



Efficient code generation for weakly ordered architectures

Reinoud Elhorst, Mark Batty, David Chisnall
{re302,mjb220,dc552} @ cam.ac.uk



UNIVERSITY OF
CAMBRIDGE

Example: message passing

```
{flag, data, x}=0;
```

Thread0:

```
data=1;
```

```
flag=1;
```

Thread1:

```
while(!flag){}
```

```
x=data;
```

Example: message passing

```
{flag, data, x}=0;
```

Thread0:

```
data=1;
```

```
flag=1;
```

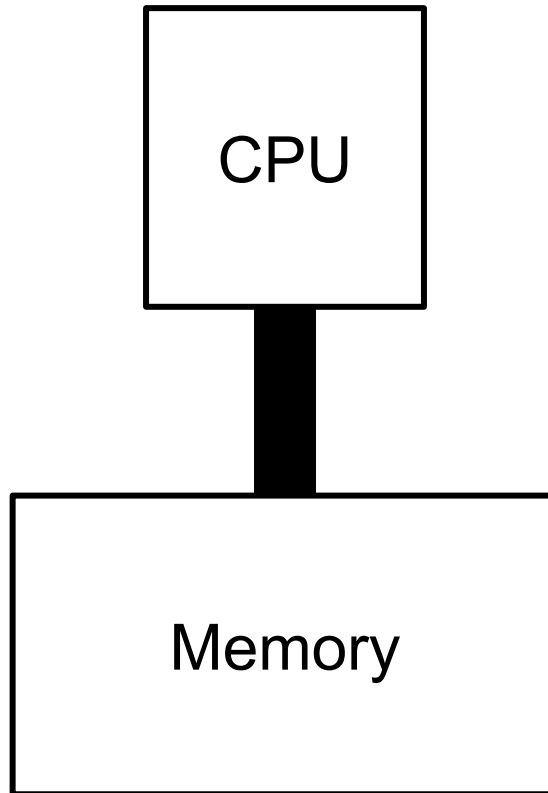
Thread1:

```
while(!flag){}
```

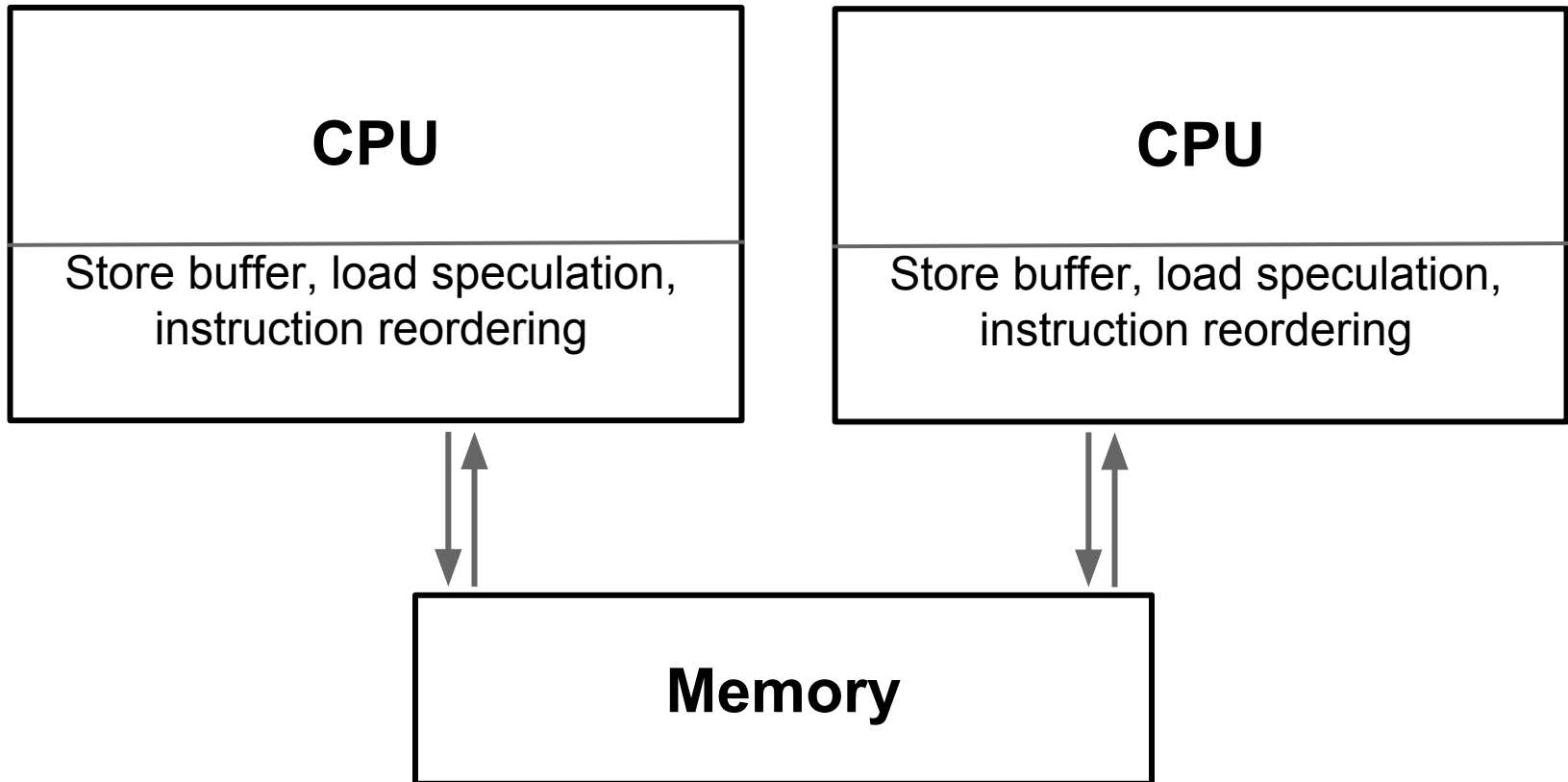
```
x=data;
```



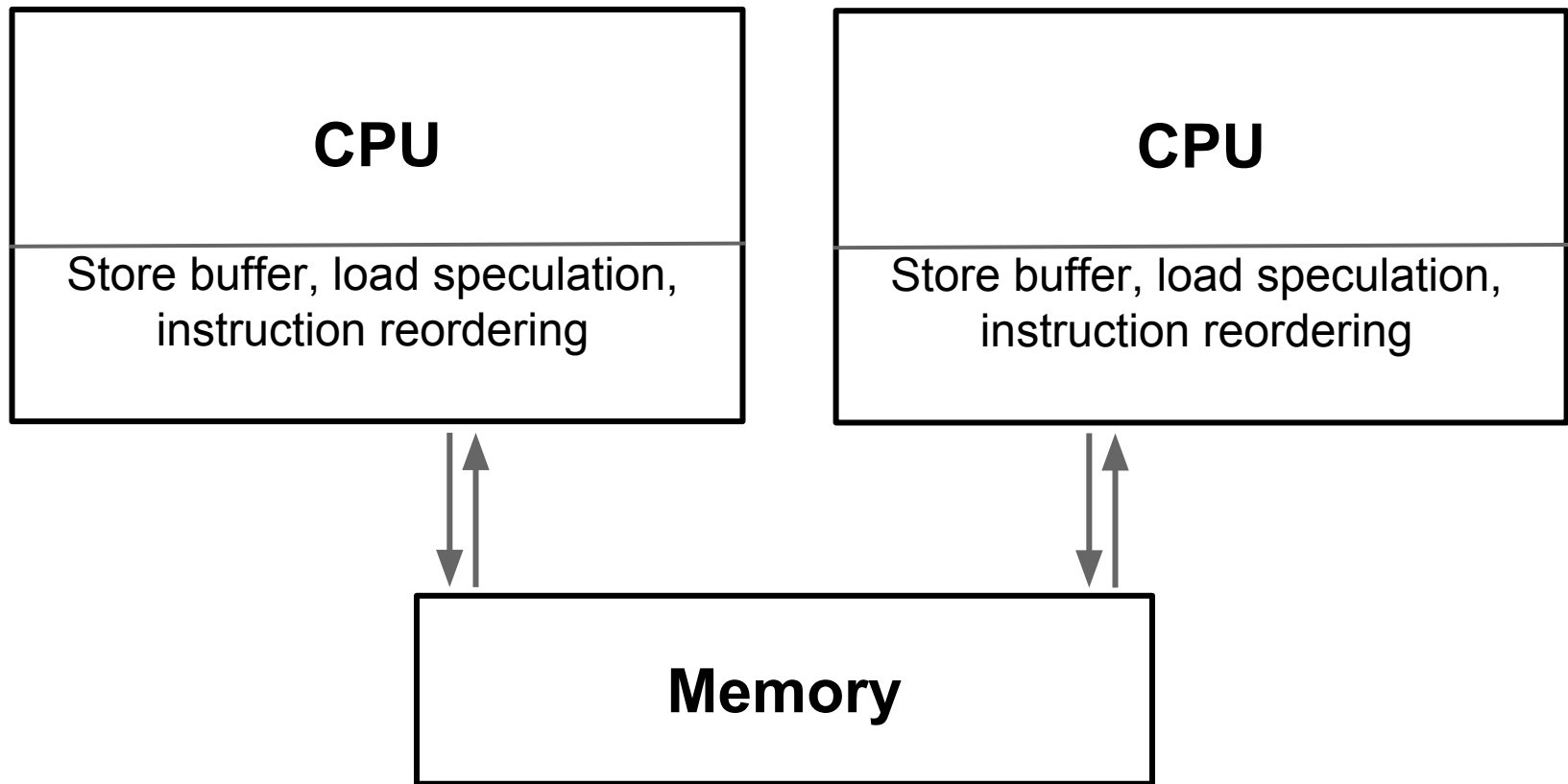
Programmer's mental model (C99)



Simplified modern architecture



Simplified modern architecture



Compiler instruction reordering

Synchronisation: Barriers and fences

Barriers give us the guarantees we want....

....however: performance

- DEC Alpha
 - Very weakly ordered, huge variety of barriers
- x86
 - Store buffer, compiler may reorder, **mfence**
- ARMv7
 - Relatively weak, **dmb**

Memory models: language support

- C89: inline asm (usually via macros)
- GCC 4.7 : `__sync_*` (portable)
- C11/C++11: `stdatomic.h` (performance)
