DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating the impact of the x86 hardware memory ordering on the Apple M1 processor through QEMU

Marcel Faltus

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating the impact of the x86 hardware memory ordering on the Apple M1 processor through QEMU

Evaluierung der Auswirkungen der x86-Hardware-Speicheranordnung auf dem Apple M1-Prozessor mit QEMU

Author:	Marcel Faltus
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Redha Gouicem
Submission Date:	15.09.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2022 Marcel Faltus

Acknowledgments

I want to thank Dr. Redha Gouicem for showing me the ropes and supporting me throughout my thesis with his ideas and constructive criticism.

Another thank you goes to Dr Tom Spink who, together with Dr. Redha Gouicem, helped me to make sense of the evaluation results and showed me new ways to analyse and debug problems with QEMU.

Also thank you to my supervisor Prof. Dr.-Ing. Pramod Bhatotia for giving me the chance to work on this thesis.

Lastly a big thank you to my family for supporting me throughout my studies at the Technical University of Munich.

Abstract

In recent years ARM computers have become more popular on the consumer side among other reasons because Apple included an ARM processor in their PC-lineup [App20a; sta21; AWS; Sha21].
One disadvantage of ARM is that previous programs, that were only written with the x86 architecture in mind, cannot be naively executed on ARM since it differs from the x86 Instruction Set Architecture (ISA) in a multitude of ways. One of the lesser know ones is the memory order. ARM allows the CPU to reorder memory accesses as it sees fit, making no guarantees in which order variables get read or written to [Lim22; Alg+21].
This causes emulation of x86 Programs on ARM to require additional protection to prevent this behaviour. Apple added this protection in hardware whereas previously this was done through software with memory barrier instructions [App20b; wikb].
This paper adds this hardware protection, also known as x86 total store order (TSO), to QEMU and measures the impact it has on Full-System Emulation (FSE) performance compared to memory barrier instructions and an implementation without any memory barriers. The latter one should be the fastest albeit incorrect implementation since it ignores the memory ordering completely. Comparisons with Rosetta can not be made yet since QEMU does not support User-Mode Emulation (UME) on Mac computers.
The performance tests show different results, seemingly at random. For example some implementation is around 40% faster as the others. While this behaviour affects is consistent within a single performance test with a one and a half hour duration, running another test shows a different implementation being around 40% faster while the previous one is within margin of error of the baseline performance test. The cause remains undiscovered even after comparing the generated transformed code for each implementation.

Contents

1 Introduction

In recent years more and more big tech companies have started to steadily move from the dominating x86 Architecture, backed by Intel and AMD, to ARM because of benefits like power efficiency, performance and license support [sta21; AWS; Sha21]. Additionally to the corporate transition to ARM, the consumer side also started to move to ARM thanks to laptop manufacturers like Apple and Lenovo [App20a; Har22].

However the incompatibility of existing x86 programs with the ARM architecture creates a problem that among others Microsoft and Apple want to solve [Mica; App20b]. One of the biggest challenges is the mismatch between the ARM and x86 memory models. Memory accesses under the x86 architecture are completed in the same order as the program instruction causing them. This behaviour is called a strong memory model. Contrary the ARM memory model is a weak memory model which allows for reordering of memory accesses as the CPU sees fit [Lim22; Alg+21].

This creates the need for the emulator to support strong-on-weak memory consistency [wikb]. Thus dynamic binary translators have to insert memory fences to restrict the CPU from freely reordering these instructions, leaving performance on the road [LZC20]. One such example is QEMU [QEMa] which attempt's to imposes a stronger restrictions on memory accesses when emulating the x86 ISA on ARM. However QEMU's attempt is flawed and does not preserve the original application's semantics [Gou+23].

Apple and Microsoft both have support for ARM computers [Micb; App20a]. Microsoft's approach lies with static translation within the WOW64 layer. To prevent repeated translation of the same binary they combine the static translation with a caching service with optimization support [Mica]. In contrast to Microsoft, Apple not only uses static translation with caching but masterfully merges this with dynamic translation and hardware assistance with ensuring strong-on-weak memory consistency [App20b; Nak21b].

This thesis proposes taking advantage of Apples hardware approach to handling strong-on-weak memory consistency which allows QEMU to operate without the use of memory fences while still maintaining strong-on-weak memory consistency. This is done by using a two part approach. Firstly QEMU is modified to translate code without generating memory fences. On its own this breaks strong-on-weak memory consistency but in combination with the second part of using a kernel module to switch Apples M1 CPU into x86 TSO mode this provides perfect compliance with the strong x86 memory model.

Following that it is attempted to measure the performance impact of hardware TSO on the Apple M1 however due to some yet unknown factors the results are either very close together or they contain an outlier with a ca. 60% performance increase. This also applies to incorrect QEMU version that are included to have an idea about the maximum achieve able performance increase. Since these results are nonsensical, it is attempted to find differences in the translated code of the tested QEMU builds but the found differences are expected.

2 Background

As a start it is important to understand the difference in strong and weak memory models with the focus on x86 TSO (from now on just TSO) and ARM's memory model. Another important part is the difference of Rosetta 2 and QEMU and why Rosetta 2 cannot fully replace QEMU.

2.1 Memory models and memory ordering

For processors built with the x86 instruction set architecture in mind memory access is strongly ordered meaning the program instruction order is preserved [Lam79]. While ARM has a weak memory model.
But how can this difference in memory models affect execution? Consider following program in Figure 2.1, where both threads run simultaneously: What is the list of

```
/* A and B have been initialized to 0 and are globally visible */
A = B = 0;
```

Thread 1:

```
A = 1;
B = 1;
```

Thread 2:

```
int X = 2;
if ( B == 1 )
{
    X = A;
}
```

Figure 2.1: Example program vulnerable to out of order execution

possible values for **X**? On first glance this problem seems simple since only after **A** is set to **1**, **B**'s value is changed to **1** too, so the only possible values for **X** are **1** and **2**. The assumption of code being executed in the same order as specified by the programmer is known as Sequential Consistency (SC) and is among others found in Intel or AMD CPUs [Lam79]. However weak-memory-models like ARM are not required to behave this way. If the code in Figure 2.1 is run with the ARM memory model a different outcome becomes possible since the CPU does not need to respect the order of memory accesses dicated by the programmer. The CPU could execute **B = 1** before **A = 1** therefore making the possibility of **X** being **0** also an option.

Effects on binary emulation This possibility of out of order execution is a problem with emulating programs with a stronger memory ordering than the host provides. Since the emulated program was written with a strong memory model, like TSO in mind it is possible for the program to behave in unexpected ways when this assumption is violated. This incorrectness can be avoided by forcing the processor to commit affected memory accesses with memory fence instructions. But in turn the added memory fences have a big performance impact [LZC20].

2.2 Full-System emulation and User-Mode emulation

Emulation usually has two modes:

1. Full-System Emulation (FSE)
2. User-Mode Emulation (UME)

As the name of the first mode (FSE) suggest a full virtual machine is emulated, this requires device emulation like CPU topology, storage and the network card. [QEMd]

On the other hand the second mode (UME) is advantageous in situations where whole virtual machine is unnecessary or even undesirable. Since UME can only emulate a program and not a full operating system, it has to translate the program's system calls to host system calls, which has the advantage of running native code. Especially when emulating across different instruction set architectures this can create a significant speedup since opposed to FSE the program directly talks to the host operating system, that is not in need of translation by the emulator [QEMc].

2.3 Rosetta 2 and QEMU

The Mac M1 is special because it recognises the problem Apple's switch from an Intel to an ARM based processor creates and they solve it by shipping with their own emulator called Rosetta 2 [App20b]. Rosetta 2 is a UME and therefore enables good backwards compatibility with Intel programs. However other emulators like QEMU also exist and they bring something to the table that Rosetta 2 does not: Full-System Emulation (FSE). Since at the moment QEMU and Rosetta 2 fill different roles in software emulation on Mac computers it is not possible to compare both emulators against each other in a fair manner since QEMU has to also emulate a full operating system in FSE mode. Rosetta 2 also uses a hardware feature built into the M1 chip that completely removes a lot of the complexity for strong-on-weak emulation by supporting Intel's strong memory model in hardware. However this feature is locked away in the kernel and requires a third-party kernel module to enable [Nak21b; Jha].

