arm

Atomic explosion: evolution and use of relaxed concurrency primitives

Kernel Recipes, Paris

Will Deacon <will.deacon@arm.com>

September, 2018

Intro

- Co-maintainer of `arm64` architecture, `ARM` perf backends, `SMMU` drivers, atomics, locking, memory model, TLB invalidation…
- Developer in the Open-Source Software group at Arm
- Close working relationship with Architecture and Technology Group
- Co-author of Armv8 architectural memory model
- Involved in C/C++ memory model working group

Unsurprisingly, I'm going to talk about concurrency.

Concurrency is the problem, not the solution

Imagine paying for an upgrade on a flight…

arm

Concurrency is the problem, not the solution

…but getting given this instead.

Concurrency is the problem, not the solution

…but getting given this instead.

We asked for performance, and they gave us concurrency.

Just say no!

arm

Concurrency is the problem, not the solution

…but getting given this instead.

We asked for performance, and they gave us concurrency.

Just say no!

Unfortunately, it's unavoidable in the kernel

arm

Low-level concurrency in Linux

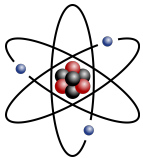
- Interrupts and preemption
- `spin_lock()`, `mutex_lock()`, `rwsem`
- `seqlock`
- RCU
- `cmpxchg()`, `xchg()`
- `lockref`
- `percpu-rwsem`
- `atomic_t`, `atomic64_t`
- `READ_ONCE()`, `WRITE_ONCE()`
- `smp_load_acquire()`, `smp_store_release()`
- `smp_mb()`, `smp_rmb()`, `smp_wmb()`

and there's more…

arm

Atomics

- Accesses to `atomic_t` guaranteed to be 'indivisible' (single-copy atomic)
- (Badly) described in `memory_barriers.txt`; `atomic_t.txt` much better.
- Core code provides lock/hash-based implementation which you probably don't want
- Traditionally, separated into three classes:



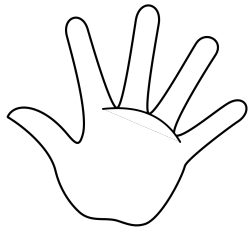
get/set Unordered access similar to `READ_ONCE/WRITE_ONCE` e.g. `atomic64_read()`

read-modify-write (rmw) Unordered posted operation e.g. `atomic_long_inc()`

value-returning rmw Returns new value with full ordering e.g. `atomic_add_return()`

Five historic limitations of `atomic_t` and friends

1. Limited set of operations
2. Unordered or fully ordered: nothing in-between
3. Implementation entirely duplicated per-arch
4. Independent of `cmpxchg()` etc
5. Not well defined or understood



Concurrency is hard: shouldn't force arch maintainers to take on burden of implementing atomics.

Milestones

- 47933ad4 ("arch: Introduce smp_load_acquire(), smp_store_release()"), Nov 2013
- e6942b7d ("atomic: Provide atomic_{or,xor,and}"), April 2014
- 654672d4 ("locking/atomics: Add _{acquire|release|relaxed}() variants of some atomic operations"), Aug 2015
- 28aa2bda ("locking/atomic: Implement atomic{,64,_long}_fetch_{add,sub,and,andnot,or,xor}{,_relaxed,_acquire,_release April 2016
- 1f03e8d2 ("locking/barriers: Replace smp_cond_acquire() with smp_cond_load_acquire()"), April 2016
- 3942b771 ("MAINTAINERS: Claim atomic*_t maintainership"), Nov 2016
- 087133ac ("locking/qrwlock, arm64: Move rwlock implementation over to qrwlocks"), Oct 2017
- 1c27b644 ("Automate memory-barriers.txt; provide Linux-kernel memory model"), Jan 2018
- c1109047 ("arm64: locking: Replace ticket lock implementation with qspinlock"), March 2018

arm

Semantics

Extensions include:

`Bitwise` operations

`*_fetch` ops return old value prior to atomic update

`*_relaxed` no ordering required

`*_{acquire,release}` message passing

`smp_cond_load_acquire()` poll with acquire semantics until condition is satisfied

Core code will generate what the arch doesn't provide!

- `cmpxchg`-based atomics in `asm-generic/atomic.h`
- `atomic`-based bitops in `asm-generic/bitops/*`

Old API remains for unordered and fully-ordered atomics.