 - clang 3.1: `__c11_atomic_*`
 - LLVM 3.1: IR C11 memory model inspired instructions
 - Atomic read/modify/write
 - Compare and exchange
 - Atomic qualifiers on load / store

C11 `_Atomic(int)` x variable access:

- Relaxed [`atomic_add(&x, 1, relaxed);`]
- Acquire/release [`atomic_load(x, acquire);`]
- Sequentially consistent [`..., seq_cst);`]
- Implicitly seq cst [`x+=3;`]

- Safe racy accesses
 T0: `x=5;` T1: `x=3;`
- Guaranteed atomic updates
 `x++;`
- Reasoning about them still hard

C11 → hardware

C11	X86	ARM
<code>load_relaxed</code>	<code>mov</code>	<code>ldr</code>
<code>store_release</code>	<code>mov</code>	<code>dmb; str</code>
<code>store_seq_cst</code>	<code>lock xchg</code>	<code>dmb; str; dmb</code>
<code>cas_strong</code>	<code>lock cmpxchg</code>	<code>loop(ldrex ... strex)</code>
<code>atomic_add</code>	<code>lock add</code>	<code>loop(ldrex; add ; strex)</code>

Example: seqlock

- Lockless data-structure
- Used in Linux, Xen, FreeBSD
- Guaranteed to read variables together that were written together
- Three identical implementations, except for memory order

