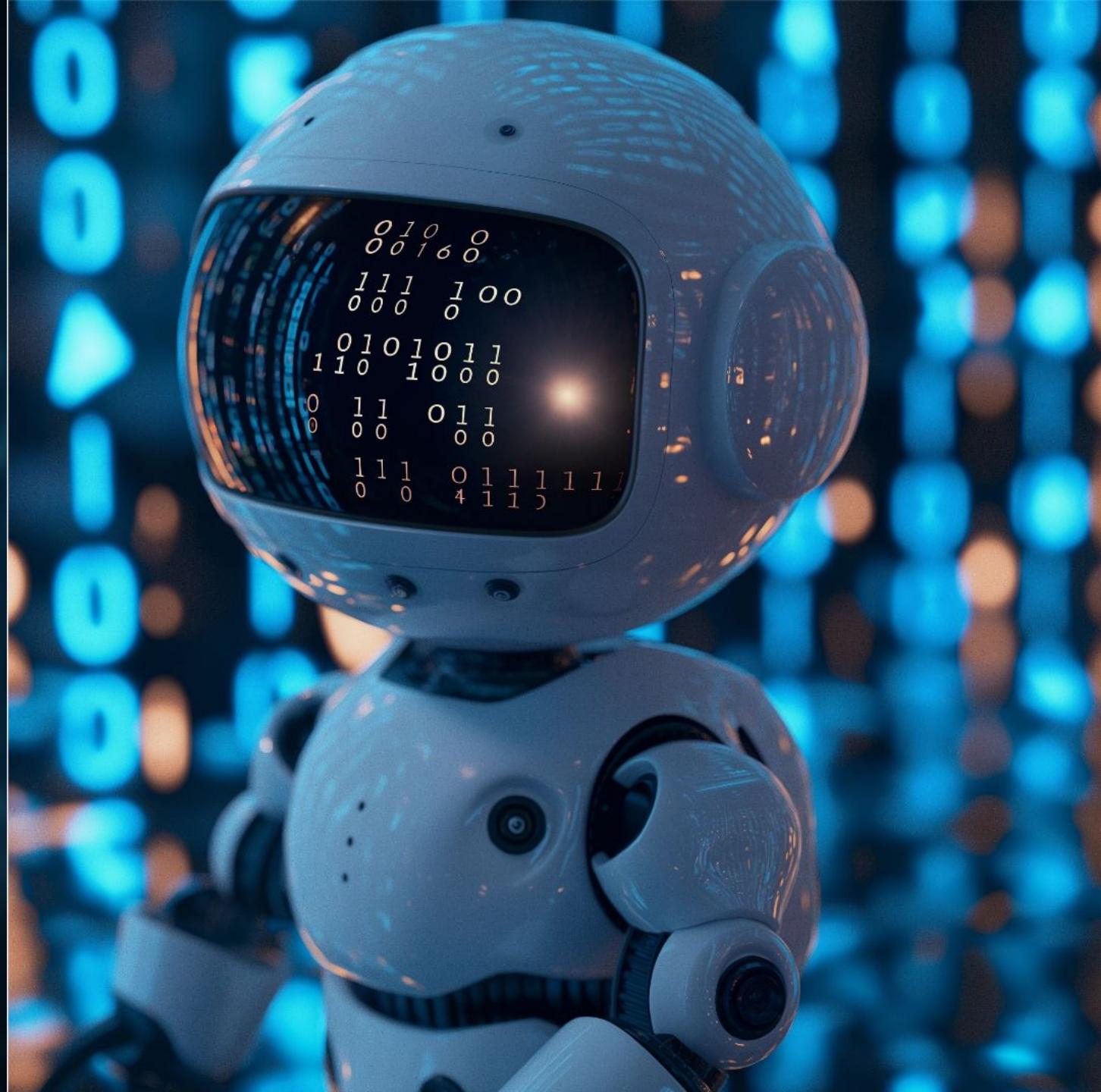


Memory Model

C++11 – C++23



About Me:

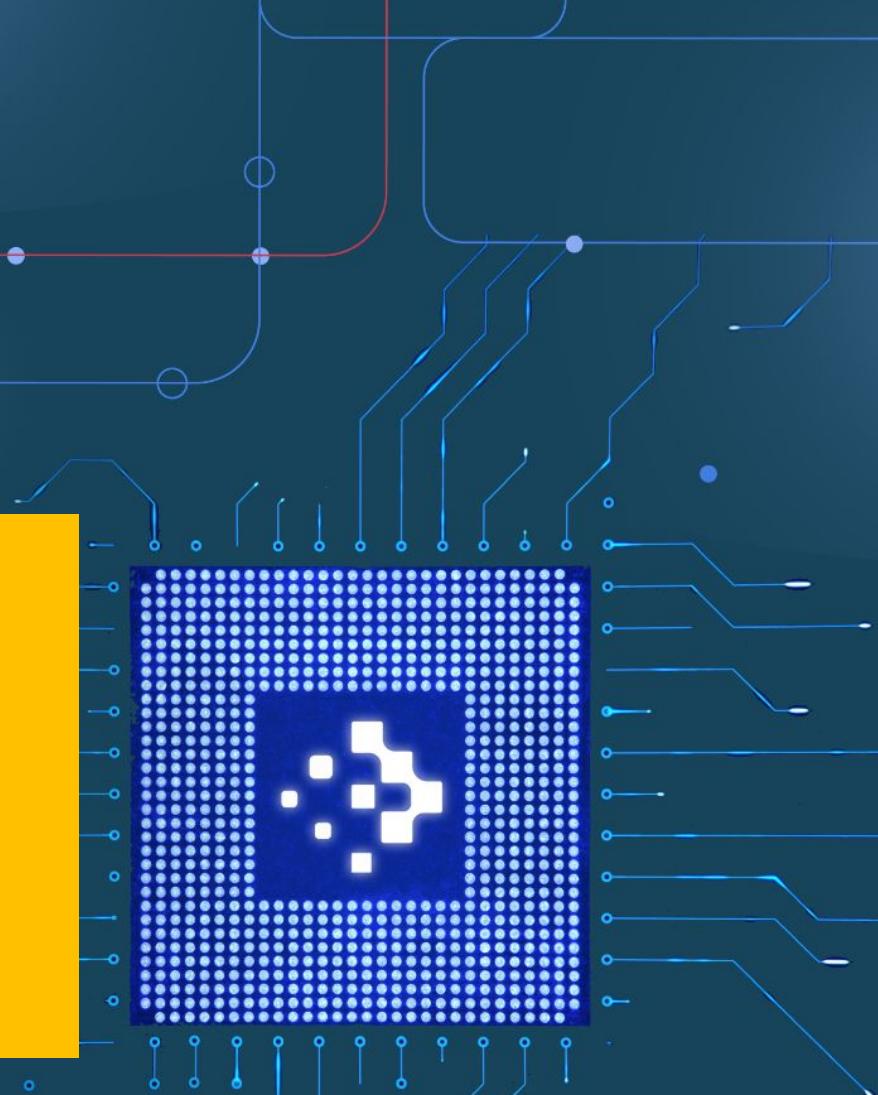


alex.dathskovsky@speedata.io

www.linkedin.com/in/alex-dathskovsky

www.cppnext.com

https://www.youtube.com/@cppnext-alex



Important Question

**Does the processor executes the
program as you wrote it?**

The Answer

Short Version

- Usually Not

Longer Version

- It will run what you have intended but:
 - Compilers reorder and change operations
 - CPU's use threading and OOO execution

Compiler Optimisations

If you were the compiler, what would you do?

```
constexpr std::size_t N = 100;
auto data = get_int_array<N>();
std::size_t p_sum{};
for (std::size_t i{}; i < N; i++){
    auto p = get_pivot(N);
    p_sum += data[i] * data[p];
}
```

If you were the compiler, what would you do?