2.4 Motivation

Rosetta 2's emulation technique combines static binary translation with dynamic binary translation and hardware memory model switching. While Rosetta 2 supports UME it lacks FSE and therefore its use is limited to "backwards compatibility" in the sense of which programs were compatible with previous Macs. But why should only Rosetta 2 have access to this hardware TSO switch?

In order to measure if this hardware implementation of TSO improves emulation for open source emulators, a patch for QEMU is proposed which is capable of enabling TSO. This removes the need for fence generation in QEMU and this should cause a measurable improvement over current QEMU. Additionally QEMUs problem with incorrect fence placement can be avoided when moving the burden of correct fence placement to the hardware side [Gou+23].

3 Overview

At the moment QEMU emulates x86 programs on ARM using memory fences. The current implementation is still being worked at and in the "Future Works" section [wikb]. In order to take advantage of the integrated TSO mode in Mac M1 CPUs and fully support multi threaded execution for strong-on-weak emulation with QEMU two things need to happen:

- First a way to disable fence generation in QEMU needs to be added for the Mac M1 platform.
- Secondly every one of QEMUs virtual CPU (vCPU) threads needs to have TSO mode enabled.

3.1 Switching CPU memory order for Mac M1

In the following part the currently known Rosetta 2's way of enabling TSO is shown and the approaches of different TSO enabler programs is discussed.

3.1.1 The Rosetta 2 way

Since Rosetta 2 is directly developed by Apple it has the privilege of custom made features to improve its execution time of x86 binaries not native to ARM. This comes with the downside that very little is known about these features and everything has to be revers engineered. One of these features is the TSO mode on the Mac M1 [App20b]. Trough currently unknown means Rosetta 2 executes code in the kernel. That code not only enables the TSO mode but also looks for a specific flag to be set for the process as checked by the "cbz" instruction in Figure 3.1 [Jha20].

From the code segment shown in Figure 3.1 can be assumed that if another program wants to switch to TSO it is not enough to find and mimic the way Rosetta 2 calls kernel code. Additionally it is necessary to set the flag, that gets check by the "cbz" instruction in the code segment, to the correct value for other processes. However the purpose of this flag is speculation since its name is not known [Jha20].

What is known is that by setting a specific bit in the s3_0_c15_c9_0 register the processor enables TSO mode [Jha20].

```
fffffe00071f7bac mrs x3, TPIDR_EL1
fffffe00071f7bb0 ldr x0, [x3, #0x4f8] ; Latency: 4
fffffe00071f7bb4 cbz x0, 0xfffffe00071f7bf8
fffffe00071f7bb8 mrs x0, ACTLR_EL1
fffffe00071f7bbc orr x0, x0, #0x2
fffffe00071f7bc0 orr x0, x0, #0x70
fffffe00071f7bc4 dsb sy
fffffe00071f7bc8 msr ACTLR_EL1, x0
fffffe00071f7bcc isb
```

Figure 3.1: Kernel snippet probably showing the code responsible for switching to TSO mode, found here: [Jha20]

3.1.2 TSO enabler by leeehui

This program developed by github user leeehui is intended to enable TSO mode on Asahi Linux, a linux distribution designed to run on Apple hardware [lee22; Lin]. Similar to the Mac kernel code segment in Figure 3.1, this program sets the required bit in the s3_0_c15_c9_0 register before just returning to user mode as can be seen in Figure 3.2.

```
#define ACTLR_EL1_EnTSO (1ULL << 1)
[...]
u64 actlr = read_sysreg(actlr_el1);
if (turnOn) {
        actlr |= ACTLR_EL1_EnTSO;
} else {
        actlr &= ~ACTLR_EL1_EnTSO;
}
write_sysreg(actlr, actlr_el1);
```

Figure 3.2: Code responsible for switching to TSO mode for the Asahi Linux distribution, source: [lee22]

3.1.3 TSO enabler by Saagar Jha

A different approach is followed by Saagar Jha. Instead of just setting the register to the required value they instead opted for just re-using the kernel code that switches the TSO mode for Rosetta 2. This kernel module is available for the Mac M1 (t8101) chip.

When loading the kernel module it searches for the start of the kernel code from where it subsequently starts its quest to find a specific sequence of instruction, which are part of the TSO mode switching code from Apple. It should also be noted that Saagar Jha only wrote the code to verify whether TSO is working and they say that the instructions that enable TSO in the kernel were originally found by gihub user osy [Jha22].

```
// Look for this sequence:
// ldr xR, [xR, #OFFSET]
// cbz xR, location
// mrs xR, actlr_el1
// xR, xR, #0x2
// xR, xR, #0x70
uint32_t *instructions = (uint32_t *)text_exec;
for (uint64_t i = 0; i < text_exec_size / sizeof(uint32_t) - 5; ++i) {
        if ((instructions[i + 0] & 0xffc00000) == 0xf9400000 &&
                (instructions[i + 1] & 0x2f000000) == 0x24000000 &&
                (instructions[i + 2] & 0xffffffe0) == 0xd5381020 &&
                (instructions[i + 3] & 0xfffffc00) == 0xb27f0000 &&
                (instructions[i + 4] & 0xfffffc00) == 0xb27c0800) {
                printf("TSOEnabler:␣Found␣thread␣pointer␣read␣at␣%p\n", instructions + i);
                // Extract the immediate from the ldr.
                int tso_offset = (instructions[i] >> 10 & 0xfff) << 3;
                printf("TSOEnabler:␣TSO␣offset␣is␣%d\n", tso_offset);
                return tso_offset;
        }
        return -1;
}
```

Figure 3.3: Code responsible for finding Apples TSO switching code used by Rosetta 2, source: [Jha]

3.2 Integration into QEMU

Integrating all this does not require much code but it requires the code to be placed at the right spots so it can coordinate to only disable fence generation when TSO is available. The details of where exactly TSO can be enabled within QEMU is discussed in section 4.2. Also the deactivation of memory fences for QEMU is shown in section 4.3.

4 Design

4.1 TSO mode

As mentioned previously in chapter 3 the integration of TSO into QEMU is split into two parts.
But this is only for the QEMU part and since enabling TSO requires a kernel module. While it is possible to create my own kernel module that switches a thread to TSO mode it will be hard to recreate the same conditions that Rosetta 2 has when emulating x86 binaries. This would create the uncertainty of whether the TSO mode had been enabled as Apple intended and whether the M1 chip behaves as expected.

Because of that the "TSOEnabler" kernel module by Saagar Jha is chosen since it uses Apple's own code to enable TSO which eliminates a lot of uncertainties. Of course it is still unclear if the "TSOEnabler" kernel module behaves exactly like Rosetta 2's TSO mode. But this is still the option which mimics Rosetta 2's way the most [Jha22].

4.2 Where to enable TSO?

The usual place for execution accelerating modifications is the "accel" directory which for example contains Kernel Virtual Machine (KVM) support or multi thread support for tiny code generator (TCG) [QEMb; wikb]. Especially the multi thread support is exactly the spot where TSO should be enabled. However these optimizations usually are platform independent and do not need to influence code translation but this patch cant fulfill these restrictions since it

- only supports Mac M1 processors
- and needs to disable fence generation

Also need to disable fence generation requires changes in the host specific TCG implementation which is discussed in more detail in section 4.3.

Changes for enabling TSO TSO is enabled/disabled on a per-thread level. Conveniently every thread that wants to initiate translation is required to register with the "tcg_register_thread" function. Since UME is unsupported on Mac for now it is a safe to assume that only threads that called the register function will translate code. Note that for UME this assumption is violated because additionally the parent thread is also authorized to initiate translation [QEMb].

