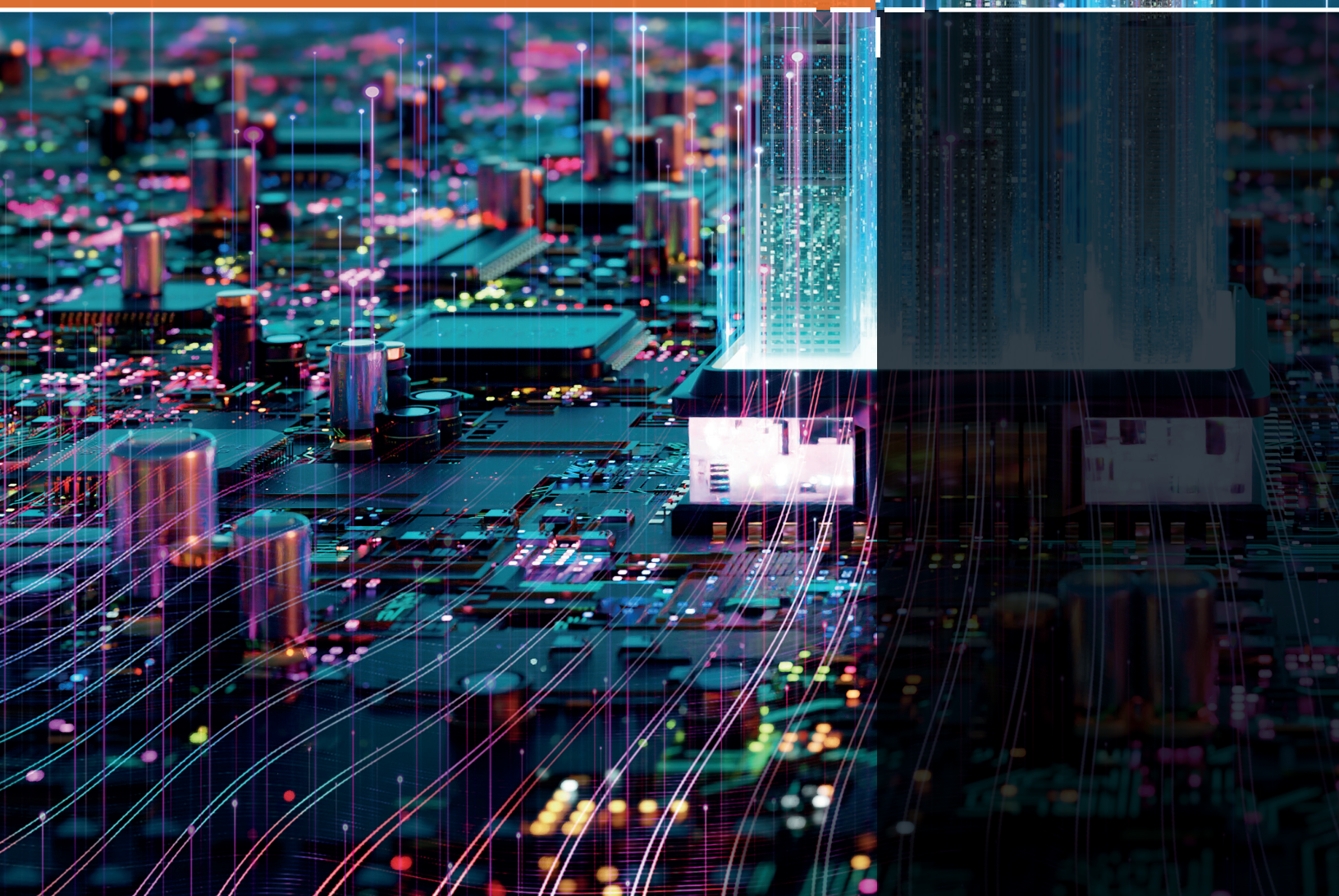
WHITE PAPER

arm

Synchronization Overview and Case Study on Arm Architecture

Ker Liu, Zaiping Bie



Contents

1–0verview

The objective of this white paper is to share synchronization knowledge on Arm architecture. The target reader of this document is those who work on synchronization with the Arm® architecture.

[Warning] When we are dealing with locking optimizations, we must be extremely careful about correctness. Bugs caused by synchronization are usually hard to root cause and the optimized code may crash on other CPUs with different or future micro-architectures design.

This document also provides information on typical enhancements and offers a deep-dive case study .

2–Synchronization Approach on Arm®v8-A Architecture

2.1 Atomic operation

Implementing synchronization (lock, mutex etc) requires atomic access.
The Arm architecture defines two types of atomic access:

1. Load and store exclusive, which has been supported since the Armv6 architecture release.
2. Atomic operation, which was introduced in Armv8.1-A large system extension (LSE).

2.1.1 Exclusive load and store

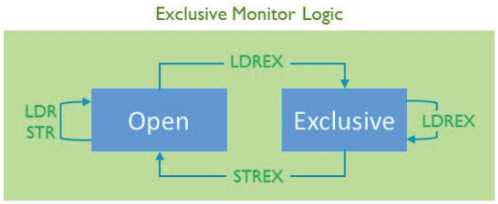
Arm supports load exclusive and store exclusive access instructions. In the A32 and T32 instruction sets, it is LDREX and STREX, and in the A64 instruction set, it is LDXR and STXR. These instructions use a try-and-test mechanism to implement atomic access.