Seqlock

```
_Atomic(int) x1, x2, lock;
```

```
void write(int v1, int v2) {  
    static int lock_1 = 0;  
    store(lock, ++lock_1);  
    store(x1, v1);  
    store(x2, v2);  
    store(lock, ++lock_1);  
}
```

```
void read(int *v1, int *v2) {  
    int lv1, lv2, lock_1;  
    while (true) {  
        lock_1 = load(lock);  
        if (lock_1 & 1) continue;  
        lv1 = load(x1);  
        lv2 = load(x2);  
        if (lock_1 == load(lock)) {  
            *v1 = lv1;  
            *v2 = lv2;  
            return;  
        }  
    }  
}
```

Seqlock

```
_Atomic(int) x1, x2, lock;
```

```
void write(int v1, int v2) {  
    static int lock_1 = 0;  
    store(lock, ++lock_1);  
    store(x1, v1);  
    store(x2, v2);  
    store(lock, ++lock_1);  
}
```

```
void read(int *v1, int *v2) {  
    int lv1, lv2, lock_1;  
    while (true) {  
        lock_1 = load(lock);  
        if (lock_1 & 1) continue;  
        lv1 = load(x1);  
        lv2 = load(x2);  
        if (lock_1 == load(lock)) {  
            *v1 = lv1;  
            *v2 = lv2;  
            return;  
        }  
    }  
}
```

Seqlock

```
_Atomic(int) x1, x2, lock;
```

```
void write(int v1, int v2) {  
    static int lock_1 = 0;  
    store(lock, ++lock_1);  
    store(x1, v1);  
    store(x2, v2);  
    store(lock, ++lock_1);  
}
```

```
void read(int *v1, int *v2) {  
    int lv1, lv2, lock_1;  
    while (true) {  
        lock_1 = load(lock);  
        if (lock_1 & 1) continue;  
        lv1 = load(x1);  
        lv2 = load(x2);  
        if (lock_1 == load(lock)) {  
            *v1 = lv1;  
            *v2 = lv2;  
            return;  
        }  
    }  
}
```

Seqlock

```
_Atomic(int) x1, x2, lock;
```

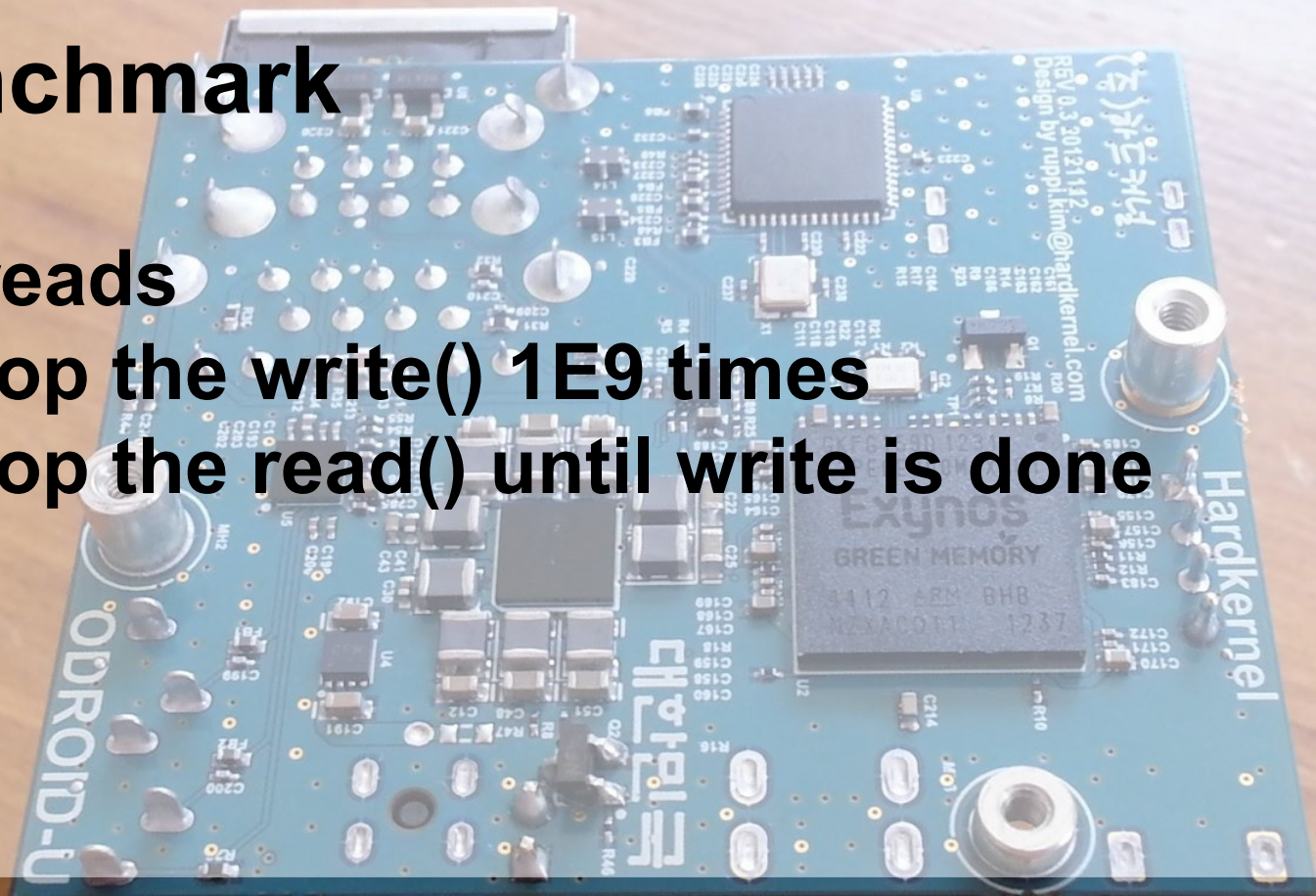
```
void write(int v1, int v2) {  
    static int lock_1 = 0;  
    store(lock, ++lock_1);  
    store(x1, v1);  
    store(x2, v2);  
    store(lock, ++lock_1);  
}
```

```
void read(int *v1, int *v2) {  
    int lv1, lv2, lock_1;  
    while (true) {  
        lock_1 = load(lock);  
        if (lock_1 & 1) continue;  
        lv1 = load(x1);  
        lv2 = load(x2);  
        if (lock_1 == load(lock)) {  
            *v1 = lv1;  
            *v2 = lv2;  
            return;  
        }  
    }  
}
```

