Current sorry state of C11 code and suggestions to fix it

PHIL YANG
ARM
2019-09-20

Agenda

- The rte_atomic APIs
- The problem
- Fix suggestions
  - Obsolete rte_atomic APIs
  - Use C11 atomics instead
  - Examples
- AArch64 support for C11 atomic APIs
  - Armv8.0-a acquire-release instructions
  - Armv8.1-a LSE atomic
  - Armv8.3-a weaker RCpc model

DPDK rte_atomic Family APIs

- rte_atomic16/32/64_init
- rte_atomic16/32/64_read
- rte_atomic16/32/64_set
- rte_atomic16/32/64_add
- rte_atomic16/32/64_sub
- rte_atomic16/32/64_inc
- rte_atomic16/32/64_dec
- …

http://gitpdk.org/dpdk/tree/lib/librte_eal/common/include/generic/rte_atomic.h

The Implementation on X86

- Based on locked instructions, explicit or implicit. e.g. “xchg”
  - Locked instructions have a strict memory order (two-way barrier)
  - Loads and stores are not reordered with locked instructions.

```
1.  static inline uint64_t
2.  rte_atomic64_exchange(volatile uint64_t *dst,
    uint64_t val)
3.  {
4.          asm volatile(
5.                  MPLOCKED
6.                  "xchgq %0, %1;"
7.                  : "=r" (val), "=m" (*dst)
8.                  : "0" (val),  "m" (*dst)
9.                  : "memory");    /* no-clobber list */
10.         return val;
11. }
```

```
1.  static inline void
2.  rte_atomic64_inc(rte_atomic64_t *v)
3.  {
4.          asm volatile(
5.                  MPLOCKED
6.                  "incq %[cnt]"
7.                  : [cnt] "=m" (v->cnt)   /* output */
8.                  : "m" (v->cnt)          /* input */
9.                  );
10. }
```

The Implementation on PPC64

- Some allowing reordering

```
static inline void
rte_atomic64_add(rte_atomic64_t *v, int64_t inc)
{
        long t;
        asm volatile(
                        "1: ldarx %[t],0,%[cnt]\n"
                        "add %[t],%[inc],%[t]\n"
                        "stdcx. %[t],0,%[cnt]\n"
                        "bne- 1b\n"
                        : [t] "=&r" (t), "=m" (v->cnt)
                        : [cnt] "r" (&v->cnt), [inc] "r" (inc), "m" (v->cnt)
                        : "cc", "memory");
}
```

- Some don't

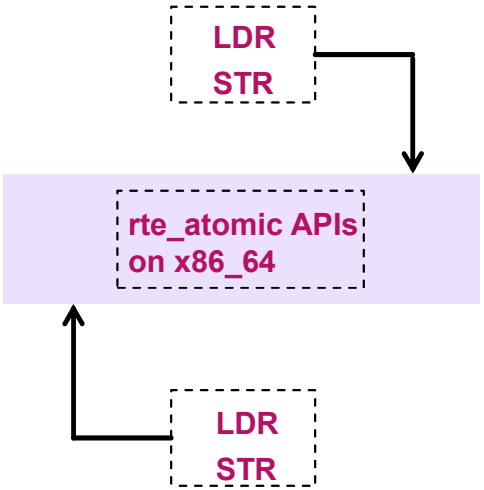
```
static inline int rte_atomic64_cmpset(…
        asm volatile (
                        "\tlwsync\n"
                        "1: ldarx %[ret], 0, %[dst]\n"
                        "cmpld %[exp], %[ret]\n"
                        "bne 2f\n"
                        "stdcx. %[src], 0, %[dst]\n"
                        "bne- 1b\n"
                        "li %[ret], 1\n"
                        "b 3f\n"
                        "2:\n"
                        "stdcx. %[ret], 0, %[dst]\n"
                        "li %[ret], 0\n"
                        "3:\n"
                        "isync\n"
                        : [ret] "=&r" (ret), "=m" (*dst)
                        : [dst] "r" (dst),  [exp] "r" (exp),       [src] "r" (src),
                          "m" (*dst)
                        : "cc", "memory");
        return ret;
}
```

