Atomic Instruction Translation towards a Multi-threaded QEMU

Alvise Rigo
Virtual Open Systems
Grenoble - France
a.rigo@virtualopensystems.com

Alexander Spyridakis
Virtual Open Systems
Grenoble - France
a.spyridakis@virtualopensystems.com

Daniel Raho
Virtual Open Systems
Grenoble - France
s.raho@virtualopensystems.com

KEYWORDS

TCG/QEMU; atomic instructions; system emulation; instructions emulation; parallel emulator; multi-threading

ABSTRACT

In the context of system emulation, the sophistication of the emulator usually grows with the complexity of the target system model. Particularly, emulating precisely a certain CPU architecture can introduce many challenges that have to be properly explored and somehow solved to reach an accurate emulation of the target system.

In this paper we present an implementation design of ARM atomic instructions for a multi-threaded version of QEMU (the *Quick EMUlator*), currently under development [1].

To prove the correctness and performance of such an implementation, some tests have been performed showcasing a high degree of accuracy and fidelity of the emulated instructions. While this paper does not cover all possible guest architectures that QEMU supports, the described new approach results in a reliable infrastructure that eventually can address all target architectures in QEMU.

I. INTRODUCTION

Multi-threading architectures have brought new challenges in the context of synchronization between parallel execution flows, requiring for specific multi-thread aware instructions.

Some of these instructions are called atomic, in that their execution is indivisible and uninterruptible. This means that any modifications to memory are always consistent, no matter how many CPUs can concurrently access it.

The main reason for such instructions is to simplify the code required to synchronize these flows, and of course to make it faster. Nowadays, almost every architecture has its own set of atomic instructions, which is often used to implement low-level synchronization routines.

These instruction sets, introduced complications for all system emulators, such as QEMU, that need to preserve the atomic nature of the instructions while emulating them. *TCG* (Tiny Code Generator), a software component parsing and translating the guest instructions to host instructions, has been originally implemented with a single-threaded design in mind, although still capable of exposing multiple cores to the guest. In this simplified context, QEMU emulates multiple guest CPUs by executing them in a round robin fashion, making the guest actually slower then the uniprocessor variant.

Moving to a real multi-threading design (multiple guest CPUs executed concurrently in different threads) is a significant improvement for QEMU, allowing various emulation use cases in today's many-core systems. This design change comes with several challenges and among the most important ones is the proper and accurate translation of atomic instructions. With multi-threading in mind, atomicity is not granted by default and must be implemented carefully instead.

Challenges to be addressed

Emulation in itself brings two considerable challenges: accurate emulation for both the processor architecture, and the target machine model, including any attached devices/peripherals. The former challenge is of crucial importance, since in the context of QEMU it requires to:

- fetch the opcode instructions from guest memory
- decode and translate instructions to an intermediate code representation (*IR*), which is handled by the TCG frontend
- express the IR code in host machine instructions, which is done in the TCG backend.

The technique listed in the steps above is also called *Dynamic Translation* [Bellard 2005] and allows to create extremely portable full system emulators. For instance, adding support for a new guest/host architecture requires the implementation of a new TCG frontend and backend respectively, making the overall emulator design quite flexible and modular.

Every modern instruction set includes a number of atomic instructions which are extremely useful when implementing synchronization functions in a shared memory system. This kind of instructions are used to implement lock-free algorithms or, in general, programs where accesses to the shared data don't necessarily require a lock.

Examples of such instructions are the *Compare and Swap* (CAS) instructions (like the x86 `LOCK CMPXCHG` [2]) or the *LoadLink/StoreConditional (LL/SC)*, introduced by Jensen, Hagensen, and Broughton [Jensen et al. 1988] as part of the *S-1 AAP* project.

Proceedings 30th European Conference on Modelling and Simulation ©ECMS Thorsten Claus, Frank Herrmann, Michael Manitz, Oliver Rose (Editors)
ISBN: 978-0-9932440-2-5 / ISBN: 978-0-9932440-3-2 (CD)

The idea of the LL/SC instructions was to perform read-modify-write operations without requiring any bus to be locked, or in general, any CPU to be temporarily halted. This approach is superior compared to CAS instructions, because it solves the so called *ABA problem* when implementing non-blocking algorithms.

In a uniprocessor system all benefits brought by atomic instructions are irrelevant, as the implementation overhead of coherency for concurrent tasks is not needed. This is why QEMU was not concerned with such complications thus far, assuming that all guest instructions can be considered atomic by default. However, modifying QEMU towards real multi-threading (capable of emulating more than one CPUs at once) requires to revisit this simplified design, in favour of a more complex one, described in Section V of this paper.

II. ATOMIC INSTRUCTIONS

This section describes the semantics of the LL and SC instructions. Excluding minor differences, these instructions are considered semantically equal for all architectures.

