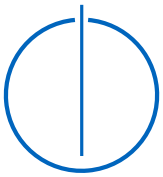
DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Guided Research Report in Informatics

Native emulation of the cmpxchg instruction on ARM by QEMU

Jasper Karl Ottomar Rühl



I am very happy for the opportunity to work on such a cool piece of software and grateful to my mentor Redha for advising me about benchmarks and implementation decisions, enduring my questions and of course conversing about CPU's or One Piece in general.

1 Introduction

With new powerful and efficient processor architectures such as ARM or RISC-V establishing themselves during the last years, the dominance of x86 processors seems to be slowly fading. With these fresh ISAs offering various benefits, they justify a migration away from x86 systems. As these ISA's are not natively compatible with each other, software that has been built for x86 systems can not easily be executed on these other platforms. As simply recompiling all necessary software for the newly adopted ISA might not be possible, as it would be the case for purchased software, the need for solutions easing such a transition arises. QEMU is such a solution, capable of emulating an x86 processor on ARM hardware through dynamic binary translation. (Many other ISA's are also supported)

Dynamic binary translation is a powerful approach to support execution of unmodified binaries across different architectures, where encountered instructions are on-the-fly translated to equivalent ones of the new ISA. With different architectures employing different memory synchronisation models, a simple mapping of instructions, more specifically atomic operations, to their equivalent is not always sufficient:

For multi-threaded synchronisation, i.e. to lock a certain semaphore, conditional read-modify-write operations are necessary. These read and conditionally modify their target address atomically, eliminating the possibility of a race condition. Under x86, this can be achieved with the CMPXCHG instruction. We improve the way QEMU emulates this instruction on ARM to gain a small boost in performance.

2 Background

In this section, we give some background information on the concepts and instructions used.

2.1 Dynamic Binary Translation

The concept of dynamic binary translation (DBT) describes the idea of emulating a binary on another ISA by translating its relevant parts just in time at execution. "Relevant parts" can be considered all instructions that are about to be executed. Given a point in the program, all following instructions leading up to a branching instruction (e.g. `JNZ`) are easily identified as such. This leads to the definition of a "basic block" (BB) made up of many non-branching instructions and one ending branching instruction.

Usually, DBT operates on the BB level, translating one BB at a time. At the end of the BB, the translation is either resumed or, incase the following BB has been encountered earlier, execution immediately continues with the other BB.

2.1.1 QEMU's TinyCodeGenerator

QEMU does this by translating encountered instructions into an ISA-independent language, also known as an intermediate representation (IR). Then, QEMU's TinyCodeGenerator (TCG) compiles the IR code into code of the host's ISA. Instructions of the TCG's IR are known as TCG-Ops.

By using the IR, the complexity in cross-architecture translation schemes is reduced from $O(n^2)$ to $O(n)$, as you now only need to add the translation scheme from and to the IR when adding another ISA to the supported architectures.

2.2 Memory Consistency Models

Memory consistency models (MCM) specify the order in which memory operations (read/write) issued by a processor become visible to other processors.

A MCM is considered "weak" if it does not guarantee such operations from a single CPU end up being seen in that order. Such MCM arise due to various architecture-specific reasons like memory hierarchy with intermediate buffers and so on.

To enforce memory synchronisation, one can use special barrier instructions, which however significantly impacts performance. ARM's MCM is weaker than x86's, meaning that for a given sequence of reads/writes, ARM's MCM allows some reorderings that X86's does not. This has some implications for emulating a x86 binary on ARM: As these "unallowed" reorderings were not taken into account when compiling the binary, it might exhibit some new, possibly unwanted outcomes. Thus, we have to manually ensure these scenarios do not occur by leveraging the aforementioned synchronising instructions.

2.3 Atomic Operations

The main characteristic of an atomic operation is that all effects of it (e.g. on memory) take place at the same time. As such, there is no chance of it's effects being partially applied due to interruptions by other threads.

For some instructions, the atomic property is essential, namely the ones used to achieve synchronisation among multiple threads. Consider this short example of a lock: A lock is a piece of memory shared among different threads. To indicate that a thread is entering a critical section, it will want to first check if any other thread is currently in a critical section. To do so, it will check the value of the lock. If the value indicates it is already locked, the thread will wait and try again. If the lock is open, the thread will lock it by writing a certain value to it and continue in the critical section.