LDREX/LDXR - The load exclusive instruction performs a load from an addressed memory location, the PE (for example, the CPU) also marks the physical address being accessed as an exclusive access. Store exclusive instructions check for the exclusive access mark.

STREX/STXR - The store exclusive instruction tries a value from a register to memory if the PE (for example, the CPU) has exclusive access to the memory address. The instruction returns a status value of 0 if the store was successful, or of 1 if no store was performed.

These instructions can be used by a lock construct, such as a mutex, to avoid race conditions when writing a lock value. Race conditions can occur in multi-threaded or multi-core systems, or during interrupt service routine (ISR) conditions.

In hardware, the core includes a logic named the exclusive monitor (sometimes, an external or global exclusive monitor might also be required). This monitor observes the core. When the core performs a load exclusive access, it records that fact in the exclusive monitor. When it performs a store exclusive, it checks whether another store exclusive has been performed successfully between the load exclusive and this store exclusive. In this case, the store exclusive fails.



A lock routine might look like this:

```
; void lock ( lock_t * ptr )
lock:
        ; Is it locked?
        LDXR    W1, [X0]            ; Load current value of lock
        CMP     W1, #LOCKED         ; Compare with "LOCKED"
        B.EQ    lock                ; If LOCKED, try again

        ; Attempt to lock
        MOV     W1, #LOCKED
        STXR    W2, W1, [X0]         ; Attempt to lock
        CBNZ    W2, lock             ; If STXR failed, try again
        DMB     SY                   ; Ensures accesses to the resource are not made
                                     ; before the lock is acquired
        RET
```

And the corresponding unlock function:

```
; void unlock ( lock_t * ptr )
unlock:
        DMB     SY                  ; Ensure accesses to protected resource have
                                    ; completed
        MOV     W1, #UNLOCKED       ; Release the lock
        STR     W1, [X0]
        RET
```

Note that STXR is not required in an unlock operation. It is only used when testing and setting it in a lock operation.

2.1.2 LSE Atomic operation

LSE atomic instructions can be used as an alternative to load-exclusive/store-exclusive instructions, for example to ease the implementation of atomic memory updates in very large systems. LSE atomics can be used with a closely coupled cache, sometimes referred to as near atomics, or further out in the memory system as far atomics.

Unlike LDXR/STXR, which uses try and test mechanism, LSE atomic is a forced atomic access. The instructions provide an atomic update of register content with memory for a range of conditions:

- Compare and swap instructions, CAS, and CASP. These instructions perform a read from memory and compare it against the value held in the first register. If the comparison is equal, the value in the second register is written to memory. If the write is performed, the read and write occur atomically. No other modification of the memory location can take place between the read and write.
- Atomic memory operation instructions, LD<OP>, and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN. Each instruction atomically loads a value from memory, performs an operation on the values, and stores the result back to memory. The LD<OP> instructions save the originally read value in the destination register of the instruction.
- Swap instruction, SWP. This instruction atomically reads a location from memory into a register and writes a different supplied value back to the same memory location.

A lock routine might look like this:

```
; void lock ( lock_t * ptr )
lock:
        MOV             W1, #LOCKED
        ; The unsigned maximum of (UNLOCKED, LOCKED) should
        ; return UNLOCKED if we have just set LOCKED
        5:  LDUMAXA     W1, W1, [X0]
        CBNZ            W1, 5b
        RET
```

And the corresponding unlock function:

```
; void unlock (lock_t *ptr)
unlock:
        MOV             W1, #UNLOCKED
        STLR            W1, [X0]
```

2.2 Arm memory ordering

Arm defines a weak memory ordering model, which means that accesses might not occur in program order. Arm memory consistency model allows:

Type	Reorder allowed?
Loads reorder after loads	Yes
Loads reorder after stores	Yes
Stores reorder after stores	Yes
Loads reorder after stores	Yes
Atomic reordered with loads	Yes
Atomic reordered with stores	Yes

Normal memory access before or after normal atomic access could be reordered, which breaks critical section rules that synchronization using atomic access requires. The Arm architecture defines barrier instructions to force memory access ordering. In reality, atomic instructions are used in pair with barrier instructions. Furthermore, there are some instructions which combine atomic and barrier functionality in one single instruction.

2.3 Arm data access barrier instructions

The Arm architecture includes barrier instructions to force access order and access completion at a specific point.

DMB – Data Memory Barrier
DSB – Data Synchronization Barrier

DMB

Explicit memory accesses before the DMB are observed before any explicit access after the DMB.

- Does not guarantee when the operations happen, just guarantees the order.

```
LDR  X0, [X1]          ;Must be seen by memory system before STR
DMB  SY
ADD  X2, #1            ; May be executed before or after memory system sees LDR
STR  X3, [X4]          ;Must be seen by memory system after LDR
```

DSB

A DSB is more restrictive than a DMB.

- Use a DSB when necessary, but do not overuse them.