LoadLink

This instruction reads the value from a shared memory location and stores the content into a register of the calling CPU. It also establishes a link and records the CPU with the accessed address (`xaddr`), to properly handle the subsequent SC operation. The LL marks the beginning of a *tentative exclusive memory region* [Jensen et al. 1988], that will be either confirmed or dismissed by the following SC instruction; each CPU defines its own EMR, only one at once is allowed. Depending on the architecture implementing the instruction, the size of the exclusive memory can be bigger than the size of the memory access (i.e. the size of the data read and then stored to a system register by the LL operation). For instance ARMv7 defines as `IMPLEMENTATION DEFINED` the *Exclusive Reservation Granule*, which is the size of the memory that will be monitored whenever a CPU issues a LL instruction.

StoreConditional

This instruction writes to the address `xaddr` only if it belongs to an exclusive memory region previously created by an LL. The SC is not always successful since another CPU can nullify the exclusive memory region by writing or reading to it. In general, the SC fails if a certain condition comes true. In its original definition, this condition has been defined specifically for the implementation of a particular processor architecture [Broughton et al. 1982], however, all recent architectures adopting these instructions implement a slightly different variant.

Invalidation of an exclusive memory region

Setting aside the actual implementation, this condition has to guarantee that the exclusive region initiated by the LL has not been invalidated by any other CPU in the system. This includes any CPU capable of reading/writing to system memory, which can result in violating the initial assumption of an *exclusive* memory region.

From now on, the following notation will be used:

- $w_{P_i}(y, val)$ write of value *val* to address *y* made by process *i*
- $w_x(y, z, val)$ write access of size *z* bytes to address *y* made by processor *x*, the value written is *val*. When the value written is not relevant, the notation $w_x(y, z)$ will be used
- $r_x(y, z)$ read access of size *z* bytes made by processor *x* to address *y*. In some generic cases, for a read access to address *x*, the notation $load(x)$ will be used
- $ll_x(y, z)$ LoadLink instruction issued by processor *x* to address *y*, resulting in a read access of *z* bytes. In some generic cases, for a LoadLink to address *y*, the notation $loadLink(y)$ will be used
- $sc_x(y, z)$ StoreConditional instruction issued by processor *x* to address *y*, resulting in a write access of *z* bytes no matter what is the value written. In some generic cases, for a StoreConditional access to address *y* that writes the value *val*, the notation $storeCond(y, val)$ will be used
- $EMR_{x,y}$ exclusive memory region created by the system after CPU *x* performed $ll_x(y, z)$. $EMR_{x,y}$ persists until $sc_x(y, z)$ is performed.

III. THE ABA PROBLEM

The ABA problem usually occurs when a CAS-based non-blocking algorithm gives a false positive result [Dechev et al. 2010]. Listings 1 and 2 give an example of the same algorithm implemented using the two types of atomic instructions (CAS instruction and LL/SC instructions). The algorithm, called *updateValue*, updates the current value of a memory location at address *addr*. The algorithm makes use of the CPU's instructions CAS, LL and SC. In case of failure, SC returns 1.

Listing 1: `updateValue()` non-blocking algorithm with possible occurrence of the ABA problem

```
input: int addr
output: none
begin:
   do
      old ← load(addr)
      new ← compute_new_val()
   while CAS(addr, old, new) ≠ old
end
```

If a process P_1 running $updateValue(addr)$ is interrupted after loading the value pointed by *addr* and before the execution of the CAS instruction, then the ABA problem will occur if a process P_2, with $P_2 \neq P_1$, changes the value pointed by *addr* to old_1, with $old_1 \neq old$, and eventually restore the value *old* (more concisely: $w_{P_2}(addr, old_1)$, $w_{P_2}(addr, old)$). In this scenario, the CAS instruction would success, without P_1 knowing that the value pointed by *addr* changed two times.

Listing 2: `updateValue()` non-blocking algorithm ABA problem resistent

```
input: int addr
output: none
begin:
    do
        old ← LL(addr)
        new ← compute_new_val()
    while SC(addr, new) = 1
end
```

On the contrary, if the implementation provided by Listing 2 was used to handle the same scenario, there would not be any false positive since the SC would fail due to the first write $w_{P_2}(addr, old_1)$ made by process P_2.

Suppose now that the code of Listing 2 is executed by a guest OS, and that TCG is used to translate the load-Link (LL) and storeCond (SC) instructions. Depending on the architecture of the host, TCG can rely on the same instructions of the host to directly map the guest instructions to host instructions. In ARM for instance, LL would be mapped to LDREX while SC to STREX.

However, for architectures such as x86 where similar instructions are not present, emulating correctly the LL/SC semantic is not obvious, and requires some additional effort. In other words, the emulation has to resolve the ABA problem using only CAS-like instructions. The simplistic emulation proposed by Listing 3 and 4 still suffers of the ABA problem, since in between the LL and SC, the value can be changed and then restored to *GLOBAL_old*.

Listing 3: `LL` emulation with CAS instruction

```
input: int *addr
output: int
begin:
    GLOBAL_old ← load(addr)
    return GLOBAL_old
end
```

Listing 4: `SC` emulation with CAS instruction

```
input: int *addr, int new
output: int
begin:
    written ← CAS(addr, GLOBAL_old, new)
    if new = written
        return 0
    else
        return 1
    end
end
```

In previous work [Dechev et al. 2010], it is proved already that it is possible to replicate correctly the LL/SC semantic without incurring the ABA problem, by the usage of additional variables which the non-blocking algorithm has to be aware of. For example [Gifford and Spector 1987], as representative of a common solution, makes use of version tags that pair the actual data to be accessed: every access made by the CAS instruction would modify also the tag, guaranteeing **uniqueness** to every access. It is worth noting that the problem is solved only at an algorithm level, since spurious writes could still change only the value of the actual data, leaving the tag unaltered.