Following the execution flow it becomes apparent that while every thread that executes code calls "tcg_register_thread" this function does not directly call anything defined in the host specific TCG files, like "tcg-target.c.inc" in "aarm64" [QEMb]. This presents a problem because the TSO enabling logic is only needed for the "aarm64" host architecture. Note that aarch64 is QEMU's code word for 64-bit ARM. To solve this I propose a new hook called "tcg_target_register_thread" which is directly called by "tcg_register_thread" and is implemented for every supported host architecture. This hook will only serve the purpose of calling the necessary functions for TSO mode for the "aarch64" architecture. For every other architecture this hook will do nothing.

4.3 QEMU tiny code generator

The QEMU TCG is responsible for translating code of any architecture to the target architecture. This is done by converting the code to an intermediate representation before translating it to the target architecture. The tcg is also responsible for ensuring memory access ordering. This is usually achieved by adding memory barrier instructions at specific locations to guarantee at least the memory ordering the source architecture provided [wika; wikb].

The decision of if and where to place fences is done while converting the code to the intermediate representation. QEMU takes the memory ordering of the target and source architecture into account and only places memory barriers for the cases where the target architecture memory ordering is weaker than the expected memory ordering of the source architecture as seen in Figure 4.1[wikb; QEMb].

The implementation in Figure 4.1 is a bit more complex so it can also support applications running in user-mode emulation. Since however user-mode emulation is not supported for the Mac M1, QEMU can only run in full-system emulation mode which defines the Pre-Processor macro TCG_GUEST_DEFAULT_MO [QEMb; wikb; QEMc].

To make QEMU generate fences like it was running on the x86 architecture the only thing that needs to be adjusted is the Macro "TCG_TARGET_DEFAULT_MO". Looking into the source files for the i386 architecture shows following value for the target memory ordering "(TCG_MO_ALL & TCG_MO_ST_LD)". Referencing this value with Figure 4.2 shows that QEMU wants to ensures the order of all memory accesses except for stores before load. This will be the same value the aarch64 has

```
static void tcg_gen_req_mo(TCGBar type)
{
#ifdef TCG_GUEST_DEFAULT_MO
    type &= TCG_GUEST_DEFAULT_MO;
#endif
    type &= ~TCG_TARGET_DEFAULT_MO;
    if (type) {
        tcg_gen_mb(type | TCG_BAR_SC);
    }
}
```

Figure 4.1: Memory barrier decision function in QEMU [QEMb]

```
typedef enum {
    /* Used to indicate the type of accesses on which ordering
       is to be ensured. Modeled after SPARC barriers.

       This is of the form TCG_MO_A_B where A is before B in program order.
    */
    TCG_MO_LD_LD = 0x01,
    TCG_MO_ST_LD = 0x02,
    TCG_MO_LD_ST = 0x04,
    TCG_MO_ST_ST = 0x08,
    TCG_MO_ALL = 0x0F, /* OR of the above */

    /* Used to indicate the kind of ordering which is to be ensured by the
       instruction. These types are derived from x86/aarch64 instructions.
       It should be noted that these are different from C11 semantics. */
    TCG_BAR_LDAQ = 0x10, /* Following ops will not come forward */
    TCG_BAR_STRL = 0x20, /* Previous ops will not be delayed */
    TCG_BAR_SC = 0x30, /* No ops cross barrier; OR of the above */
} TCGBar;
```

Figure 4.2: Values for memory barrier types used in QEMU [QEMb]

to pretend to have as target memory ordering. Also as mentioned previously the value for "TCG_TARGET_DEFAULT_MO" must only change when TSO support is available. Since we have to use a third-party kernel module the existence of said kernel module is not guaranteed and has to be verified every time when QEMU is started. The verification process is not difficult and will be shown in detail in paragraph 5.2.

5 Implementation

5.1 TSO kernel module installation and enabling

The usage of a third-party kernel module requires manual compilation and installation. Additionally this kernel module is not signed requiring further modification of the installed macOS. The following is a short summary explaining the installation process which can also be found in the github repository cited here [Jha].

After installing Xcode and using that to compile the kernel module the generated artifact needs to be put in the directory "/Library/Extensions". Now you may have to confirm a prompt but in case you do not get a prompt you need to change the permissions of the file with "chown -R root:wheel". After that the Security & Privacy preferences tab within Settings should have an option to allow this kernel module. If after a restart the command in Figure 5.1 enables the kernel module you are done and the setup and installation is complete. However if this did not work there is a section in the github repository that explains one last option that usually works [Jha].

```
sudo kextload /Library/Extensions/TSOEnabler.kext
```

Figure 5.1: Enabling the TSOEnabler kernel module, source: [Jha]

Also it is important to remember that after every restart or shutdown of the Mac M1 this kernel module has to be loaded again with the command in Figure 5.1. After the kernel module is loaded a new system call is available and its usage is explained in the next part paragraph 5.2 [Jha].

5.2 QEMU part

Test for TSO availability Since enabling TSO requires the TSOEnabler kernel module by Saagar Jha to be installed and running, at least a simple check for the availability of said kernel module should to be in place. This could serve as a check to switch back to using regular memory fences or simply to warn users about the unavailability of TSO.

Figure 5.2 shows an example system call for enabling TSO. The return value can be used to determine the availability of TSO, which equals zero on success. Depending on where this check is made it may be necessary to call this function again to disable TSO for this thread if it is available, since an unwanted side effects might be the pinning of the current thread to a performance core [Jha; App20b].

```
int tsoEnable = 1;
size_t sizeOfFlag = sizeof(tsoEnable);
int result = sysctlbyname("kern.tso_enable", NULL, &sizeOfFlag,
                          &tsoEnable, sizeOfFlag);
```

Figure 5.2: Calling the syscall for enabling TSO by

Enabling TSO for vCPU threads As mentioned in section 4.2 in order to switch the required threads to TSO mode a hook at the initialization of the vCPU threads is required. The hook "tcg_target_register_thread" is placed at the end of the function "tcg_register_thread", which allows the translation of code after it was called, see Figure 5.3. The downside of this approach is that for every other supported TCG host architecture like sparc, riscv or i386 this hook has to be also present and serves no purpose for them.

```
void tcg_register_thread(void)
{
    TCGContext *s = g_malloc(sizeof(*s));
    [...]
    tcg_target_register_thread();
}
```

Figure 5.3: Hook for vCPU initialization in QEMU [QEMb]

Additionally since not only some Apple computers can run ARM the actual system call to enable TSO needs to be safeguarded by checks that allow the compilation on other platforms that do not support the function "sysctlbyname", which seems to not exist for Linux but for Darwin and FreeBSD [die; Fre; App]. Figure 5.4 shows a possible implementation for the hook which differs slightly from my code on github but it behaves the same way [Fal22b].

```c
static void tcg_target_register_thread(void)
{
    /* Enable TSO if TSO is supported by current platform */
   if (TSO_ENABLED == TSO_Status) {
#ifdef __APPLE__
        int tsoEnable = 1;
        size_t sizeOfFlag = sizeof(tsoEnable);
        int errNum = sysctlbyname("kern.tso_enable", NULL,
                                 &sizeOfFlag, &tsoEnable, sizeOfFlag);
        if (errNum != 0) {
            printf("Called␣tso_enable␣on␣thread␣%08x␣with␣result␣%d\n",
                   *(unsigned int *)((void *) pthread_self()), errNum);
        }
#endif
    }
}
```

Figure 5.4: Possible implementation for the hook to enable TSO only when compiling for the Apple platform on ARM

6 Evaluation

6.1 Research Questions

While the promise of strong-on-weak emulation with perfect QEMU sounds pretty good, the actual functionality of the TSOEnabler kernel module has not been tested. Also it is unknown whether the Apple M1's TSO mode has a positive impact on performance. These questions have been condensed into the following research questions:

1. Does the TSOEnabler kernel module work as expected?
2. Does TSO have an impact on execution and how big is the impact?

6.2 Functionality

Before evaluating the performance impact on QEMU, first the functionality of the TSO kernel module has to be verified. This is done with the test program developed by Saagar Jha, designed to detect store reordering [Jha].