Benchmark

2 threads

- loop the write() 1E9 times
- loop the read() until write is done



ODROID-U2

Exynos 1.7 GHz quad-core Cortex-A9 (ARMv7)

1MB Shared L2 cache

Implement with three memory orders

- **Implicit**

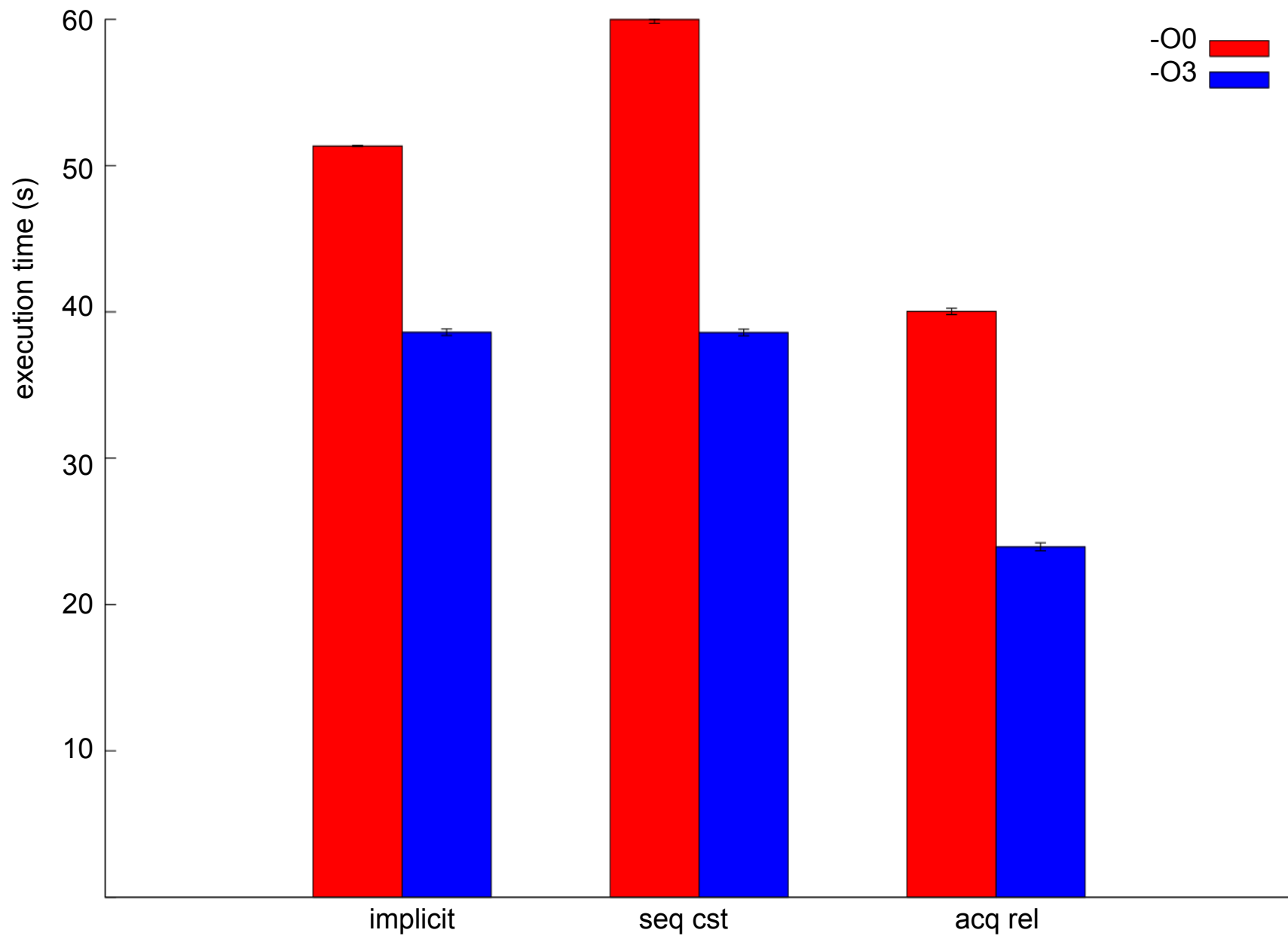
```
#define load(x) (x)
#define store(x,y) (x=y)
```