No instruction after a DSB executes until:

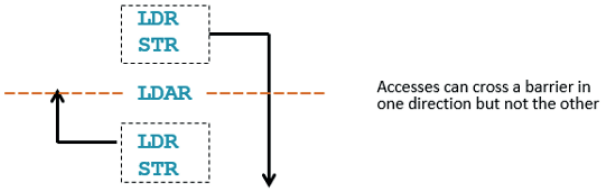
- All explicit memory accesses before the DSB in program order have completed.
- Any outstanding cache/TLB/branch predictor operations complete.

```
DC  ISW              ; Operation must have completed before DSB can complete
STR  X0, [X1]        ; Access must have completed before DSB can complete
DSB  SY
ADD  X2, X2, #3      ; Cannot be executed until DSB completes
```

DMB and DSB are two-way barriers, which force the ordering of memory accesses before and after the barrier instruction. It might be too strong, considering the performance impact of using DMB and DSB. Armv8-A also introduced load-acquire and store-release mechanisms, which are used by LDAR and STLR instructions. These are one-way barriers, which only force ordering in one direction, so they can reduce performance impact in some cases.

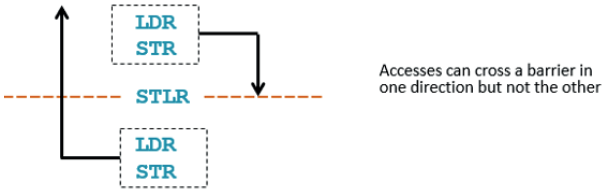
Load-Acquire (LDAR)

- All accesses after the LDAR are observable by the memory system after the LDAR executes.
- Accesses before the LDAR are not affected.

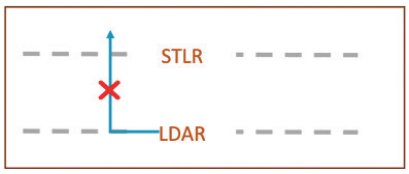


Store-Release (STLR)

- All accesses before the STLR are observable by the memory system before the STLR executes.
- Accesses after the STLR are not affected.



Armv8 LDAR and STLR implement RCsc (Release Consistency sequentially consistent https://en.wikipedia.org/wiki/Consistency_model#Release_consistency:_RCsc_and_RCpc). This means that a store-release followed by a load-acquire cannot be reordered with respect to each other.



Armv8.3-A introduced the LDAPR (Load-Acquire RCpc Register) instruction to implement RCpc (Release Consistency processor consistent, see above link). This means that a store-release followed by a load-acquire to a different address can be reordered with respect to each other.

Some atomic instructions combine atomic and load-acquire/store-release functionality. For example:

A64 Instruction	Description
LDAXR	Load acquire and exclusive
STLXR	Store release and exclusive
CASA	CAS atomic and acquire
CASL	CAS atomic and release
CASAL	CAS atomic and acquire-release

The barrier instructions are employed in the previous lock or unlock implementation examples:

1. For the first example, DMB is used in both lock and unlock routines to ensure memory ordering.
2. For the second example, LDUMAXA, which is with load-acquire attribute was used in lock; STLR, which is with store-release attribute was used in unlock.

3–Use Case Study

In this section, we look at several use cases on how new memory ordering instructions can help improve existing code. The intention of these examples is to provide more background, history, and details on the reason for the code changes, which can help inspire readers of this white paper to make similar changes in future.

3.1 Case1: OpenJDK

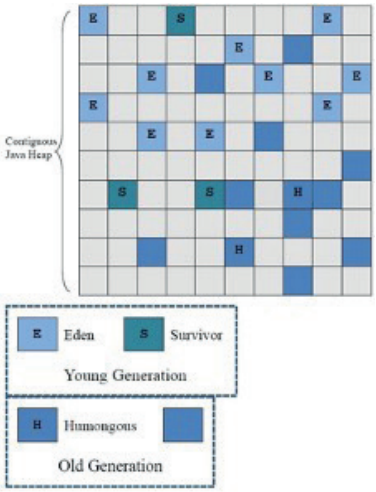
A G1GC bug is reported which triggers an OpenJDK crash during a Cassandra stress test. G1 garbage collector(G1GC) is a server-style garbage collector, targeted for multi-processor machines with large memories.

3.1.1 Description

AArch64 OpenJDK8u crashed at OtherRegionsTable::add_reference() because of BUS_ADRALN or SEGV_MAPERR during Cassandra load test.
The problematic frame listed as follows:

```
# A fatal error has been detected by the Java Runtime Environment:
# SIGBUS (0x7) at pc=0x0000ffff8da2e554, pid=35024, tid=0x0000ffff6a9ff1d0
Stack: [0x0000ffff6a800000,0x0000ffff6aa00000],  sp=0x0000ffff6a9fe5d0,  free space=2041k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V [libjvm.so+0x51f554]  OtherRegionsTable::add_reference(void*, int)+0xbc
V [libjvm.so+0x533ae4]  InstanceKlass::oop_oop_iterate_nv(oopDesc*,
FilterOutOfRegionClosure*)+0xe4
V [libjvm.so+0x5198e8]  HeapRegion::oops_on_card_seq_iterate_careful(MemRegion,
FilterOutOfRegionClosure*, bool, signed char*)+0x488
V [libjvm.so+0x4be9b4]  G1RemSet::refine_card(signed char*, unsigned int, bool) [clone
.part.73] [clone .constprop.114]+0x15c
V [libjvm.so+0x4a65a0]  RefineCardTableEntryClosure::do_card_ptr(signed char*, unsigned
int)+0x30
V [libjvm.so+0x462f08]  DirtyCardQueueSet::apply_closure_to_completed
buffer(CardTableEntryClosure*, unsigned int, int, bool)+0xe0
V [libjvm.so+0x3d6554]  ConcurrentG1RefineThread::run()+0x154
V [libjvm.so+0x7971bc]  java_start(Thread*)+0xf4
C [libpthread.so.0+0x6fb4]  start_thread+0xa4
```

3.1.2 Analysis



For G1GC, the heap is partitioned into a set of equal-sized heap regions (Eden-E, Survivor-S, Old-O, Humongous-H all represent different logical generation types), and each has a contiguous range of virtual memory.