Relaxed

- Unordered – even the compiler can reorder!
- Single-copy atomic
- Fiddly to use (esp. value-returning variants) but indispensable at times
- Often (but not always) used in conjunction with fences

```
P0                                      P1
atomic_fetch_inc_relaxed(&x); | atomic_fetch_inc_relaxed(&x);
```


arm

Adoption of `_relaxed` atomics in mainline

Unfortunately, adoption of the atomic extensions has been slow…

Adoption of `_relaxed` atomics in mainline

Unfortunately, adoption of the atomic extensions has been slow…

Author	Number of `_relaxed` atomics
Will Deacon:	12
Catalin Marinas:	5
Peter Z:	3
Robin Murphy:	2
Kevin Brodsky:	1
David Howells:	1
Waiman Long:	1
Davidlohr Bueso:	1
Trond Myklebust:	1

`smp_load_acquire`, `smp_store_release` are doing much better, but have a headstart and are generally ‘safer’.

Fully-ordered

- As if there's an `smp_mb()` on either side of the operation
- (See `smp_mb__{before,after}_atomic`)
- Orders all access types across the operation (inc. ST->LD)
- Expensive on all architectures (inc. x86)
- Sometimes referred to as 'SC-restoring'
- Even in the presence of racy writes:

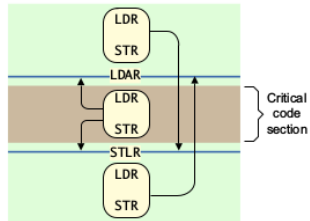
```
P0                              P1
WRITE_ONCE(*x, 1);          |   WRITE_ONCE(*y, 2);
atomic_inc_return(&p);      |   atomic_inc_return(&q)
WRITE_ONCE(*y, 1);          |   READ_ONCE(*x)
```

arm

Acquire/Release

Middle-ground between relaxed and fully-ordered:

- Appeals to “message-passing” idiom
- Producer thread writes/releases data
- Consumer thread reads/acquires the same data
- Maps efficiently to existing architectures and C/C11
- ‘Roach-motel’ semantics

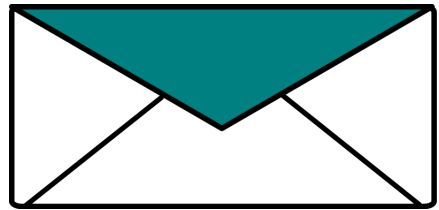


Everthing before a release is visible to everything after an acquire that reads from the release.

More flexible than `smp_wmb()/smp_rmb()` but without enforcing ST->LD ordering of `smp_mb()`.

arm

Acquire/Release

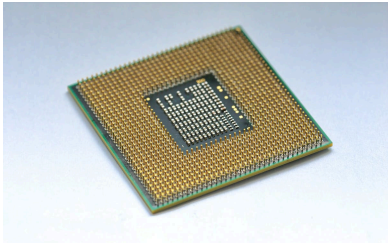


Acquire/release operations can be chained together without loss of cumulativity:

```
P0                          P1                                P2
WRITE_ONCE(*x,1);         | atomic_read_acquire(y);       | atomic_xchg_acquire(z,2);
atomic_set_release(y,1); | atomic_fetch_inc_release(z); | READ_ONCE(*x);
```

Try doing this with fences.

Show me the code!



	x86	arm64	ppc
`smp_load_acquire`	`MOV`	`LDAR`	`LD; LWSYNC`
`smp_store_release`	`MOV`	`STLR`	`LWSYNC; ST`
`atomic_fetch_add_release`	`LOCK XADD`	`LDADDL`	`LWSYNC; LL/SC`
`smp_mb()`	`LOCK ADDL`	`DMB ISH`	`SYNC`

RISC-V also has native support.

Generic locking code: `kernel/locking/*`

Generic locking implementations

Can we really have our cake and eat it?

Portability: implemented entirely using in-kernel concurrency APIs. No need for additional assembly code! Can also be ported to userspace/bare-metal.

Performance: use of relaxed atomics to implement complex, scalable, fair algorithms

Correctness: formal modelling as well as extensive testing on multiple architectures

Let's look at some examples…

qrwlock layout

```
typedef struct qrwlock {
    union {
        atomic_t            cnts;
        struct {
            u8 wmode;       /* Writer mode: 0 or LOCKED (0xff) */
            u8 __lstate[3]; /* 23-bit reader count + WAITING bit */
        };
    };
    arch_spinlock_t     wait_lock;
} arch_rwlock_t;
```

Put the writer count in its own byte and use a spinlock for implicit queueing.

qrwlock

`write_lock()` `cmpxchg` on lockword `0 => LOCKED` (acquire)

`write_unlock()` Clear `wmode` to 0 (release)

`read_lock()` Increment reader count if `wmode` is 0 (acquire)

`read_unlock()` Decrement reader count (release)

If a `lock()` operation fails, then take the `wait_lock` which gives us queueing for free!