Test systems. The functionality test was executed on a Mac M1 with 4 performance, 4 efficiency cores and 16 GB of memory. Additionally the comparison system for x86 has an AMD Ryzen 7 3800X (8-Cores, 2 Hyper-threads each) with 32 GB of memory.

Setup. The test program was run in a loop and the final value of the "count" variable, after the program detected a reorder and terminated, was recorded.

Results. In the about 17000 test runs done without TSO on the Mac M1 this program never reached a "count" value higher than 1600. With Rosetta 2 on the Mac the program had to be manually terminated after fifteen minutes with a "count" value of more than ten million and no reorders were detected. Similarly when using the "TSOEnabler" kernel extension on the Mac and running the program a similar amount of time to the Rosetta 2 version, around the same "count" value was observed and also no reorders could be detected. The same behaviour was found for the native x86 platform which had a "count" value of more than 80 million after it was manually terminated after ten minutes without any reorders. Note that this test program has been executed on the QEMU builds to verify the working of TSO in QEMU but no thorough test has been made for the working of the TSO system call in the QEMU build.

6.3 Performance Impact of TSO

6.3.1 Shared test conditions

Test system. The performance tests were executed on a Mac M1 with 4 performance, 4 efficiency cores and 16 GB of memory. QEMU was run in full-system emulation mode. As guest operating system the virtual machine build of Arch Linux from 1.6.2022 was used. Due to gitlab job artifact rules this exact build is no longer available. The benchmarks were run and timed on the guest OS. For better understanding Figure 6.1 shows the arrangement of the different layers required to execute the benchmarks.

Benchmark suite. The evaluation was performed using the PARSEC benchmark suite [Bie11]. Since PARSEC does not run natively on Mac no direct comparison was made between native and QEMU performance.

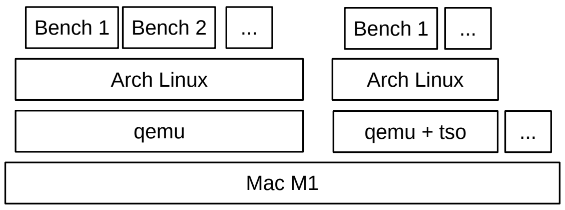

Figure 6.1: Layered setup for the benchmark runs

Setups. The benchmark suites were executed on a custom QEMU build based on the master branch after version "v7.0.0" [Fal22b]. Some of these builds have no changes ("SW Fences (compile time)") or minimal changes ("SW Fences (runtime)") but keep standard QEMU fence generation. The additive "compile time" or "runtime" refer to the point in time when the fence generation method is fixed. This was added to evaluate the possible impact of making TSO enabling a program flag that can be toggled at will. "Compile time" thus is the place in time when the QEMU binaries get build, giving the compiler the option to optimize code paths related to fence generation. "No Fences" and "HW Fences" both disable fence generation but "HW Fences" additionally enables TSO on every vCPU thread. Since the "No Fences" binaries disable fence generation they are not correctly emulating the x86 memory model on ARM however they represent the maximum possible performance with the used QEMU build. Note that in QEMU build and QEMU implementation both refer to these QEMU binary versions. In every plot the build for "SW Fences (compile time)" is used as the baseline, the average performance of "SW Fences (compile time)" is displayed with a red line and the raw values are shown in red with the base unit of seconds.

Mac shortcomings. Due to the host being a Mac, there was no way of pinning threads to individual CPU cores or to fix the CPU frequency. However the TSO kernel module pinned the QEMU vCPU threads to the performance cores, preventing them of running on the efficiency cores. As compiler Apple clang with version 13.1.6 (clang-1316.0.21.2.5) was used.

6.3.2 Performance test

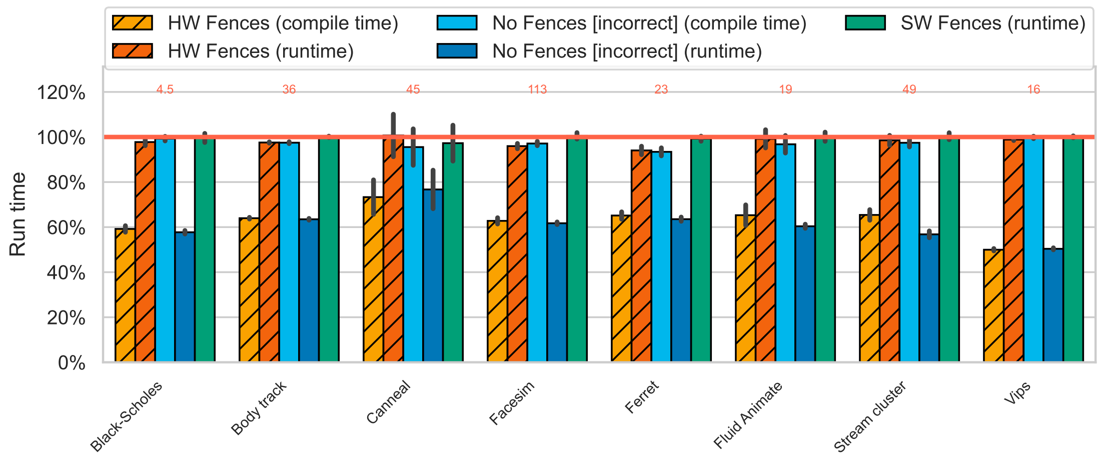

Figure 6.2: Runtime of PARSEC benchmark suite, running on QEMU with no fence generation (No Fences), QEMU with TSO enabled (HW Fences) and QEMU with no changes (SW Fences). Fence generation was fixed at QEMU binary compile time (compile time) or decided at runtime (runtime). All (compile time) runs were done on a *different day* than the (runtime) runs. Lower is better

Methodology. Each benchmark gets executed 20 times using the "simlarge" data set and the speedup to the baseline unchanged changed QEMU "SW Fences (compile time)" is calculated. The virtual machine had access to 4 vCPU cores that are backed by 4 host threads and 1 GB of RAM.

Execution order. The order in which the different QEMU builds were timed differs from performance test to performance test. For Figure 6.2 the execution order was as follows:

1. HW Fences (runtime)
2. SW Fences (runtime)
3. No Fences [incorrect] (runtime)
4. SW Fences (compile time)
5. No Fences [incorrect] (compile time)
6. HW Fences (compile time)

Performance. On first glance the run times in Figure 6.2 look plausible. But on closer inspection a few Questions come to mind:

- Why is "HW Fences (compile time)" about 60% faster than "No Fences [incorrect] (compile time)"?
- Why is "HW Fences (compile time)" so much faster than "HW Fences (runtime)" while for "No Fences [incorrect]" this difference exists too but instead of "(compile time)" being faster it's slower?

Also these speedups are present for the entire Benchmark Suite run, which takes about 1 hour and 30 minutes for the baseline. Due to the length of the runs an the setup as seen in Figure 6.1, CPU features such as caching or branch prediction cannot cause these speedups in subsequent runs since the time it takes for the same benchmark to be run again on a different binary is more than an hour and more importantly if this speedup were caused by such a CPU feature the effect should be visible after the first benchmark suite run and should persist until the last run.

6.3.3 Performance test with increased memory

Narrowing down the cause. To completely rule out any problems with the source code, from here on all QEMU executable were refactored by using pre-processor macros to ensure almost identical treatment for the different Fence generation settings. Additionally confusing the binaries has been made harder by reporting the compile settings at runtime [Fal22a][Fal22b].