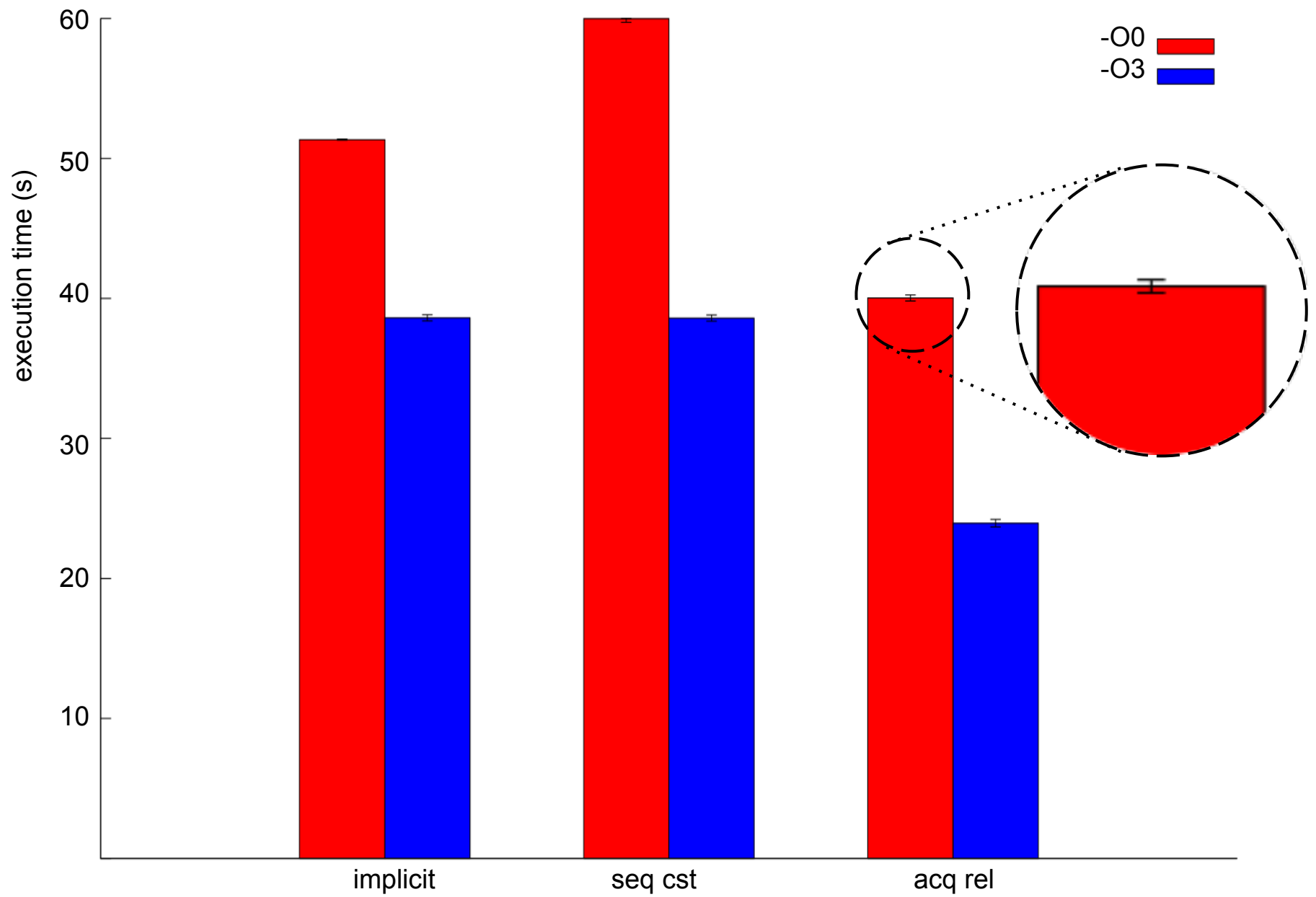
- **Sequential consistency**

```
#define load(x) atomic_load_explicit(&x, seq_cst);
#define store(x,y) atomic_store_explicit(&x, y, seq_cst);
```

- **Acquire / release**

```
#define load(x) atomic_load_explicit(&x, acquire);
#define store(x,y) atomic_store_explicit(&x, y, release);
```





ARM assembly

dmb ish	barrier
ldr / str	load / store
add / sub	ALU operators
cmp	compare
bne	branch not equal
<hr/>	
r12	register
[r1]	pointed to by register
#1	literal

C11 → hardware

C11	X86	ARM
load_relaxed	mov	ldr
store_release	mov	dmb; str
store_seq_cst	lock xchg	dmb; str; dmb
cas_strong	lock cmpxchg	loop(ldrex ... strex)
atomic_add	lock add	loop(ldrex; add ; strex)

Seq. cst.

```

.LBB0_1:
dmb ish
sub r4, r1, #1
str r4, [r2]
add r0, r0, #1
dmb ish
cmp r0, r3
dmb ish
str r0, [r12]
dmb ish
dmb ish
str r0, [1r]
dmb ish
dmb ish
str r1, [r2]
add r1, r1, #2
dmb ish
bne .LBB0_1

```

Acquire release

```

.LBB0_1:
dmb ish
sub r4, r1, #1
str r4, [r2]
add r0, r0, #1
dmb ish
str r0, [r12]
dmb ish
str r0, [1r]
dmb ish
str r1, [r2]
add r1, r1, #2
dmb ish
cmp r0, r3
bne .LBB0_1

```

```

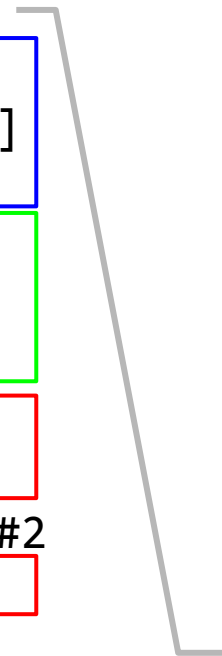
for (i=0; i<1E9; i++)
store(lock, ++1_lock);

store(x1, i);

store(x2, i);

store(lock, ++1_lock);

