# 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

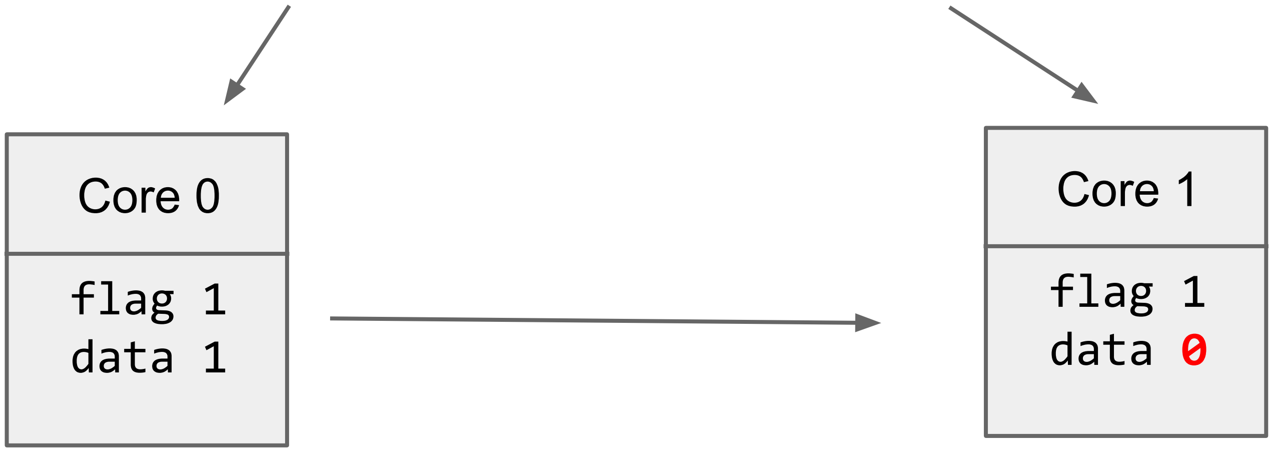
```
{flag,data,x}=0;
```

**Thread0:**

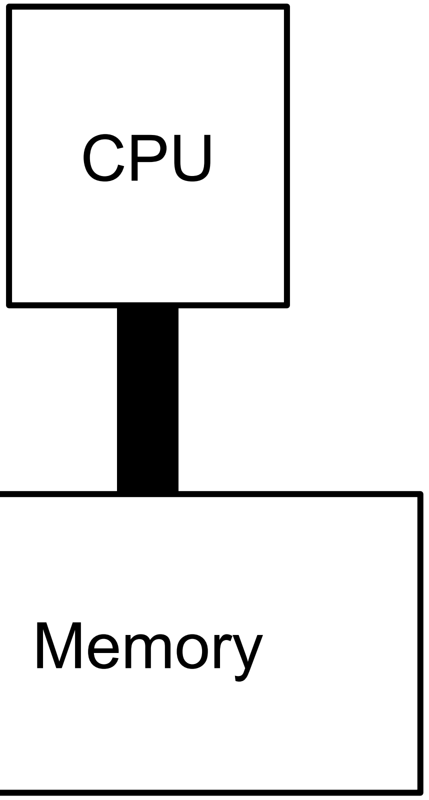
```
data=1;
flag=1;
```

**Thread1:**

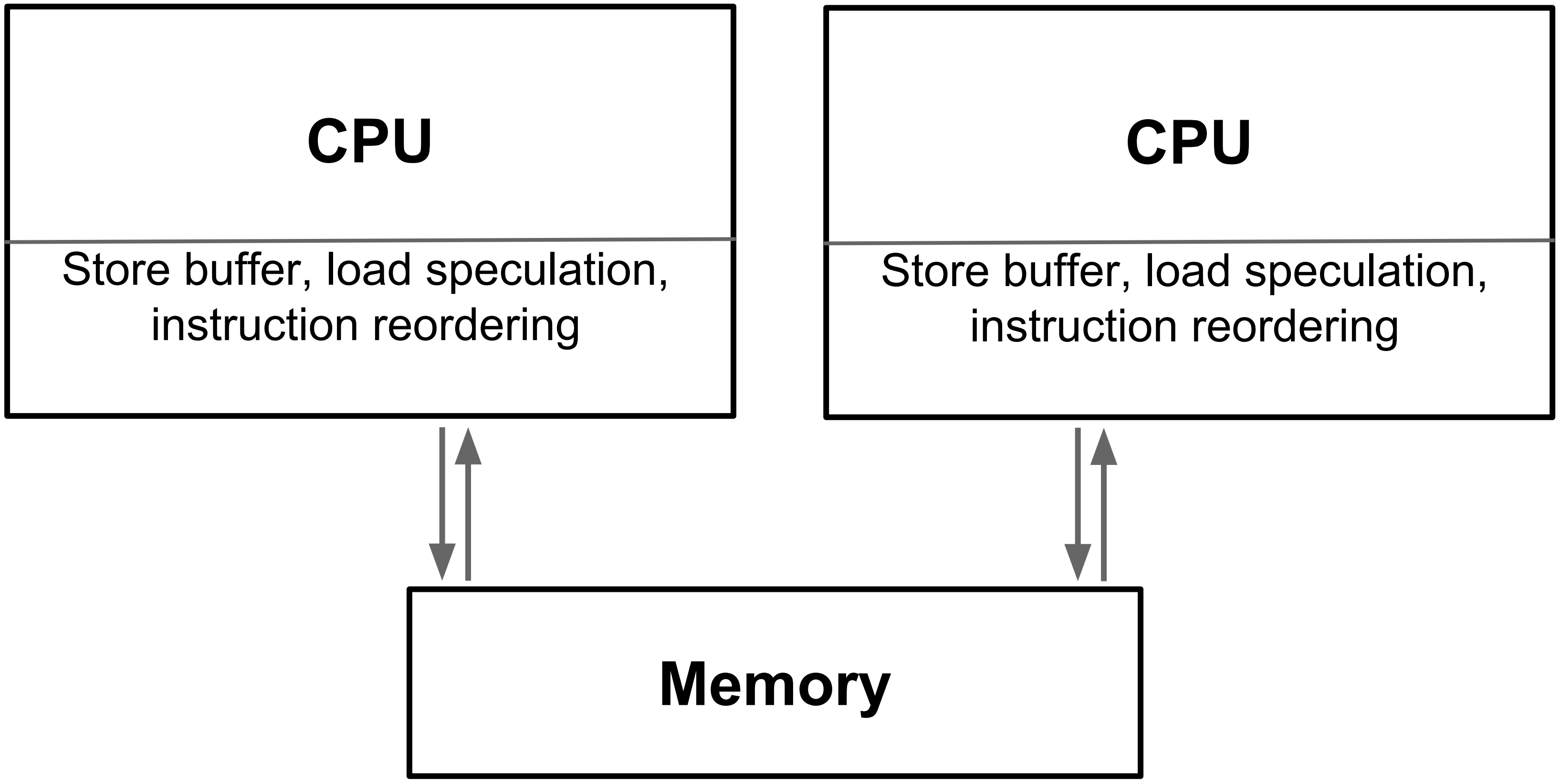
```
while(!flag){}
x=data;
```



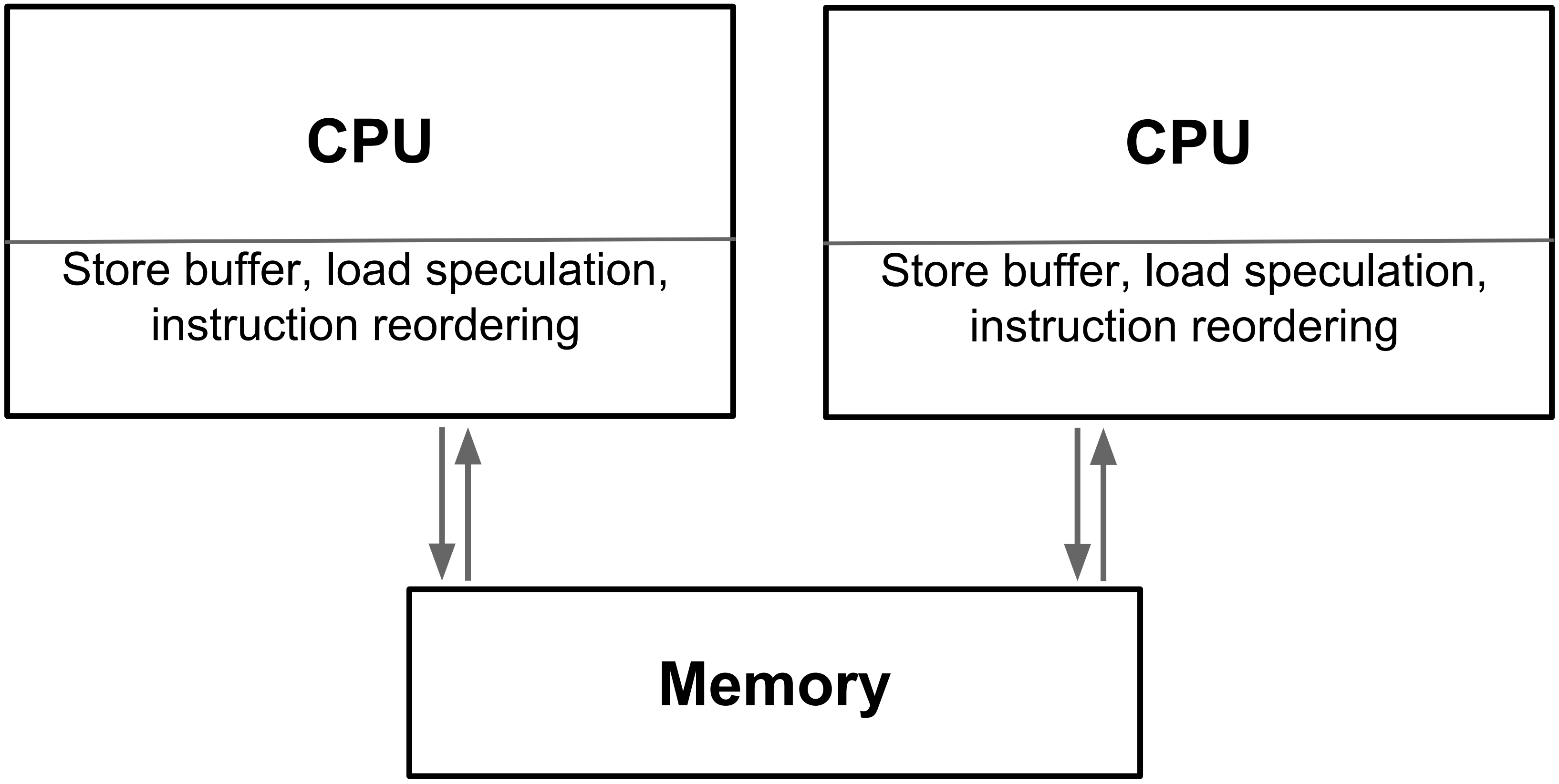
## Programmer’s mental model (C99)



## Simplified modern architecture


CPU
Store buffer, load speculation, instruction reordering
CPU
Store buffer, load speculation, instruction reordering
Memory

## Simplified modern architecture


CPU
Store buffer, load speculation, instruction reordering
CPU
Store buffer, load speculation, instruction reordering
Memory

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 |
|---|---|---|
| `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)` |