Given the nature of emulation, we can not make the guest aware of such an additional tag, and the extra care of emulating properly the LL/SC semantic has to be left entirely on the host. This makes the adoption of CAS-like instructions hardly feasible, especially if corner cases have to be handled. One such example is when 128bit wide LL/SC guest instructions have to be emulated relying on 32bit or 64bit cmpxchg host instructions, which would be the case for the emulation of ARMv8 LDXP/STXP instructions on x86_64. For reasons explained above, the emulation of atomic instructions can not be implemented in a straightforward way, imposing several challenges in which a major one is an actual solution to the ABA problem.

IV. QEMU INTERNALS

Before diving into details, some QEMU concepts are presented for completeness in the following paragraphs.

Instruction translation

In QEMU, the process of translating guest instructions to host instructions, is covered by TCG, which first translates guest instructions to an intermediate representation (defined by TCG instructions), and finally to a host representation. An example of this process is depicted in Figure 1, where TCG is translating ARM instructions to native (host) x86 instructions.

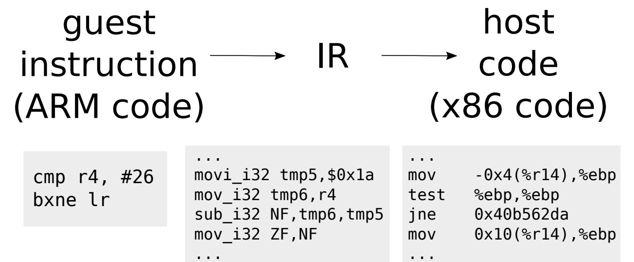


Fig. 1: Example of code translation for a TB

Host code is always contained in *Translation Blocks* (TBs), that although part of QEMU's address space, they are not normal functions but rather auto generated code of the QEMU process itself. Every TB contains a basic block of the guest code translated for the host machine. This generated code is not always enough to emulate all guest features (devices or instructions affecting the machine model state for instance); for this reason TCG allows to jump out of a TB to execute additional emulation functions. Figure 2 illustrates the TB structure, where the *prologue* and *epilogue* are the entry point and the exit point of the TB respectively.

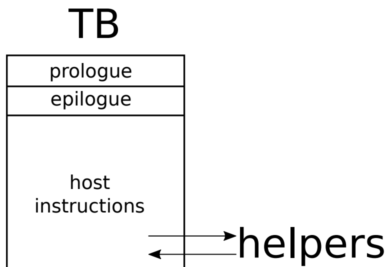


Fig. 2: Scheme of a TB

QEMU software MMU

QEMU offers two emulation modes, user and system. The first mode allows to run Linux user space applications and is not the topic of this paper. System mode instead is the most commonly used, but also the part of QEMU that receives most of the community effort. This mode of execution allows for full system emulation as the name suggests, relying on a software MMU (*softmmu*) that is able to solve the disparity between guest and host addresses. More specifically, the softmmu translates guest RAM (virtual) addresses to host pointers. The softmmu, in turn, relies on a TLB (Translation Lookaside Buffer) to cache the newly translated addresses (in the form of TLB entries). TLB entries are used to prevent guest table walks for addresses that are frequently accessed.

QEMU uses different bitmaps to track the different properties of each guest memory page. For example, one bitmap is used to distinguish memory mapped IO (MMIO) pages of the guest. These bitmaps are usually monitored when a new TLB entry is created, since the whole TLB is accessed directly from TCG code each time the guest wants to do a load or store operation. This happens mostly because reading and writing to memory requires translating the guest address into a host compatible, and the TLB serves exactly this purpose. While executing TB code, TLB entries are constantly checked to perform the translation; if the TLB entry does not exist or is meant for a different page, the execution exits the TB in order to generate the missing entry.

In other cases, e.g. MMIO pages, the corresponding TLB entries will always force the execution to leave the TB to execute, for instance, the emulation code of some device.

According to QEMU terminology, we will call *slow-path* the execution that exits from the current TB, either due to a missing TLB entry or for a TLB entry that requires an exit (as per MMIO entries). In all the other cases, the execution will follow the so called *fast-path*.

V. TOWARDS MULTI-THREADING

Extending QEMU's infrastructure to properly translate atomic instructions in a real multi-threaded implementation, can result in two notable design options: One, where the translation of atomic instructions is fully in charge of the guest frontend, eventually using helpers to implement additional steps that are not directly covered by TCG-generated code. This option would result in modifying all QEMU supported guest architectures in a consistent way, thus making the transition to multi-threading even more demanding. The other solution, proposed in this paper, aims at providing a unified code infrastructure for all guest frontends, providing an accurate implementation that abides more closely to the semantics dictated by the architecture specification.