```

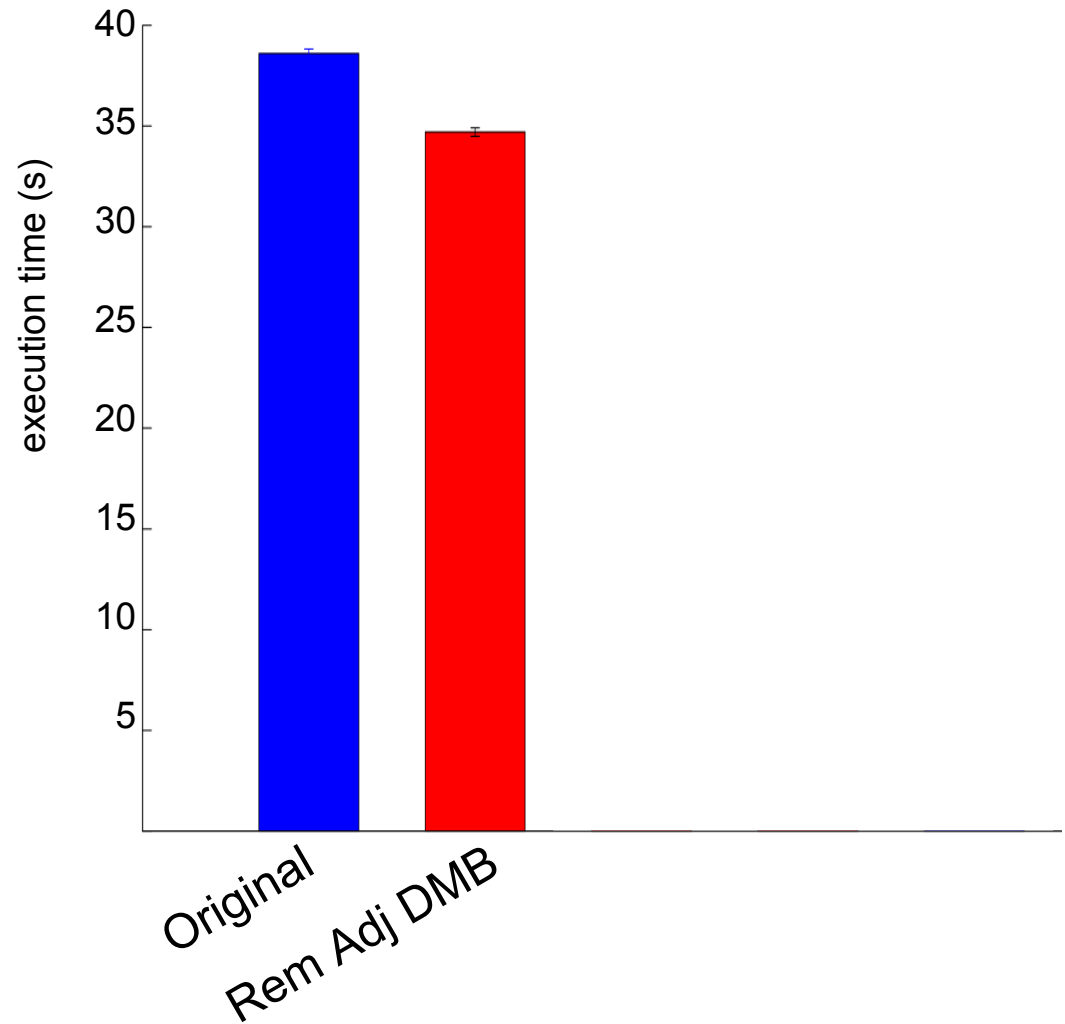


Optimise the seq cst version

- **dmb** is a memory barrier
- ARM memory model: two **dmbs** in a row does not increase synchronization

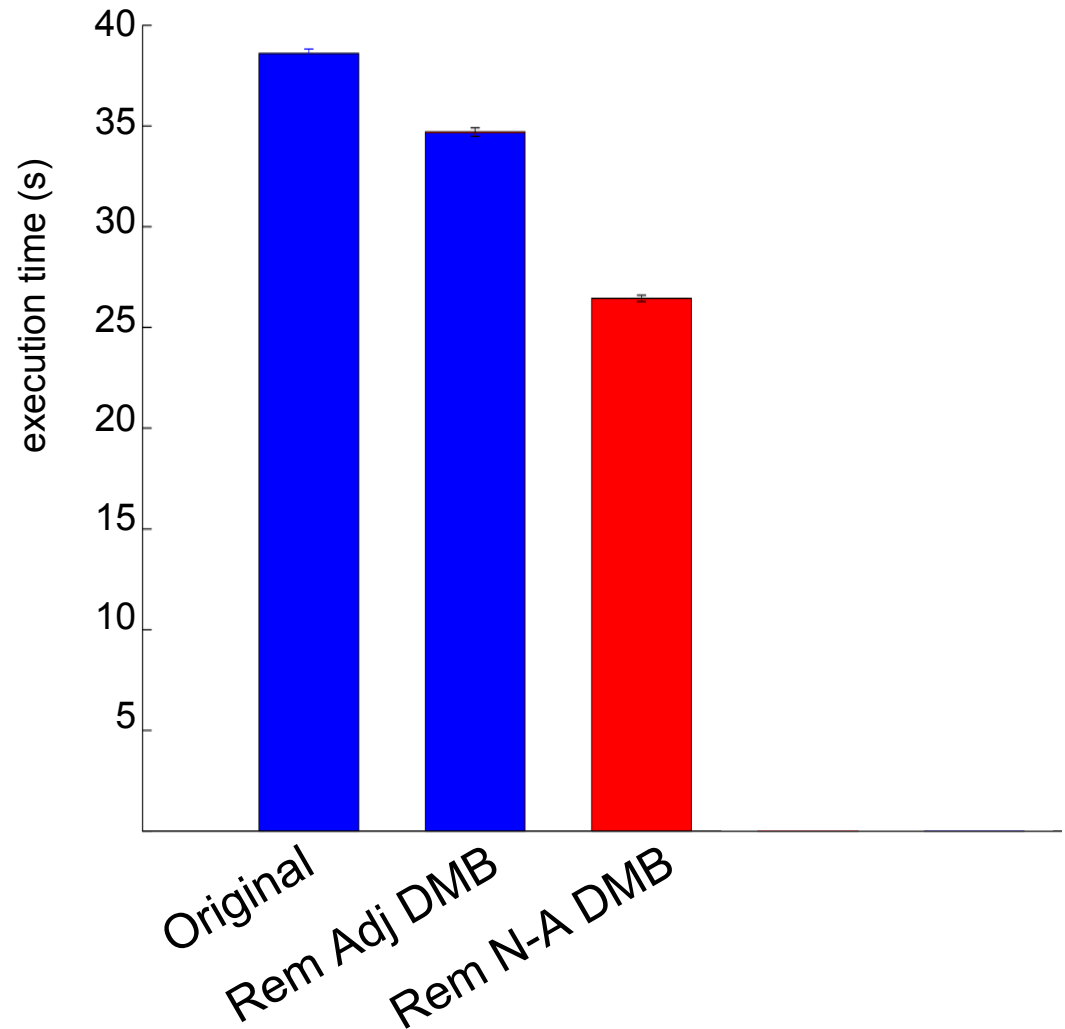
Pass 1: Remove adjacent barriers

```
.LBB0_1:  
  dmb      ish  
  sub      r4, r1, #1  
  str      r4, [r2]  
  add      r0, r0, #1  
  dmb      ish  
  cmp      r0, r3  
  dmb      ish  
  str      r0, [r12]  
  dmb      ish  
dmb      ish  
  str      r0, [1r]  
  dmb      ish  
dmb      ish  
  str      r1, [r2]  
  add      r1, r1, #2  
  dmb      ish  
  bne     .LBB0_1
```