- Compiler detects that `get_pivot` yields the same value
- It's better to call `get_pivot` once and call the value from register again and again
- `data[p]` has the same value, it will be stored in a register

Compiler can do even better

- Modern (X86) CPUs have vectorization

```
.LCPI1_5:  
    .long 20          # 0x14  
    .long 21          # 0x15  
    .long 22          # 0x16  
    .long 23          # 0x17  
.LCPI1_6:  
    .long 24          # 0x18  
    .long 25          # 0x19  
    .long 26          # 0x1a  
    .long 27          # 0x1b  
.LCPI1_7:  
    .long 28          # 0x1c  
    .long 29          # 0x1d  
    .long 30          # 0x1e  
    .long 31          # 0x1f
```

```
movaps  xmm0, xmmword ptr [rip + .LCPI1_0] # xmm0 = [0,1,2,3]  
movaps  xmmword ptr [rsp - 128], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_1] # xmm0 = [4,5,6,7]  
movaps  xmmword ptr [rsp - 112], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_2] # xmm0 = [8,9,10,11]  
movaps  xmmword ptr [rsp - 96], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_3] # xmm0 = [12,13,14,15]  
movaps  xmmword ptr [rsp - 80], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_4] # xmm0 = [16,17,18,19]  
movaps  xmmword ptr [rsp - 64], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_5] # xmm0 = [20,21,22,23]  
movaps  xmmword ptr [rsp - 48], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_6] # xmm0 = [24,25,26,27]  
movaps  xmmword ptr [rsp - 32], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_7] # xmm0 = [28,29,30,31]  
movaps  xmmword ptr [rsp - 16], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_8] # xmm0 = [32,33,34,35]  
movaps  xmmword ptr [rsp], xmm0  
movaps  xmm0, xmmword ptr [rip + .LCPI1_9] # xmm0 = [36,37,38,39]  
movaps  xmmword ptr [rsp + 16], xmm0
```

Compiler can do even better

- If everything can be known in compile time

```
main:                                # @main
    mov      eax, 247500
    ret
```

Algebraic Simplifications

$$(v + 1) + 3 \rightarrow$$

$$v + (1+3) \rightarrow$$

$$v + 4$$

Strength Reduction

```
uint64_t add_and_devide_u(uint64_t a, uint64_t b){  
    return (a+b)/2;  
}
```



```
add_and_devide_u(unsigned long, unsigned long):  
    lea      rax, [rdi + rsi]  
    shr     rax  
    ret
```

To INT Or
To UI NT



Loop unrolling

- Compiler may unroll your loops

```
for (int i{}; i<5; i++){
    std::cout << "Hello World!\n";
}
```

Loop unrolling

- Compiler may unroll your loops

```
mov    edx, 13
mov    rdi, rbx
mov    rsi, r14
call   std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >
mov    edx, 13
mov    rdi, rbx
mov    rsi, r14
call   std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >
mov    edx, 13
mov    rdi, rbx
mov    rsi, r14
call   std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >
mov    edx, 13
mov    rdi, rbx
mov    rsi, r14
call   std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >
mov    edx, 13
mov    rdi, rbx
mov    rsi, r14
call   std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<char, std::char_traits<char> >
```

Dead Code Removal

- If there is unnecessary code, it will probably be removed

```
int dead_code(int size){  
    int i{}, j{};  
    for (; i < size; i++){  
        j++;  
    }  
  
    return i;  
}
```

Dead Code Removal

- If there is unnecessary code, it will probably be removed

```
dead_code(int):  
    xor      eax, eax  
    test     edi, edi  
    cmovg   eax, edi  
    ret
```

inlining

- inlines function calls

```
int add1(int a) {  
    return a + 1;  
}  
  
int main(int c, char** argv){  
    return add1(c);  
}
```

inlining

- inlines function calls

```
main:                                # @main
    lea      eax, [rdi + 1]
    ret
```

inlining recursion

- Tail recursion can be inlined

```
int add(int a, int n){  
    if (n==0){  
        return a;  
    }  
    return add(a+1, n-1);  
}  
  
int main(int c, char** argv){  
    return add(c, c+10);  
}
```

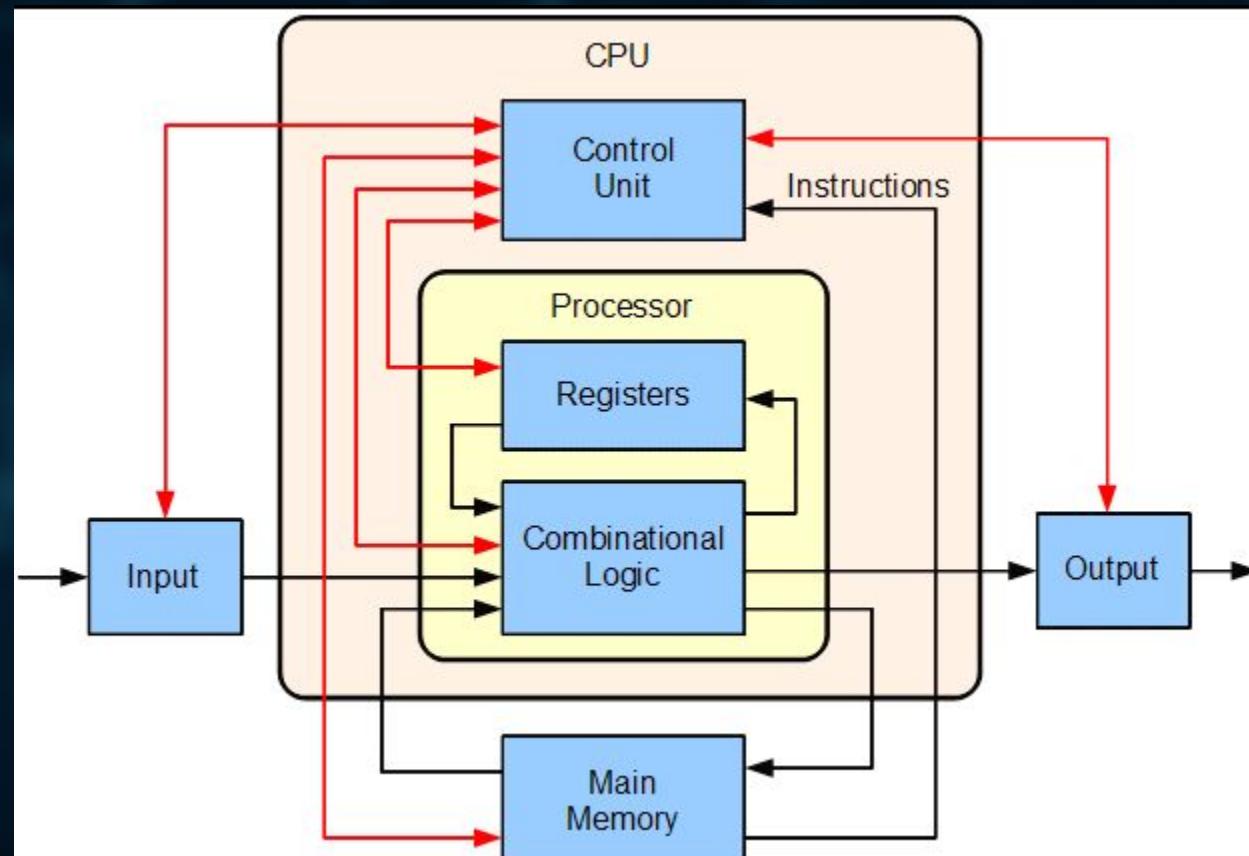
inlining recursion

- Tail recursion can be inlined

```
1 add(int, int):  
2     lea    eax, [rdi + rsi]  
3     ret  
4 main:  
5     lea    eax, [2*rdi + 10]  
6     ret
```