The reference from Old Generation to Young Generation is maintained by the data structure remembered set (RemSet). This structure can be concurrently accessed by multiple GC threads.

The earlier buggy function (marked with red text) is operating on this structure and crashes.

Pseudocode for the bug-related code path:

```
1 OtherRegionsTable::add_reference(index)
2   prt_pointer =  PerRegionTable_pointer_array[index]
3   if prt_pointer is not null:
4         read prt_pointer reference PerRegionTable structure
5   else:
6         alloc PerRegionTable structure which address is prt_pointer
7         write PerRegionTable structure
8         PerRegionTable_pointer_array[index] = prt_pointer
```

This pseudo function is executed by different software threads to operate the PerRegionTable structure simultaneously. The bug code is related to line 7 and line 8. The content of the prt structure is available to the other thread once it is put into regions array, but in the Arm architecture, the other thread can see line 8 first before it sees line 7. For this scenario, one thread accesses prt structure's contents, which may still not be ready, then the crash or error happens. It may be easier to reproduce this condition on systems with high core counts.

Regarding the read operation, there is an address dependency between the line 2 read structure reference and the line 4 read structure data. The read order can be guaranteed.

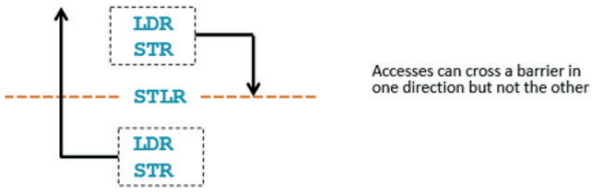
3.1.3 Solution

The fix is straightforward. Replace the direct assignment with store-release to enhance the store order. The concurrent thread does not use the item contents until it has been fully populated.

Updated code:

```
1 OtherRegionsTable::add_reference(index)
2   prt_pointer =  PerRegionTable_pointer_array[index]
3   if prt_pointer is not null:
4         read prt_pointer reference PerRegionTable structure
5   else:
6         alloc PerRegionTable structure which address is prt_pointer
7         write PerRegionTable structure
8         PerRegionTable_pointer_array[index] = prt_pointer
          // Need store release semantic here.
          OrderAccess::release_store(PerRegionTable_pointer_array [index], prt_pointer)
```

As the following figure shows, STLR prohibits all the code before line 8 to cross it. Therefore, the contents of prt are populated before it is visible to other thread.



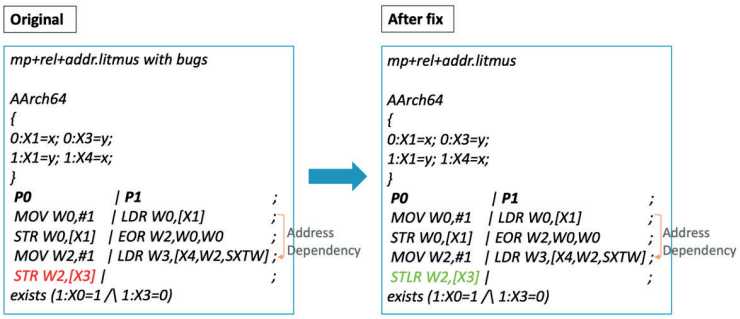
Result: after we apply this patch, the issue disappears, and has not been reproduced. This issue has been fixed after OpenJDK8u172 (include).

Patch link:
http://hg.openjdk.java.net/jdk8u/jdk8u-dev/hotspot/rev/4edb0f406a2c

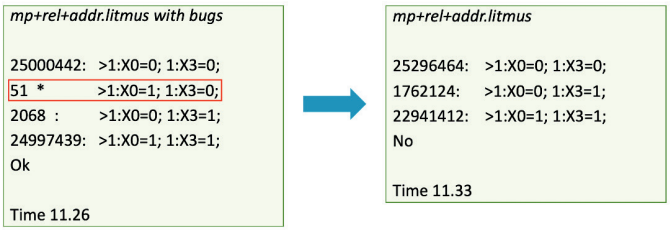
3.1.3.1 Corresponding litmus test

As depicted in Appendix A – Memory Model Tool section, litmus test is a good tool which has some typical use cases to help understand the memory model for different scenario on different architectures.

For the OpenJDK case mentioned above, we can use the typical use cases in litmus to clearly represent the updates made by the following case:



Litmus test results (executed 50000000 times on a certain Arm server) are shown below. The test results of original code contain some (X0=1; X3=0) cases (51 times in all the 50000000 test), which violate the essential program logic. No such (X0=1; X3=0) violation cases are found for the fixed code.



3.14 Thoughts

Memory barrier-related changes should be made very carefully. We must make sure the code has been well reviewed and heavily tested.

The correctness and performance of synchronization cases must be carefully balanced. We should first make sure the logic is correct, then go further to improve performance by eliding redundant barriers or using a lightweight barrier.

