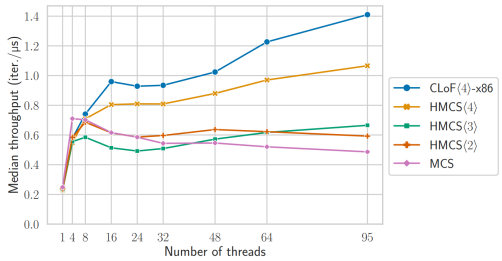
queue – **BBQ**: up to 10x improved throughput, latency





work stealing – **BWoS**: better latency, more throughput

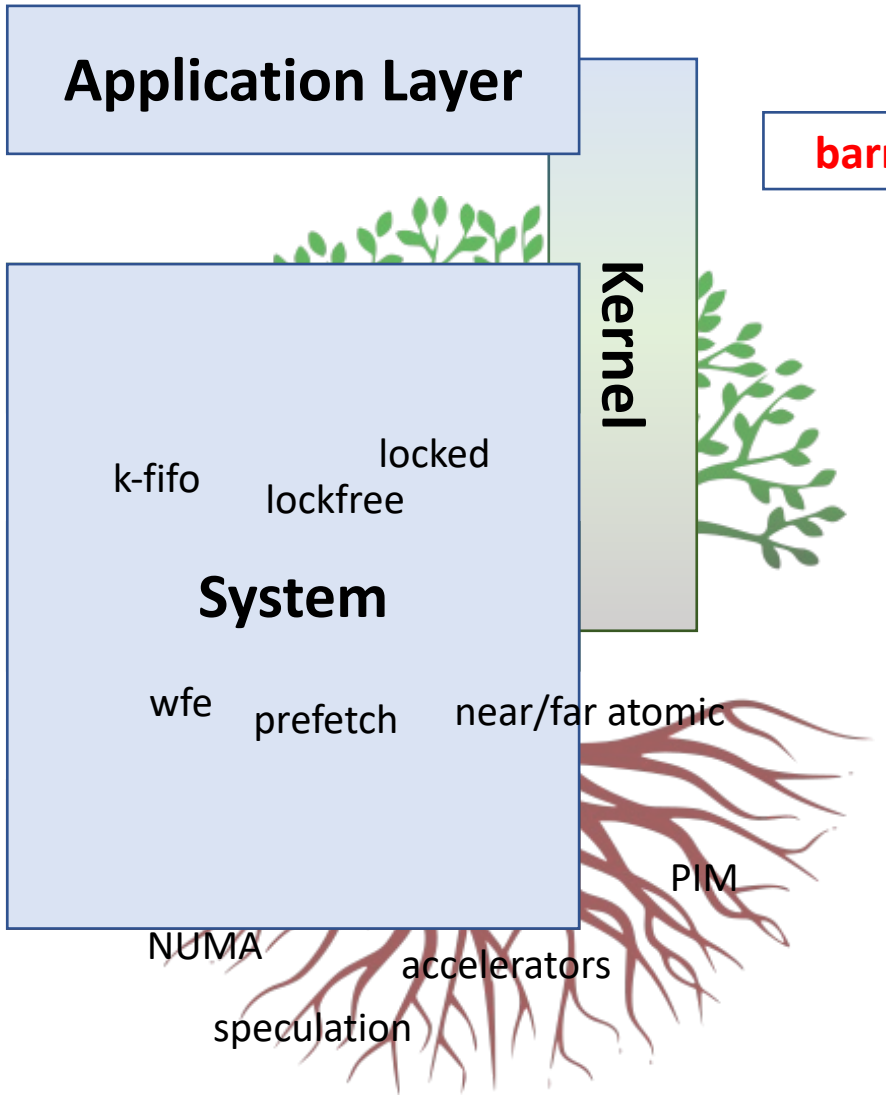




lock – **CloF**: automatically configurable, improved throughput

ASPLOS'21, OOPSLA'21, NETYS'21, SOSP'21, ATC'22, OOPSLA'22, ASPLOS'23, OSDI'23 - code used in several product lines