CPU Optimisations

INO Execution



Source https://en.wikipedia.org/wiki/Central_processing_unit

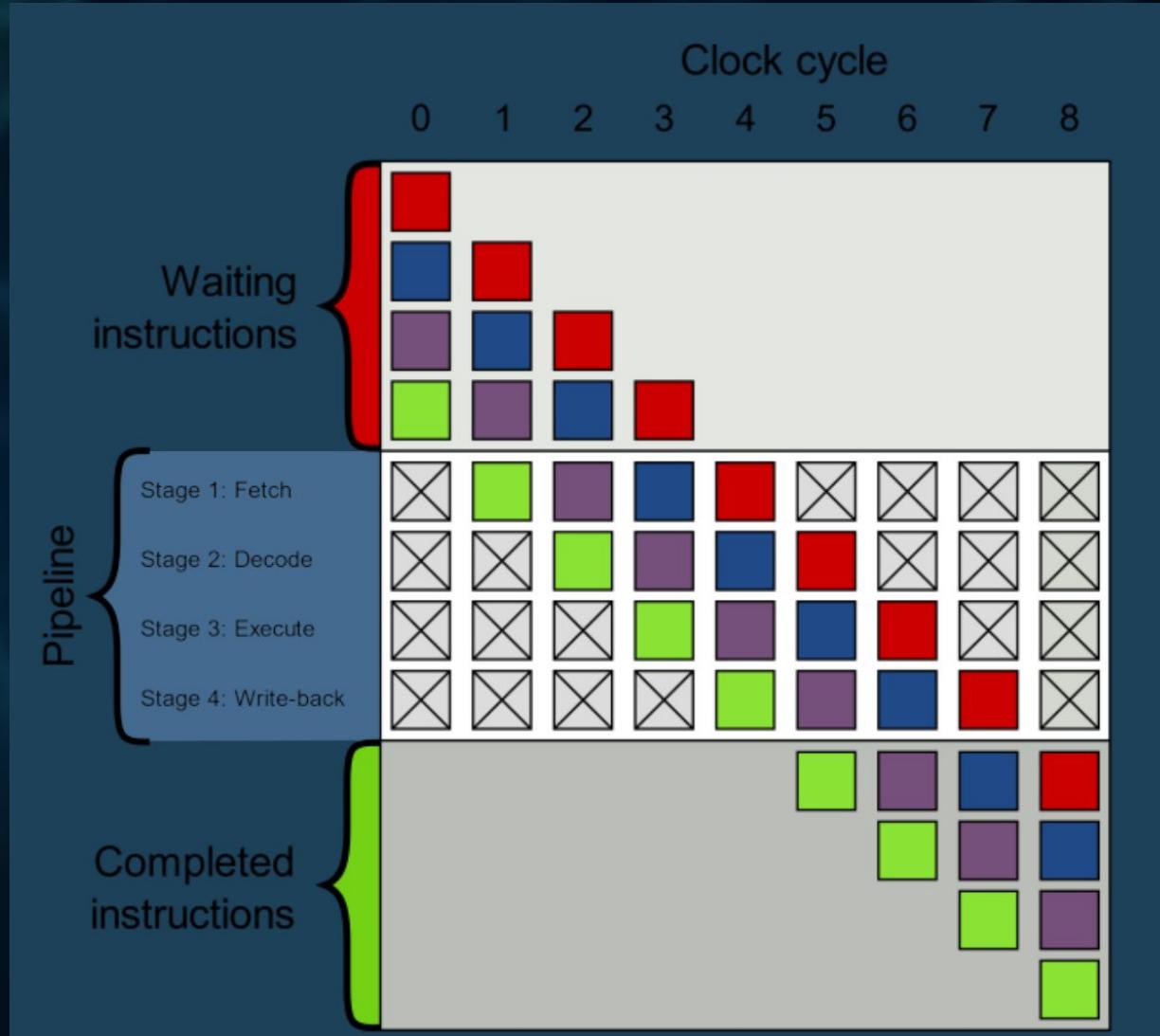
+ 22

High Speed Query Execution with Accelerators and C++

ALEX DATHSKOVSKY



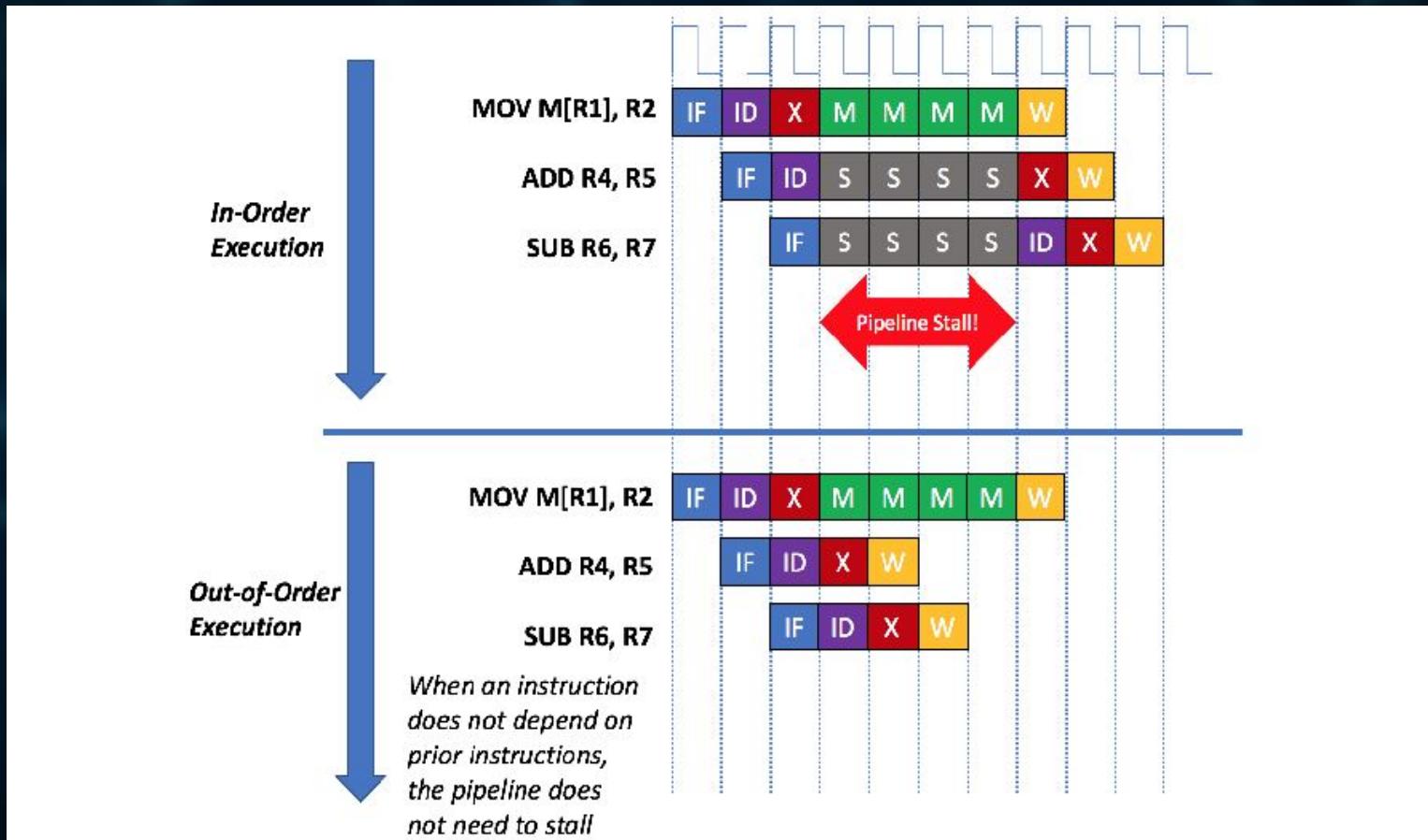
INO Execution



INO Execution

- instruction fetch
- if operands available execute it if not fetch them
- The instruction is executed by the functional unit
- The functional unit writes the result back to the register or memory