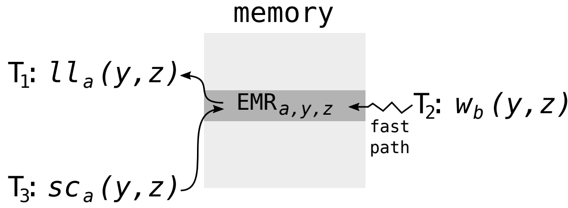
LL/SC helpers

Two helper functions consist the core of atomic instruction translation. *LoadLink* and *StoreConditional*, which are functionally equal to the analogous instructions described in Section II. The two helpers have been designed to have an one to one mapping to the corresponding atomic instructions present in architectures such as ARM, that adopted the LL/SC paradigm. In other cases, for example x86 `LOCK CMPXCHG`, the instruction can be easily achieved by means of LL/SC instructions as presented in [Maged 2004] and [Anderson and Moir 1995]. Even if the work presented in this paper is focused on ARM `ldrex` and `strex`, it nevertheless provides the means to translate atomic instructions for all other architectures that are supported by QEMU.

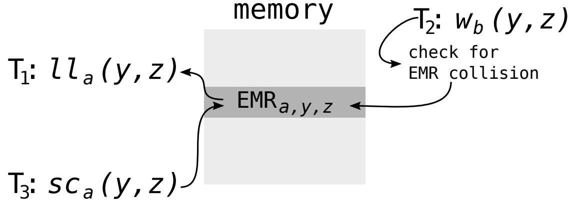
These two helpers rely heavily on QEMU's softmmu. Since being extremely deep-rooted in QEMU's MMU emulation layer, it is the perfect tool to address all the features that the atomic instructions require. From now on, we will refer to the notation that has been introduced in Subsection II, where $ll_x(y, z)$ and $sc_x(y, z)$ map respectively to `ldrex` and `strex`, $w_x(y, z)$ and $r_x(y, z)$ map to normal load and store instructions (namely `ldr` and `str`).

The translation of $ll_x(y, z)$ has to define the correlated $EMR_{x,y}$, in such a way that the link established by the first instruction is honoured by all CPUs in the system. In practice, the system has to react properly to all write accesses made to $EMR_{x,y}$ invalidating the link. Figure 3a depicts what would happen if all the fast-path accesses are not taken into account.

To protect against this, a new bitmap, called `exclusive`, has been added to QEMU's softmmu, which flags all the pages containing an active EMR. Whenever QEMU generates a TLB entry for a guest page, the corresponding exclusive bit in the bitmap is checked. A set bit on a page will make all the TLB entries created for that page to be such that, if evaluated by the guest code for an address translation (something that occurs for every $r_x(y, z)$ and $w_x(y, z)$), they will always force the execution to exit from the TB. This will be used as a hook to trap from a TB to a QEMU function that evaluates if the access conflicts an existing EMR. If such a conflict occurred, the subsequent SC instruction will fail. In any case, the normal write access that triggered the failure will always succeed. Accesses to pages containing EMRs are depicted in Figure 3a.



(a) Fast-path access colliding an existing EMR



(b) Access to EMR properly trapped to eventually invalidate other CPU SC operations

Fig. 3: In both (a) and (b) $T_1 < T_2 < T_3$ is applicable. In (a) guest `CPU 2` is writing to the EMR through the fast-path, the following $sc_a(y, z)$ will anyhow succeed. In (b) access $w_b(y, z)$ makes `CPU 2` evaluate a TLB entry that will require to leave the fast-path

The definition of the EMR is done by storing its range inside the guest CPU state variable: this is perfectly in line with the impossibility to have nested LL/SC operations like $ll_x(y_1, z), ll_x(y_2, z), sc_x(y_1, z), sc_x(y_2, z)$.

VI. Implementation details

In this section all complications regarding multi-threading execution will be described, together with the design choices of the proposed implementation. One major problem when dealing with multi-threaded programs is the occurrence of race conditions. In the context of this work, a race condition can be associated to an inconsistency of the whole machine state, which is in charge of translating atomic instructions. The direct negative result of such a state is the failure of a SC operation that should have succeeded, or even worse, the success of a SC operation that had to fail. In the following sections, all critical points that result in race conditions are explored, where the implemented approach is also documented.

Updates of the exclusive bitmap can lead to inconsistencies due to the out-of-order execution of load/store operations as seen, for instance, on ARM architectures [3]. For this reason all accessors to such a bitmap are atomic, an outcome that is possible by means of host atomic instructions. It is important to note, that this can be possible only in the case where bitmap accessors are QEMU functions and not implemented through TCG generated code.

Setting a bit in the exclusive bitmap to enforce slow path execution for the TLB entries overlapping an EMR, can result in another problem. In fact, other guest CPUs, different from the one issuing the LL, could have already generated TLB entries for the same page, forcing the execution to follow the fast-path (as what happens in Figure 3a). To avoid this dangerous behaviour, TLB entries of these CPUs will be flushed, forcing them to recreate the TLB entry that covers the page in the EMR. This flush request will also prevent race conditions, that are related to the delayed new state propagation of the exclusive bit.