This sequence of checking the lock and then conditionally locking it requires atomicity: If two threads observe that the lock is free, either due to one thread being interrupted after reading the lock or due to the threads running concurrently, both will write to it and enter their respective critical section, which is undesirable. To prevent such a scenario, instructions supporting the reading and conditional modifying of memory in one cycle were introduced. In the following, we describe an x86 instruction commonly used for this case and its ARM counterpart.

2.3.1 x86 CMPXCHG

The CMPXCHG instruction is a read-modify-write instruction. Given an address, it compares the value at that address with an operand specified value. If they are equal, another operand specified value is written to said address.

The zero flag is set according to the comparison, and the return value indicates the value at said address after the operation. CMPXCHG's return value is always in the RAX register. One input operant is always passed via RAX, the other two can be varying registers.

Due to the x86 memory consistency model and CMPXCHG being both a read and a write operation, all preceding stores will have commited by the time cmpxchg commits. Also, all succeeding reads on the specified address will observe the value resulting from the CMPXCHG operation.

2.3.2 ARM CAS

The family of CAS instructions was introduced to the ARM ISA with version 6.1. The semantics are similar to CMPXCHG. Here, however, all three input operands can be freely specified. Just like under x86, the results output is written to the register specifiying the value with which $[addr]$ should be compared. The family is made up of variants of this instruction differing in accessing width and memory semantics.

To emulate CMPXCHG's behaviour, we use the CASAL* variants, as these use aquire-release semantics, mirroring the x86 consistency model.

2.3.3 QEMUs translation scheme

Due to mismatching memory consistency models of the varying supported architectures, QEMU relied on functions provided by GCC to emulate the the CMPXCHG instruction. When encountering a CMPXCHG in the guest binary, TCG generates a call to the GCC "atomic_cmpxchg" function. This requires a lot of setup to pass the necessary operands to the function etc, as can be seen in lines 5-9 in Listing 1. Note the IR call or translated call to the helper function in line 10 or lines 23-24, respectively.

Further, a function call leads to a jump in execution, with the return from said function adding a second one. Thus, such a call could lead to an unnecessary penalty in performance.

```
1  IN:
2   0x004156a3: lock cmpxchgl %edx, 0xbad05(%rip)
3
4  OP:
5   mov_i64 tmp16,rdx
6   mov_i64 tmp17,rax
7   mov_i64 tmp2,$0x4d03b0
8   mov_i32 tmp11,tmp17
9   mov_i32 tmp19,tmp16
10  call atomic_cmpxchgl_le,$0x2,$1,tmp20,env,tmp2,tmp11,tmp19,$0x20
11  extu_i32_i64 tmp15,tmp20
12  ext32u_i64 rax,tmp15
13  mov_i64 cc_src,tmp15
14  mov_i64 loc10,tmp17
15  sub_i64 cc_dst,tmp17,tmp15
16
17 OUT:
18  mov       x0, x19
19  mov       w1, #0x3b0
20  movk      w1, #0x4d, lsl #16
21  mov       w2, #0
22  mov       w3, #1
23  mov       w4, #0x20
24  ldr       x30, #0xffffb001e1e8
25  blr       x30
26  mov       w20, w0
27  str       x20, [x19]
28  str       x20, [x19, #0x98]
29  neg       x20, x20
30  str       x20, [x19, #0x90]
```

Listing 1: *original translation of a cmpxchg instruction*

3 Design

In this section, we describe the design and implementation of our native CAS translation scheme.

3.1 How to change the translation scheme

In the beginnings of the project, we were not sure where to insert our changes in the code. Initially, we planned to do a case distinction on the code that encounters the TCG IR call and generates a call to the GCC function, to reduce the number of code files I would need to adapt. As this would have greatly increased the complexity of that code, inserted some very different functionality, and reducing the number of code files touched is not the criteria for a good commit, we decided to go a different way. Instead, we chose to define a new TCG instruction. Although this means changing the code at many different locations, it is the "cleaner" way to do it, as the necessary changes are slimmer and allow us to include our changes nicely in the translation pipeline.