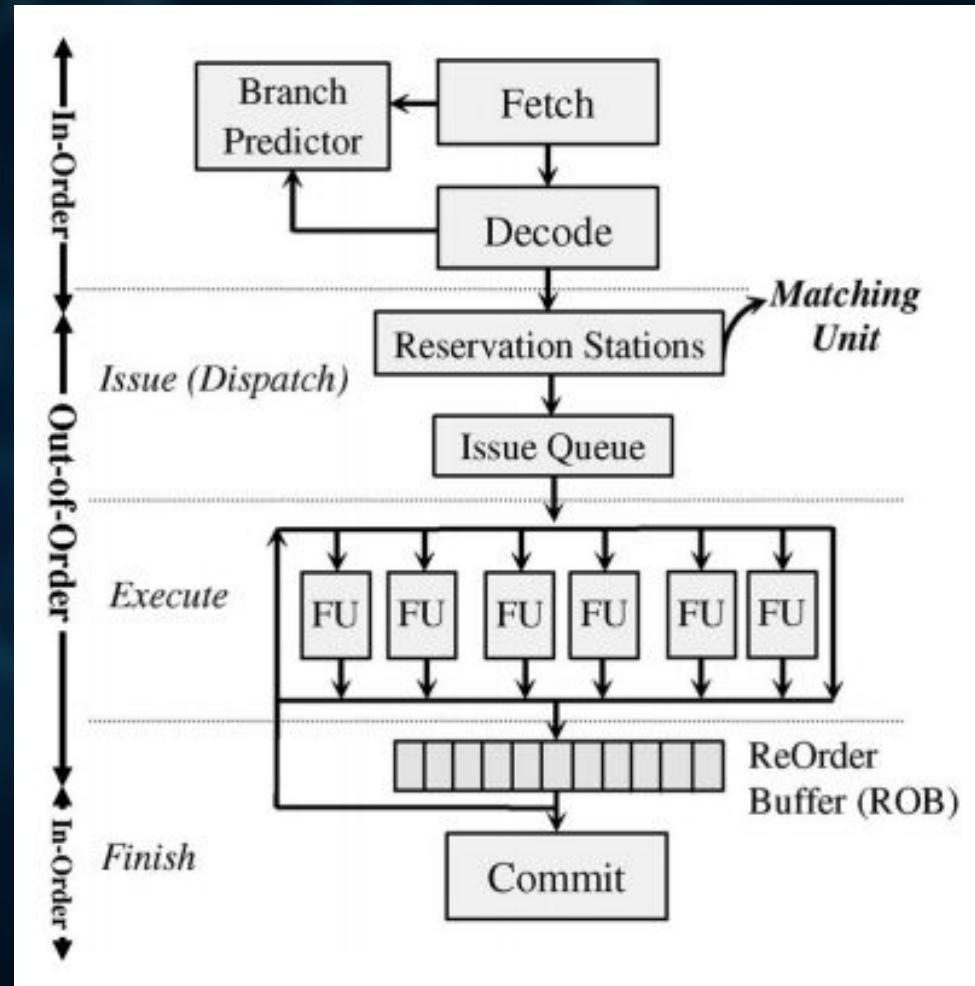
INO vs OOO Execution



Source:

<https://www.semanticscholar.org/paper/RISC-V-Reward:-Building-Out-of-Order-Processors-in-Zekany-Tan/f7f6d27f334604c3c85f0b8d21d2a9b4df22a983>

OOO Execution



From F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez and Y. Etsion, "Hybrid Dataflow/von-Neumann Architectures," in IEEE Transactions on Parallel and Distributed Systems

OOO Execution : Dynamic Scheduling

- instruction fetched
- instruction dispatched to instruction queue
- the instruction waits in the queue until its input operand are available
- if operands available instruction is allowed to leave the queue before other instructions
- the instruction is issued to a functional unit
- only if all older instructions have completed the operation the result is written to register file

Dynamic Scheduling

- **Check for structural hazards**
 - an instruction can be issued if a functional unit is available
 - an instruction stalls if no appropriate functional unit available

Dynamic Scheduling

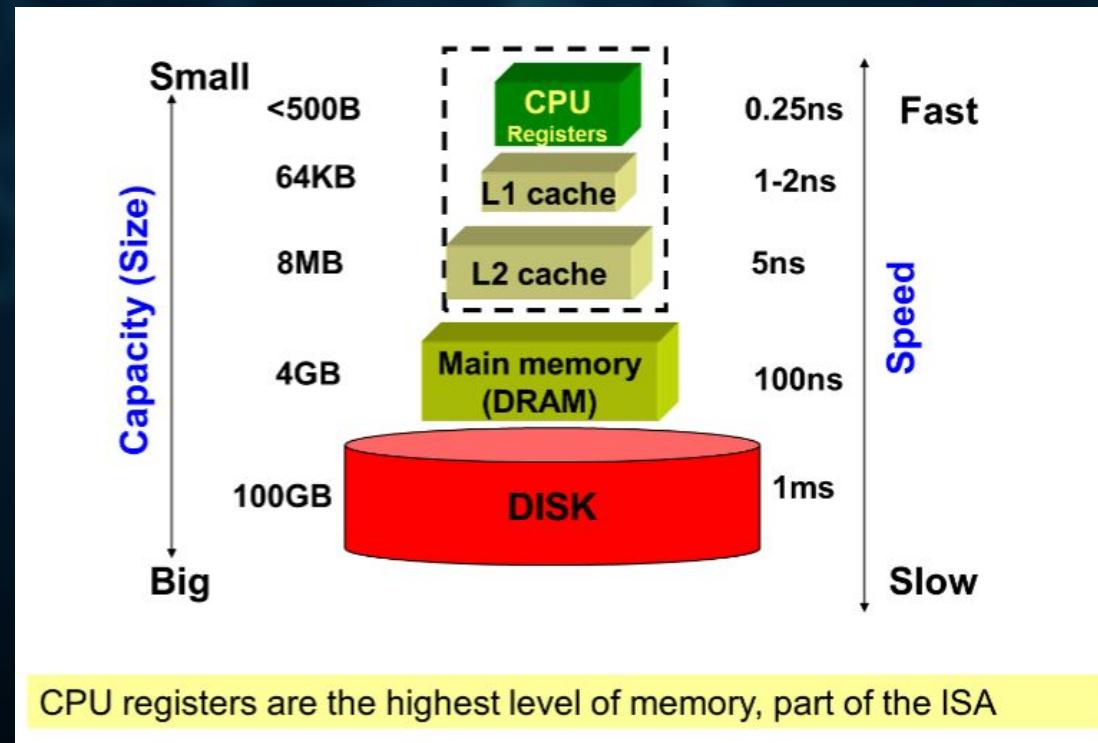
- **Check for data hazards**
 - an instruction can be executed when its operands have been calculated or loaded from memory
 - an instruction stalls if operands are not available

Dynamic Scheduling

- instruction can be executed out of order if there is no dependency on previous instructions
- ready instructions can execute before earlier instructions that are stalled

Cache \$

- Cache can accelerate reads and make writes longer and more complex



Reordering and C++

Reordering Types

- Data dependencies must be honored
- C++ compiler may reorder any memory access under the as-if rule
- Different processors have different reordering guarantees

AS-IF Rule

- Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread.

AS-IF Rule

- At program termination, data written to files is exactly as if the program was executed as written
- Prompting text which is sent to interactive devices will be shown before the program waits for input
- Executing function from external libraries

AS-IF Rule - Not Always

- programs with undefined behaviour
- copy elision, the compiler may remove calls to move and copy-constructor and destructors of temporary objects even if those calls have side effects

Multi Threading and Reordering

Reordering between two threads

Thread 1

```
flag1 = 1;  
if (flag2 == 0){  
    critical section}
```

Thread 2

```
flag2 = 1;  
if (flag1 == 0){  
    critical section}
```

Sequential Consistency

Sequential Consistency

- Result of any execution is the same as if the operations of all the processors were executed in some sequential order
- Operations of each individual processor appear in the sequence in the order specified by the program

SQC: Do we Have It?

- SQC is very expensive
- Modern compilers do not offer it (for free)

What Does C++ and Modern CPU guarantee?

SC-DRF:

- A system guaranteeing DRF-SC must define specific instructions called *synchronizing instructions*, which provide a way to coordinate different processors (equivalently, threads)
- Programs use those instructions to create a “happens before” relationship between code running on one processor and code running on another.