Concurrency



Thread-Safe Object

channels, maps, lists

Shared memory

barrier

```
void T1() {
    n = 20;
    release(flag = true);
}
```

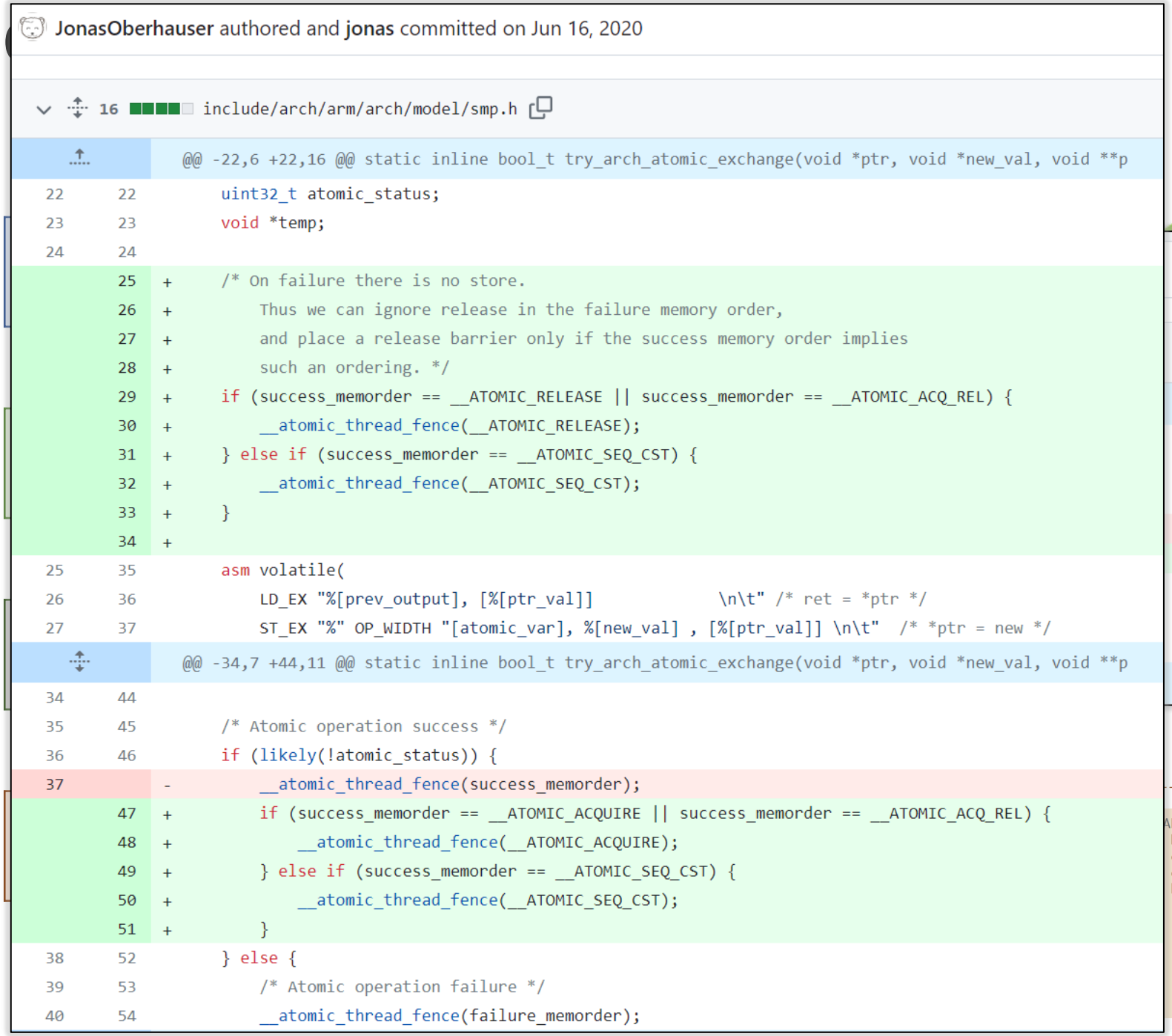
```
void T2() {
    while (! acquire(flag));
    assert(n==20);
}
```

→ **W**eak **M**emory **M**odel (**WMM**)

Highly optimized – program now crashes after *only 100µsec*!