Lastly for this implementation, the evaluations and updates of the EMRs have been safeguarded using a mutex. This is mandatory because updating this structure is not possible with a single atomic instruction. Another related aspect that requires additional caution, relates to the actual memory accesses made by the LL and SC instructions. More specifically, the results on memory brought by these instructions has also to be done jointly with the update of the EMR values. The Listings 5, 6 and 7 represent respectively the LL, SC and normal store access. In these examples, the critical region is delimited by two calls `LOCK` and `UNLOCK`.

For instance, consider the Listing 5, which only works as long as the normal load is done inside the critical section, otherwise the loaded value can be potentially updated by another CPU, which might or might not be inside the critical region. For the same reason, the SC operation (Listing 6) has also to rely on the same critical region to be consistent with the rest of the atomic instruction emulation. Without entering the critical region, it can potentially declare the operation as successful (returning 0), but performing the store after another CPU modified the value. Similarly, the store operation (Listing 7) enters the critical region to check for a possible conflict in EMR, but also to perform the regular access.

Listing 5: `LoadLink` pseudo code, `load()` denotes a plain load from memory of size `z`

```
input: int y, int z, int x
output: int
begin:
    LOCK()
    CPU[x].EMR ← [y, y + z]
    ret ← load(y,z)
    UNLOCK()
    return ret
end
```

Listing 6: `StoreCond` pseudo code, `store()` denotes a plain store to memory of size `z`

```
input: int y, int z, int x,
      int val
output: int
begin:
    LOCK()
    if CPU[x].EMR = [y, y + z]
        store(y,z,val)
        ret ← 0
    else
```

```
        ret ← 1
    end
    UNLOCK()
    return ret
end
```

Listing 7: Plain write access trapped by the slow-path

```
input: int y, int z, int val
begin:
    LOCK()
    for each CPU
        if CPU.EMR overlaps [y, y + z]
            CPU.EMR ← NULL
        end
    store(y, z, val)
    UNLOCK()
end
```

VII. EXPERIMENTAL RESULTS

The implementation has been evaluated under two main points of view: correctness of the emulation and performance. Both of them are relevant aspects that have to be properly considered: from one side, the translation of atomic instructions needs to behave in accordance with the architecture instruction set specification. From the other side, the performance evaluation has also to be taken into consideration, as too much overhead would slow down the guest execution considerably. Atomic instructions in fact, are mostly used for synchronization routines, e.g. the Linux implementation of the spin lock functions, and are designed to keep as short as possible the lifespan of the EMR in order to minimize the number of conflicts.

[Wang et al. 2011] and [Ding et al. 2011] could have been two candidates for comparison against the presented work, however, only the upstream version of QEMU (whose source code is available at [4]) will be taken into account. In fact, QEMU has been significantly evolving over the last years and as such, results from these previous attempts would be considered deprecated and outdated. In addition, as it has been described in Chapter V, the presented implementation sets the objective of providing a common infrastructure for atomic instruction translation for all supported architectures, relying as much as possible on the current QEMU code base, as well as its features and components. While the proposed implementation results in a slightly greater overhead than a guest specific implementation, at the same time it is also more beneficial to the QEMU community by offering a unified implementation and a faster upstreaming process.

ABA problem occurrence test

With the purpose of verifying that the ABA problem could actually occur in the context of emulation (as explained in Section III), a specific test has been implemented. The test is used as a proof that LL/SC instructions can not be translated by relying only on a straight application of the host's CAS instruction. The C code of the test is presented in Listing 8, each thread is pinned to one ARMv7 guest CPU. The value of *ADDEND* has been chosen explicitly to make the identification of the ABA occurrence easy to verify on the host emulation side, not in the guest. In fact, in case the problem occurs, the STREXD instruction will fail as it should, *but it would not fail in the case* ADDEND *was 1* (in such a case, the occurrence would not be noticed by the guest). Listing 9 shows the naive implementation of the STREXD instruction used in the multi-threaded QEMU code; the method *lsb32* returns the 32 least significant bits of the argument.

Listing 8: ABA problem triggering test

```
/* shared counter */
uint64_t global_cnt

/* thread 1 */
for (i = 0; i < LOOP_SIZE; i++) {
      __sync_fetch_and_add(&global_cnt, 2);
}

/* thread 2 */
#define ADDEND ((1 << 32) | 1)
for (i = 0; i < LOOP_SIZE; i++) {
      __sync_fetch_and_add(&global_cnt,
                              ADDEND);
      __sync_fetch_and_sub(&global_cnt, 1);
}
```

Listing 9: ABA problem: STREXD emulation code

```
input: int *addr, int new
output: int
begin:
    written ← CAS(addr, GLOBAL_old, new)
    if new = written
        return 0
    else
        if lsb32(written) = lsb32(new)
            aba_errors++
        end
        return 1
    end
end
```