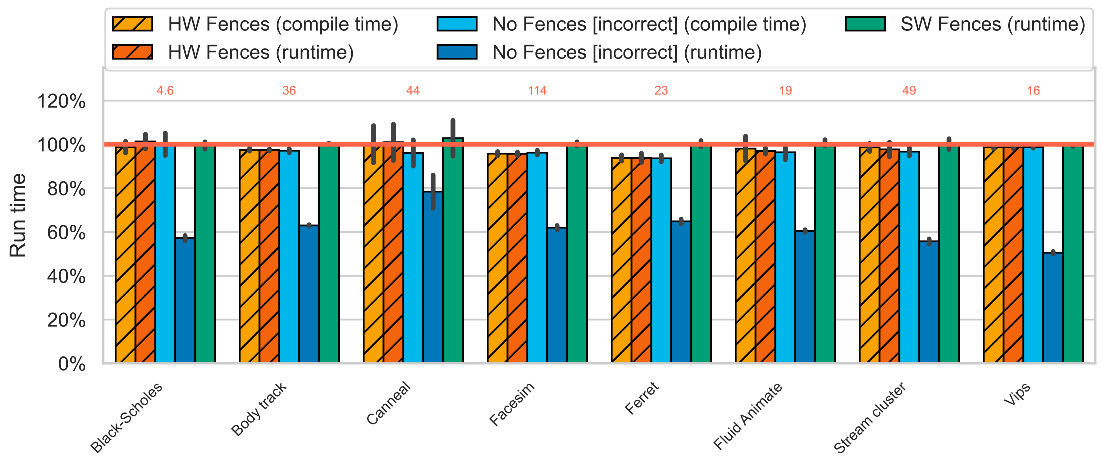

Figure 6.3: Runtime of PARSEC benchmark suite, running on QEMU with no fence generation (No Fences), QEMU with TSO enabled (HW Fences) and QEMU with no changes (SW Fences). Fence generation was fixed at QEMU binary compile time (compile time) or decided at runtime (runtime). All runs took place on the *same day*. Lower is better

Methodology. As before each benchmark was executed 20 times using the "simlarge" data set and the speedup to the baseline unchanged changed QEMU "SW Fences (compile time)" is calculated. Additionally the usable memory for the guest OS was increased from 1 GB up to 4 GB. So now the virtual machine had access to 4 vCPU cores that are backed by 4 host threads and 4 GB of RAM.

Execution order. For Figure 6.3 the execution order was as follows:

1. No Fences [incorrect] (compile time)
2. No Fences [incorrect] (runtime)
3. SW Fences (runtime)
4. SW Fences (compile time)
5. HW Fences (compile time)
6. HW Fences (runtime)

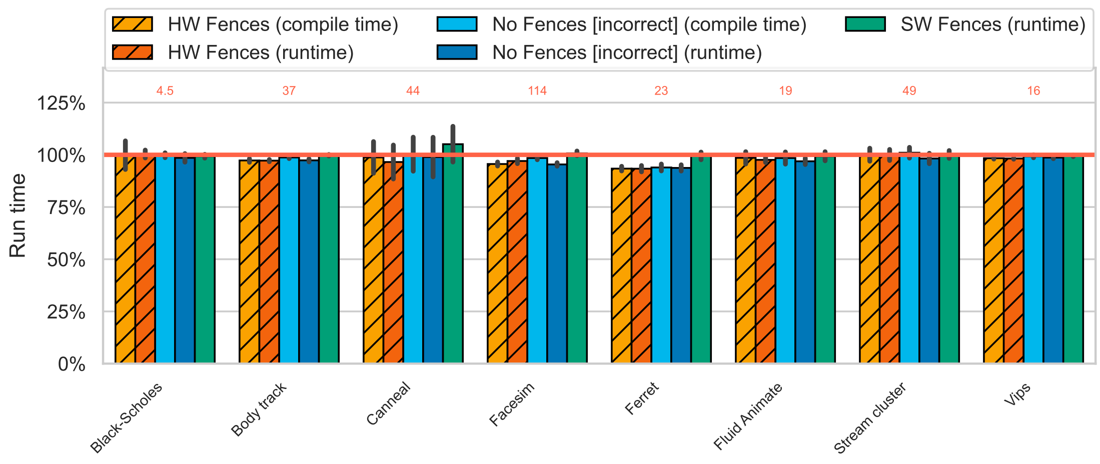

Figure 6.4: Re-run of the Figure Figure 6.3 (previous run) while CPU metrics are being collected. Runtime of PARSEC benchmark suite, running on QEMU with no fence generation (No Fences), QEMU with TSO enabled (HW Fences) and QEMU with no changes (SW Fences). Fence generation was fixed at QEMU binary compile time (compile time) or decided at runtime (runtime). All runs took place on the *same day*. Lower is better

6.3.4 Performance test with CPU monitoring

Methodology. Like before each benchmark was executed 20 times using the "simlarge" data set and the speedup to the baseline unchanged changed QEMU "SW Fences (compile time)" is calculated. The virtual machine had access to 4 vCPU cores that are backed by 4 host threads and 4 GB of RAM.

Execution order. Each section that ends with the overall Performance cluster usage dropping to almost 0% in Figure 6.5 corresponds to a Benchmark suite run. Similar patterns can be witnessed in Figure 6.6 and Figure 6.7 where instead of the usage dropping, the CPU frequency or the Power draw drop drastically. Following is the order in which the Benchmark suites were run. The first item in the list matches the first section, the second item matches the second section and so on. Note that the Benchmark suite run named "SW Fences (compile time)" is used as the baseline in Figure 6.4.

1. No Fences [incorrect] (compile time)
2. No Fences [incorrect] (runtime)

3. SW Fences (runtime)
4. SW Fences (compile time)
5. HW Fences (compile time)
6. HW Fences (runtime)

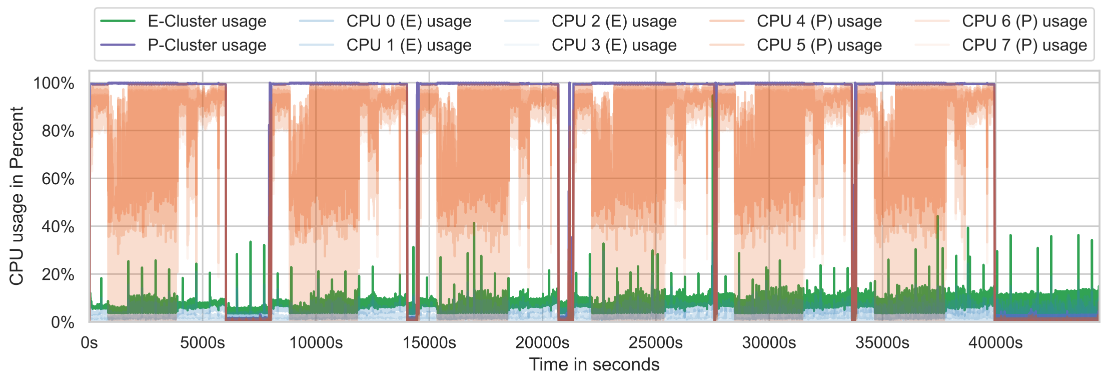

Figure 6.5: CPU usage while running Benchmarks of Figure 6.4. Each CPU core is plotted individually. Orange shades belong to the Performance Cores while light blue shades are for the efficiency cores. The green (Efficiency cluster) and purple (Performance cluster) values are collected by the tool powermetrics

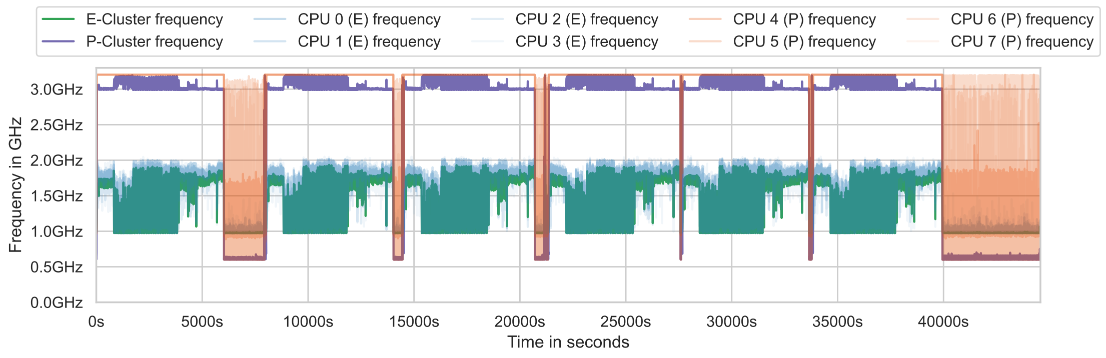

Figure 6.6: CPU frequency while running Benchmarks of Figure 6.4. Each CPU core is plotted individually. Orange shades belong to the Performance Cores while light blue shades are for the efficiency cores. The green (Efficiency cluster) and purple (Performance cluster) values are collected by the tool powermetrics