The Implementation on AArch64

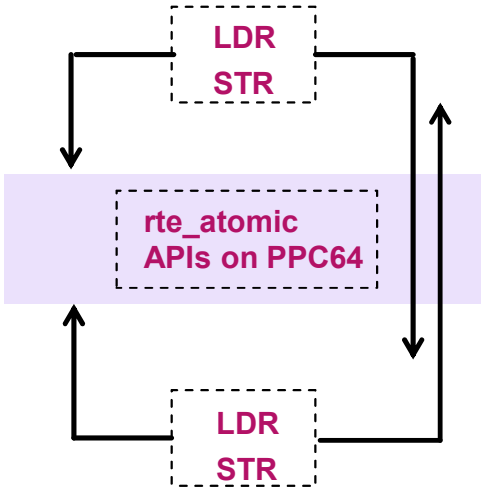
- For AArch64
  - The generic implementation in use.
  - Implemented with the “__sync” builtins
    - Implicitly 2-way full barriers

```
static inline void
rte_atomic16_add(rte_atomic16_t *v, int16_t
inc)
{
        __sync_fetch_and_add(&v->cnt, inc);
}
```

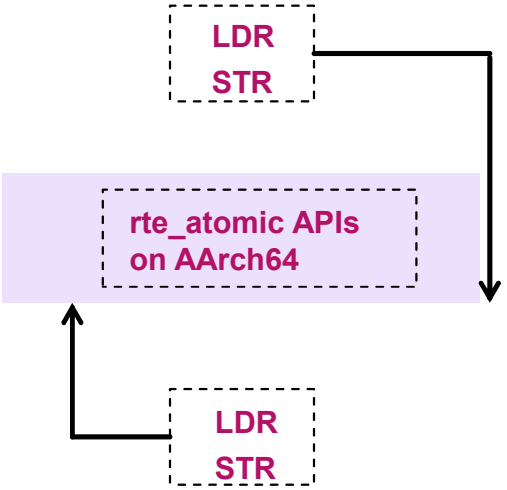
Ordering Semantics for Different Arches



X86 – Total Order



PPC64 – Partial Order



AArch64 – Bottom Line Order

What's The Problem?

- **Flexibility** - The APIs, by (lack of) definitions, don't have memory ordering semantics
- **Inconsistency** - Different arches implemented with different implicit ordering semantics
- **Performance** - May decrease performance if stronger order than required
- **Correctness** - Not correct if user requires stronger than actually provided

How Can We Fix It?

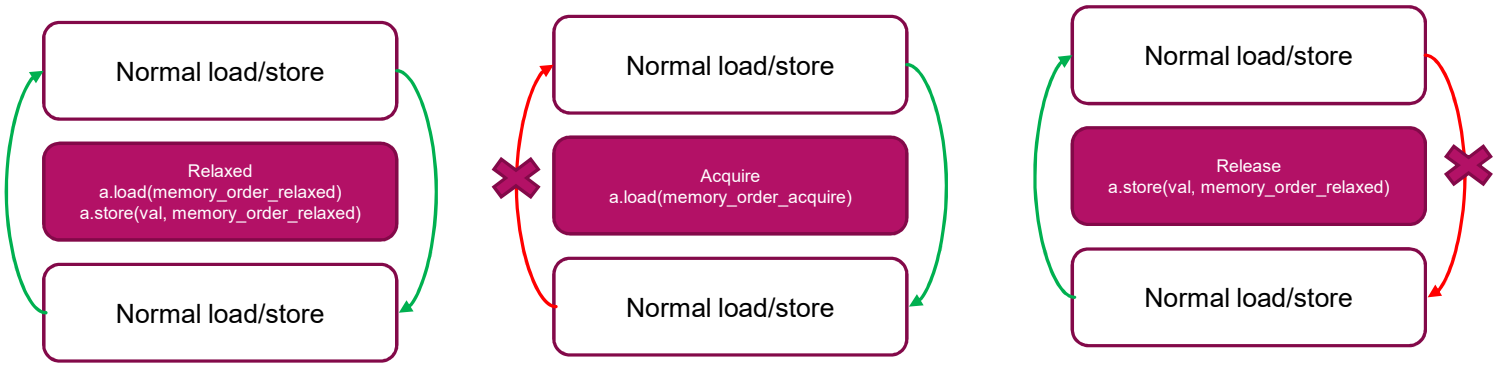
- **Stop using it!**

- Use C11 atomic APIs instead.
  - C11 offers different memory orderings semantics
  - All the arches aligned to the semantics
    - __ATOMIC_RELAXED
    - __ATOMIC_ACQUIRE
      __ATOMIC_RELEASE
    - __ATOMIC_CST_SEQ
    - …