The cycle we need to identify is (here *ll* does not report the access size, the second argument of *sc* is the value written): $ll_1(addr) \rightarrow ll_2(addr) \rightarrow sc_2(addr, val_1 + ADDEND) \rightarrow ll_2(addr) \rightarrow sc_2(addr, val_2 - 1) \rightarrow sc_1(addr, val_1)$, where val_1 is the value retrieved by both $ll_1(addr)$ and the first $ll_2(addr)$, $val_2 = val_1 + ADDEND$ and $addr$ is the address of the global counter. More specifically, the first and last instructions implement the *__sync_fetch_and_add* of thread 1, the inner instructions instead implements *__sync_fetch_and_add* and *__sync_fetch_and_sub* of thread 2. When such a cycle happens, the 32 most significant bits are incremented by $(1 \ll 32)$, while the least significant half returns to the original value (e.g.: 0x00000001 00000001 $\rightarrow$ 0x00000010 00000010 $\rightarrow$ 0x00000010 00000001): when this particular sequence of values takes place, the code registers the event incrementing the ABA errors counter.

$LOOP_SIZE$	occurrence (%)
$10x10^6$	0.0008
$8x10^6$	0.0008
$6x10^6$	0.0007
$2x10^6$	0.0005
$1x10^6$	0.0004

Table 1: Occurrence ratio of ABA problem (referring the test code of Listing 8)

Table 1 reports the percentage of errors according to the value of `LOOP_SIZE`, verifying the actual risk derived from an improper translation of LL/SC instructions. It is worth to note that thread 2 could have used plain load and store accesses; this would have probably made the occurrence of the ABA flaw even more consistent, though greatly complicating the algorithm to detect its occurrence.

Benchmarking tests

A benchmark bare metal application has been developed to evaluate the correctness and performance of the proposed solution, which is compared against the current version of QEMU. The application is also a stress test, as it puts the whole machinery under heavy load, implementing a scenario characterized by significant contention of a shared resource. The source code has been kept at a bare minimum and designed without any operating system dependencies.

Listing 10: Code used to benchmark the emulation overhead, executed for every core of the guest.

```
for i ← 0 to LOOP_SIZE
   LOCK()

   if global_a = cpu_index % 2
      global_a ← 1
      global_b ← 0
   else
      global_a ← 0
      global_b ← 1
   end

   if global_a = global_b
      errors ← errors + 1
   end

   UNLOCK()
end
```

Listing 11: Simplified ARM assembly code of the `LOCK()` function. The register `r2` contains the address of the shared lock.

```
lock:
mov r1, #1
repeat:
ldrex r3, [r2]
strex r0, r1, [r2]
cmp r0, #0
bne repeat
cmp r3, #0
bne repeat
```

In Listing 10 the pseudo code of the stress test is reported, the variable `global_a` and `global_b` are shared among all the processors, while the functions `LOCK` and `UNLOCK` are used to define the critical region, providing means to hold/release a basic lock.

The purpose of the `if` statement is to swap the two values according to the index (identifier) of the CPU that has the lock. The aim of the test is to verify the correctness of the execution: in case the critical region was not respected, then the assignments to the global variables would overlap with the likely outcome of the two variables being set to the same value.

The implementation of `LOCK` can be found in Listing 11; as far as `UNLOCK` is concerned, its implementation is not reported as it resolves in a normal store to memory. Given the particular design of the test, the multi-threading feature of QEMU doesn't offer any advantage since the single iteration of the test case is spent almost entirely inside the critical region.

The results of the stress tests are reported in Figure 4a and 4b where `LOOP_SIZE` is respectively one million and ten million. The number of guest CPUs (reported in the x-axis) was always lower or equal to the number of host CPUs.

The host machine used for the tests is an Intel Core i7-4710MQ clocked at 2.5GHz while the machine model used in QEMU is ARM *virt*.

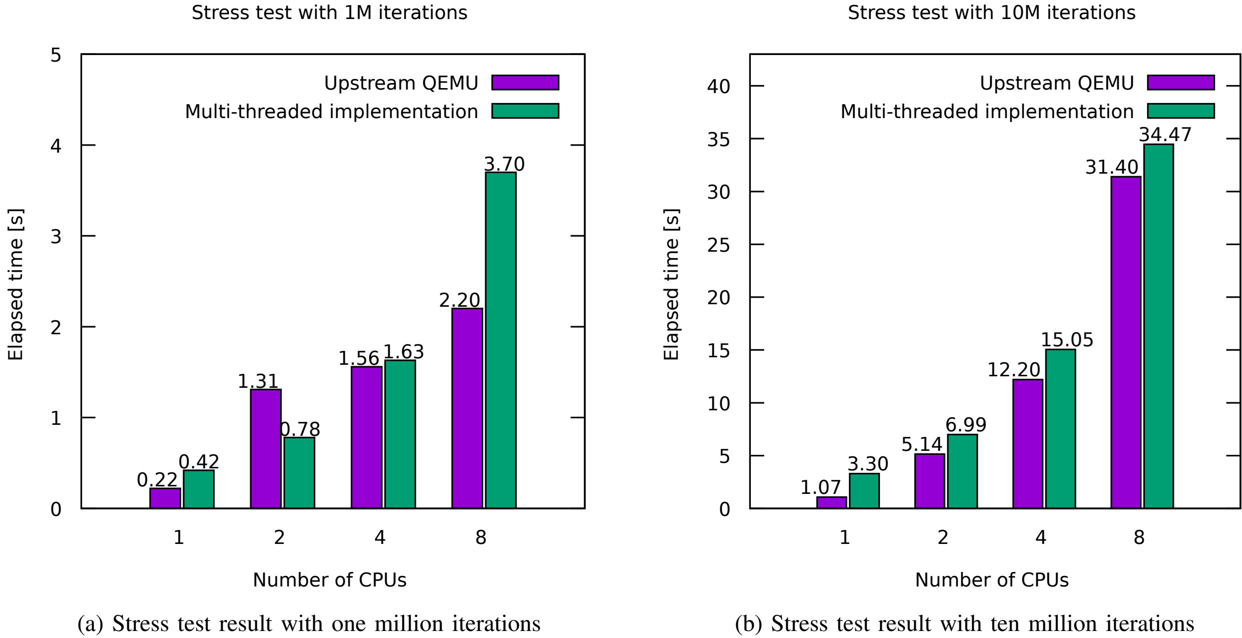
With the aim of testing the code in a real use case scenario, the boot time of the Linux kernel has also been measured. The results are reported in Figure 5.

