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

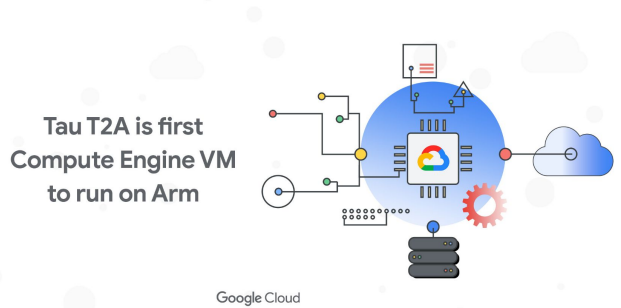
Marcel Faltus
Advisor: Dr. Redha Gouicem
Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

15.05.2022 – 15.09.2022

Motivation: Rise of ARM

Rise of ARM

- AWS Graviton
- Tau T2A
- Chromebooks
- Mac M1



Motivation: Research context

Rise of ARM

- AWS Graviton
- Tau T2A
- Chromebooks
- Mac M1

Incompatibility with existing x86 Applications

Solution:

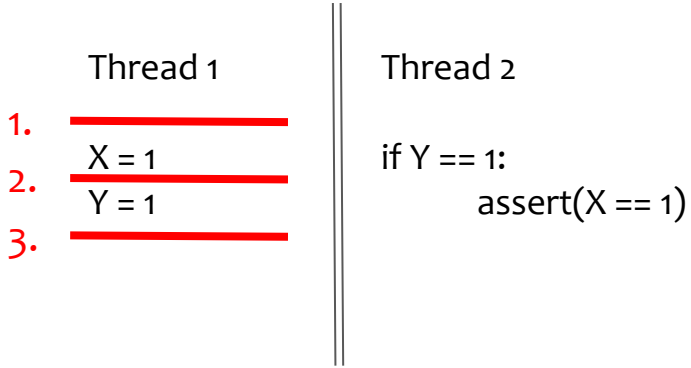
- Emulation of x86-Programs on ARM

Background-topic

Problems with x86 on ARM

x86	1.	“if” fails
	2.	“if” fails
	3.	“assert” successful

X = Y = 0



Background-topic

Problems with x86 on ARM

X = Y = 0

	Thread 1	Thread 2
4.	X = 1 Y = 1	if Y == 1: assert(X == 1)

x86	1.	“if” fails
	2.	“if” fails
	3.	“assert” successful
ARM	4.	“assert” fails

x86’s memory order is shortened to Total Store Order (TSO) from now on

ARM introduces new behaviours

Research gap

- QEMU x86 on ARM+macOS:
  - not fully supported
  - incorrect memory fences[1]
  - no user mode emulation

- Rosetta 2
  - no full system emulation
  - in hardware memory fences (TSO)

Are hardware fences faster than software fences?

[1] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, Pramod Bhatotia. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. *ASPLOS* 2023.

Design

How can we test this?

Design: How can we test this?