- `spin_lock()` acquisition implies head of queue
- Writers poll for all others to drain (set `WAITING` bit)
- Readers poll for writers to drain

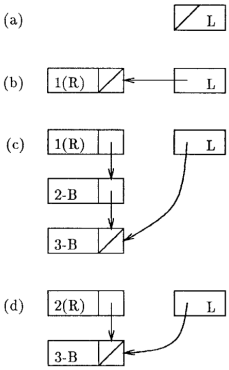
qrwlock results

```
// locktorture 2w/8r/rw_lock_irq
rwlock: (191:1)
  Writes:  Total: 6612  Max/Min: 0/0   Fail: 0
  Reads :  Total: 1265230  Max/Min: 0/0   Fail: 0
  Writes:  Total: 6709  Max/Min: 0/0   Fail: 0
  Reads :  Total: 1916418  Max/Min: 0/0   Fail: 0
  Writes:  Total: 6725  Max/Min: 0/0   Fail: 0
  Reads :  Total: 5103727  Max/Min: 0/0   Fail: 0

qrwlock: (6:1)
  Writes:  Total: 47962  Max/Min: 0/0   Fail: 0
  Reads :  Total: 277903  Max/Min: 0/0   Fail: 0
  Writes:  Total: 100151  Max/Min: 0/0   Fail: 0
  Reads :  Total: 525781  Max/Min: 0/0   Fail: 0
  Writes:  Total: 155284  Max/Min: 0/0   Fail: 0
  Reads :  Total: 767703  Max/Min: 0/0   Fail: 0
```

`qspinlock`: generic spinlock implementation

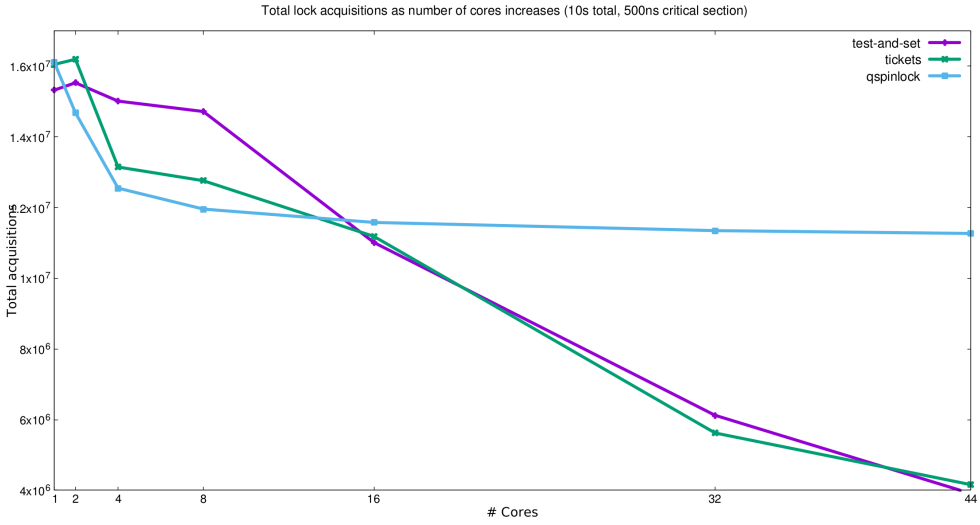
Complex locking implementation based around MCS locks:



- Lockword points to end of linked waiter list
- Each CPU spins on their own cacheline within their list node
- When unlocking, write to the next node in the queue
- Linux implementation optimises the low-contention case, avoids dynamic node allocation and squeezes everything into a 32-bit word (`atomic_t`)

Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors – Mellor-Crummey & Scott, 1991

qspinlock: scaling under contention



arm

Verification tools

‘Beware of bugs in the above code; I have only proved it correct, not tried it.’

LKMM

'Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel' – https://dl.acm.org/citation.cfm?id=3177156

```
C MP+polocks

P0(int *x, int *y, spinlock_t *mylock)
{
  WRITE_ONCE(*x, 1);
  spin_lock(mylock);
  WRITE_ONCE(*y, 1);
  spin_unlock(mylock);
}
```

```
P1(int *x, int *y, spinlock_t *mylock)
{
  int r0;
  int r1;

  spin_lock(mylock);
  r0 = READ_ONCE(*y);
  spin_unlock(mylock);
  r1 = READ_ONCE(*x);
}

exists (1:r0=1 /\ 1:r1=0)
```

tools/memory-model/

```
$ herd7 -conf linux-kernel.cfg litmus-tests/MP+polocks.litmus
Test MP+polocks Allowed
States 3
1:r0=0; 1:r1=0;
1:r0=0; 1:r1=1;
1:r0=1; 1:r1=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:r0=1 /\ 1:r1=0)
Observation MP+polocks Never 0 3
Time MP+polocks 0.01
Hash=602e4c28ae61714bf6072f8a98078bd7
```