Results analysis

By implementing a test designed to showcase the ABA problem, the risk of false positives in the field of atomic instruction emulation is confirmed. The same test could not have been possible with the implementation object of this work, since its overall design guarantees the eviction of the ABA problem from the start. However, the most complicated implementation of atomic instruction translation based on a multi-threaded QEMU is slightly slower than the upstream counterpart. In upstream QEMU there is no real CPU concurrency and all the iterations are serialized inherently by QEMU, through its round-robin scheduling of guest CPUs. Moreover, the emulation of *ldrex* and *strex* is done entirely inside a TB, which is a huge performance advantage since it does not require any branches to C code. As we can see in 4a, the relatively low number of iterations pronounces the overhead by the proposed implementation significantly. However, Figure 4b shows how the gap between the two implementations is lower with an increased number of iterations. With ten million iterations the overhead introduced by the new atomic translation is less than 9%. The stress test has also proved the correctness of the emulation, since the error counter never left the initial value of 0.

In the case of Linux kernel boot-up times, the overhead with one guest CPU increases the boot time by 13.5%. However, as soon as the benefits of multi-threaded

Fig. 4: Stress test result



(a) Stress test result with one million iterations

(b) Stress test result with ten million iterations

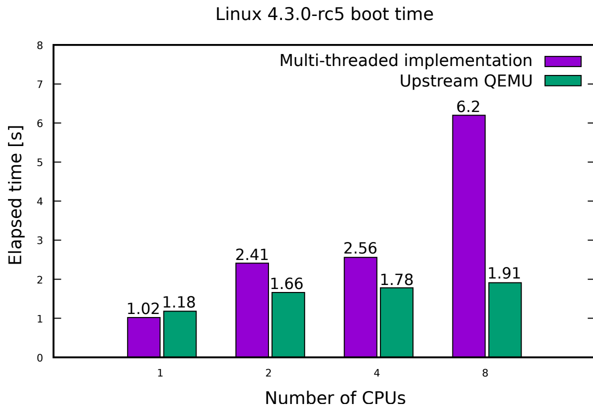


Fig. 5: Boot time of the Linux kernel

QEMU come into play, upstream QEMU is outperformed as we can see from Figure 5.

VIII. Related work

There have been already some attempts to make QEMU multi-threading in the past, all of them had to address the problem of atomic instruction translation. PQEMU [Ding et al. 2011] implemented the translation of ARM's LDREX and STREX instructions enclosing their actual memory access in a mutex. This solution, although capable of serializing all the overlapping atomic accesses, is unable to emulate completely the LL/SC semantic since it does not solve the ABA problem. In fact, in the sequence $ll_a(y, z, val_1) \rightarrow w_b(y, z, val_2) \rightarrow w_b(y, z, val_1) \rightarrow sc_a(y, z, val_3)$ the SC does not fail. COREMU [Wang et al. 2011] also does not address at all the ABA problem, without even handling concurrent LDREX or STREX instruction.

Other related works like [Hong et al. 2012], [Chipounov and Candea 2010], [Scheller 2008] and [Brad et al. 2012] combined (or entirely replaced) the current dynamic binary translator of QEMU with LLVM, offloading to an additional thread the compilation and optimization of the generated code. These works, although proposing indeed a multi-threaded version of QEMU, keep the emulation of multiple guest cores in one single thread, as in the vanilla version of QEMU.

Broadening the scope, another work that relates to QEMU, since it adopts dynamic binary translation, is ARCSim [Almer et al. 2011] that translates the ARCompact atomic exchange wrapping its emulation with spin locks. While this solution can work to emulate CAS-like instructions, it would not work with LL/SC instructions. Similar conclusions can be drawn for [Almer et al. 2012] that instead maps the ARCompact atomic instruction directly to x86 compare and exchange. This is perfectly fine for the scope of the ARCompact ISA, but would not work for LL/SC instructions emulation due to the reasons presented in Listing 3 and 4.