Comparison between software fences and hardware TSO needed

2 Choices:

1. Rosetta 2 (impossible)
2. QEMU

Add support for hardware TSO in QEMU

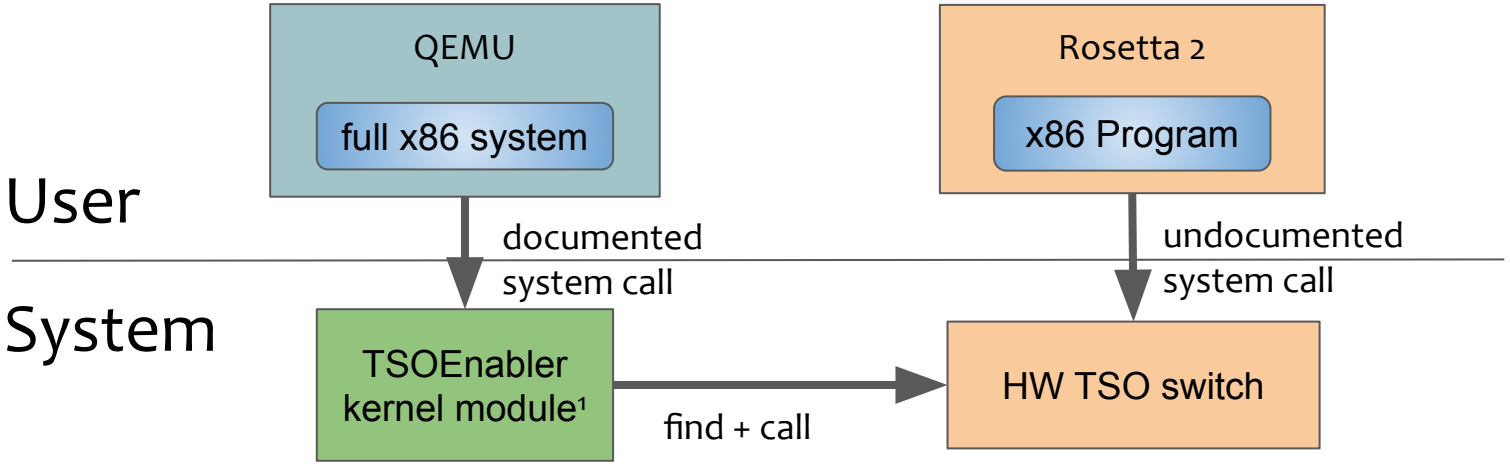
Implementation

How to enable hardware TSO for QEMU?

Implementation: Enabling hardware TSO for QEMU

1. Find a way to enable hardware TSO for QEMU
2. Disable memory fences in QEMU

Implementation: Enabling hardware TSO



TSOEnabler works per thread.
⇒ Every vCPU thread must enable hardware TSO

[1] TSOEnabler by Saagar Jha: https://github.com/saagarjha/TSOEnabler

Implementation: Disabling memory fences

Disable memory fence generation

```
#define TCG_TARGET_DEFAULT_MO  0
```

`(TCG_MO_ALL & ~TCG_MO_ST_LD)`

constant

can be used for optimizations
at compile time

`target_memory_ordering`

variable

can be toggled at runtime
(e.g. program argument)

Why not test both?

Implementation: Generated instructions comparison

HW Fences

```
---- 00000000000fcf84 0000000000000016
mov_i64 tmp2,$0x61b8
ext16u_i64 tmp2,tmp2
add_i64 tmp2,tmp2,cs_base
ext32u_i64 tmp2,tmp2
qemu_ld_i64 tmp1,tmp2,leuw,2
add_i64 tmp2,tmp2,$0x2
ext32u_i64 tmp2,tmp2
qemu_ld_i64 tmp0,tmp2,leul,2
and_i64 tmp0,tmp0,$0xffffff
st_i64 tmp0,env,$0x198
st32_i64 tmp1,env,$0x1a0
```

```
set_label $L0
exit_tb $0x280005e83
```

SW Fences

```
---- 00000000000fcf84 0000000000000016
mov_i64 tmp2,$0x61b8
ext16u_i64 tmp2,tmp2
add_i64 tmp2,tmp2,cs_base
ext32u_i64 tmp2,tmp2
mb $0x31
qemu_ld_i64 tmp1,tmp2,leuw,2
add_i64 tmp2,tmp2,$0x2
ext32u_i64 tmp2,tmp2
mb $0x31
qemu_ld_i64 tmp0,tmp2,leul,2
and_i64 tmp0,tmp0,$0xffffff
st_i64 tmp0,env,$0x198
st32_i64 tmp1,env,$0x1a0
```

```
set_label $L0
exit_tb $0x280005f03
```

Evaluation

Does the TSOEnabler work as expected?