## 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_l = 0;
  store(lock, ++lock_l);
  store(x1, v1);
  store(x2, v2);
  store(lock, ++lock_l);
}
```

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

# Seqlock

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

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

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

## Seqlock

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

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

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

## Seqlock

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

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

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

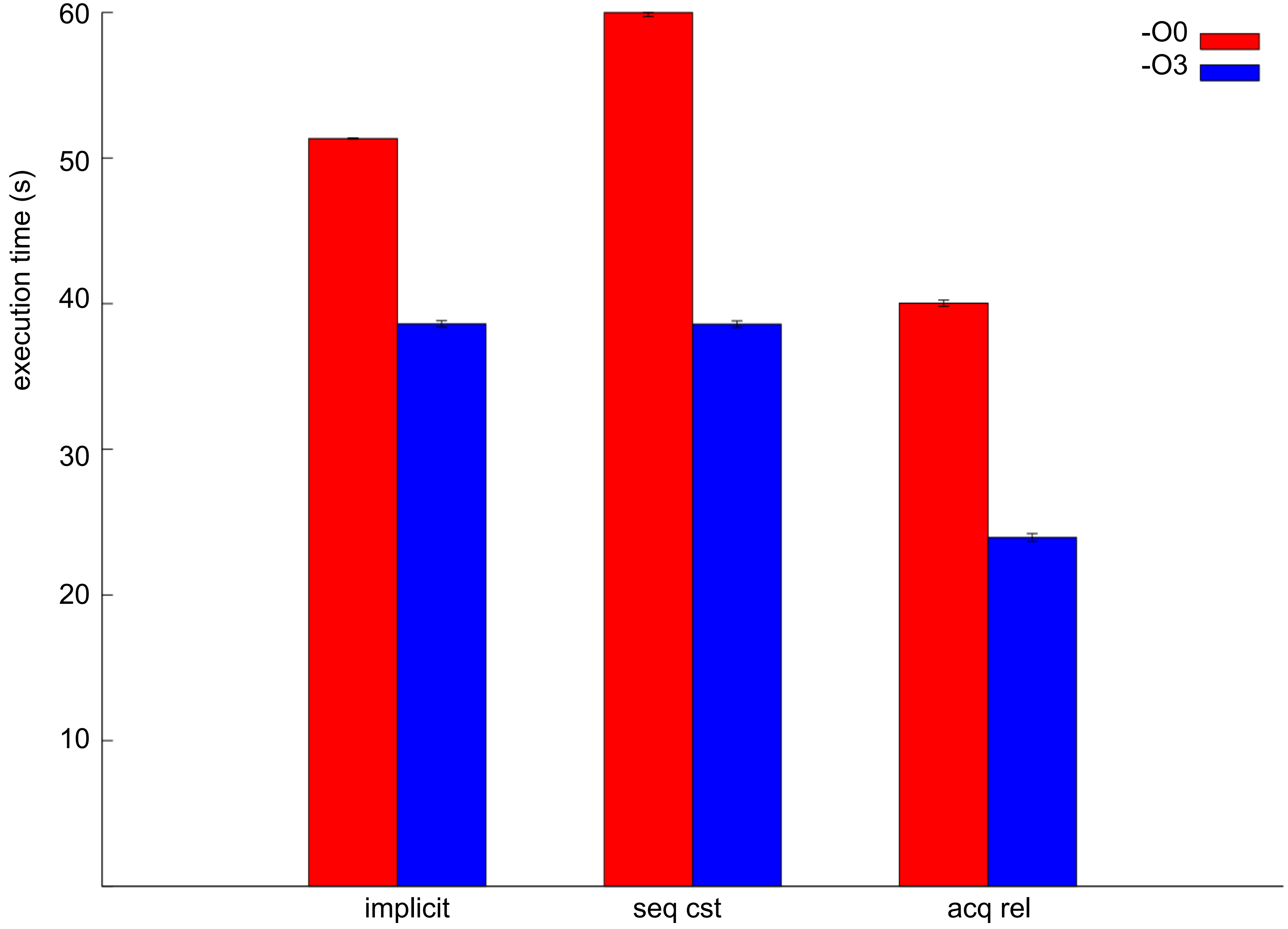
```
#define load(x) (x)
#define store(x,y) (x=y)
```

- Sequential consistency

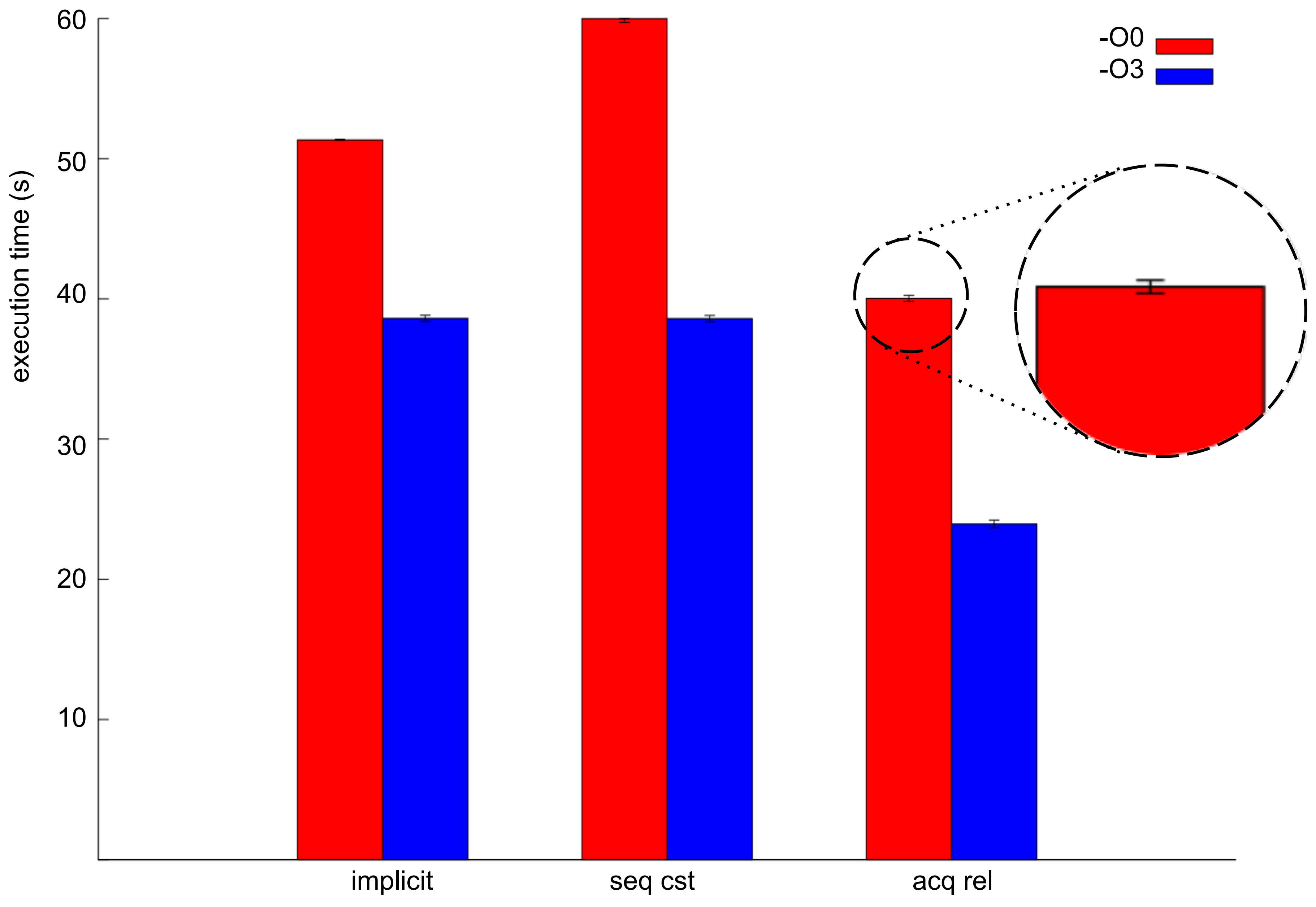
```
#define load(x) atomic_load_explicit(&x, seq_cst);
#define store(x,y) atomic_store_explicit(&x, y, seq_cst);
```

- Acquire / release

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



execution time (s)
60
50
40
30
20
10
-O0
-O3
implicit
seq cst
acq rel



execution time (s)
60
50
40
30
20
10
-O0
-O3
implicit
seq cst
acq rel

## ARM assembly

| | |
|---|---|
| **`dmb ish`** | barrier |
| **`ldr / str`** | load / store |
| **`add / sub`** | ALU operators |
| **`cmp`** | compare |
| **`bne`** | branch not equal |
| **`r12`** | registry |
| **`[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, [lr]
  dmb      ish
  dmb      ish
  str      r1, [r2]
  add      r1, r1, #2
  dmb      ish

  bne .LBB0_1
```