Challenge

- design application layer & system to **leverage evolving hardware & compiler optimizations**
- manage resulting **complexity**
- **disable optimizations selectively** when they make the application incorrect (e.g. with **barriers**)



JonasOberhauser authored and jonas committed on Jun 16, 2020

16 include/arch/arm/arch/model/smp.h

```
@@ -22,6 +22,16 @@ static inline bool_t try_arch_atomic_exchange(void *ptr, void *new_val, void **p
22 22      uint32_t atomic_status;
23 23      void *temp;
24 24
+ 25      /* On failure there is no store.
+ 26          Thus we can ignore release in the failure memory order,
+ 27          and place a release barrier only if the success memory order implies
+ 28          such an ordering. */
+ 29      if (success_memorder == __ATOMIC_RELEASE || success_memorder == __ATOMIC_ACQ_REL) {
+ 30          __atomic_thread_fence(__ATOMIC_RELEASE);
+ 31      } else if (success_memorder == __ATOMIC_SEQ_CST) {
+ 32          __atomic_thread_fence(__ATOMIC_SEQ_CST);
+ 33      }
+ 34
25 35      asm volatile(
26 36          LD_EX "%[prev_output], [%[ptr_val]]             \n\t" /* ret = *ptr */
27 37          ST_EX "%" OP_WIDTH "[atomic_var], %[new_val] , [%[ptr_val]] \n\t"  /* *ptr = new */
@@ -34,7 +44,11 @@ static inline bool_t try_arch_atomic_exchange(void *ptr, void *new_val, void **p
34 44
35 45      /* Atomic operation success */
36 46      if (likely(!atomic_status)) {
- 37          __atomic_thread_fence(success_memorder);
+ 47          if (success_memorder == __ATOMIC_ACQUIRE || success_memorder == __ATOMIC_ACQ_REL) {
+ 48              __atomic_thread_fence(__ATOMIC_ACQUIRE);
+ 49          } else if (success_memorder == __ATOMIC_SEQ_CST) {
+ 50              __atomic_thread_fence(__ATOMIC_SEQ_CST);
+ 51          }
38 52      } else {
39 53          /* Atomic operation failure */
40 54          __atomic_thread_fence(failure_memorder);
```

missing release-barrier – two threads can enter critical section!

release-barrier is implemented wrong – two threads can *still* enter critical section!

```
ARM_ERRATA_742230
bool "ARM errata: DMB operation may be faulty"
depends on CPU_V7 && SMP
depends on !ARCH_MULTIPLATFORM
help
  This option enables the workaround for the 742230 Cortex-A9
  (r1p0..r2p2) erratum. Under rare circumstances, a DMB instruction
  between two write operations may not ensure the correct visibility
  ordering of the two writes. This workaround sets a specific bit in
  the diagnostic register of the Cortex-A9 which causes the DMB
  instruction to behave as a DSB, ensuring the correct behaviour of
  the two writes.
```