These related works can certainly give some hints on how to translate specific guest atomic instructions, but do not help in building an infrastructure that could address different types of guest instructions emulated by different host architectures.

IX. Conclusion and Future Works

This paper described the key points for a new unified approach towards atomic instruction translation in the QEMU emulator, focusing on the set of problems introduced by parallel execution of guest cores, like the ABA problem.

One of the most important aspect discussed, relates to the use of the LL/SC semantic to implement a set of helper functions. These helpers can be used to translate ARM's *ldrex* or *strex* directly, but also additional ones e.g. CAS instructions.

As expected, the overhead of the emulated instructions turned to be higher than the non multi-threaded counterpart, but still under 15% in a real use case scenario. Important to note is that the proposed implementation did not produce any errors and resulted in a flawless boot of Linux (which makes considerable use of atomic instructions). While there is a cost in performance, the new infrastructure provides means for a real parallel execution of SMP guest environments, as we can see in Figure 5.

The source code for the proposed implementation, which is currently at its seventh iteration, is available at [5] git repository. The patch series will be updated according to the feedback of the community that already helped to advance the work to this stage. In the future releases, the support of other architectures will be added to extend the multi-threaded execution for further QEMU supported targets.

X. Acknowledgment

The authors of this paper would like to thank Huawei Technologies Duesseldorf GmbH that made possible the realization of this work.

References

[Almer et al. 2011] O. Almer, I. Bohm, T. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham, *Scalable multi-core simulation using parallel dynamic binary translation*, in Embedded Computer Systems (SAMOS), 2011 International Conference on, july 2011, pp. 190-199

[Almer et al. 2012] Oscar Almer, Igor Bhm, Tobias Edler von Koch, Bjrn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, Nigel Topham, *A Parallel Dynamic Binary Translator for Efficient Multi-Core Simulation*, International Journal of Parallel Programming, 2012, Volume 41, Number 2, pp. 212-235

[Anderson and Moir 1995] James H. Anderson and Mark Moir. 1995, *Universal constructions for multi-object operations*, fourteenth annual ACM symposium on Principles of distributed computing (PODC '95). ACM, New York, NY, USA, 184-19

[Bellard 2005] Bellard, Fabrice, *QEMU, a Fast and Portable Dynamic Translator*, In USENIX Annual Technical Conference, FREENIX Track, pp. 41-46. 2005.

[Brad et al. 2012] Brad Alexander, Sean Donnellan, Andrew Jeffries, Travis Olds, and Nicholas Sizer. 2012. *Boosting instruction set simulator performance with parallel block optimisation and replacement* In Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122 (ACSC '12), Mark Reynolds and Bruce Thomas (Eds.), Vol. 122. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 11-20.

[Broughton et al. 1982] J. M. Broughton, P. M. Farmwald, T. M. McWilliams, *S-1 Multiprocessor System*, SPIE Technical Symposium East, Arlington, Virginia, May 3-7, 1982

[Chipounov and Candea 2010] Chipounov, Vitaly, and George Candea, *Dynamically Translating x86 to LLVM using QEMU* No. EPFL-REPORT-149975. 2010.

[Dechev et al. 2010] Damian Dechev, Peter Pirkelbauer, Bjarne Stroustrup, *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-Free Designs*, 16th IEEE International Symposium (ISORC 2013), pp. 185-192, 2010

[Ding et al. 2011] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. 2011, *PQEMU: A Parallel System Emulator Based on QEMU*, ICPADS '11, Washington, DC, USA, 276-283

[Gifford and Spector 1987] David Gifford and Alfred Spector. *Case study: IBM's system/360-370 architecture.*, Commun. ACM 30, 4 (April 1987), 291-307, 1987

[Hong et al. 2012] Hong, Ding-Yong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung, *HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores* In Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 104-113. ACM, 2012.

[Jensen et al. 1988] E. H. Jensen, G. W. Hagensen, J. M. Broughton, *A New Approach to Exclusive Data Access in Shared Memory*, S-1 Project. 15th Annual International Symposium on Computer Architecture, 1988

[Maged 2004] Maged M. Michael, *ABA Prevention Using Single-Word Instructions*, IBM Research Division, RC23089, Tech. Rep., January 2004

[Scheller 2008] T. Scheller. *LLVM-QEMU*, Google Summer of Code Project, 2008.

[Wang et al. 2011] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen Weihua Zhang and Binyu Zang. *COREMU: a Scalable and Portable Parallel Full-system Emulator*, ACM SIGPLAN PPoPP 2011. San Antonio, USA, February, 2011

[1] *QEMU's wiki web page*, http://wiki.qemu.org/Features/tcg-multithread

[2] *Intel 64 and IA-32 Architectures Software Developers Manual*, http://www.intel.fr/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, September 2015

[3] ARM white paper *Memory access ordering - an introduction*, http://community.arm.com/groups/processors/blog/2011/03/22/memory-access-ordering–an-introduction

[4] *QEMU git repository* git://git.qemu-project.org/qemu.git

[5] Source code GIT repository: https://git.virtualopensystems.com/qemu_tcg/mttcg_qemu/tree/slowpath-for-atomic-v5