- Is TSO enforced on ARM?
- Sanity check with micro benchmark

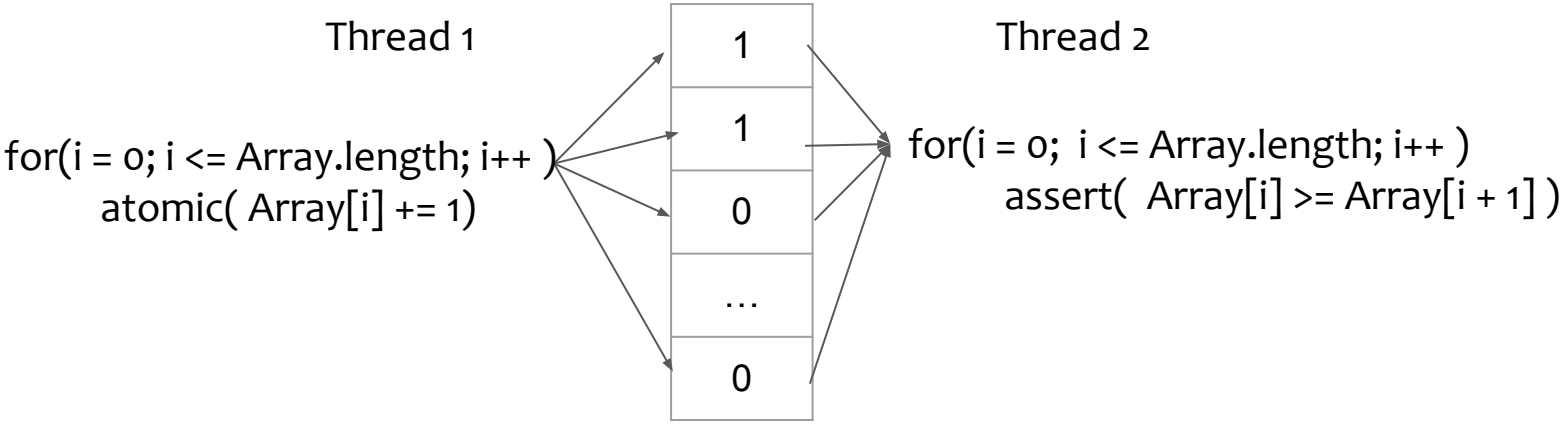
How big is the performance impact of HW TSO?

- Comparison of software vs hardware fences
- PARSEC Benchmark Suite[1]

[1] PARSEC Benchamrk Suite: https://parsec.cs.princeton.edu/

Evaluation: Does TSOEnabler work as expected?

TSO test program

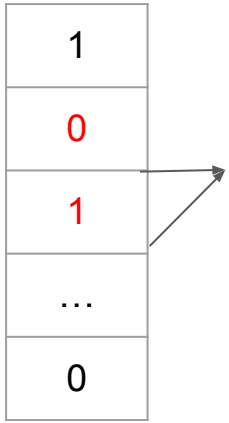


Evaluation: Does TSOEnabler work as expected?

TSO test program

Thread 1

```
for(i = 0; i <= Array.length; i++ )
    atomic( Array[i] += 1)
```



Thread 2

```
for(i = 0; i <= Array.length; i++ )
    assert(  Array[i] >= Array[i + 1] )
```

HALT

Count iterations over the whole array

Evaluation: Does TSOEnabler work as expected?

Experimental setup:

- Mac M1 8 Core CPU (3.2 GHz, 4 Performance, 4 Efficiency)
- AMD Ryzen 7 3800X 8 Core CPU (4.5 GHz, 16 Threads)

TSO test program until crash

	Max iterations
ARM	≤1,600
ARM + HW TSO	>10,000,000 - no reorder
x86 AMD	>80,000,000 - no reorder

repeated ~17k times

Stopped after 15 mins

TSO Enabler works ✔

Evaluation: Performance impact of hardware TSO

Experimental setup:

- Mac M1 8 Core CPU (3.2 GHz, 4 Performance, 4 Efficiency), 16 GB RAM
- PARSEC Benchmark Suite
- Arch Linux guest