## Acquire release

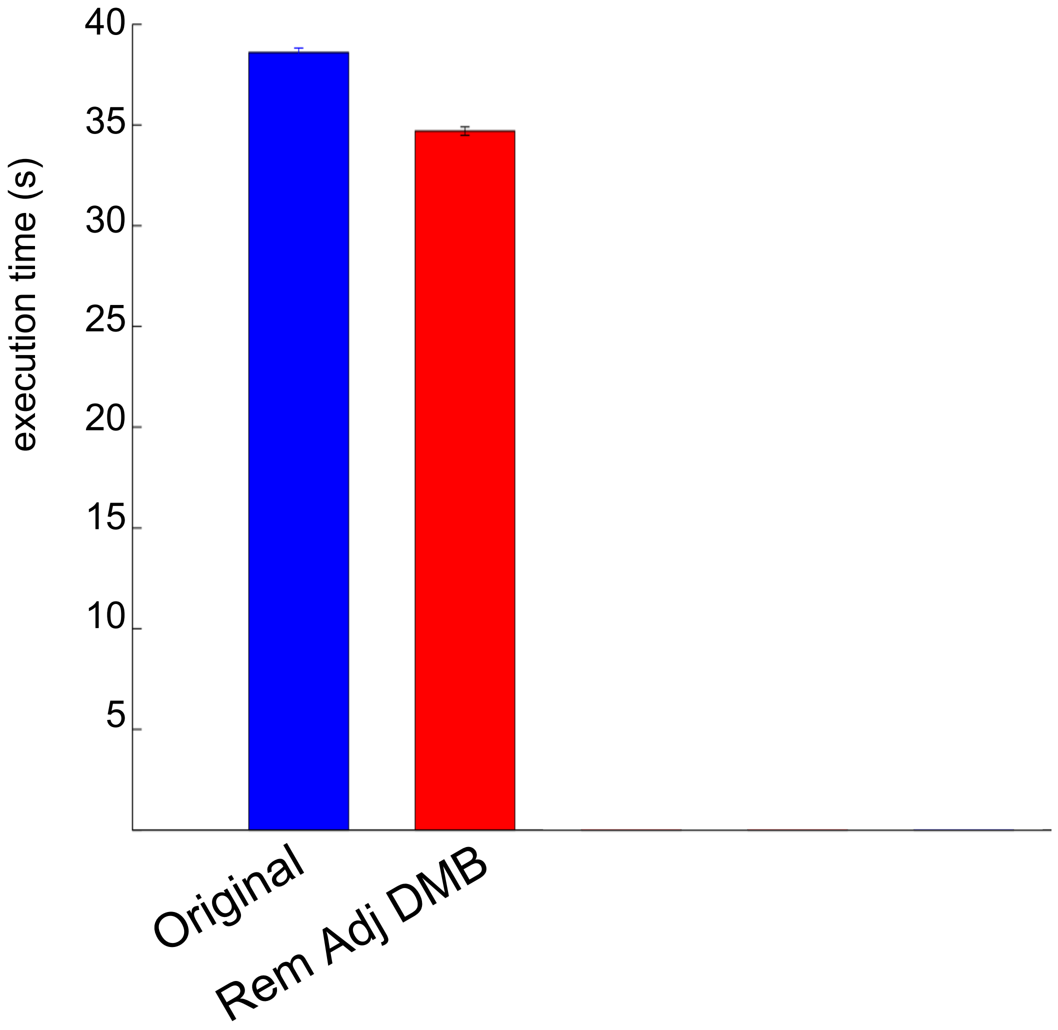
```
                                 for (i=0; i<1E9; i++)
.LBB0_1:
  dmb      ish
  sub      r4, r1, #1
  str      r4, [r2]              store(lock, ++l_lock);
  add      r0, r0, #1


  dmb      ish
  str      r0, [r12]             store(x1, i);

  dmb      ish
  str      r0, [lr]              store(x2, i);

  dmb      ish
  str      r1, [r2]              store(lock, ++l_lock);
  add      r1, r1, #2

  cmp      r0, r3
  bne      .LBB0_1
```