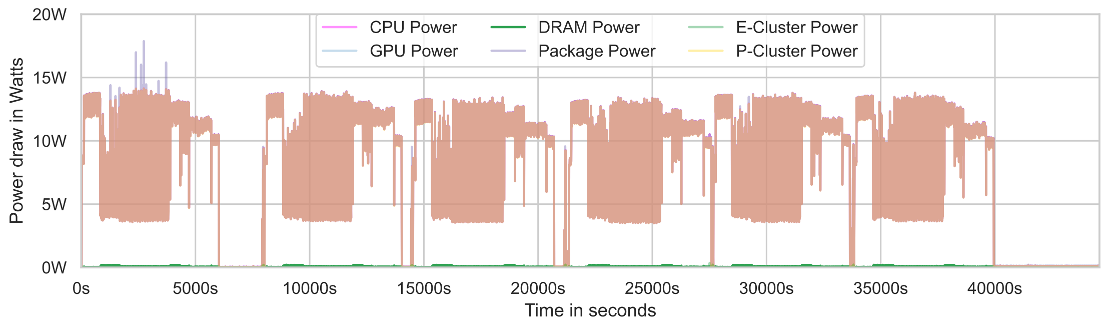

Figure 6.7: M1 chip power draw while running Benchmarks of Figure 6.4. CPU power, Package Power and P-Cluster Power overlap creating the orange area. Like before values are collected by the tool powermetrics

6.3.5 Performance test with delays between QEMU executions

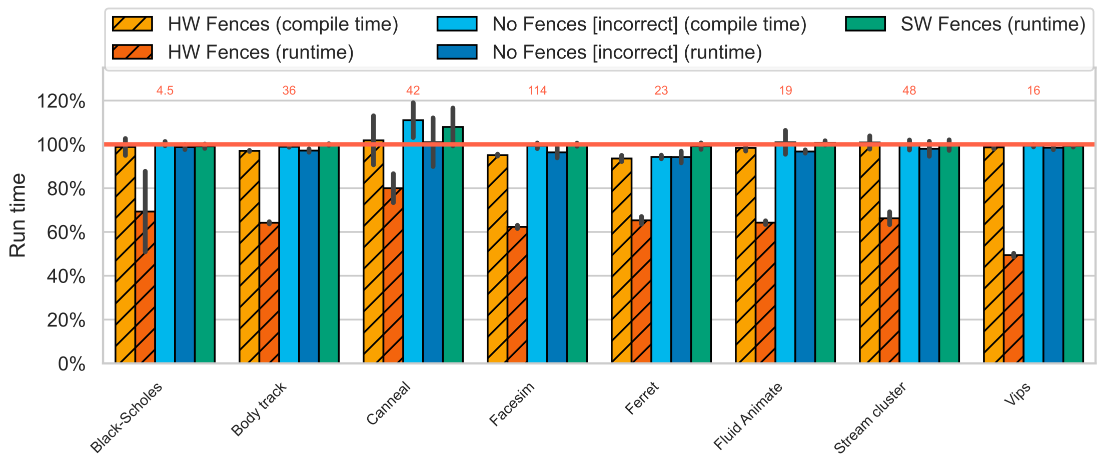

Figure 6.8: Runtime of PARSEC benchmark suite, running on QEMU with no fence generation (No Fences), QEMU with TSO enabled (HW Fences) and QEMU with no changes (SW Fences). Fence generation was fixed at QEMU binary compile time (compile time) or decided at runtime (runtime). Between the end of the previous and the start of the next run a 1 hour wait was added with the exception being "HW Fences (runtime)" which had a 2+ hour wait before it was started. Lower is better

Methodology The number of execution times has been cut down to 10 and the dataset has been changed to "simmedium". Like before the speedup to the baseline unchanged QEMU was calculated. The virtual machine parameters are unchanged at 4 vCPU cores that are backed by 4 host threads and 4 GB of RAM. To reduce interference from factors like CPU throttling a 1 hour wait was placed between each Benchmark suite execution with the exception being "HW Fences (runtime)" which had a 2+ hour wait before it was started.

Execution order. For Figure 6.8 the execution order was as follows:

1. No Fences [incorrect] (compile time)
2. No Fences [incorrect] (runtime)
3. SW Fences (runtime)

4. SW Fences (compile time)
5. HW Fences (runtime)
6. HW Fences (compile time)

6.4 Comparison of code generated by used QEMU builds

Even with the refactored QEMU builds the performance evaluation randomly contains faster benchmark suite runs. Another possible reason could be that different QEMU builds translate the x86 instructions in different ways and these differences may cause the variation in the benchmark runtimes.

TCG intermediate instructions Instead of directly translating x86 instructions to ARM QEMU first converts them to an intermediate representation called TCG ops. Later these are transformed to ARM instructions on the Mac M1 [wika]. So any difference in the TCG ops directly causes a difference in the ARM instructions. In Table 6.1 the first 500 by QEMU transformed blocks have been analyzed and these two examples show the only two types of differences that were found. Comparing "runtime" variants to "compile time" variants shows no difference in TCG ops except for arguments of "exit_tb" like the second part in Table 6.1.

Table 6.1: Comparison of the generated TCG instructions of the custom QEMU builds. Any not colored text is unchanged between QEMU builds, green colored text shows instructions only present in one QEMU build and blue shows differences within a line

No Fences [incorrect] (runtime)	HW Fences (runtime)	SW Fences (runtime)
ext16u_i64 tmp2,tmp2	ext16u_i64 tmp2,tmp2	ext16u_i64 tmp2,tmp2
add_i64 tmp2,tmp2,cs_base	add_i64 tmp2,tmp2,cs_base	add_i64 tmp2,tmp2,cs_base
ext32u_i64 tmp2,tmp2	ext32u_i64 tmp2,tmp2	ext32u_i64 tmp2,tmp2
		mb $0x31
qemu_ld_i64 tmp1,tmp2,leuw,2	qemu_ld_i64 tmp1,tmp2,leuw,2	qemu_ld_i64 tmp1,tmp2,leuw,2
add_i64 tmp2,tmp2,$0x2	add_i64 tmp2,tmp2,$0x2	add_i64 tmp2,tmp2,$0x2
goto_ptr tmp16	goto_ptr tmp16	goto_ptr tmp16
set_label $L0	set_label $L0	set_label $L0
exit_tb $0x280000083	exit_tb $0x280000083	exit_tb $0x103720083

Output ARM instructions For the generated ARM instructions as visible in Table 6.2 the differences were similarly to Table 6.1 with the addition of different virtual memory addresses for the instructions. These virtual memory addresses have been omitted as we want to focus on the ARM instructions. This time comparing "runtime" variants to "compile time" variants shows the same differences as seen in Table 6.2.

Table 6.2: Comparison of the generated ARM instructions of the custom QEMU builds. Any not colored text is unchanged between QEMU builds, green colored text shows instructions only present in one QEMU build and blue shows differences within a line

No Fences [incorrect] (runtime)	HW Fences (runtime)	SW Fences (runtime)
528c2f15 mov w21, #0x6178	528c2f15 mov w21, #0x6178	528c2f15 mov w21, #0x6178
8b150294 add x20, x20, x21	8b150294 add x20, x20, x21	8b150294 add x20, x20, x21
2a1403f4 mov w20, w20	2a1403f4 mov w20, w20	2a1403f4 mov w20, w20
		d50339bf dmb ishld
a97e0660 ldp x0, x1, [x19, #-32]	a97e0660 ldp x0, x1, [x19, #-32]	a97e0660 ldp x0, x1, [x19, #-32]
data: [size=16]	data: [size=16]	
.quad 0x0000000100daf78c	.quad 0x00000001010ada74	
.quad 0x0000000100cddb30	.quad 0x000000010117f74c	

6.5 Discussion of the results

The first research question was **"Does the TSOEnabler kernel module work as expected?"**. The results in section 6.2 show that the TSOEnabler kernel module indeed works as expected however it remains unclear how big the performance impact of using TSO is.