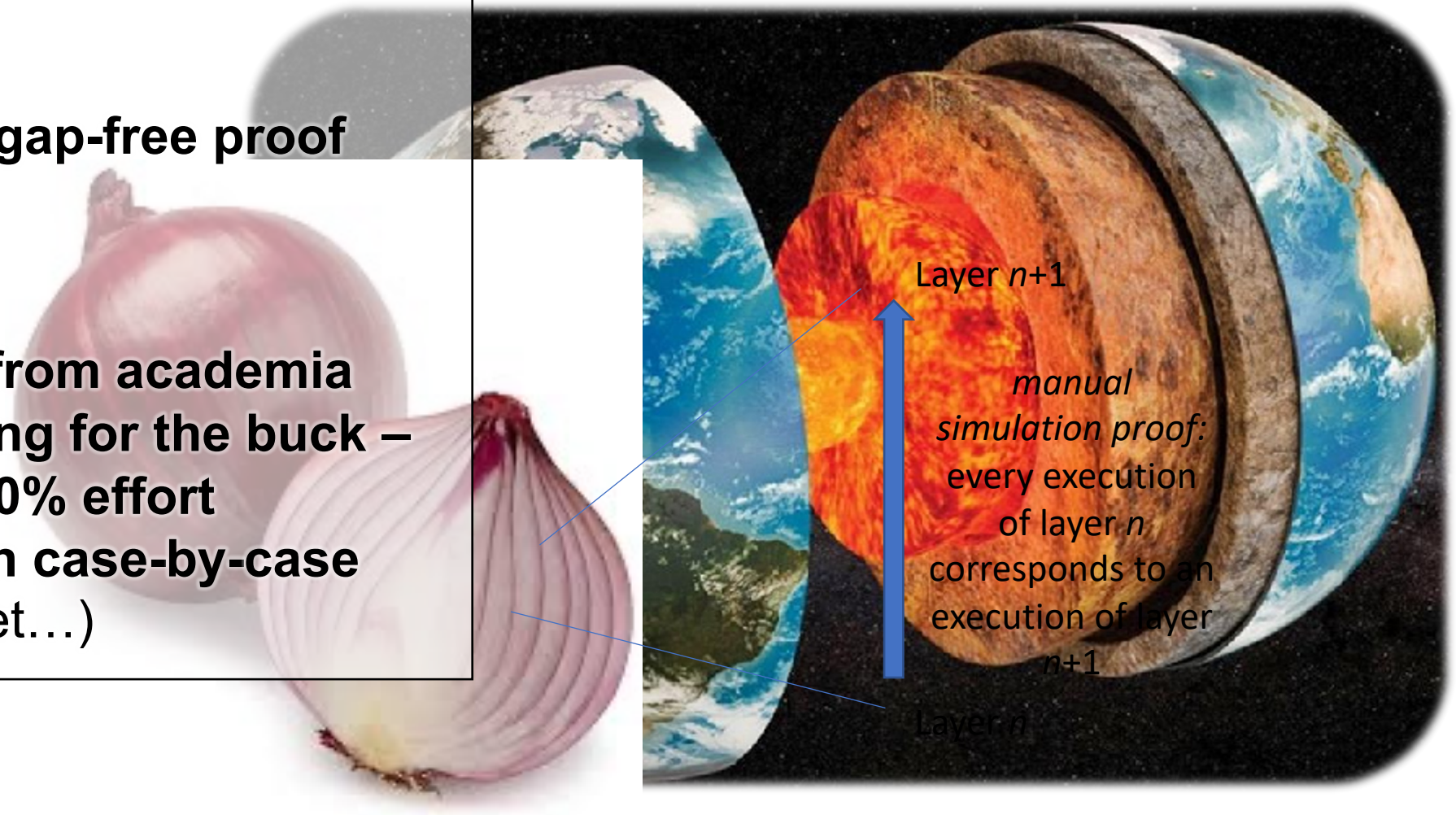
How to do end-to-end Verification?

- Much larger problem scale
- High speed of change

→ **manual, unbounded, gap-free proof can't keep up!**

What to do?

1. **learn how to think from academia**
2. **look for biggest bang for the buck – 80% verified with 20% effort**
3. **choose methods on case-by-case basis** (no silver bullet…)