- For the missing APIs, we can add, e.g. 128-bit compare and exchange:
  - http://patches.dpdk.org/patch/57675/

Acquire/Release Semantics

- One way barriers.
- Allows for ordering in the other direction
  - Ideal for producer/consumer type use cases (pairing!!).
  - After an ACQUIRE on a given variable, all memory accesses preceding any prior RELEASE on that same variable are guaranteed to be visible.
  - All accesses of all previous critical sections for that variable are guaranteed to have completed.
  - C11's __ATOMIC_RELAXED/__ATOMIC_RELEASE/ATOMIC_ACQ_REL.



AArch64 Implementation Revisited

- Source code in rte_atomic.h

```
static inline void
rte_atomic16_add(rte_atomic16_t *v, int16_t inc)
{
        __sync_fetch_and_add(&v->cnt, inc);
}
```

- Assembly code on AArch64

```
15          add     x0, sp, 8
16      .L6:
17          ldxr    x1, [x0]
18          add     x1, x1, 1
19          stlxr   w2, x1, [x0]
20          cbnz    w2, .L6
21          dmb     ish
```

Release order, which may be unnecessary

Two-way barrier, which is unnecessary

https://gcc.godbolt.org/z/lKQEAW

Example Revisited

- rte_atomic_add re-implementation with "__atomic" builtins
  - "__sync" builtins version

```
15          add     x0, sp, 8
16      .L6:
17          ldxr    x1, [x0]
18          add     x1, x1, 1
19          stlxr   w2, x1, [x0]
20          cbnz    w2, .L6
21          dmb     ish
```

two-way barrier

armv8.0-a

  - "__atomic" builtins version

```
4           add     x0, sp, 8
5       .L2:
6           ldxr    x1, [x0]
7           add     x1, x1, 1
8           stlxr   w2, x1, [x0]
9           cbnz    w2, .L2
```

one-way barrier

https://gcc.godbolt.org/z/lKQEAW

Memif PMD

- Using 'rte_mb' to synchronize the ring shared data
- Will stall pipeline and decrease performance
- Optimized with c11 one-way barriers gave increased performance (throughput)

http://patchwork.dpdk.org/patch/57960/

More Examples

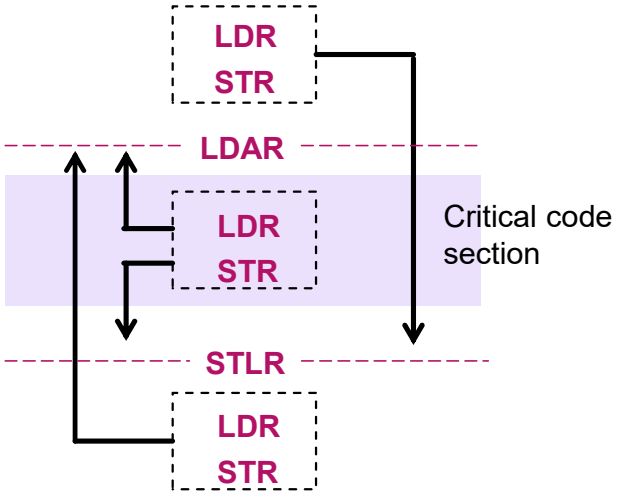
- rte_ring lib
  - lib/librte_ring/rte_ring_c11_mem.h
- rte_stack lib
  - lib/librte_stack/rte_stack_lf_c11.h
- Locks
  - generic/rte_spinlock.h
  - generic/rte_mcslock.h
  - generic/rte_rwlock.h
- …