In this case, the DMB also works, but it is too heavy and may hurt performance. Achieving correct and high-performance synchronization requires a deep understanding of the Arm memory model and related instructions.

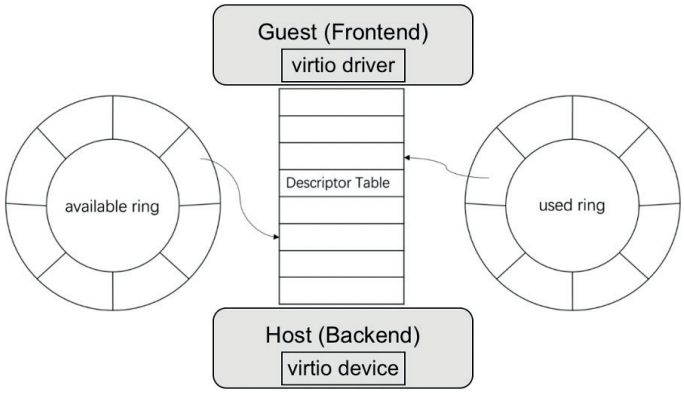
As most server software code is written and tested under x86 arch, ported memory instructions may fail on Arm-based platforms due to memory model differences. The litmus test suite (Appendix A – Memory Model Tool section) can help with understanding the memory model and verify the program on different architectures.

3.2 Case 2: DPDK

Optimized virtio split ring barriers with 20% performance uplift for PVP(PHY-VM-PHY) case on AArch64.

Virtio was developed as a standardized open interface for virtual machines (VMs) to access simplified devices such as block devices and network adapters. The virtio specification defines a bi-directional notifications mechanism:

- **Available Buffer Notification:** Used by the driver to signal there are buffers that are ready for processing by the device. Available idx indicates where the driver would put the next descriptor entry in the ring.
- **Used Buffer Notification:** Used by the device to signal that it has finished processing buffers. Used idx indicates where the device would put the next descriptor entry in the ring.



3.2.1 Description

Before the enhancement, the logics are as follows:

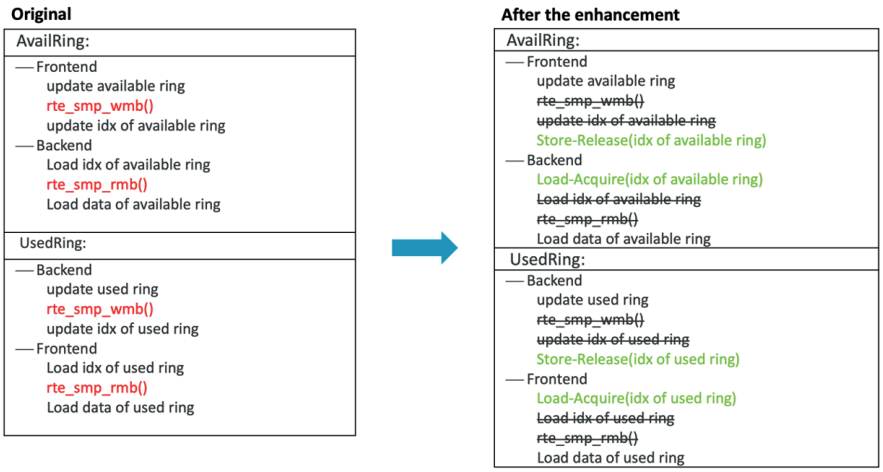
AvailRing:
- Frontend update available ring rte_smp_wmb() update idx of available ring - Backend Load idx of available ring rte_smp_rmb() Load data of available ring
UsedRing:
- Backend update used ring rte_smp_wmb() update idx of used ring - Frontend Load idx of used ring rte_smp_rmb() Load data of used ring

The Armv8-based CPU memory model is weakly ordered and its cores support out-of-order memory instructions. Using rte_smp_wmb() and rte_smp_rmb(), which are two-way barriers, can make sure the logic is correct.

3.2.2 Analysis

For the previous case, it is possible to enhancement performance further by replacing the existing two-way barriers with C11 one-way barriers for used index and available index in split ring.

3.2.3 Solution



Result: replacing the two-way barriers with C11 one-way barriers we can achieve about a 20% performance improvement.

Patch link: https://patches.dpdk.org/patch/69226/

3.2.3.1 Corresponding litmus test

For the DPDK case mentioned previously, there is also a typical litmus use case to clearly represent the updates made by this case as follows:

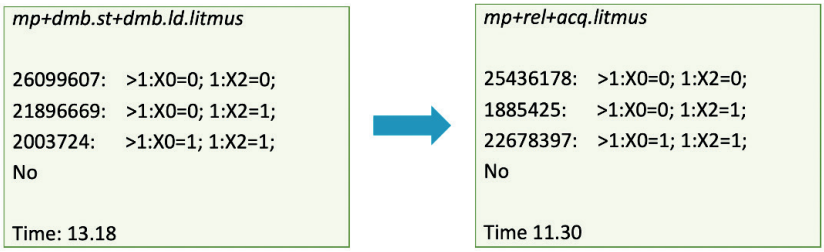
Original

```
mp+dmb.st+dmb.ld.litmus

AArch64
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0            | P1            ;
 MOV W0,#1     | LDR W0,[X1]   ;
 STR W0,[X1]   | DMB LD        ;
 DMB ST        | LDR W2,[X3]   ;
 MOV W2,#1     |               ;
 STR W2,[X3]   |               ;
exists (1:X0=1 /\ 1:X2=0)
```

After enhancement

```
mp+rel+acq.litmus

AArch64
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0            | P1            ;
 MOV W0,#1     | LDAR W0,[X1]  ;
 STR W0,[X1]   | LDR W2,[X3]   ;
 MOV W2,#1     |               ;
 STLR W2,[X3]  |               ;
exists (1:X0=1 /\ 1:X2=0)
```

Litmus test results (executed 50000000 times on an Arm server) as shown below. The results of both tests do not contain (X0=1; X2=0) as expected, and the enhanced version has advantage on performance aspect (11.30 vs 13.18 seconds runtime).



