

Emulating x86 memory access on ARM64

André Almeida

LPC 2026





Steam Frame, bringing Windows/x86 games to Linux/ARM64!



Proton

The compatibility layer:

- Wine: Windows calls to Linux
- DXVK: DirectX calls to Vulkan
- **FEX-Emu**: JIT x86 to ARM64

`mov rax, [addr] -> ldr x0, [addr]` 

- Memory instructions need a careful handling
- x86 TSO vs ARM64 weak model

`mov rax, [addr] -> ldar x0, [addr]` 

- FEX uses sync instructions (barriers, LL/SC, atomics) to get stronger memory ordering
- This creates much more “atomics” instructions than the original code

- Both x86 and ARM64 allows simple instructions to be issued on unaligned addresses
- x86 allows it atomic instructions, while ARM64 doesn't*
- Plus, x86 even allows for cross-cache line atomics, the so called **split locks**, as we will discuss later

- Calling an unaligned atomic instruction in ARM64 raises a SIGBUS exception
- FEX currently traps this and handles it in userspace
- All this context switch is costly: our experimental work shows that is more than 5x faster to do it directly on the kernel side

- For instance, we replace an unaligned 32bit atomic with an aligned 64bit atomic, and do the bit shifting to get the result

- ARM released extensions for Armv8.3+ to help with TSO emulation: FEAT_LRCPC/2/3
- **LDAPR** acts like load + barrier (*AcquirePC* semantics)
- Apple Silicon implemented full TSO emulation on hardware
 - Which create much less “new” atomic instructions

- LRCPC extensions can't solve one corner case: the split locks.
- x86 can do cross-cache line atomic instructions
- The hardware issues some kind of “bus lock” that blocks a whole cache
- Adds latency to all unrelated tasks
- Linux has some mitigation to avoid a DoS

Guess which application uses split locks? Games!



- There's no simple way to lock two cachelines
- Atomic instructions can't cross a cacheline
- Doing it with two instructions can cause a "memory tear"

- Let's imagine a 64-bit region that crosses a cache boundary
- Thread A wants to do a 64-bit compare and swap on it, Thread B wants to do two 32-bit compare and swaps

	1...32	32...64
cache line 1	0x...	0x00000000
cache line 2	0x00000000	0x...

```

A
CAS64 0 0 -> 2 2

B
CAS32 0 -> 1
CAS32 0 -> 1

```

A) CAS64 (0 0, 2 2), return 2 2
B) CAS32 (0, 1), fails and returns 2
B) CAS32 (0, 1), fails and returns 2

...

2

2

...

```
B) CAS32 (0, 1), returns 1
A) CAS64 (0 0, 2 2), fails and return 0 1
...
```

...	1
1	...

- 1 0 or 1 1 are other valid permutations here

- Both valid scenarios
- For our emulation we need to split SWP64 into two SWP32

A	B
CAS32 0 -> 2	CAS32 0 -> 1
CAS32 0 -> 2	CAS32 0 -> 1

```
A) CAS32 (0, 2), return 2
B) CAS32 (0, 1), return 1
A) CAS32 (0, 2), fails and return 1
B) CAS32 (0, 1), fails and returns 2
```

...	2
1	...

- Now combine both A) CAS32 returns to give a final A) CAS64 return to userspace
- CAS64 fails and returns 2 1
- Abnormal result, memory tearing

Discussion points

- We need someway to execute two CAS32 “atomically”
 - Halt every other CPU while this CPU executes this?
 - Have a Giant Lock in userspace?
 - What else?
- Should mainline support this kind of usecase?

Thanks!

We are hiring

<https://www.igalia.com/jobs/>