List of builds

- SW Fences (Standard QEMU)
- HW Fences (usage of TSO)
- No Fences (incorrect)

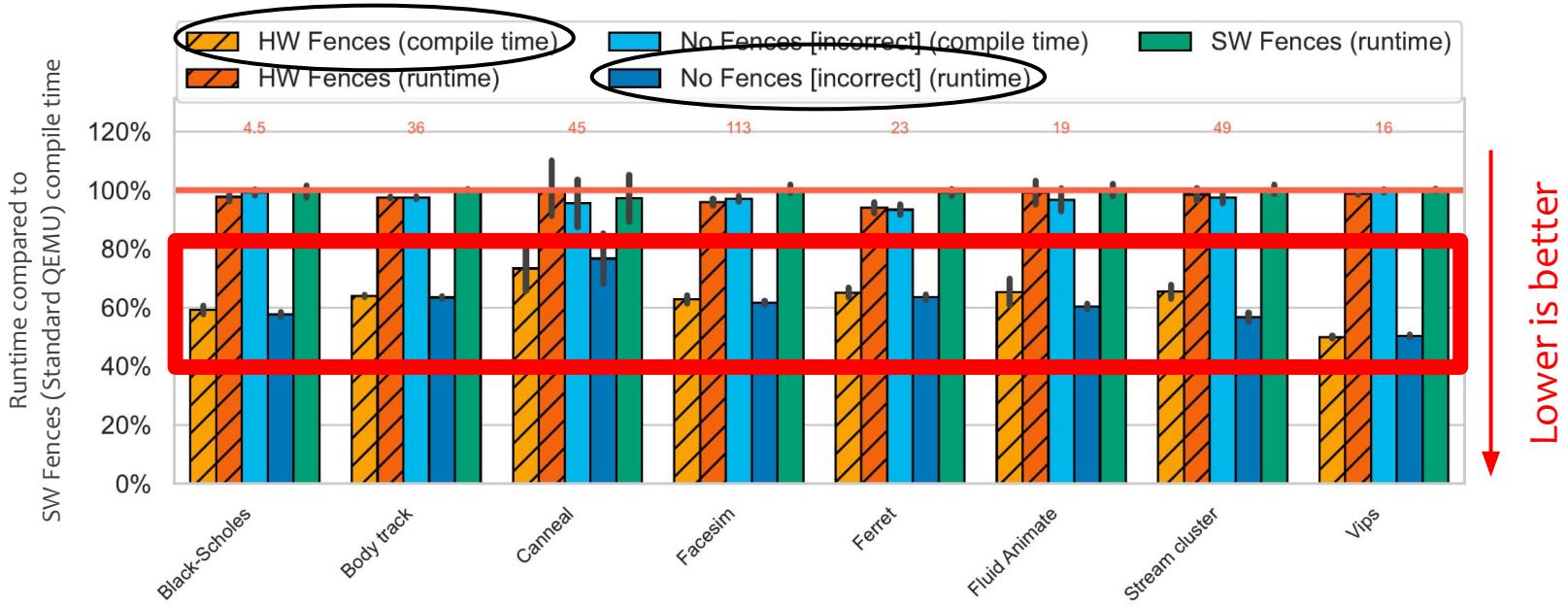
2 different version for each build:

- "compile time"
  QEMU build time
- "run time"
  QEMU run time

Baseline: SW Fences (Standard QEMU) compile time

Evaluation: Performance impact of hardware TSO

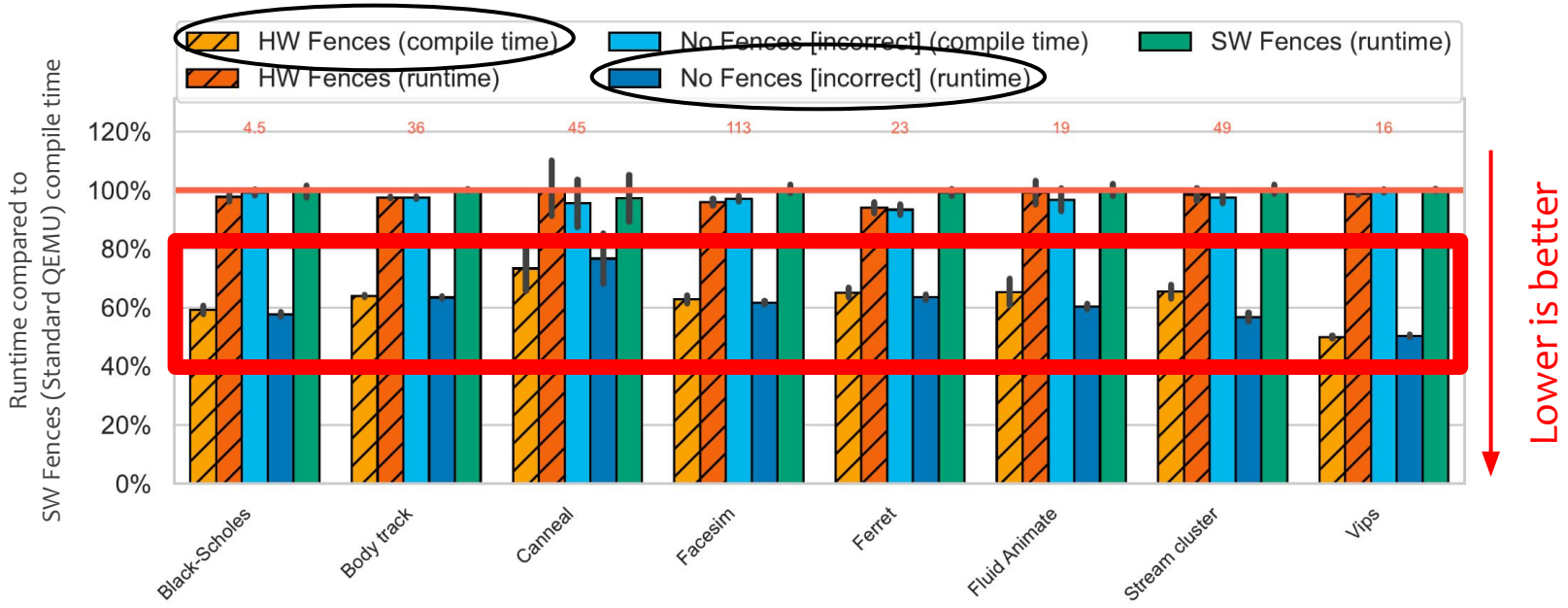
QEMU resources: 4 CPUs; 1 GB RAM



HW Fences ≃ No Fences ⇒ HW TSO is cheap!

Evaluation: Performance impact of hardware TSO

QEMU resources: 4 CPUs; 1 GB RAM



Runtime variants ≠ Compile time variants?

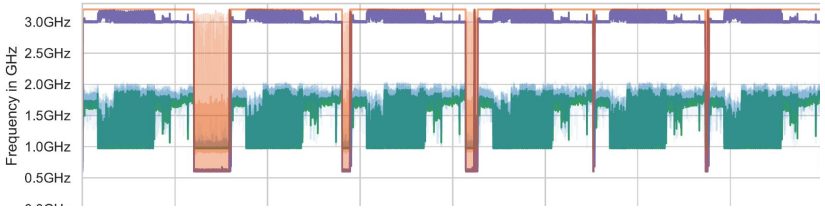
Investigation

- QEMU argument differences? No
- Benchmarking differences? No
- QEMU binaries swapped? Sadly not
- Thermal throttling? Nope
- Not using Performance Cores? Also no
- Background tasks? No
- CPU Feature (e.g. caching)? No

```
root@archlinux:/test/ ✔ > ./run_benchmarks.sh
Please enter a prefix for the results: No Fences
Benchmark iterations [20]:
Benchmark threads [4]:
Dataset [simlarge]:
```

```
Fence type: none, const: no
TSO support: no
```