Pass 1: Remove non-adjacent barriers

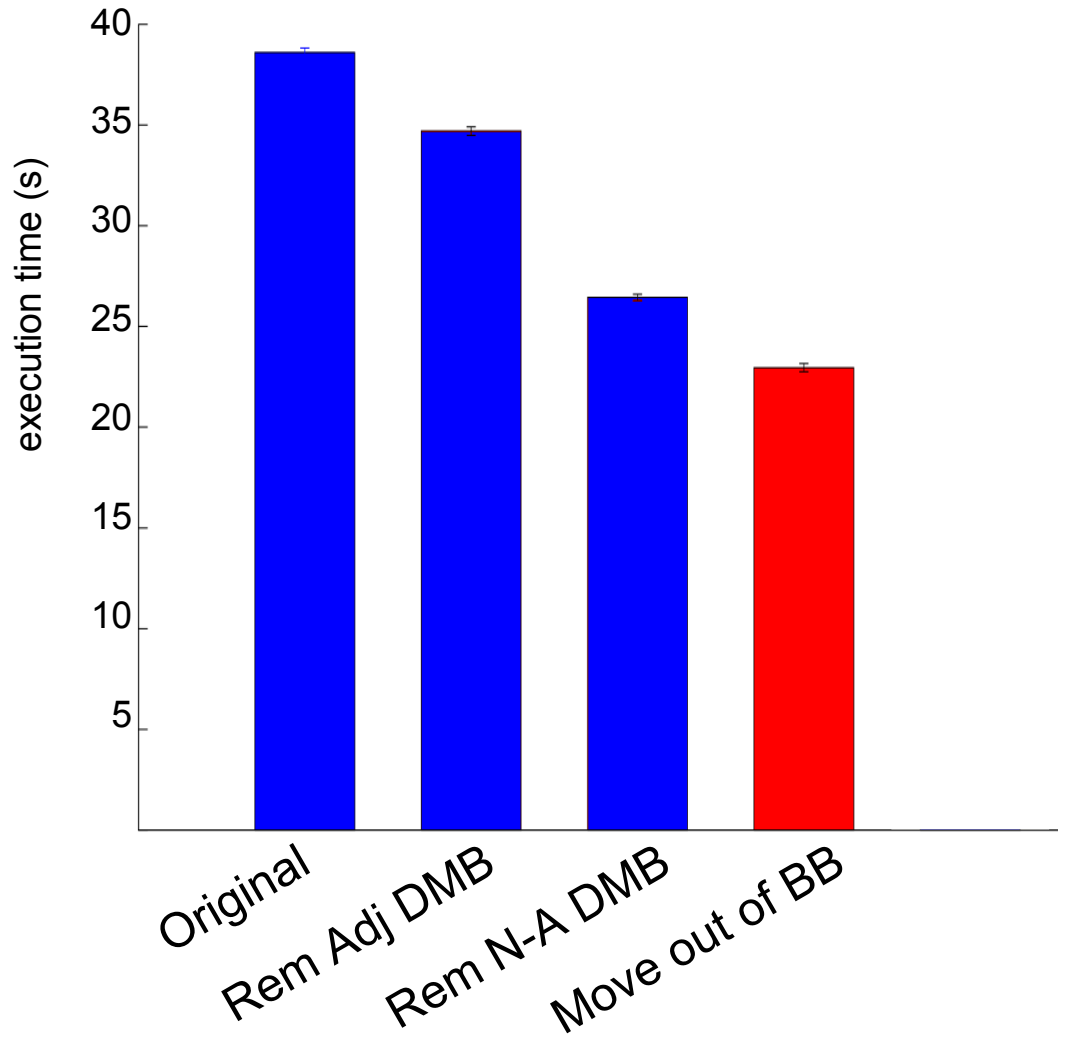
```
.LBB0_1:  
  dmb      ish  
  sub      r4, r1, #1  
  str      r4, [r2]  
  add      r0, r0, #1  
  dmb      ish  
  cmp      r0, r3  
  dmb   ish  
  str      r0, [r12]  
  dmb      ish  
  str   r0, [1r]  
  dmb      ish  
  str      r1, [r2]  
  add      r1, r1, #2  
  dmb      ish  
  bne     .LBB0_1
```