3.2.4 Thoughts

Two-way barriers can guarantee logical correctness, but they can be too heavy with respect to memory performance. We can use lightweight barriers (for example: store-release, load-acquire) to yield better performance.

3.3 Case3: MySQL

Use CAS to improve MySQL performance in high contention workloads.

3.3.1 Description

MySQL currently uses TAS (test and set) to implement a spinlock and its derived mutex. This implementation slows down the performance of high core-count systems with high lock contention.

3.3.2 Analysis

This enhancement uses compare and swap (CAS) to implement a spinlock for AArch64. As the following table shows, TAS uses the GCC built-in function __atomic_exchange which is compiled as (ldaxr+stlxr+cbnz). And CAS uses __atomic_compare_exchange which is compiled as (ldaxr+cmp+bne+stlxr). TAS is not efficient on AArch64 compared to CAS, especially in high core-count systems.

Using TAS means that for each contending thread it must write the lock first. This action introduces more cache line snoops when using the MESI or MOESI cache coherency protocol, which extends critical sections and increases memory access traffic.

3.3.3 Solution

<table>
<tr><td>Current implementation:
TAS (Test and set): ldaxr+stlxr+cbnz instruction.</td><td>Enhance implementation:
CAS (Compare and swap): ldaxr+cmp+bne+stlxr instruction.</td></tr>
<tr><td>

```
bool tas_lock()
{
    return(TAS(&m_lock_word, MUTEX_STATE_LOCKED)
        == MUTEX_STATE_UNLOCKED);
}

int TAS(volatile int* ptr, int new_val)
{
  int ret;
  __atomic_exchange(ptr, &new_val,  &ret,
__ATOMIC_SEQ_CST);
  return(ret);
}
```

</td><td>

```
bool cas_lock()
{
  return (CAS(&m_lock_word, MUTEX_STATE_UNLOCKED,
MUTEX_STATE_LOCKED) == MUTEX_STATE_UNLOCKED);
}

int CAS(volatile int* ptr, int old_val, int new_val)
{

  __atomic_compare_exchange(ptr, &old_val, &new_val, false,
          __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST);
  return(old_val);
}
```

</td></tr>
<tr><td>

```
Dump of assembler code for function TAS:
  0x00000000004005dc <+0>: sub sp, sp, #0x20
  0x00000000004005e0 <+4>: str x0, [sp,#8]
  0x00000000004005e4 <+8>: str w1, [sp,#4]
  0x00000000004005e8 <+12>:   ldr w0, [sp,#4]
  0x00000000004005ec <+16>:   mov w2, w0
  0x00000000004005f0 <+20>:   ldr x1, [sp,#8]
  0x00000000004005f4 <+24>:   ldaxr  w0, [x1]
  0x00000000004005f8 <+28>:   stlxr  w3, w2, [x1]
  0x00000000004005fc <+32>:   cbnz   w3, 0x4005f4 <TAS+24>
  0x0000000000400600 <+36>:   str w0, [sp,#28]
  0x0000000000400604 <+40>:   ldr w0, [sp,#28]
  0x0000000000400608 <+44>:   add sp, sp, #0x20
  0x000000000040060c <+48>:   ret
```

</td><td>

```
Dump of assembler code for function CAS:
  0x0000000000400640 <+0>: sub sp, sp, #0x10
  0x0000000000400644 <+4>: str x0, [sp,#8]
  0x0000000000400648 <+8>: str w1, [sp,#4]
  0x000000000040064c <+12>:   str w2, [sp]
  0x0000000000400650 <+16>:   ldr w0, [sp]
  0x0000000000400654 <+20>:   mov w4, w0
  0x0000000000400658 <+24>:   ldr x2, [sp,#8]
  0x000000000040065c <+28>:   add x0, sp, #0x4
  0x0000000000400660 <+32>:   ldr w3, [x0]
  0x0000000000400664 <+36>:   ldaxr  w1, [x2]
  0x0000000000400668 <+40>:   cmp w1, w3
  0x000000000040066c <+44>:   b.ne   0x400678 <CAS+56>
  0x0000000000400670 <+48>:   stlxr  w5, w4, [x2]
  0x0000000000400674 <+52>:   cbnz   w5, 0x400664 <CAS+36>
  0x0000000000400678 <+56>:   cset   w2, eq
  0x000000000040067c <+60>:   cmp w2, wzr
  0x0000000000400680 <+64>:   b.ne   0x400688 <CAS+72>
  0x0000000000400684 <+68>:   str w1, [x0]
  0x0000000000400688 <+72>:   ldr w0, [sp,#4]
  0x000000000040068c <+76>:   add sp, sp, #0x10
  0x0000000000400690 <+80>:   ret
```

</td></tr>
</table>

Patch link:
https://bugs.mysql.com/bug.php?id=88399

Result: Sysbench OLTP showed a ~15% performance improvement.