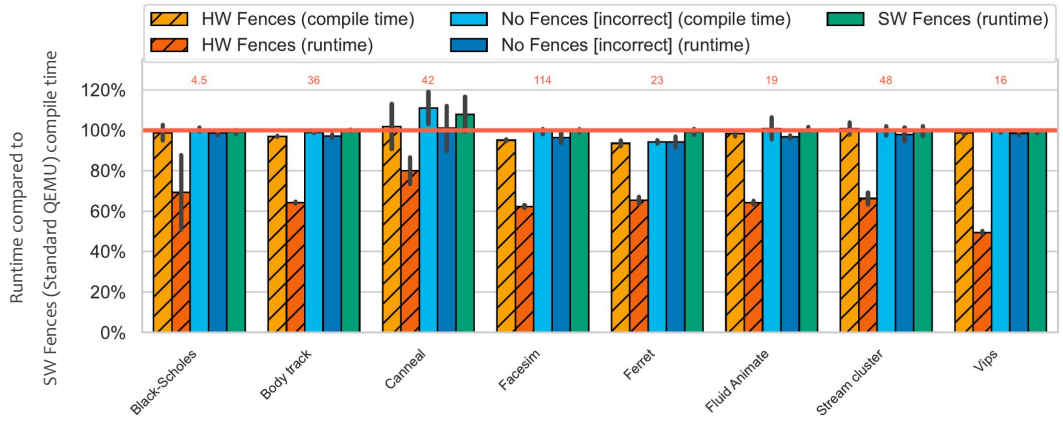
```
Fence type: hardware, const: yes
TSO support: yes
Additional Info(m/v):  d/d (x86: d)     Called tso_enable on thread 1272e9b1 with result 0
Called tso_enable on thread 118b29b1 with result 0
Called tso_enable on thread 118069b1 with result 0
Called tso_enable on thread 1198a9b1 with result 0
```



Ongoing Investigations

Impact between runs?

- 1 hour wait between runs
- one time 2 hour wait before a run



Ongoing Investigations

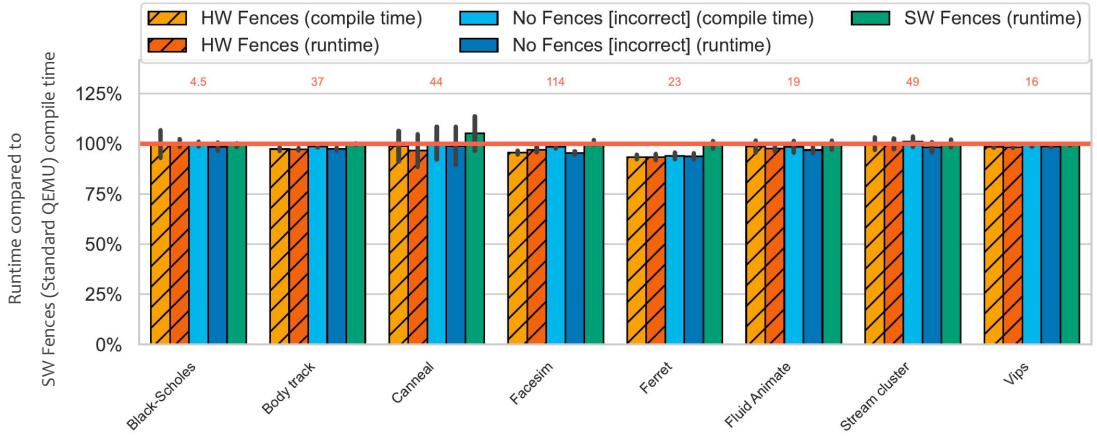
Impact between runs?

- 1 hour wait between runs
- one time 2 hour wait before a run

- run with cpu monitoring

Other options:

- Performance Profiler
- Different Mac M1



Summary

QEMU SW fences slow + incorrect …..