AArch64 evolving to better support C11 atomics

- Armv8.0-a - LDAXR/STLR instructions
- Armv8.1-a – Atomics instructions
- Armv8.3-a – RCPC, ldapr instructions

Armv8.0-a Instructions

- No explicit barrier instructions like **DMB**, **DSB**
- **LDAR** and **STLR** instructions may be used as a pair
  - To protect a critical section of code
  - May have lower performance impact than a full **DMB**
  - No ordering is enforced *within* the critical section
- Exclusive versions also available
  - **LDAXR**, **STLXR**
  - Remove the need for explicit barrier instructions



Fit for the requirements of C11 memory_order_release, memory_order_acquire and memory_order_acqrel

Armv8.1-a LSE Atomic Instructions

- ARMv8.1-A introduces new atomic instructions
  - **CAS** - Compare and swap
  - **LD<OP>** - Load and <operation>
  - **SWP** - Swap
- Eliminate the need of a loop with LSE enabled on armv8.1-a
  - https://gcc.godbolt.org/z/eNd8Zl
- Atomics can optionally have an ordering specifier
  - **A**=Acquire, **L**=Release or **AL**=Acquire & Release
- rte_atomic_add disassembly code
- "__sync" builtins version

```
add x0, sp, 12
mov w1, 1
ldaddal w1, w1, [x0]
```

armv8.1-a

- "__atomic" builtins version

```
add x0, sp, 12
mov w1, 1
ldaddl w1, w1, [x0]
```

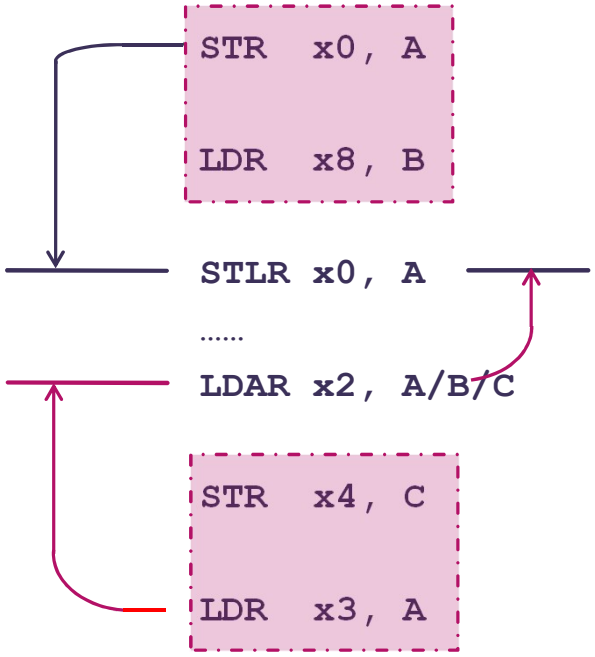
Armv8.3-a RCpc Model

- The base **armv8.0-a** architecture supports **RCsc** (Release Consistency **sequentially** consistent)
  - A Store-Release followed by a Load-Acquire cannot be re-ordered with respect to each other
  - Good fit for the requirements of C++11/C11 memory_order_seq_cst

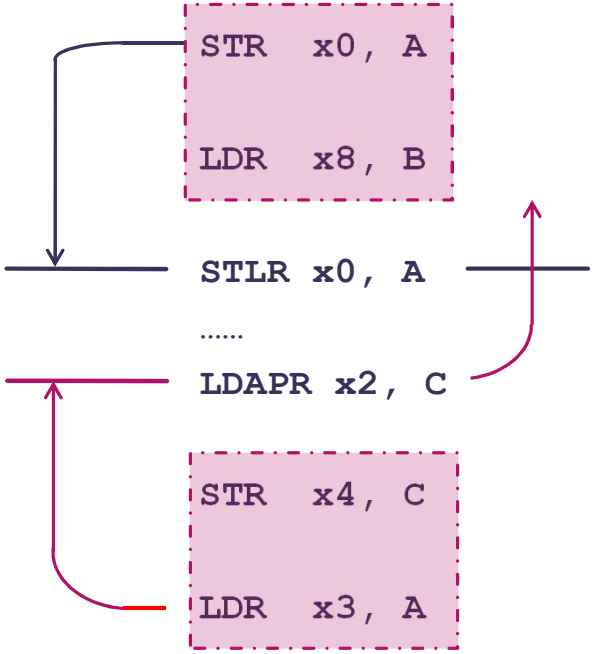
- **armv8.3-a** add the **LDAPR** instruction
  - Based on the weaker **RCpc** (Release Consistent **processor** consistent) model
  - LDAPR can be
  - Good fit for the requirements of C11 memory_order_release, memory_order_acquire and memory_order_acqrel
  - **No change to existing barriers**