Level of detail at each level

Application Layer

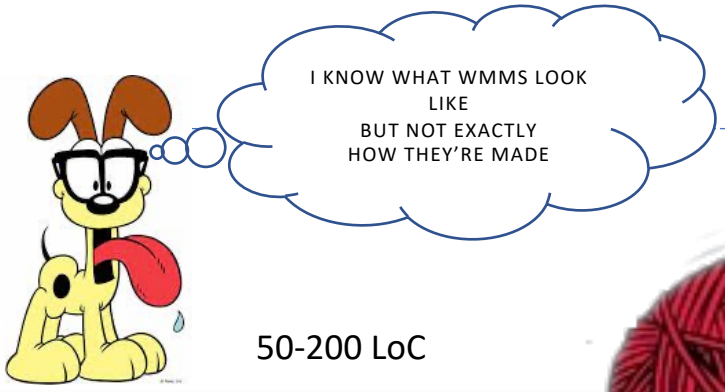
Concurrency Library

Kernel

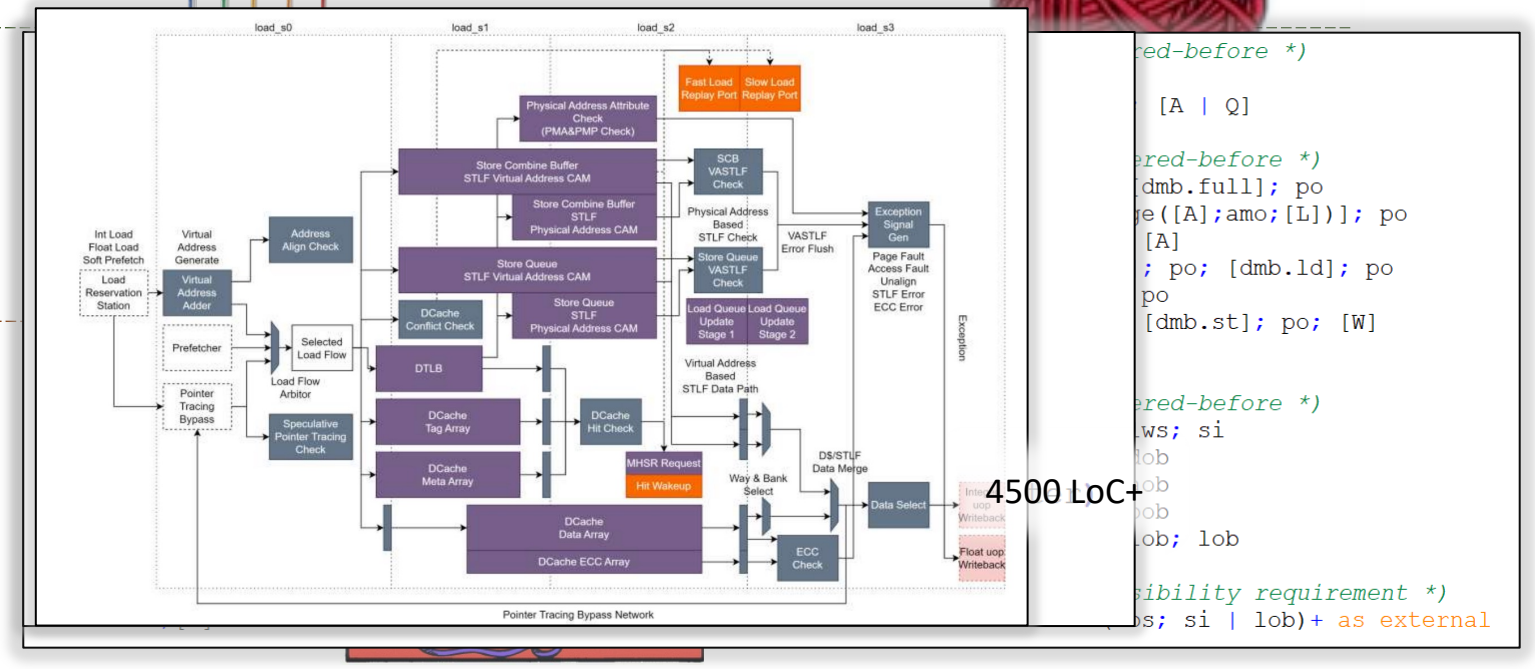
HW Mapping

Hardware

LEVEL OF DETAIL



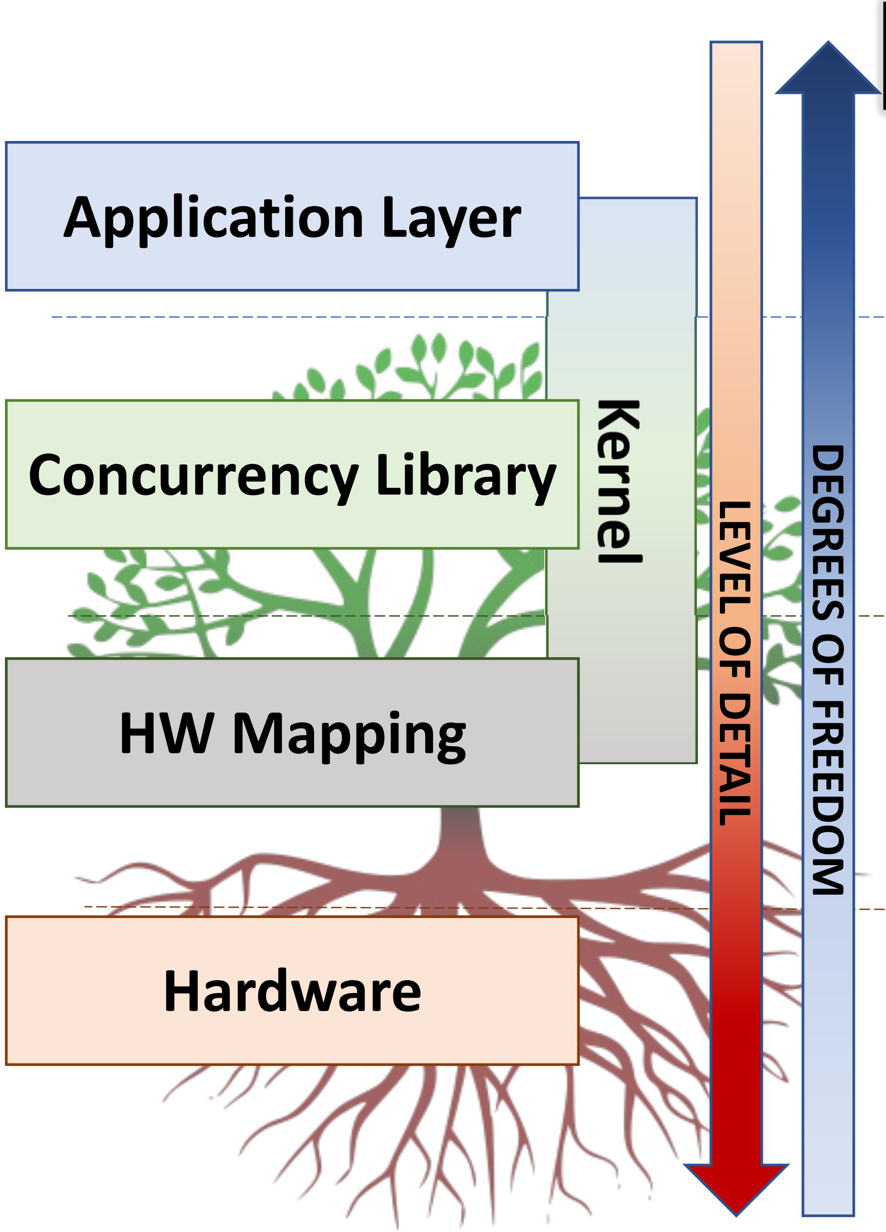
50-200 LoC



4500 LoC+

```
red-before *)

 [A | Q]

ered-before *)
dmb.full]; po
ge([A];amo;[L])]; po
[A]
; po; [dmb.ld]; po
po
[dmb.st]; po; [W]

ered-before *)
ws; si
ob
ob
ob
ob; lob

sibility requirement *)
s; si | lob)+ as external
```

Degrees of freedom at each level



???

Nobody knows what the spec is even
Other than “don't crash, don't hang”
0 false-positive deadlock prediction based on formal *happens-before* analysis

QEMU

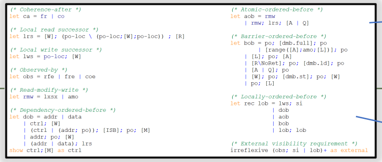
Queue<T>
::enq(a : T) ***(blocking)***
Q' := Q :: a
::deq() -> T
Q == [] -> return nil
Q == a::QR -> Q' := QR; return a

Up to millions LOC
probabilistic systematic testing based on order reduction theory

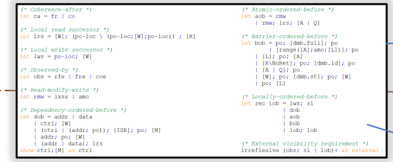
Many different specs, but each spec is small
Exploratory model checker on small test cases written by engineer

VSync
Dat3M
GenMC

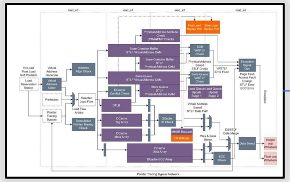
```
void T1() {
    q.enq(5);
    r1 = q.deq();
}

void T2() {
    q.enq(3);
    q.enq(4);
}

void T3() {
    r2 = q.deq();
    r3 = q.deq();
    if (r2 == 4) assert(r3 == 5);
}
```



Relatively standard spec (atomic add, atomic mult, …, atomic store…, release barrier, …)
Manually build proof robots for each of ≈6 classes
limited implementation space: ≈6 classes of implementations