Example:

Thread 1

Foo(Flag)

S(a)

Bar(Flag)

Thread 2

S(a)

Bar(Flag)



Pop Quiz: can this happen?

Thread 1

$x = 1$

Thread 2

$x = 2$

Thread 3

$y1 = x$
 $y2 = x$

Thread 4

$y3 = x$
 $y4 = x$

$y1=1, y2=2, y3=2, y4=1?$

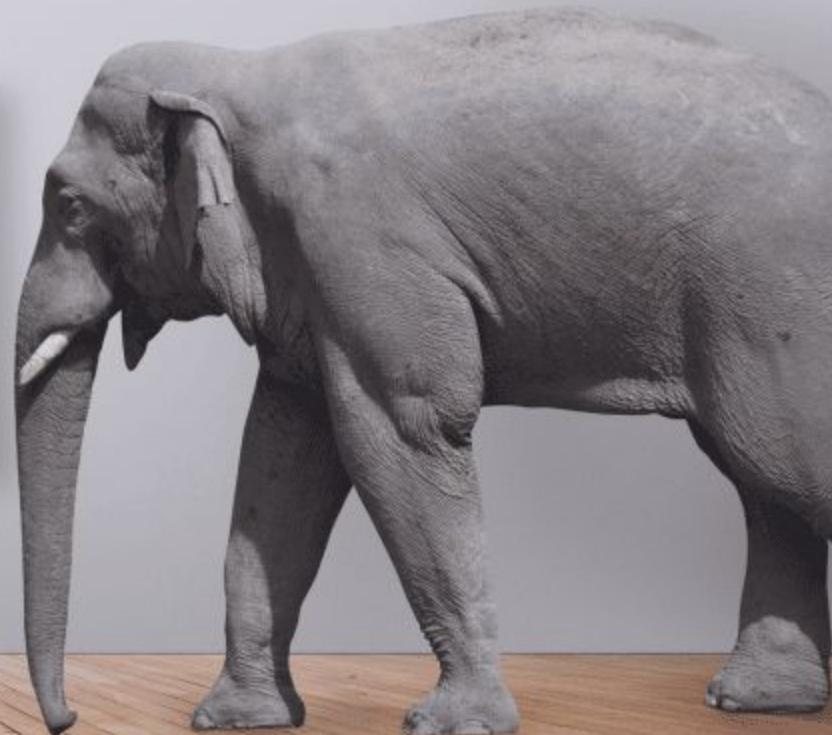
Pop Quiz: can this happen?

$y1=1, y2=2, y3=2, y4=1?$

Thread 1	Thread 2	Thread 3	Thread 4
1: $x = 1$	4: $x = 2$	2: $y1 = x$	3: $y4 = x$
		5: $y2 = x$	6: $y3 = x$

Synchronization

ADDRESSING THE ELEPHANT IN THE ROOM



Volatile

- Volatile is not a synchronization tool
- Volatile doesn't affect threading
- It's not an atomic value
- It doesn't add barriers

Volatile: what is it good for?

- Tells the compiler it shouldn't optimize the memory reads and writes order.
- Alas the CPU can still reorder

Volatile: Bad Example

```
int main(){
    int y = 1;
    volatile int* x = &y;

    auto w =std::thread([x](){ for (int i = 0; i < 50; i++){
        *x += 1;}});
}

auto r =std::thread([x](){for (int i = 0; i < 50; i++){
    fmt::print("{}\n", *x);}});

r.join();
w.join();
}
```

Volatile: Most uses deprecated with C++20

- Is += a single/atomic instruction? How about ++?
- How many reads/writes are needed for compare_exchange? What if it fails
- foo(int volatile n) int volatile foo() are meaningless.

Volatile: Most uses deprecated with C++20

- For more information watch JF's talk about it

The image shows a video frame from JF Bastien's talk at Cppcon 2019. On the left, JF Bastien is standing at a podium, wearing a dark t-shirt with 'Engineer' printed on it. A nameplate on the podium identifies him as 'JF Bastien'. Below the nameplate, the title of his talk is 'Deprecating volatile'. On the right, a large slide with a dark background features the text 'What does volatile mean?' in large white font, with the word 'Colloquially' written in a smaller script font below it. In the bottom right corner of the slide, the number '22' is visible.

Cppcon | 2019
The C++ Conference | cppcon.org

JF Bastien

Deprecating volatile

What does
volatile
mean?
Colloquially

22

Synchronization cont

Compiler Code Barriers

Prevent compiler from moving reads or writes
across the barrier

Compiler Code Barriers

Prevent compilation
across the barrier

Thread 1

```
value = very_long_calculation()
asm volatile("mfence" :: "memory")
done = true
```



reads or writes

Thread 2

```
done = false (not done){}
asm volatile("mfence" :: "memory")
something
```

Locks and Atomics

Most Locks and atomic operation will act like barriers

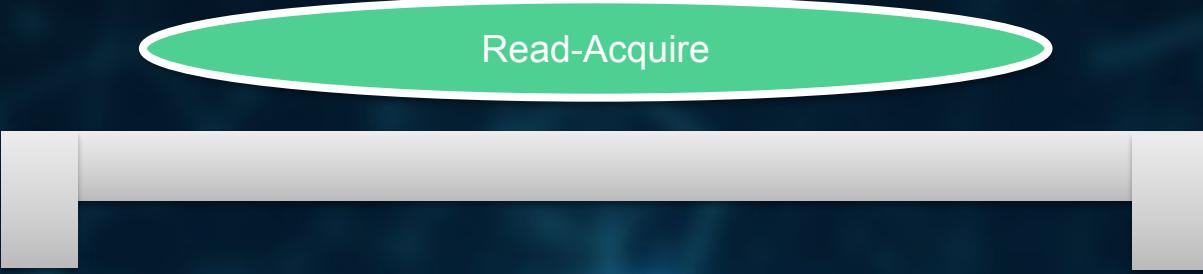
Understanding Memory Barriers / Memory Ordering

Acquire Semantics

- *property that can only apply to operations that **read** from shared memory, whether they are **read-modify-write** operations or plain loads. The operation is then considered a **read-acquire**. Acquire semantics prevent memory reordering of the read-acquire with any read or write operation that **follows** it in program order.*

Acquire Semantics

Read-Acquire



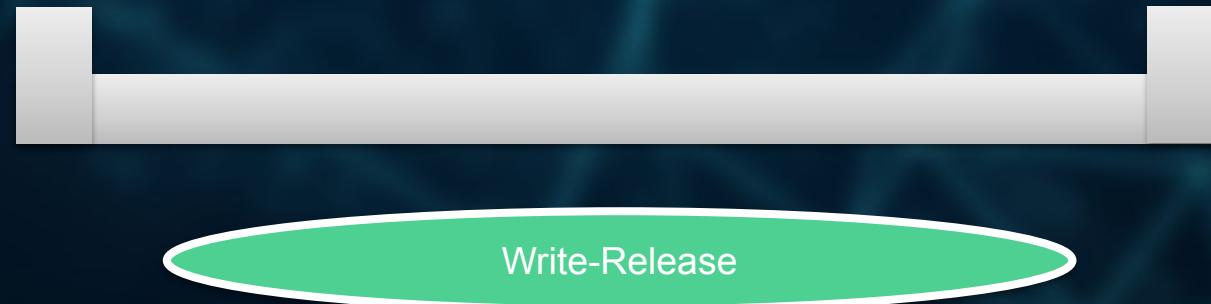
All memory operations
stay below the line