- Complex implementation
- performance on the slow end

QEMU + HW TSO:

- CPU responsible for memory order
- Simple implementation

BUT:

Performance benefit is unknown!

Try it out!

https://github.com/fdevx/qemu

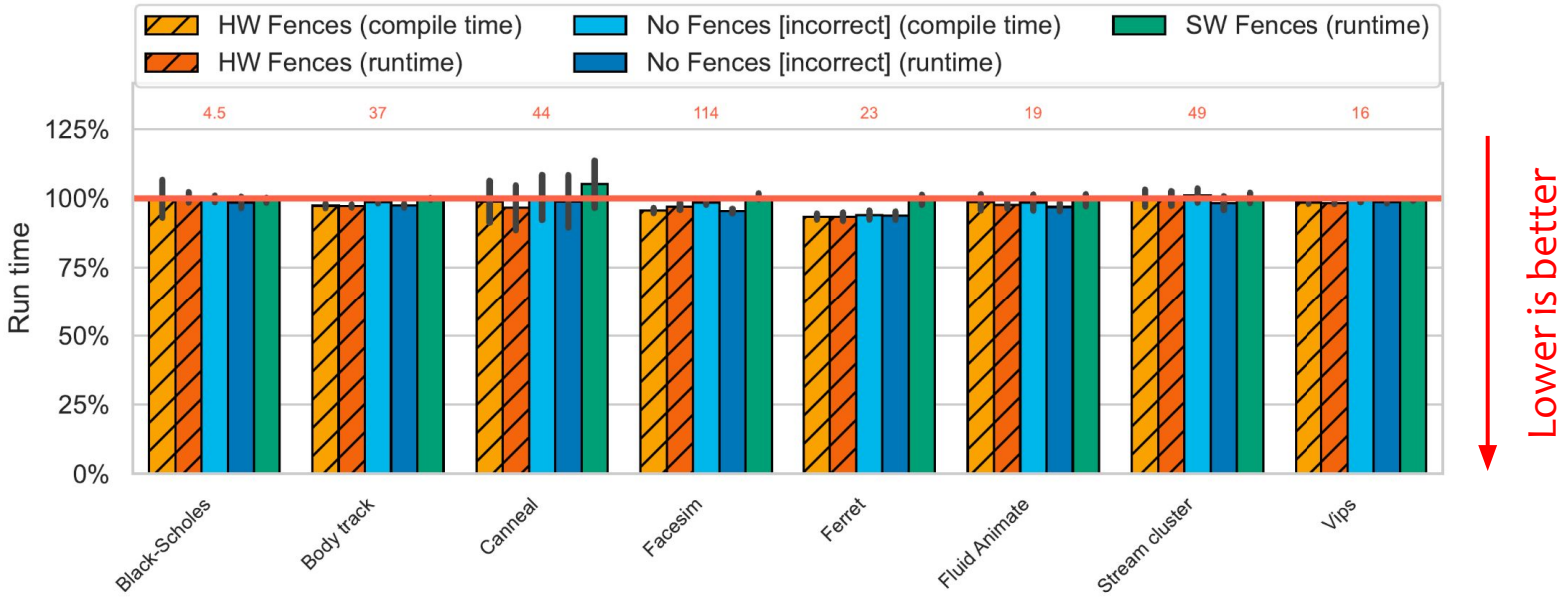
Backup

Evaluation: Performance impact of hardware TSO

QEMU resources:
- 4 vCPU threads
- 4 GB RAM

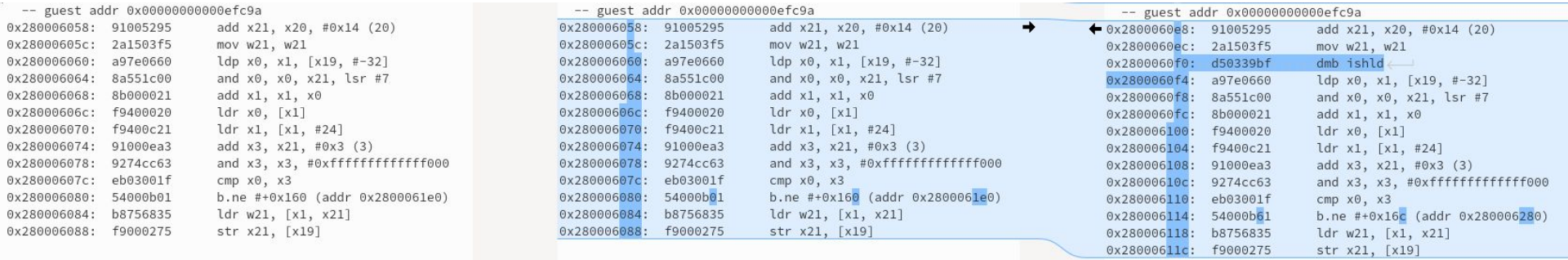
PARSEC preset: “simlarge”

run times of the PARSEC benchmark suite



Everything is slow?

Generated instructions comparison



```
 -- guest addr 0x00000000000efc9a
0x280006058:  91005295      add x21, x20, #0x14 (20)
0x28000605c:  2a1503f5      mov w21, w21
0x280006060:  a97e0660      ldp x0, x1, [x19, #-32]
0x280006064:  8a551c00      and x0, x0, x21, lsr #7
0x280006068:  8b000021      add x1, x1, x0
0x28000606c:  f9400020      ldr x0, [x1]
0x280006070:  f9400c21      ldr x1, [x1, #24]
0x280006074:  91000ea3      add x3, x21, #0x3 (3)
0x280006078:  9274cc63      and x3, x3, #0xfffffffffffff000
0x28000607c:  eb03001f      cmp x0, x3
0x280006080:  54000b01      b.ne #+0x160 (addr 0x2800061e0)
0x280006084:  b8756835      ldr w21, [x1, x21]
0x280006088:  f9000275      str x21, [x19]
```

```
 -- guest addr 0x00000000000efc9a
0x280006058:  91005295      add x21, x20, #0x14 (20)