3.2 Introducing the CAS instruction

We introduce a new IR instruction by defining it in `include/tcg/tcg-op.h`:

```
DEF(cas, 1, 3, 0, IMPL(TCG_TARGET_HAS_cas)
        | TCG_OPF_SIDE_EFFECTS)
```

Listing 2: *definition of the CAS TCG instruction*

This macro is included in different C files. It indicates the name, number of outputs and inputs, and two flags: The first indicates whether the instruction is supported by the host, the second flag means that the instruction can lead to side effects, as CAS writes to memory.

Besides defining the number of input and output operands, we need to define the *register constraints*, which tell TCG how to handle the register assignment when generating the IR code.

```
case INDEX_op_cas:
    return C_O1_I3(r,r,r,r);
```

Listing 3: *constraints for the CAS TCG instruction*

The r's indicate that all in- and output operands can be placed in regular registers.

3.3 Translating CMPXCHG

With the CAS instruction defined, we next examine how it is used to translate the x86 CMPXCHG instruction to an ARM instruction. We are going to inspect the generation of the IR instruction and how these are finally translated to cas-al instructions.

3.3.1 Translating to IR

The code responsible for translating the x86 binary to the TCG IR is found in *target/i386/tcg/translate.c* As we only focused on translation to ARM in the scope of this project, we only implemented our changes for this host architecture. Essentially, we replace the existing code with code that always generates a TCG CAS.

As CMPXCHG supports different widths, we actually do not just introduce one CAS instruction, but one for each of the possible widths, respectively ones of 1,2,4 and 8 byte. The *tcg_gen_cas* function generates the appropriate IR instruction.

```
void tcg_gen_cas(TCGv oldv, TCGv cmpv,
        TCGv newv, TCGv addr, MemOp memop){
    memop = tcg_canonicalize_memop(memop, 1, 1);
    int op=0;
    switch (memop){
        case MO_8:
            op=INDEX_op_cas8;
            break;
        case MO_16:
            op=INDEX_op_cas16;
            break;
        case MO_32:
            op=INDEX_op_cas32;
            break;
        case MO_64:
            op=INDEX_op_cas64;
            break;
        default:
            tcg_abort();
    }
    tcg_gen_op4(op,
        tcgv_i64_arg(oldv), tcgv_i64_arg(cmpv),
        tcgv_i64_arg(newv), tcgv_i64_arg(addr));
}
```

Listing 4: *tcg_gen_cas*

3.3.2 Generating the ARM instruction

During the second stage of the DBT, the TCG consecutively translates each IR instruction into an ARM instruction. The only thing left for us is to specify how an encountered CAS instruction is translated. Similar to the generation of the IR instruction, we define one function to cover all four cases of CASALs to generate.

To determine the value of the CASAL constants, the ARM instruction manual was consulted.

```
1  static void tcg_out_cas(TCGContext *s, TCGReg cmpreg,
2          TCGReg writereg, TCGReg addrreg, MemOp op){
3      AArch64Insn casins=CASAL32;
4      switch (op){
5          case MO_8:
6              casins = CASAL8;
7              break;
8          case MO_16:
9              casins = CASAL16;
10             break;
11         case MO_32:
12             casins = CASAL32;
13             break;
14         case MO_64:
15             casins = CASAL64;
16             break;
17         default:
18             tcg_abort();
19     }
20     tcg_out32(s,
21     casins | cmpreg << 16 | writereg | addrreg << 5
22     );
23 }
```

Listing 5: *tcg_out_cas*

```
1  IN:
2  0x00416e86:  cmpxchgl %edx, 0xb9593(%rip)
3
4  OP:
5   mov_i64 tmp14,rdx
6   mov_i64 tmp15,rax
7   mov_i64 tmp2,$0x4d0420
8   cas32 rax,rax,tmp14,tmp2
9   mov_i64 cc_src,rax
10  mov_i64 loc10,tmp15
11  sub_i64 cc_dst,tmp15,rax
12
13 OUT:
14  ldr       x20, [x19]
15  mov       x21, x20
16  mov       w22, #1
17  mov       w23, #0x420
18  movk      w23, #0x4d, lsl #16
19  casal     w20, w22, [x23]
20  str       x20, [x19]
21  str       x20, [x19, #0x98]
22  sub       x20, x21, x20
23  str       x20, [x19, #0x90]
```

Listing 6: *new translation of a cmpxchg instruction*

Note the changes to the original scheme, manifested by the `cas32` and `casal` instructions in lines 8 and 19.