Release Semantics

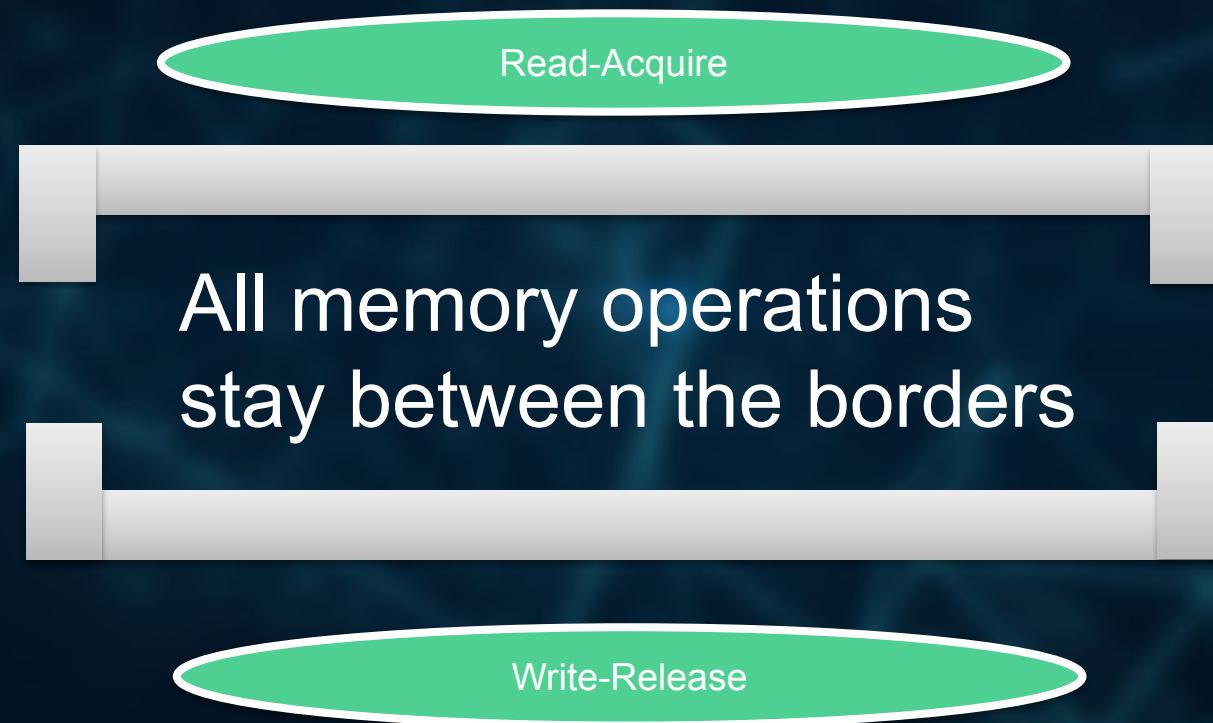
- *property that can only apply to operations that **write** to shared memory, whether they are read-modify-write operations or plain stores.*
*The operation is then considered a **write-release**. Release semantics prevent memory reordering of the write-release with any read or write operation that **precedes** it in program order.*

Release Semantics

All memory operations
stay above the line



Memory Barrier / Acquire Release Order



Sequential Consistency

- *This is the default and the most strict mode. Enforcing that all operations are seen in a globally consistent order across all threads, as if they were executing in a single sequential thread. This means that any read or write operation that uses this memory order will be sequenced before or after all other operations in the program, providing a globally consistent view of memory.*

Sequential Consistency

SQC

All operations will be in
the same order

SQC

What's The Difference?

Thread 1

$x = sr(1)$

Thread 2

$y = sr(2)$

Thread 3

$y1 = la(x)$
 $y2 = la(y)$

Thread 4

$y3 = la(y)$
 $y4 = la(x)$

$sr = \text{store_release}$

$la = \text{load_acquire}$

T3: $y1 = 1$ $y2 = 0$

T4: $y3 = 2$ $y4 = 0$

What's The Difference?

Thread 1

$x = ss(1)$

Thread 2

$y = ss(2)$

Thread 3

$y1 = ls(x)$

$y2 = ls(y)$

Thread 4

$y3 = ls(y)$

$y4 = ls(x)$

$ss = store_seq_cst$

$ls = load_seq_cst$

T3: $y1 = 1$ $y2 = 2$

T4: $y1 = 1$ $y2 = 2$

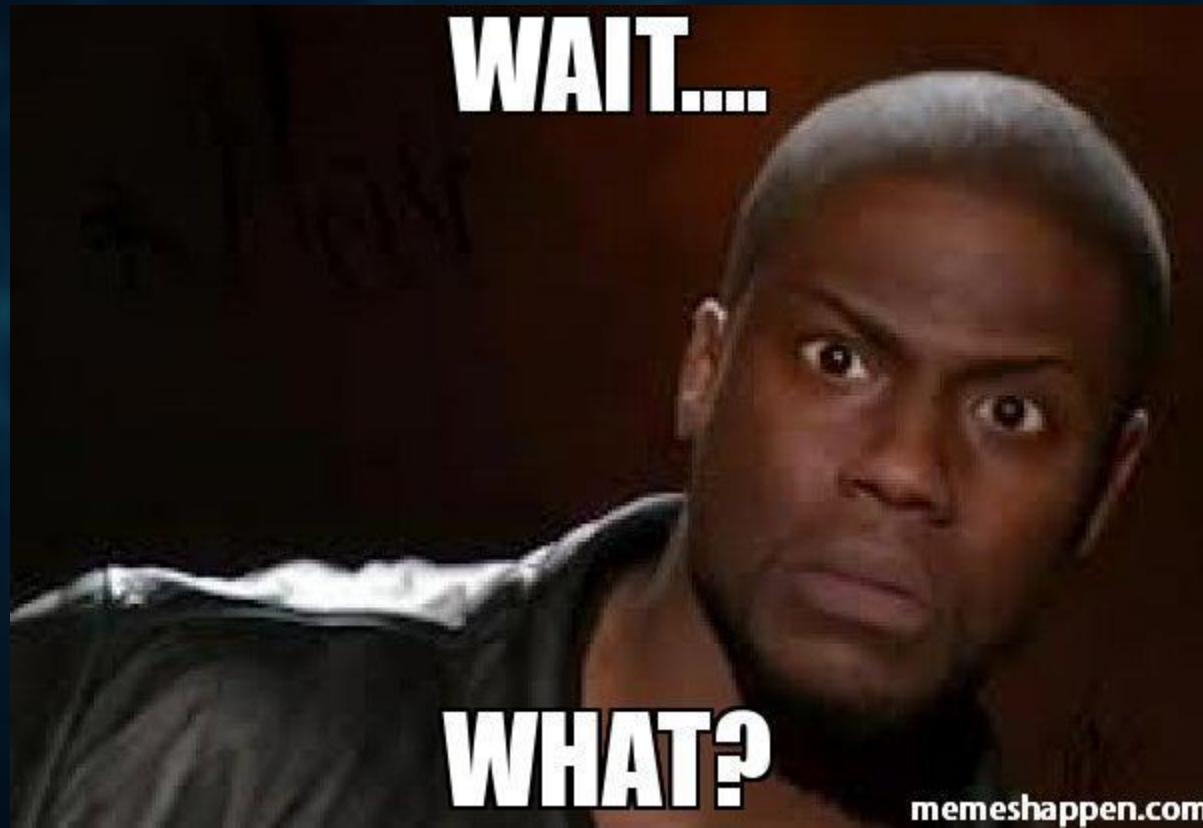
Relaxed Semantics

- *This semantics offers the weakest guarantees among memory ordering constraints. There are no guarantees about the relative order of memory accesses as observed by different threads. This means that operations performed by one thread might not be immediately visible to other threads, and the order in which operations appear to occur might not reflect the actual order in which they were executed.*

Relaxed Semantics



Consume Semantics



Consume Semantics

designed to exploit data dependencies between threads to provide synchronization guarantees.