Armv8.3-a RCpc Model

- Load acquire and Store release have a strict ordering.
- RCpc introduces instructions to support a weaker load-acquire.



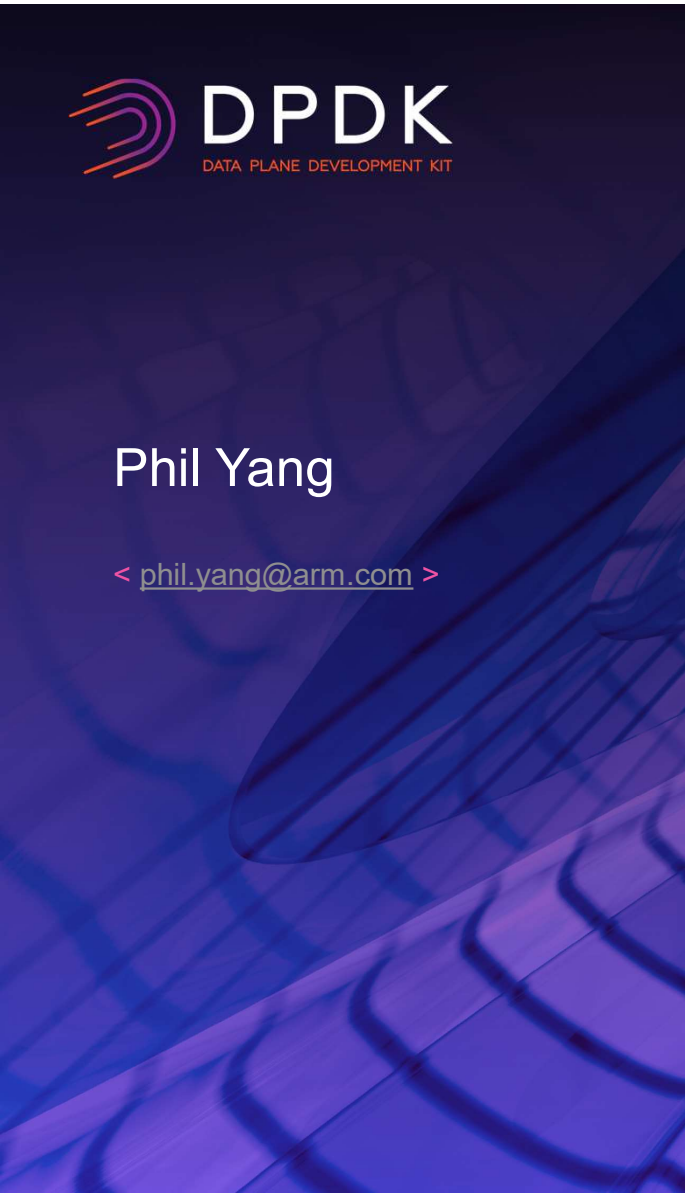
STLR always observed before LDAR



if A ≠ C, then it is permissible for the LDAPR to be observed before the STLR.

Key Takeaways

- The rte atomic APIs are defined without ordering semantics
  - Implemented with different ordering implications on different architectures
  - Hurt performance
  - May cause synchronization problems
  - Applications are not portable between architectures!
- GCC/C11 __atomic APIs are more flexible and relaxed
  - Performance gains manifested through examples like locks, rings, etc.
  - Programmers explicitly specify the required orderings
- AArch64 support for C11 atomic APIs
  - Take full advantage of out-of-ordering execution



Phil Yang

< phil.yang@arm.com >

Thanks !