3.4 Bugs with Optimization enabled

With the above described code, we started to run some benchmarks in order to measure the performance improvements. However, some benchmarks would run indefinitely, hanging in a particular routine. The confusion was intensified when the problem manifested itself only in some versions of our project, although there seemed to be no relevant difference between these. We noticed that the problem only occured in versions where the optimization of the generated IR ops

was enabled. However, we were not able to find a solution to the problem, promting us to consult the QEMU community.

Shortly after, we got a responding email, pointing out various issues with our implementation, most notably that our *constraints* were incorrect. Instead of (r,r,r,r) they should be (r,0,r,r). Adopting the proposed fix solved the problem.

3.5 Missing Aspects

The email also pointed out some missing aspects of our implementation we had not thought about, namely:

1. host vs guest adress size differences
2. host vs guest address space mapping
3. host vs guest address alignment

None of these had crossed our minds when developing the implementation as for our testing scenario, nothing of these were relevant. In the following, we will explain the issues and the reasons for missing them.

3.5.1 Address Size Differences

ARM and x86 both feature 64bit wide addresses, which however is not the case for all architectures supported by QEMU. As our focus was on x86 to ARM translation, we neglected architectures with different address sizes. Our solution is thus not easily transferable to such.

3.5.2 Address Space Mapping

QEMU has different ways of mapping the guests virtual memory to the host memory. The simplest and most efficient way is to simply use the same addresses, as this requires no additional overhead by adding a constant offset or using the TLB. Since we only tested our implementation in a mode using the zero-map-through user-mode, we did not consider adapting our solution to the other scenarios.

3.5.3 Address Alignment

x86's CMPXCHG can target any address, such as 0x1000, 0x1001, 0x1002... whereas ARM requires them to be 8 aligned, meaning that our naive translation scheme can not emulate a CMPXCHG targeting a non-aligned address. However, with compilers usually placing addresses at 8-aligned addresses, we did not encounter this problems while benchmarking.

Nevertheless, it means that our solution would need to be expanded beyond a simple CASAL instruction to accurately emulate all possible CMPXCHG usages.

3.6 Current Solution

To tackle non-aligned addresses, the existing translation scheme is two-fold: As the helper function is executed, the target address is checked for alignment. If it is aligned, the GCC helper function leveraging CASAL is called. Incase it is not aligned, however, this instruction can not be used.

Instead, the translation scheme for single threaded applications is applied. The current translation for the BB is discarded, and the CMPXCHG instruction is retranslated in a single-threaded context, allowing the use of a simple scheme. To eliminate the chance of other threads interfering, all other threads are stopped, which is done by throwing a QEMU exception and exiting the emulation of the thread.

```
void* fun(void * arg_passed){
    struct Thread_arg * arg=arg_passed;

    int id = arg->id;
    int * var = arg->var;
    int zero=0;
    long misses=0;

    for (int i=0; i<ITERATIONS; i++){
        __atomic_compare_exchange_4(var, &zero, id, 1, memcon, memcon);
        *var = 0;
    }

}
```

Figure 1: *The function each thread executes*

4 Evaluation

To evaluate the effectiveness of our solution, we developed a small microbenchmark. Each thread performs a fixed number of locking operations distributed across the variables. A small program creates T threads, which all try to first `CAS` variable v_1, then v_2 and so on until they wrap around back to v_1. We opted to have all threads access all variables to maximize the contention on the V variables serving as locks. See Figure 1 for the code executed by each thread.