## Optimise the seq cst version

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

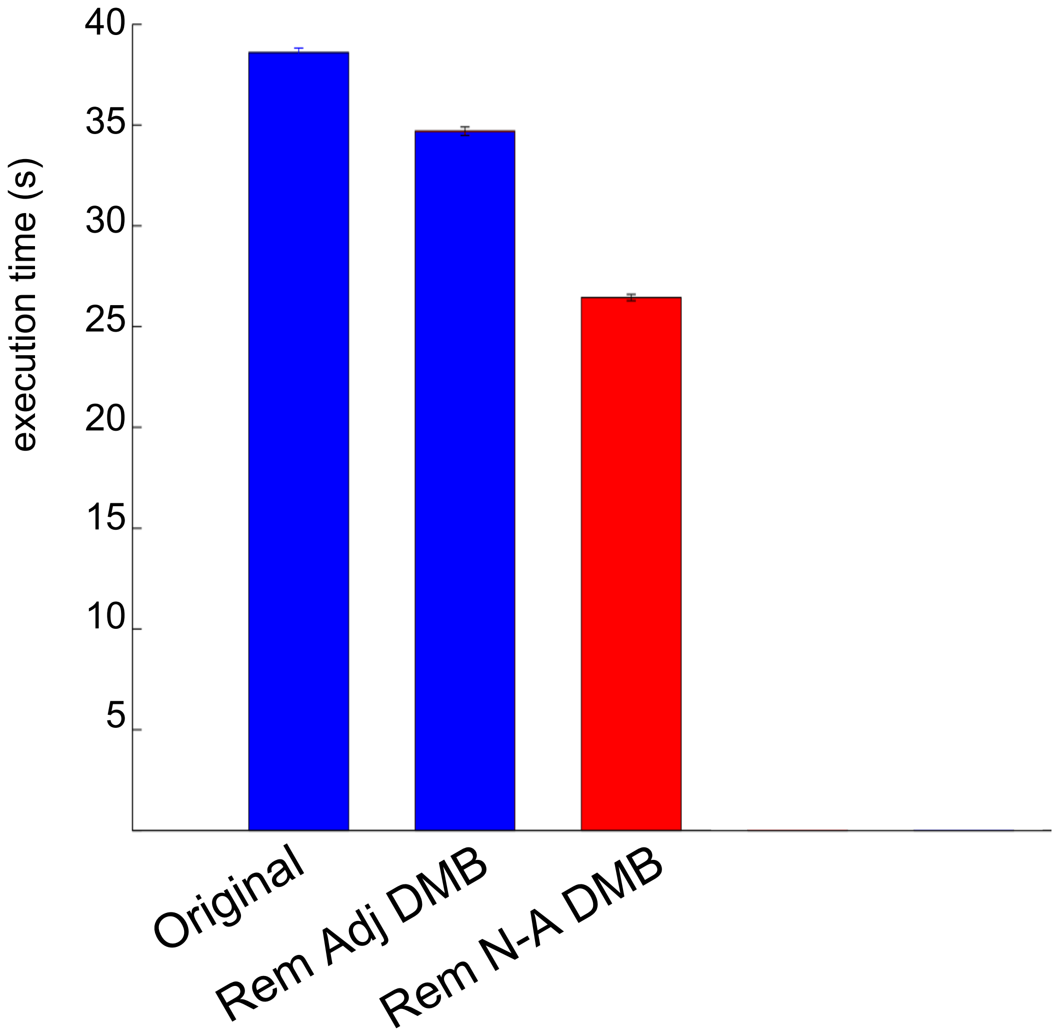
# Pass 1: Remove adjacent barriers

```
.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, [lr]
  dmb     ish
  ~~dmb     ish~~
  str     r1, [r2]
  add     r1, r1, #2
  dmb     ish
  bne .LBB0_1
```