Note: the result is verified on ARMv8.0 servers. For different micro-arch implementation of newer ARMv8 version, it is better to run benchmarks to compare the result.

3.3.4 Final Thoughts

There are intricate hardware memory model differences between different computer architectures. For most cases, it is better for the programmer to stick to programming language memory models they know (such as Appendix B C++ Memory Model). And rely on the complier to produce high-efficient and error-free instructions, other than writing assembly by themselves.

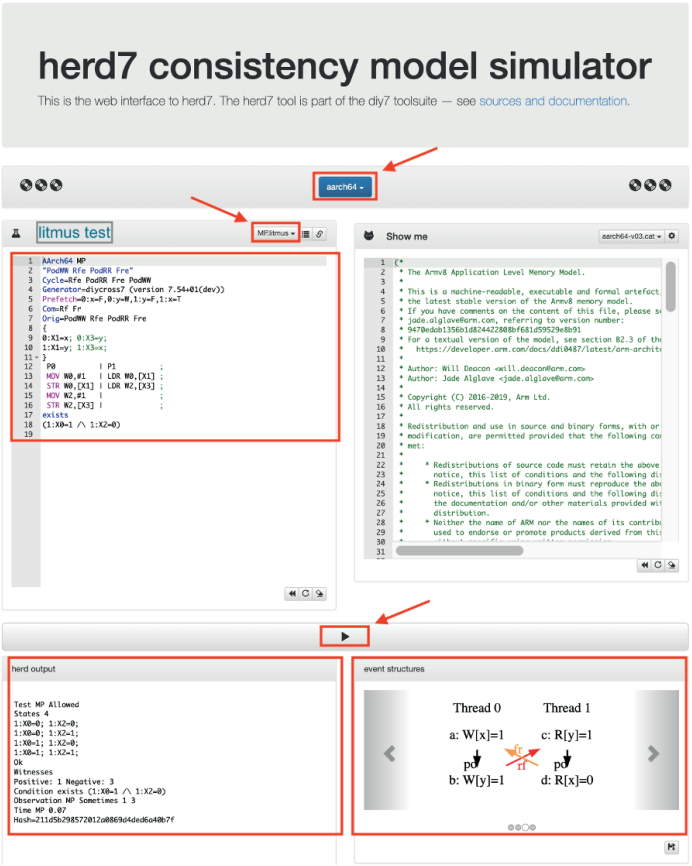
Appendix A - Memory Model Tool

Memory Model Tool (a DIY software suite) is a set of tools for testing memory consistency models. It helps develop an intuitive understanding of how it works and contains the following tools:

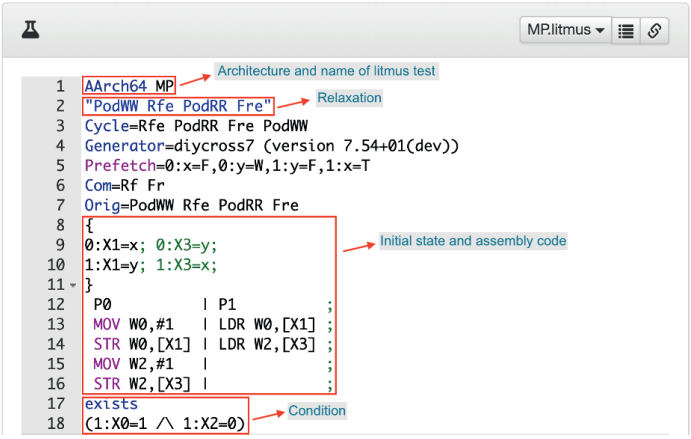
- **The diy7 tool**, which generates litmus test based on different parameters. It takes a list of candidate relaxations as input and generates the litmus test.
- **The litmus7 tool**, which runs litmus test on specific hardware. It takes the litmus test as input and prints the observed execution states.
- **The herd7 tool**, which designs and simulates memory models. The herd7 distribution already includes some models which we can learn from.

For detailed information, please refer to Ref [3] [4] [5].

We can take mp.litmus (Message Passing litmus test) in herd7 distribution as an example to learn the usage of Memory Model Tool. First, we access herd7 web interface from (https://diy.inria.fr/www/). Select “AArch64” and “MP.litmus” as below screenshot and click on the play button. We, then receive the herd output and event structures.

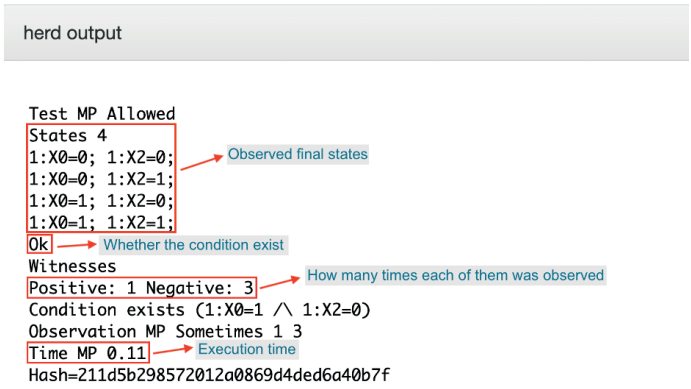


The litmus test is the core component of the memory model tool. The diy7 tool can generate the litmus test and the litmus7 tool can execute the litmus test. For the "litmus test" frame, please see the explanations as below graph:

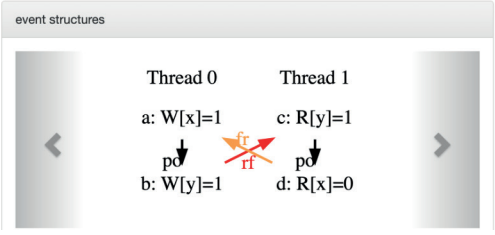


In the mp.litmus test, the "PodWW Rfe PodRR Fre" is the examined sequence of candidate relaxation (memory operation relation). Then, the assembly code is generated corresponding to the specific candidate relaxations. The last "exists" statement is the final state to be checked whether existing.

For the herd output, please see the explanations in the graph below:



In the "event structures" frame, there are different figures corresponding to different execution types, which includes the memory event and their relation arrows. Take the following figure as an example to explain the basic memory event orders:



- Po: Program order, the order to happen in the same thread.
- Rf: Read-from, the read operation reads the value which was written to the same location by the write operation. We can consider the write happens before the read.
- Fr: From-read, the write operation writes the value after the read operation reads the value from the same location. The read happens before the write.

Besides the memory events orders, we can add the thread (inter or intra) and location (same or distinct) information to get the candidate relaxation. This candidate relaxation can be used as input of diy7 tool.

For the "PodWW Rfe PodRR Fre" line in the mp.litmus, each word separated by space is a candidate relaxation. The brief introduction is as follows:

- PodWW: two write operations to distinct locations in program order.
- PodRR: two read operations to distinct locations in program order.
- Rfe: a read operation that reads a preceding write operation written value from the same location on distinct processors.
- Fre: a write operation that writes a value to the same location after a read operation reads the value on distinct processors.

For more details on the candidate relaxations, please see the diy7 tool documentation<link: http://diy.inria.fr/doc/gen.html>.

We can use the diy7 tool to generate litmus test based on relaxation. For example, this command generates the message passing litmus test:

~/bin/diyone7 -arch AArch64 "PodWW Rfe PodRR Fre"

The litmus7 tool can be used to execute the litmus test. For example, to run the litmus test in the file mp.litmus, we execute the command:

~/bin/litmus7 mp.litmus

We can also use litmus7 to generate source code using the following command, which can be executed on different computers.

~/bin/litmus7 -o mp.tar mp.litmus

For more detailed explanation, refer to the litmus7 tutorial http://diy.inria.fr/doc/litmus.html.

Appendix B - C++ Memory model

With the advent of programming language (C/C++ or other programming language) memory models, in most cases, developers can avoid writing architecture-dependent assembly code. This means developers can write high-quality code in C/C++ or other programming languages without thinking too much about the architecture differences.

The following are C++ Memory model constants defined in header <atomic>.
https://en.cppreference.com/w/cpp/header/atomic

Value	Explanation
memory_order_relaxed	Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only the atomicity of this operation is guaranteed.
memory_order_consume	A load operation with this memory order performs a consume operation on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only.
memory_order_acquire	A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.
memory_order_release	A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.
memory_order_acq_rel	A read-modify-write operation with this memory order is both an acquire operation and a release operation. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.
memory_order_seq_cst	A load operation with this memory order performs an acquire operation. A store performs a release operation. And a read-modify-write operation performs both an acquire operation and a release operation. In addition, a single total order exists in which all threads observe all modifications in the same order.

The following table shows the mapping between C++ memory model and Armv8-A implementation.

Memory type	Operation	Example	Armv8-A implementation
Memory_order_relaxed	Load Store RMW	a.load(std::memory_order_relaxed);	Atomic instruction without barrier
		a.store(1, std::memory_order_relaxed);	
		a.fetch_add(1,std::memory_order_relexed);	
Memory_order_acquire	Load RMW	a.load(std::memory_order_acquire);	LDAR/LDAPR
		a.fetch_add(1,std::memory_order_acquire);	LDADDA
Memory_order_release	Store RMW	a.store(1, std::memory_order_release);	STLR
		a.fetch_add(1, std::memory_order_release);	LDADDL
Memory_order_acq_rel	RMW	a.fetch_add(1,std::memory_order_acq_rel);	LDADDAL
Memory_order_seq_cst	Load Store RMW	a.fetch_add(1, std::memory_order_seq_cst);	LDADDAL
		a.store(1, std::memory_order_seq_cst);	STLR
		a.load(std::memory_order_seq_cst);	LDAR

Acknowledgments

We would like to thank Zenon Xiu, Steven Miao, Yibo Cai, Zheng Xu, Ningsheng Jian, and Joyce Kong for their guidance and review of this paper.

References

1. Arm®, "Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile Documentation" https://developer.arm.com/docs/ddi0487/latest
2. "The software suite diy7" http://diy.inria.fr/
3. "A working example of how to use the herd7 Memory Model Tool" https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/how-to-use-the-memory-model-tool
4. "How to generate litmus tests automatically with the diy7 tool" https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/generate-litmus-tests-automatically-diy7-tool
5. "Running litmus tests on hardware using litmus7" https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/running-litmus-tests-on-hardware-litmus7