Figure 2 depicts our microbenchmark's results. Note how our "cas" solution is slightly faster than the original "helper_cas" approach in most cases. The gain in performance is most significant in the 1-1, 4-4, 8-8 and 16-16 configurations. With these featuring a high threads-per-lock ratio, we can assume that the high contention lead to more the CAS instruction failing more often. Our native CAS seems to be performing better than the helper solution in these scenarios, probably because the native instruction takes less time on failure. This leads us to the conclusion that our implementation mostly offers a benefit when a lot of misses are to be expected.

Given that our scenario is highly constructed, and real-world application's performance is dependant on far more instructions, it can be assumed our solution offers a tiny benefit for actual use cases.

5 Future Work

The non-aligned addresses will most likely prove themselves to be the biggest challenge when trying to advance our implementation to fully emulate CMPXCHG. To leverage both the advantage of a native CAS with no helper function and the ability to stop other threads and restart the translation, we could generate some code performing the alignment check at runtime and only in case of non-alignment jump to an external function. For the sake of prototyping, the same function could be reused. Prior to the jump, however, the arguments would have to be set up correctly which might be complicated to do in the scope of generated IR code.

Note that it must not be a simple call to the QEMU function, as we do not want to return to the generated bytecode. As the called function will re-translate the current basic block in a single-threaded context, the generated code from which it is called is no longer needed. It remains to be seen whether such a complexity inducing feature is worth the gain in performance.

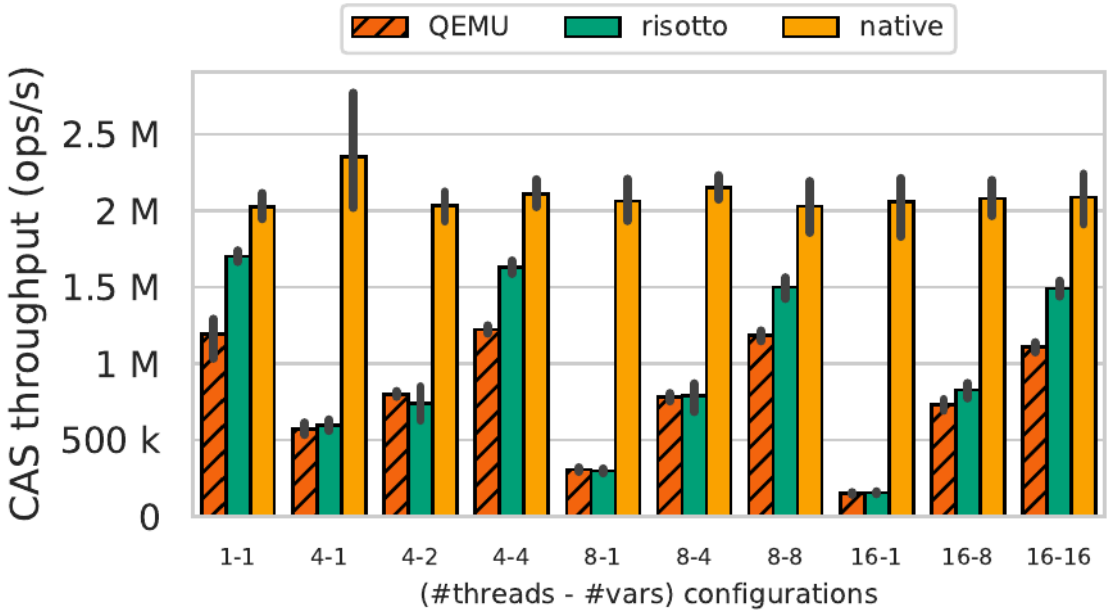


Figure 2: *Results of our microbenchmark (higher is better)*

6 Conclusion

Differing memory consitency models continue to be a challenge for efficient dynamic binary translation. To test our approach, we developed a small microbenchmark and compared it to the existing mechanism as well as a native execution of the benchmark. While our approach did provide a small benefit, its' adoption is ultimatively prevented by the edge case of unaligned addresses.

Working on such a close-to-the-hardware project was a dream of mine since the very first semester. It is amazing to finally have put what I have learnt to use and being able to say that I understand how an emulator works. The hardest part about it and something I have never done before was navigating and understanding QEMU's large codebase. This learning will surely come in handy many times during my career.