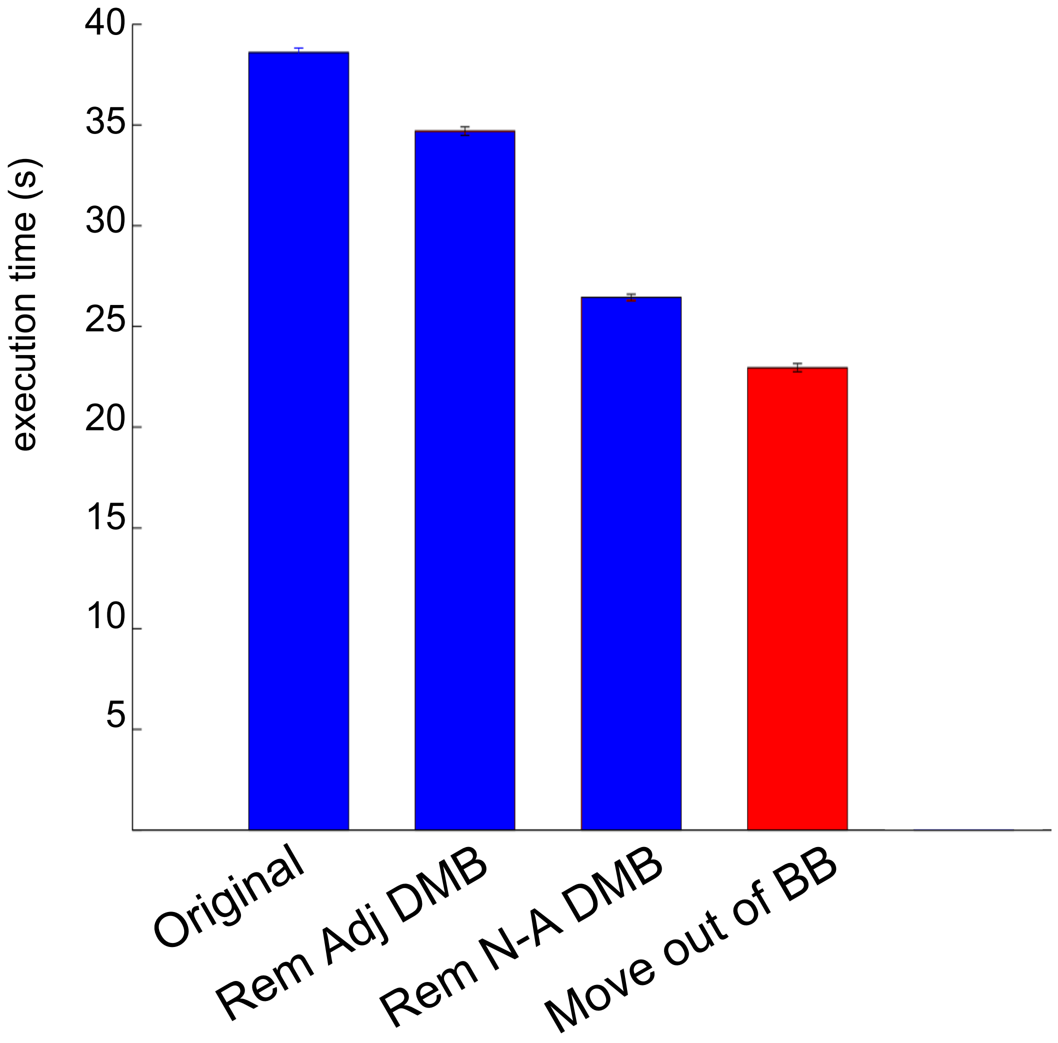
## Pass 1: Remove non-adjacent barriers

```
.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, [lr]
  dmb     ish
  str     r1, [r2]
  add     r1, r1, #2
  dmb     ish
  bne .LBB0_1
```