Data Dependency Ordering

- Some CPU have to emit barriers to order between memory instructions (armv7, risc v)
- Data dependencies make the order defined

```
ldr    w8, [x8, :lo12:val]  
add    w8, w8, #1  
str    w8, [x9, :lo12:out]
```

Data Dependency Ordering

- *Data dependencies can be on memory and chained*

ldr	w0, [x3]
ldr	w2, [x2]
add	w0, w0, w2
add	w0, w0, 40
str	w0, [x1]

Consume Semantics : How does it works

- *The compiler exploit the data dependency chains*
- *Instructions have to carry-a-dependency (lucky it's defined in the standard)*
- *to summarize: one evaluation carry-a-dependency if value of the first is defined in the second evaluation as an operand*

Example

```
a = atomic_bool_value{false}
```

```
x = 0
```

```
Thread 1
```

```
x = 1
```

```
a.set(true)
```

$z = a.load()$

$\text{if } (z) \ y = x$

Good Example

```
a = atomic_int*_value{null}
```

Thread 1

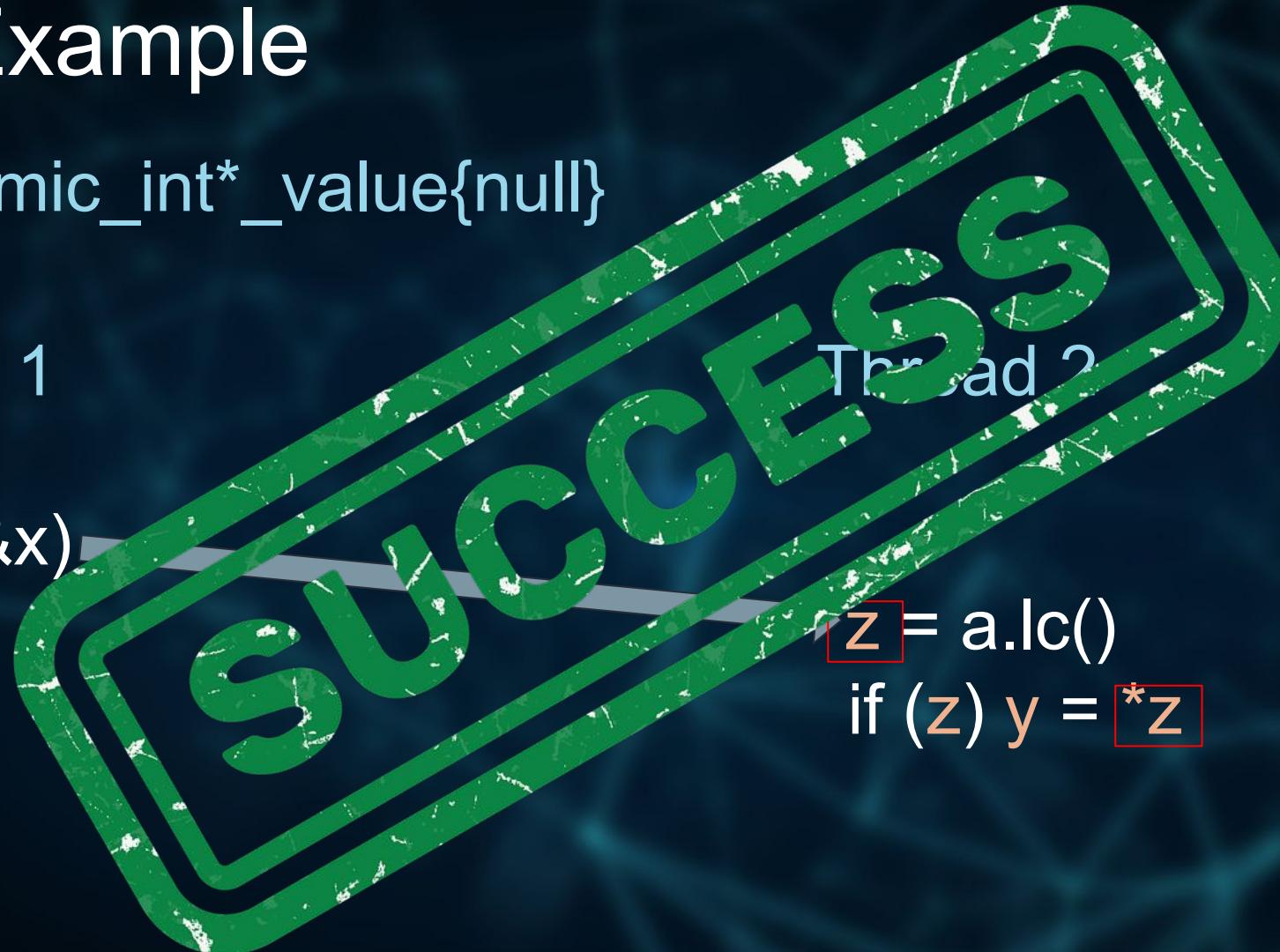
```
x = 1
```

```
a.sr(&x)
```

Thread 2

```
z = a.lc()
```

```
if (z) y = *z
```



Consume Semantics

Good news consume is almost like acquire, acquire promises same things and its stronger.

Most compilers do not support consume and will emit release-acquire

What may happen here ?

A1.store(4, release)
Y = A2.load(acquire)

What may happen here ?

```
Y = A2.load(acquire)  
A1.store(4, release)
```

Why is it possible ?

A1.store(4, release)

Y = A2.load(acquire)

Why is it possible ?

```
Y = A2.load(acquire)
```

```
A1.store(4, release)
```

C++ Concurrency tools

std::thread

- Cross platform thread class

```
std::thread([](){ std::this_thread::sleep_for(1s); }).join();
```

thread_local

- storage of the variable is defined by the thread
- helps with data races, each thread has its own copy

```
20  thread_local uint64_t inc = 0;
21
22  void do_count(std::string_view name, int count, uint64_t& ret){
23      for
24      |
25  }      ret a: 10
26
27      ret b: 20
28 }
29
30  int main
31  {
32      uint64_t inc;
33      std::thread a{do_count, "a", 10, inc};
34      std::thread b{do_count, "b", 20, inc};
35
36      a.join(); b.join();
37      fmt::print("ret a: {}\nret b: {}\nmain inc: {}\n", ra, rb, inc);
38 }
```