The performance results cannot conclusively answer the second research question **"Does TSO have an impact on execution and how big is the impact?"**. Instead these results showed that something was interfering with the benchmark runs.

A lot of potential errors have been ruled out by the fact that in Figure 6.2 "HW Fences (compile time)" and "No Fences [incorrect] (runtime)" were both significantly faster than the baseline but in Figure 6.3 only "No Fences [incorrect] (runtime)" managed to be drastically faster than the baseline. This indicates that "HW Fences (compile time)" is a fluke and something must have been wrong with the run but Figure 6.4 has no outlier at all. This is strange in itself since "No Fences [incorrect]" should always be faster than normal QEMU since it does not enforce the stronger guest memory ordering, as can be seen in the evaluation of "Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures" [Gou+23]. Also in the last run Figure 6.8 for some reason "HW Fences (runtime)" was an outlier for the first time.

Additionally the comparison of the TCG ops and ARM instructions between the different QEMU versions showed no unexpected differences between those either.

7 Related Work

In this section I want to focus on three amazing works, one takes a deep dive into Rosetta 2's working (section 7.1), another work which creates a QEMU based dynamic binary translator that iterates on QEMU by among other things adding support for dynamic linking with native shared libraries (section 7.2).
And lastly as a comparison a dive into how Windows 10 emulation for ARM works (section 7.3).

7.1 Project Champollion: Reverse engineering Rosetta 2

This project takes a deep dive into Rosetta 2 and dissects a lot of the inner workings for the purpose of finding a vulnerability that can be exploited. This proves to be a great and interesting summary of how Rosetta 2 works [Nak21b].

Rosetta 2's static translation part At the first a look into the static translation part of Rosetta 2 is performed. It is discovered that when a x86_64 program is executed with the **exec** call, a daemon, called **oahd** seeming responsible for caching the static translation artifacts, searches for a previously generated file.
This file has the ending **.aot**, which is an abbreviation of Ahead-Of-Time **AOT**, indicating that it contains the result of Rosetta 2's static translation.
This is indeed the case as a program named **oahd-helper** translates the original x86 code into this format. The location of this **AOT**-file is dependant on two SHA-256 hash values, which are computed from the original x86 code and the execution path [Nak21b].

AOT file format While checking the contents of the **AOT** file it is discovered that this file' is in the Mach-O, which is Apple's own executable format. The contained instructions are for the arm64 architecture however Apple seems to employ a proprietary calling convention. Through analyzing of sample programs the calling convention is reverse engineered.
Additionally the **AOT** file on its own has a lot of undefined references to non existing parts of the program memory, which are discovered to originate from either the original

x86 program or are related to Rosetta 2's just in time translation process [Nak21b].

Rosetta 2 runtime The Rosetta 2 binary is, unlike the Windows 10 emulation program, not a library but executed instead of the x86 program. This Rosetta 2 runtime performs the address resolving necessary for the **AOT** files, performs just in time translation and some other features.
It is discovered that Rosetta 2 uses lazy binding for some functions in the **AOT** file. Further the inner workings of Rosetta 2's just in time translate is uncovered which dynamically can translate x86 code to ARM. Additionally a bunch of debugging features are discovered, which can among other things were use by Apple to trace translated x86 code [Nak21b].

AOT shared cache file The file type seems to be not known publicly and is therefore analyzed further by deciphering a function named **load_aot_shared_cache** within the Rosetta 2 runtime.
In the end the **AOT** shared cache file contains code fragments of x86 code and its translated ARM counterpart alongside with additional information like branch data [Nak21b].

7.2 Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures

This paper aims to improve upon QEMU by implementing "correct and efficient emulation for weak memory model architectures" [Gou+23]. This is done two ways:

- Formally verified memory mapping scheme for strong-on-weak memory consistency [Gou+23]
- "cross-architecture dynamic linking of native shared libraries" [Gou+23]

QEMU's memory mapping scheme While QEMU inserts fences at the corrects locations to mimic the x86 memory model it has two flaws.
First they do not recreate the x86 memory order perfectly and disallow some reorders allowed on x86, while not disallowing reorders of some atomic memory accesses. The disadvantage of enforcing a stronger memory model than necessary is in the performance loss because the CPU is not allowed to maximize utilization by reordering some

accesses.
Secondly QEMU made a shortcut for Read Modify Write calls by using built-in compiler functions designed for this kind of atomic access. Further the paper shows that these atomic access functions can change with compiler versions resulting in different instructions, while the functions comply with the memory model it may lead to inconsistencies when mimicking the x86 memory model.
Additionally it is possible for ARM processors to for example speculatively execute atomic access instructions before other write instructions causing behaviour that is impossible on x86 [Gou+23].

Fixing incorrect QEMU mappings In order to fix these incorrect mappings they decide to strengthen the formal memory model slightly. Also atomic accesses are mapped to the expect accesses on the host system and they try to only use the minimal amount of fences needed to correctly order memory accesses.
They also discover that QEMU can sometimes incorrectly optimize code which is cause by some fence instructions and they fix it by avoiding the generation of these offending fence instructions [Gou+23].

Shared library translation QEMU requires all shared libraries to be present in the guest ISA to then sequentially translate them to the host ISA when emulating a program that requires these libraries. This is still the case when these libraries are installed natively for the host ISA causing a potential loss in performance since these native libraries are in general faster than translated code.
Since shared libraries provide a clear API this is the perfect spot to transform calls done by the emulated program to calls that work with the host ISA while correctly converting arguments and return values between guest and host ISAs.
Naturally this comes with a few challenges.

1. Capturing shared library calls
2. Function signatures are needed
3. Calling conventions may differ between guest and host ISA

To solve the first challenge they decided to exploit the dynamic linking part for ELF binaries that instead of calling the dynamic linker are now rerouted to the Risotto emulator which can then call the native library function.
For the next problem they settled for a Interface Definition Language (IDF) that contains all needed information about the function signatures. To avoid having to check at runtime if a given library function is available this is done when Risotto loads an ELF

binary by comparing the signatures in the IDF with the functions the program imports. The last challenge is solved by mapping the guest calling convention to the host calling convention whenever and transform the arguments and return values accordingly. [Gou+23]

7.3 Project Chameleon

This work focuses on the successor of Microsoft's x86 32 bit emulation engine for ARM. In particular the successor also allows for 64-bit binaries. Like the "Project Champollion" this work also has the goal to uncover unknown vulnerabilities in emulation.
Also this work highlights the different choices Apple made with Rosetta 2 by showing a different and interesting approach [Nak21a].

CHPEV2 ARM64X file format The CHPE file format is Microsoft's solution to compatibility with x86 emulation processes and native ARM processes. The new CHPEV2 version allows for a mismatch between architecture of the process loading a DLL and the DLL machine type. This file type was called "Chameleon binaries" by a Microsoft developer for reasons seen in the dynamic value relocation part [Jus].

Within these CHPE files is mixed x86 32-bit and ARM code. While the ARM code makes up the majority the function prologues are kept in the x86 format supposedly to maintain backwards compatibility with some applications that mess with function prologues.
The CHPEV2 format allows for x86 64-bit and ARM 64-bit code, while also allowing for 32-bit code. However only 32-bit or 64-bit code can be present in a single CHPEV2 file. CHPEV2 has two different sub-types ARM64EC, which has the x86 64-bit machine type and ARM64X, that has the ARM 64-bit machine type.
The suffix "EC" in ARM64EC stands for "Emulation Compatible" and is **only** used by 64-bit emulation processes.
In contrast the ARM64X can be used for both emulation and native processes. It is discovered that the CHPEV2 ARM64X type exports different function addresses based on the architecture of the process that is using it [Nak21a].