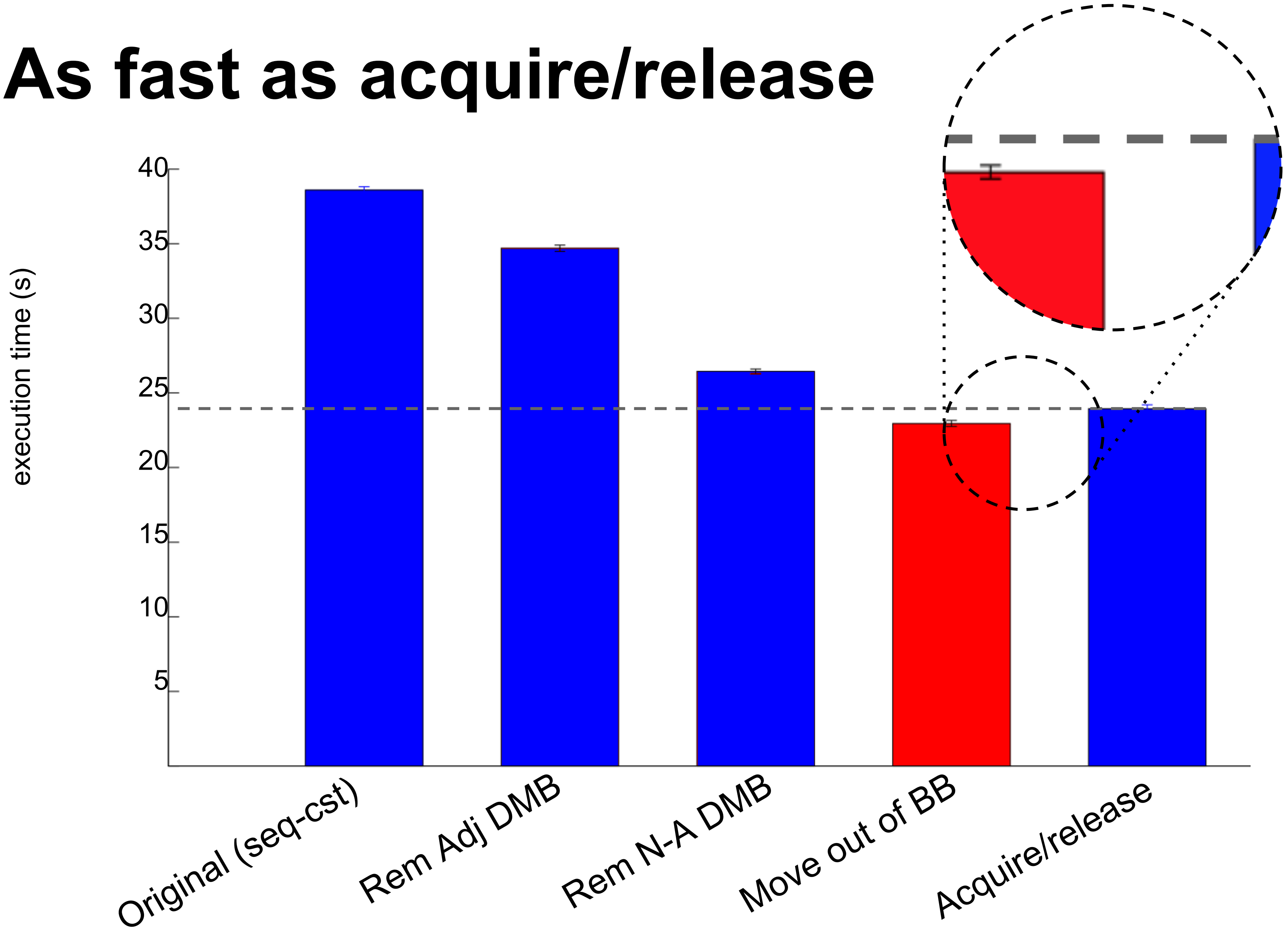


# Pass 2: Move DMB out of Basic Block

```
.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, [lr]
   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, [lr]
  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, [lr]
  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

## Seqlock as a benchmark

```
void writer() {
  int i, lock_l = 0;
  for (i=0; i<1E9; i++) {
    store(lock, ++lock_l);
    store(x1, i);
    store(x2, i);
    store(lock, ++lock_l);
  }
}
```

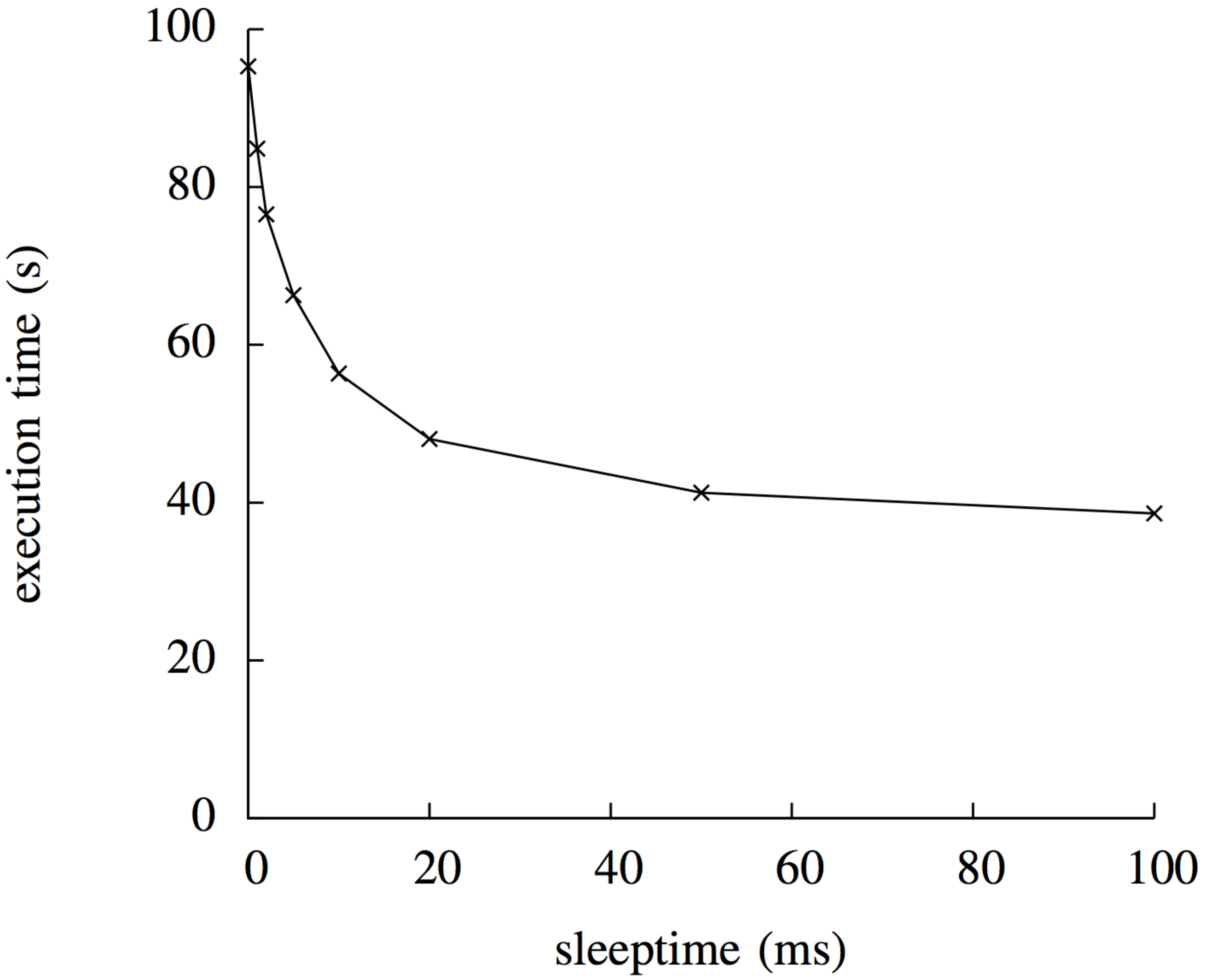
```
void reader() {
  int i, v1, v2, lock_l;
  while (v2 < 1E9) {
    while (true) {
      lock_l = load(lock);
      if (lock_l & 1) continue;
      v1 = load(x1);
      v2 = load(x2);
      if (lock_l == load(lock)) {
        break;
    }}
    assert(v1==v2);
    short_sleep();
}}
```

# Seqlock -O0 implicit/explicit seq cst