0x28000605c:  2a1503f5      mov w21, w21
0x280006060:  a97e0660      ldp x0, x1, [x19, #-32]
0x280006064:  8a551c00      and x0, x0, x21, lsr #7
0x280006068:  8b000021      add x1, x1, x0
0x28000606c:  f9400020      ldr x0, [x1]
0x280006070:  f9400c21      ldr x1, [x1, #24]
0x280006074:  91000ea3      add x3, x21, #0x3 (3)
0x280006078:  9274cc63      and x3, x3, #0xfffffffffffff000
0x28000607c:  eb03001f      cmp x0, x3
0x280006080:  54000b01      b.ne #+0x160 (addr 0x2800061e0)
0x280006084:  b8756835      ldr w21, [x1, x21]
0x280006088:  f9000275      str x21, [x19]
```

```
 -- guest addr 0x00000000000efc9a
0x2800060e8:  91005295      add x21, x20, #0x14 (20)
0x2800060ec:  2a1503f5      mov w21, w21
0x2800060f0:  d50339bf      dmb ishld
0x2800060f4:  a97e0660      ldp x0, x1, [x19, #-32]
0x2800060f8:  8a551c00      and x0, x0, x21, lsr #7
0x2800060fc:  8b000021      add x1, x1, x0
0x280006100:  f9400020      ldr x0, [x1]
0x280006104:  f9400c21      ldr x1, [x1, #24]
0x280006108:  91000ea3      add x3, x21, #0x3 (3)
0x28000610c:  9274cc63      and x3, x3, #0xfffffffffffff000
0x280006110:  eb03001f      cmp x0, x3
0x280006114:  54000b61      b.ne #+0x16c (addr 0x280006280)
0x280006118:  b8756835      ldr w21, [x1, x21]
0x28000611c:  f9000275      str x21, [x19]
```