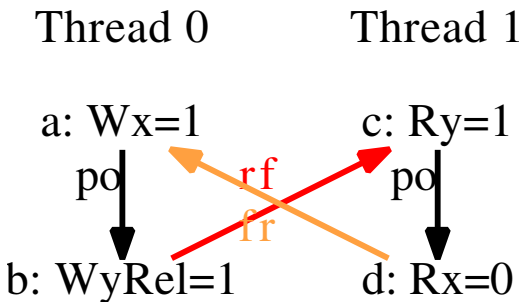
- Strong vs weak
- Compiler transforms
- Preemption
- I/O
- Tests as modules

TLA+

- *TLA*$^+$ (Temporal Logic of Actions) is a formal specification language developed by Leslie Lamport
  - Based on set theory and temporal logic, can specify invariant and liveness properties
  - Specification written in formal logic is amenable to finite model checking (using *TLC* model checker)
  - Can also be used for machine-checked proofs of correctness
- *PlusCal* is a formal specification language which transpiles to *TLA*$^+$
  - Pseudocode like, better suited to specify sequential algorithms
  - Simple to describe SC concurrent threads/processes
- Used to model `qrwlock`, `qspinlock` and parts of the arm64 kernel!
  - `git://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/kernel-tla.git`
  - Proved exclusiveness of locking algorithms
  - Proved that forward progress is always made by each thread
  - `qrwlock`: 2+2 reader/writer
  - `qspinlock`: 3 lockers

https://github.com/herd/herdtools7

```
AArch64 MP+popl+po
"PodWWPL RfeLP PodRR Fre"
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0            | P1           ;
 MOV W0,#1     | LDR W0,[X1] ;
 STR W0,[X1]   | LDR W2,[X3] ;
 MOV W2,#1     |              ;
 STLR W2,[X3] |              ;
exists
(1:X0=1 /\ 1:X2=0)
```



arm

Example litmus test: MP+popl+po

```
AArch64 MP+popl+po
"PodWWPL RfeLP PodRR Fre"
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0            | P1           ;
 MOV W0,#1     | LDR W0,[X1] ;
 STR W0,[X1]   | LDR W2,[X3] ;
 MOV W2,#1     |              ;
 STLR W2,[X3] |              ;
exists
(1:X0=1 /\ 1:X2=0)
```

```
Test MP+popl+po Allowed
States 4
1:X0=0; 1:X2=0;
1:X0=0; 1:X2=1;
1:X0=1; 1:X2=0;
1:X0=1; 1:X2=1;
Ok
Witnesses
Positive: 1 Negative: 3
Condition exists (1:X0=1 /\ 1:X2=0)
Observation MP+popl+po Sometimes 1 3
Time MP+popl+po 0.01
Hash=75d804cb38f3f607de6ab3cc9925140e
```


arm

Testing

Ongoing work in academia to improve formal tools, but until then…

locktorture to stress mutex, spinlock, rwlock, rwsem

rcutorture to stress RCU, CPU hotplug

lkmm modules to run a ‘litmus test’ from within the kernel

Generic locking implementations automatically get cross-arch testing!

But what does this have to do with YOU?

Patch review

So you've received a patch using relaxed/weak atomics?

- Most people don't need this stuff: use RCU, locking or existing high-level interfaces where possible
- Acquire/release in preference to `smp_*mb()`
- Discourage legacy `atomic_*_return()` ops
- Acquire/release should be paired; don't mix-and-match with barriers if you can avoid it
- Require comments showing the pairing
- Heavy fences generally only needed for racy writes
- Try to express the problem as a litmus test for LKMM.

and last, but not least…

Who are we?

We're here to help!

Will Deacon <will.deacon@arm.com>

Boqun Feng <boqun.feng@gmail.com>

Paul McKenney <paulmck@linux.vnet.ibm.com>

Ingo Molnar <mingo@redhat.com>

Alan Stern <stern@rowland.harvard.edu>

Peter Zijlstra <peterz@infradead.org>

…and others in `MAINTAINERS`.

arm

Conclusion

The kernel's low-level concurrency primitives have never looked so good:

- Portable and efficient abstraction of the underlying machine
- Parity with modern programming languages
- Off-the-shelf synchronisation code suitable for production
- Ability to reason about concurrent behaviours
- Active group of maintainers

Generic concurrent code doesn't have to suck!

Questions?