```
for.body:
  %1 = load i32* %local_lock, align 4
  %inc = add nsw i32 %1, 1
  store i32 %inc, i32* %local_lock, align 4


  store atomic i32 %inc, i32* @lock seq_cst, align 4
  %2 = load i32* %i, align 4


  store atomic i32 %2, i32* @x1 seq_cst, align 4
  %3 = load i32* %i, align 4


  store atomic i32 %3, i32* @x2 seq_cst, align 4
  %4 = load i32* %local_lock, align 4
  %inc2 = add nsw i32 %4, 1
  store i32 %inc2, i32* %local_lock, align 4


  store atomic i32 %inc2, i32* @lock seq_cst, align 4
  br label %for.inc
```

```
for.body:
  %1 = load i32* %local_lock, align 4
  %inc = add nsw i32 %1, 1
  store i32 %inc, i32* %local_lock, align 4
  store i32 %inc, i32* %.atomictmp
  %2 = load i32* %.atomictmp, align 4
  store atomic i32 %2, i32* @lock seq_cst, align 4
  %3 = load i32* %i, align 4
  store i32 %3, i32* %.atomictmp2
  %4 = load i32* %.atomictmp2, align 4
  store atomic i32 %4, i32* @x1 seq_cst, align 4
  %5 = load i32* %i, align 4
  store i32 %5, i32* %.atomictmp3
  %6 = load i32* %.atomictmp3, align 4
  store atomic i32 %6, i32* @x2 seq_cst, align 4
  %7 = load i32* %local_lock, align 4
  %inc5 = add nsw i32 %7, 1
  store i32 %inc5, i32* %local_lock, align 4
  store i32 %inc5, i32* %.atomictmp4
  %8 = load i32* %.atomictmp4, align 4
  store atomic i32 %8, i32* @lock seq_cst, align 4
  br label %for.inc
```

## Influence of read sleeptime



## weak cmpxchg: C11

```
void mul(_Atomic(int)*a, int b)
{
        *a *= b;
}
```

# weak cmpxchg: IR

```
atomic_op:                                        ; preds = %atomic_op, %entry
  %0 = phi i32 [ %atomic-load, %entry ], [ %1, %atomic_op ]
  %mul = mul nsw i32 %0, %b
  %1 = cmpxchg i32* %a, i32 %0, i32 %mul seq_cst
  %2 = icmp eq i32 %1, %0
  br i1 %2, label %atomic_cont, label %atomic_op
```

## weak cmpxchg: x86

```
LBB0_1:
        movl    %ecx, %edx
        imull   %esi, %edx
        movl    %ecx, %eax
        lock
        cmpxchgl        %edx, (%rdi)
        cmpl    %ecx, %eax
        movl    %eax, %ecx
        jne     LBB0_1
```

## weak cmpxchg: ARM

```
.LBB0_1:
        dmb     ish
        cmp     r2, lr
        popeq   {lr}
        bxeq    lr
.LBB0_2:
        mov     lr, r2
        dmb     ish
        mul     r12, lr, r1
.LBB0_3:
        ldrex   r2, [r0]
        cmp     r2, lr
        bne     .LBB0_1
@ BB#4:
        strex   r3, r12, [r0]
        cmp     r3, #0
        bne     .LBB0_3
        b       .LBB0_1
```

## weak cmpxchg: ARM (as it should be)

```
        dmb     ish
.retry:
        ldrex   r2, [r0]
        mul     r12, lr, r1
        strex   r3, r12, [r0]
        cmp     r3, #0
        bne     .retry

        dmb     ish
```

# Weak cmpxchg: MIPS64

```
# BB#0:                         # %entry
        daddiu  $sp, $sp, -16
        sd      $fp, 8($sp)     # 8-byte Folded Spill
        move     $fp, $sp
        addiu   $3, $zero, 0
$BB0_1:                         # %entry
                                # =>This Inner Loop
        ll      $2, 0($4)
        bne     $2, $3, $BB0_3
        nop
# BB#2:                                # %entry
                                       #   in Loop:
        addiu   $6, $zero, 0
        sc      $6, 0($4)
        beqz    $6, $BB0_1
        nop
$BB0_3:                                # %entry
        sync 0
$BB0_4:                                # %atomic_op
                                       # =>This Loop
                                       #     Child

Loop BB0_5 Depth 2
        move     $3, $2
        mul     $6, $3, $5
        sync 0
$BB0_5:                                # %atomic_op
                                #   Parent Loop BB0_4
        ll      $2, 0($4)
        bne     $2, $3, $BB0_7
        nop
```

```
# BB#6:                                        # %atomic_op
                                               #   in Loop:
        move     $7, $6
        sc      $7, 0($4)
        beqz    $7, $BB0_5
        nop
$BB0_7:                                        # %atomic_op
                                   #   in Loop: Header=BB0_4
        sync 0
        bne     $2, $3, $BB0_4
        nop
# BB#8:                                   # %atomic_cont
        move     $sp, $fp
        ld      $fp, 8($sp)      # 8-byte Folded Reload
        jr      $ra
        daddiu  $sp, $sp, 16
```

# Weak cmpxchg MIPS64 as it should be

```
sync 0            # Ensure all other loads / stores are globally visible
retry:
        ll   $t4, $a0     # Load the current value of the atomic int
        mult $t4, $a1     # Multiply by the other argument
        mflo $t4          # Get the result
        sc   $t4, $a0     # Try to write it back atomically
        bnez $t4, entry   # If we failed, try the whole thing again
        sync 0            # branch delay slot - ensure seqcst behaviour here
```