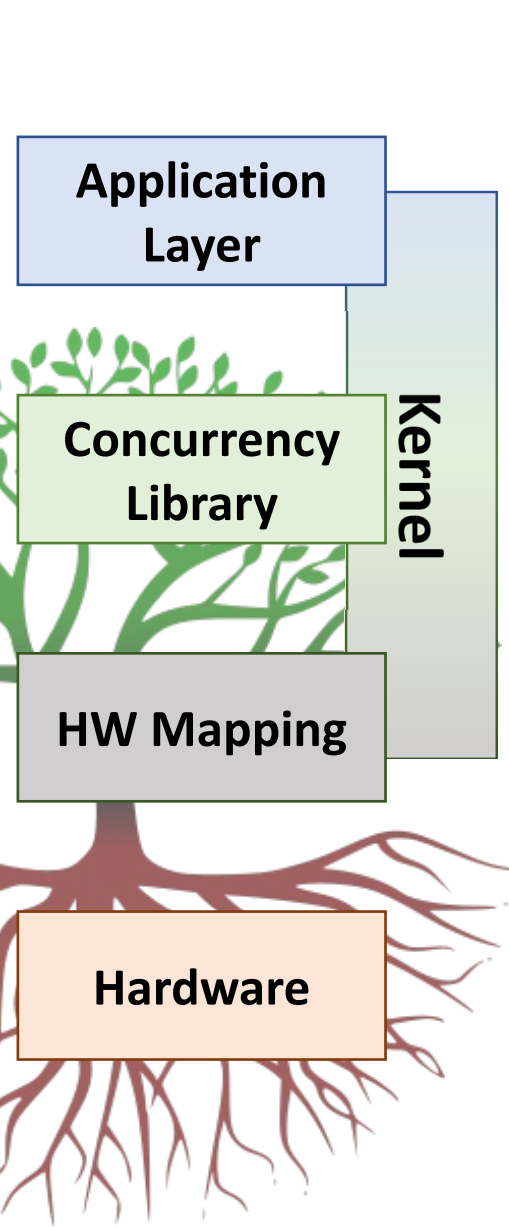
Extremely fixed HW spec (≈ ISA), “easy” to prove parts manually
Break down WMM into high-level … Arm, RISC-V, x86



Completely fixed at runtime
“new StoreBuffer()”
prove parts semi-automatically with SMT-solver to make sure details are right

Z3
CVC5

Case Study: Concurrency Library

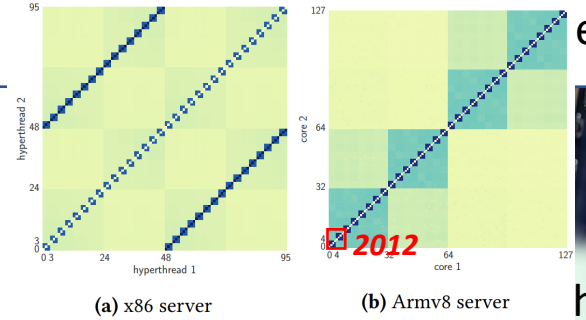


Opensource standard 2012-2021:

The Safest Car in the world!

performance of racing on same variable by two threads

Efficient Model Checkers can only check based on CPU ID



eness!

hole

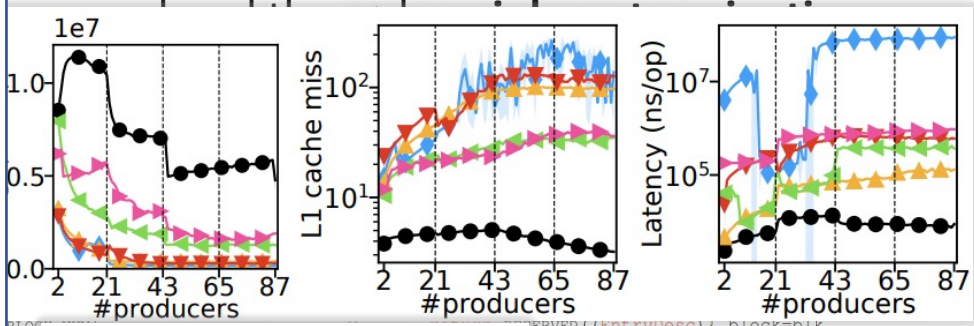
BBQ@ATC'22

avoid those races by working

- on different "blocks"
- 99% of algorithms only hang in spinloops

relax MCS lock

```
while (! my_node.go);
```



Found many bugs

- Used by our normal engineers when designing new complex concurrent algorithms to improve product performance
- 0 bugs reported from product lines stress tests

initial version:

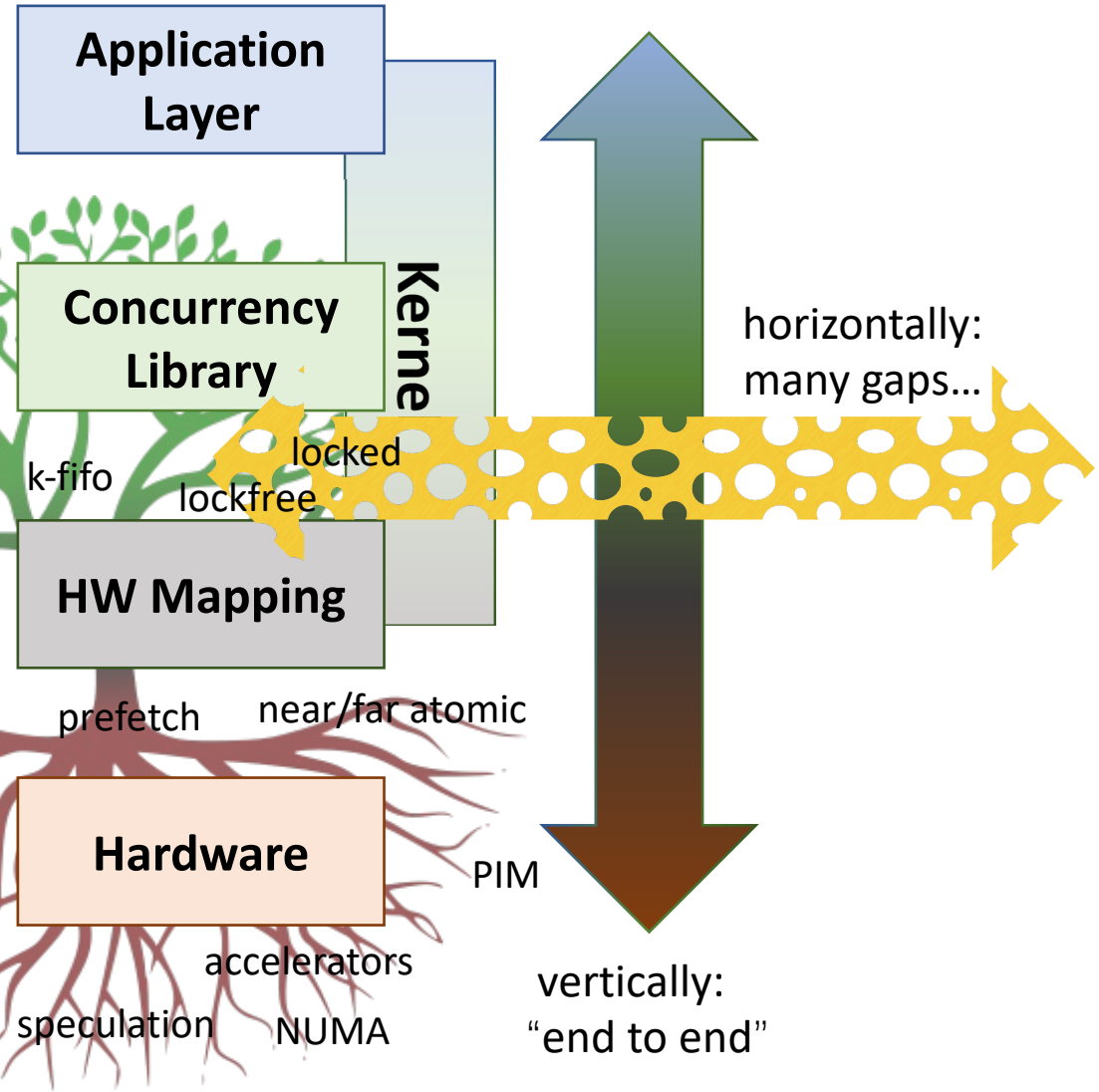
full of subtle bugs

even after 24 hour

```
class BBQ<T> {
  shared<Head> phead, chead;
  Block<T>[] blocks;
}
class Block<T> {
  shared<Cursor> allocated, comm
  shared<Cursor> reserved, consu
  T[] entries;
}
class EntryDesc {
  Block block; Offset offset; Version version; }
```

```
MAX(nblk.reserved, Cursor{.vsn=committed.vsn});
MAX(chead, ch + 1);
return true;
}
```

Summary & Lessons Learned



- Each layer has a different point in “detail” vs “degrees of freedom” making different methodologies and different amount of gaps effective
- Learned a lot about the architecture at each layer by opening “system box” and doing verification
- Could build more efficient concurrency algorithms due to that knowledge
- More efficient concurrency **needs** verification to manage complexity