Dynamic Value Relocation Table (DVRT) The CHPEV2 ARM64X file format contains a Dynamic Value Relocation Table (DVRT) which is applied at runtime by the kernel depending on what architecture is needed.
This table contains all the information needed to transform the file to the x86 64-bit architecture at runtime without the need to translate code [Nak21a]. Only three

replacement operations are supported by the table:

1. Zero fill
2. Assign value
3. Add/Sub delta

these operations are enough to move addresses of functions to different parts of the file and even change some header values like the machine type from ARM 64-bit to x86 64-bit [Nak21a].

8 Summary and Conclusion

This thesis shows its possible to take advantage of Apple's implementation of the x86 memory model in hardware [App20b]. This was done by selecting a way to enable TSO which most closely resembles Rosetta 2's approach to avoid any unknown side effects. With the help of the TSOEnabler kernel module by Saagar Jha and their TSO test program included with the module it was possible to enable TSO mode on any thread. By adapting QEMU to detect the availability of the TSOEnabler kernel module at runtime and the subsequent deactivation of memory barrier generation for the ARM 64-bit architecture, the only thing left for full support of Apple's TSO implementation was to find the right place and time in the TCG code to call the system call added by the TSOEnabler kernel module. Ultimately it was decided to add a hook, which gets called when a thread initializes translation with TCG.

While the verification of the TSOEnabler kernel module worked flawlessly, the different QEMU binaries did not cooperate that easily. Repeated performance benchmarks showed different results and the theoretically best performing QEMU build without any memory fences had sometimes the same performance as the unchanged QEMU build with memory fences. This heavily suggested other problems with thermal throttling or the M1 chip hitting its power cap for example. However collecting this CPU information with another run showed similar CPU usage, frequency and power draw across the board while the benchmark times hardly differed from the baseline QEMU. Also a comparison of the translated instruction by the different QEMU builds only showed that the problem must lie elsewhere. The source code for the QEMU fork is available on github: `https://github.com/fdevx/qemu` [Fal22b].

9 Future Work

The performance test results in section 6.3 are strange and the source of the time gain or the reason for the time losses for some benchmark suite executions is still a big question. The fact that the translated code only slightly differs between runtime and compile time version and yet these can have vastly different timing results suggests that the problem does not lie in the implementation but in other factors. In a follow-up study of this implementation it should be possible to narrow down the cause trough for example the use of a CPU profiler.

Another point of interest is a comparison between Apple's Rosetta 2 and QEMU. While QEMU does not yet support UME on macOS, with the help of the Risotto paper it should be feasible to map system calls to the macOS architecture[QEMc; Gou+23].

An additional point of improvement for QEMU is that it uses dynamic translation only and with the help of additional static translation like in Rosetta 2 further performance optimizations and better transformed code should be possible. This may also save QEMU a fair bit of runtime work that can be offloaded to happen in the static translation part [QEMe; Nak21b].

List of Figures

List of Tables

Bibliography

[Alg+21] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget. "Armed Cats: Formal Concurrency Modelling at Arm." In: *ACM Transactions on Programming Languages and Systems* 43 (2 2021). ISSN: 15584593. DOI: `10.1145/3458926`.

[App] Apple. *sysctlbyname*. URL: `https://developer.apple.com/documentation/kernel/1387446-sysctlbyname`.

[App20a] Apple. "Apple unleashes M1." In: (Nov. 2020). URL: `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`.

[App20b] Apple. "WWDC2020 Keynote (at 1:39:25)." In: (June 2020).

[AWS] A. AWS. *AWS Graviton Processor*. URL: `https://aws.amazon.com/ec2/graviton/`.

[Bie11] C. Bienia. "Benchmarking Modern Multiprocessors." Princeton University, Jan. 2011.

[die] die.net. *Search for sysctlbyname in linux manpages*. URL: `https://www.die.net/search/?q=sysctlbyname`.

[Fal22a] M. Faltus. *Custom QEMU build switches*. 2022. URL: `https://github.com/fdevx/qemu/commit/2d6c2849c775283101e87c8b141728e2fd13c0be`.

[Fal22b] M. Faltus. *QEMU + TSO on Mac M1*. 2022. URL: `https://github.com/fdevx/qemu`.

[Fre] FreeBSD. *sysctl, sysctlbyname, sysctlnametomib*. URL: `https://www.freebsd.org/cgi/man.cgi?sysctl(3)`.

[Gou+23] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia. "Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures." In: *ASPLOS* (2023). URL: `https://redha.gouicem.fr/publication/asplos23/`.

[Har22] S. Harding. *Lenovo announces the first Arm-based ThinkPad*. Feb. 2022. URL: `https://arstechnica.com/gadgets/2022/02/lenovo-announces-the-first-arm-based-thinkpad/`.

[Jha] S. Jha. *TSOEnabler*. URL: `https://github.com/saagarjha/TSOEnabler`.

[Jha20] S. Jha. *How to find TSO Offset in the kernel?* 2020. URL: `https://github.com/saagarjha/TSOEnabler/issues/1`.

[Jha22] S. Jha. *@saagarjha @peterzheng98 Apple has released [xnu source code from macOS 11.0.1](https://opensource.apple.com/source/xnu/xnu-7195.50.7.100.1/) and there's this bit in ACTLR_EL1 to enable TSO*. 2022. URL: `https://github.com/saagarjha/TSOEnabler/issues/8#issuecomment-1210247259`.

[Jus] P. Justo. *Chameleon binaries*. URL: `https://mobile.twitter.com/itanium_guy/status/1461465563176194051`.

[Lam79] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." In: *IEEE Transactions on Computers* C-28 (9 1979). ISSN: 00189340. DOI: `10.1109/TC.1979.1675439`.

[lee22] leeehui. *tso-enabler*. 2022. URL: `https://github.com/leeehui/tso-enabler`.

[Lim22] A. Limited. "Arm Architecture Reference Manual for A-profile architecture (Section B2.1)." In: *Arm* (2022).

[Lin] A. Linux. *Asahi Linux*. URL: `https://asahilinux.org/`.

[LZC20] N. Liu, B. Zang, and H. Chen. "No barrier in the road: A comprehensive study and optimization of arm barriers." In: 2020. DOI: `10.1145/3332466.3374535`.

[Mica] Microsoft. *How x86 emulation works on Arm*. URL: `https://docs.microsoft.com/en-gb/windows/arm/apps-on-arm-x86-emulation`.

[Micb] Microsoft. *Windows on Arm*. URL: `https://docs.microsoft.com/en-us/windows/arm/overview`.

[Nak21a] K. M. Nakagawa. *Project Chameleon*. 2021. URL: `https://github.com/FFRI/ProjectChameleon`.

[Nak21b] K. M. Nakagawa. *Project Champollion: Reverse engineering Rosetta 2*. 2021. URL: `https://github.com/FFRI/ProjectChampollion`.

[QEMa] QEMU. *QEMU A generic and open source machine emulator and virtualizer*. URL: `https://www.qemu.org/`.

[QEMb] QEMU. *QEMU Source Code*. URL: `https://github.com/qemu/qemu`.

[QEMc] QEMU. *QEMU User space emulator*. URL: `https://www.qemu.org/docs/master/user/main.html`.

[QEMd] QEMU. *System emulation*. URL: `https://www.qemu.org/docs/master/system/index.html`.

[QEMe] QEMU. *Translation Internals*. URL: https://www.qemu.org/docs/master/devel/tcg.html.

[Sha21] A. Shah. "We're closing the gap with Arm and x86, claims SiFive: New RISC-V CPU core for PCs, servers, mobile incoming." In: *the register* (Oct. 2021). URL: https://www.theregister.com/2021/10/21/sifive_riscv_cpu/.

[sta21] A. B. staff. *Ten Products and Trends from the Arm Ecosystem in 2021*. 2021. URL: https://www.arm.com/blogs/blueprint/arm-ecosystem-trends-2021.

[wika] Q. wiki. *Documentation/TCG*. URL: https://wiki.qemu.org/Documentation/TCG.

[wikb] Q. wiki. *Features/tcg-multithread*. URL: https://www.qemu.org/docs/master/devel/multi-thread-tcg.html.