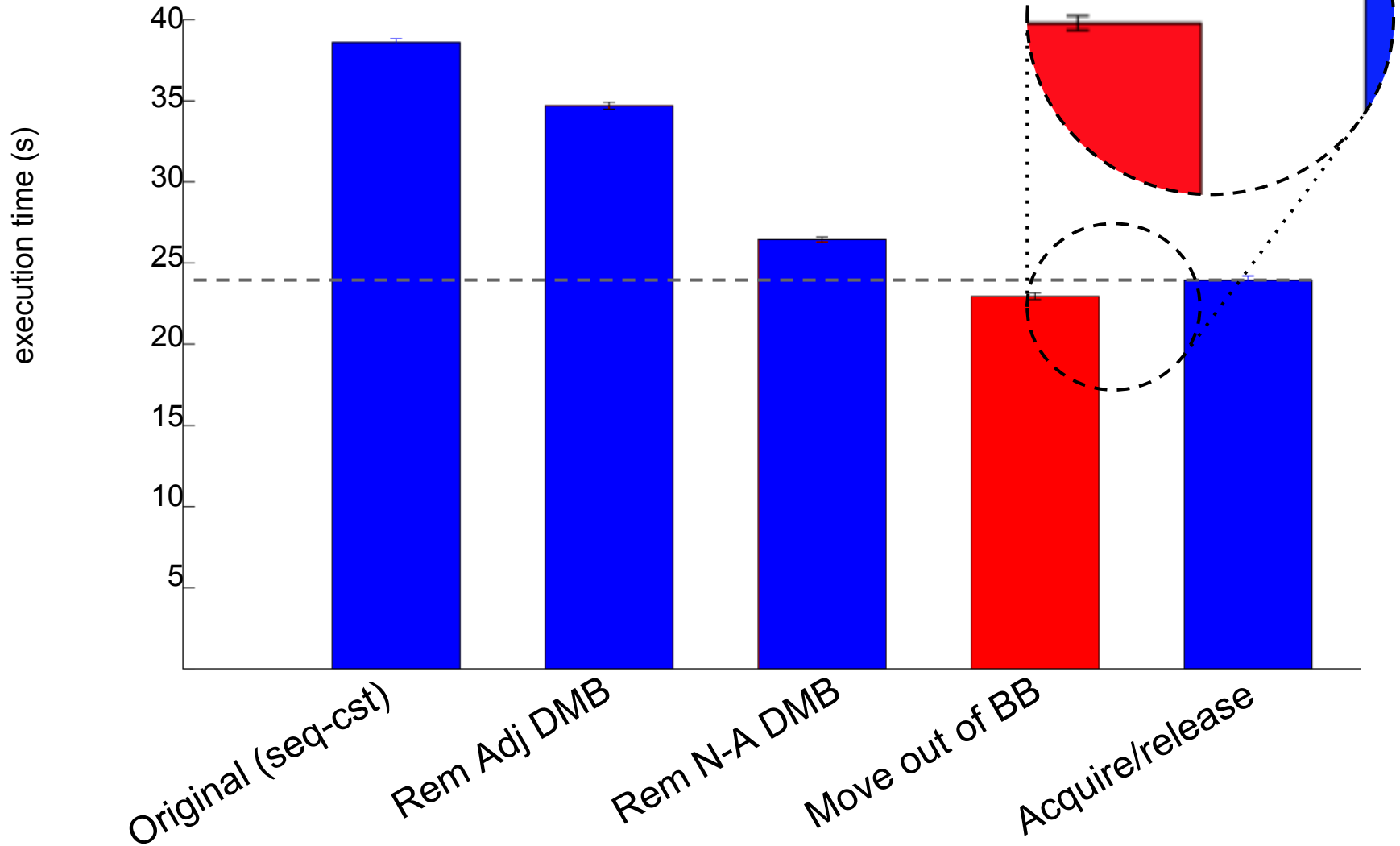
Pass 2: Move DMB out of Basic Block

.LBB0_1:

```
dmb ish
sub    r4, r1, #1
str    r4, [r2]
add    r0, r0, #1
dmb ish
cmp    r0, r3
str    r0, [r12]
dmb ish
str    r0, [1r]
dmb ish
str    r1, [r2]
add    r1, r1, #2
dmb ish
bne   .LBB0_1
dmb ish
```



As fast as acquire/release

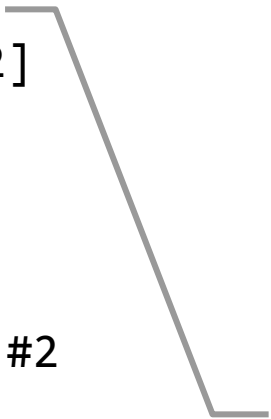


Optimised Seq.cst

```
.LBB0_1:  
  dmb      ish  
  sub     r4, r1, #1  
  str     r4, [r2]  
  add     r0, r0, #1  
  dmb     ish  
  cmp     r0, r3  
  str     r0, [r12]  
  dmb     ish  
  str     r0, [1r]  
  dmb     ish  
  str     r1, [r2]  
  add     r1, r1, #2  
  
  bne    .LBB0_1  
  dmb     ish
```

Acquire release

```
.LBB0_1:  
  dmb      ish  
  sub     r4, r1, #1  
  str     r4, [r2]  
  add     r0, r0, #1  
  dmb     ish  
  
  str     r0, [r12]  
  dmb     ish  
  str     r0, [1r]  
  dmb     ish  
  str     r1, [r2]  
  add     r1, r1, #2  
  cmp     r0, r3  
  bne    .LBB0_1
```



Next steps

- Can we take these optimisations further?
- Can we use the C11 semantics to improve optimisations?
- Is the LLVM IR expressive enough to facilitate all optimisations?

Can we take these optimisations further?

- Pass 2: Move DMB out of Basic Block
 - Consider the call graph
 - Do tests to see if it actually becomes faster
- Similar optimisations for other architectures
 - Mips (**sync**)

Can we use the C11 semantics to improve optimisations?

- Can we come up with better moving rules if we know where the barriers came from?
- OpenMP
 - Atomic variables local to parallel loops

Is the LLVM IR expressive enough to facilitate all optimisations?

Atomic read-modify-write

- implicit

```
  _Atomic(int) a; a *= b;
```

- explicit

```
  expected = current.load();
```

```
  do {
```

```
    desired = do_something(expected);
```

```
  } while (!compare_swap_weak(current, expected, desired));
```

LLVM IR only has strong `cmpxchg`, which *itself* generates a loop on ARM / MIPS

Thank you

Paper available on EuroLLVM site

Seqlock code & paper: <https://github.com/reinhrst/ARMBarriers>

Instruction-mappings: <https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html>

Reinoud Elhorst, Mark Batty, David Chisnall
{re302,mjb220,dc552} @ cam.ac.uk