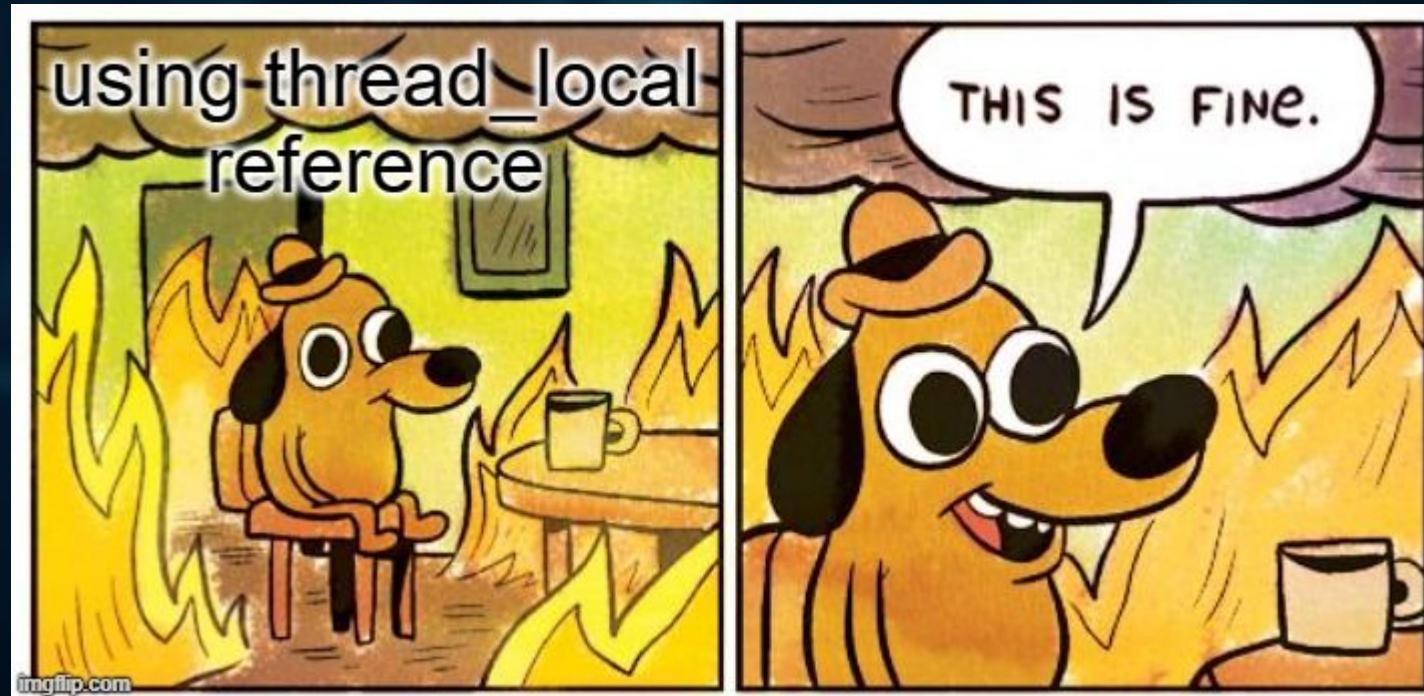
thread_local

- Please don't abuse it

```
38     thread_local uint64_t inc = 0;
39
40     void do_count(std::string_view name, int count,
41                     uint64_t& ret, uint64_t* val){
42         for
43             ret a: 30
44         }
45
46         ret b: 20
47     }
48
49     int main() {
50         std::cout << "main inc: " << inc << std::endl;
51         std::thread t(do_count, "A", 20, std::ref(rb), &inc);
52         t.join();
53         std::thread t2(do_count, "B", 10, std::ref(rb), &inc);
54         t2.join();
55         std::cout << "ret a: " << ra << std::endl;
56         std::cout << "ret b: " << rb << std::endl;
57         std::cout << "main inc: " << inc << std::endl;
58     }
```

thread_local

- Please don't abuse it



std::atomic

- Provides a portable way to perform low level atomic operations
- No torn reads and no torn writes
- Provides read-acquire, write-release and full memory barriers

Atomic Memory Order

- `memory_order_relaxed` – there are no sync or ordering constraints, only this operation is guaranteed to be atomic
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_seq_cst` – this will give us sequential consistency, and this is the default mode

Atomic Memory Fence

Int a = 0

Thread 1

```
a = 42;  
atomic_thread_fence(memory_order_release)
```

Thread 2

```
atomic_thread_fence(memory_order_acquire)  
  
int r2 = a ? a : -1;
```

std::atomic Without a Fence

```
Int a = 0  
std::atomic<int> ready(0)
```

Thread 1

```
a = 42;  
  
ready.store(1, memory_order_release)
```

Thread 2

```
int r1 = ready.load(memory_order_acquire)  
  
int r2 = a ? a : -1;
```

std::atomic: Default

```
Int a = 0  
std::atomic<int> ready(0)
```

Thread 1

```
a = 42;  
  
ready.store(1)
```

Thread 2

```
int r1 = ready  
  
int r2 = a ? a : -1;
```

std::atomic Performance: Loads

- On X86 atomic loads are just loads (primitive types)
- Can be more expansive and cause locking on other systems

std::atomic Performance: Stores

- On X86 atomic stores use xchg, and it is a full barrier

std::atomic Spinlock Example

```
35 void spin_bool_foo(std::atomic<bool>& flag){  
36     while(flag.exchange(true)){  
37         //spin;  
38     }  
39     fmt::print("Worker Spin lock opened doing stuff \n");  
40     //doing stuff  
41     fmt::print("Worker releasing lock \n");  
42     flag = false;  
43 }
```

std::atomic Spinlock Example Cont

```
60  int main(){
61      std::atomic<bool> flag{true};
62      auto holder = std::thread([&flag](){
63          std::this_thread::sleep_for(1s);
64          fmt::print("Holder releasing lock \n");
65          flag = false;});
66
67      auto worker = std::thread(spin_bool_foo, std::ref(flag));
68
69      holder.join();
70      worker.join();
71  }
```

std::atomic Spinlock Example

```
Holder releasing lock
```

```
Worker Spin lock opened doing stuff
```

```
Worker releasing lock
```

`std::atomic<std::shared_ptr>`



A presentation slide for Cppcon 2023. The slide has a dark blue background with a large white graphic element on the left side. The title is "Lock-free Atomic Shared Pointers Without a Split Reference Count? It Can Be Done!" by Daniel Anderson. The date is October 01 - 06, 2023. The Cppcon logo is at the bottom left, and a small "i" icon is in the top right corner.

**Lock-free Atomic Shared Pointers
Without a Split Reference Count?
It Can Be Done!**

DANIEL ANDERSON

2023 | October 01 - 06

Cppcon
The C++ Conference

+ 23 i

Atomic Builtins

Atomic Builtins

- Most Compilers have implementations for C++11 compatible atomic operations
- GCC and Clang use the `__atomic` prefix

Volatile vs Atomics

Volatile vs Atomics

```
3  uint64_t sink;
4
5  uint64_t func_vol() {
6      constexpr volatile uint64_t *ptr = &sink;
7      return *ptr;
8  }
9
10 uint64_t func_atom() {
11     return __atomic_load_n(&sink, __ATOMIC_RELAXED);
12 }
```

X86-64 disassembly

```
1 func_vol():                                # @func_vol()
2     mov     rax, qword ptr [rip + sink]
3     ret
4 func_atom():                                # @func_atom()
5     mov     rax, qword ptr [rip + sink]
6     ret
7 sink:
8     .quad    0                                # 0x0
```

X86-64 32 bits disassembly

```
1 func_vol():                                # @func_vol()
2         call    .L0$pb
3 .L0$pb:
4         pop    eax
5 .Lttmp0:
6         add    eax, offset _GLOBAL_OFFSET_TABLE_+(_.Lttmp0-.L0$pb)
7         mov    edx, dword ptr [eax + sink@GOTOFF+4]
8         mov    eax, dword ptr [eax + sink@GOTOFF]
9         ret
```

X86-64 32 bits disassembly

```
10    func_atom():                                # @func_atom()
11        call    .L1$pb
12    .L1$pb:
13        pop     eax
14    .Ltmp3:
15        add     eax, offset _GLOBAL_OFFSET_TABLE_+(_.Ltmp3-.L1$pb)
16        movq    xmm0, qword ptr [eax + sink@GOTOFF] # xmm0 = mem[0],zero
17        movd    eax, xmm0
18        pshufd xmm0, xmm0, 85                      # xmm0 = xmm0[1,1,1,1]
19        movd    edx, xmm0
20        ret
21    sink:
22        .quad   0                                     # 0x0
```

RISC V-64 disassembly

```
1 func_vol():                                # @func_vol()
2 .Lpcrel_hi0:
3     auipc    a0, %pcrel_hi(sink)
4     addi    a0, a0, %pcrel_lo(.Lpcrel_hi0)
5     ld      a0, 0(a0)
6     ret
7 func_atom():                                # @func_atom()
8 .Lpcrel_hi1:
9     auipc    a0, %pcrel_hi(sink)
10    ld      a0, %pcrel_lo(.Lpcrel_hi1)(a0)
11    ret
12 sink:
13     .quad    0                                # 0x0
```

RISC V-32 disassembly

```
1 func_vol():                                # @func_vol()
2     lui    a0, %hi(sink)
3     lw     a1, %lo(sink+4)(a0)
4     lw     a0, %lo(sink)(a0)
5     ret
6 func_atom():                                # @func_atom()
7     addi   sp, sp, -16
8     sw     ra, 12(sp)                      # 4-byte Folded Spill
9     lui    a0, %hi(sink)
10    addi   a0, a0, %lo(sink)
11    li     a1, 0
12    call   __atomic_load_8
13    lw     ra, 12(sp)                      # 4-byte Folded Reload
14    addi   sp, sp, 16
15    ret
```

Fence Example

Relaxed

```
227     std::atomic<int> flag{0};  
228     uint64_t shared_val{};  
  
232     void IncsharedXtimeNoAquire(){  
233         for (int count{0}; count < 10000000; ) {  
234             int expected{};  
235             if (flag.compare_exchange_strong(expected, 1,  
236                         std::memory_order_relaxed)){  
237                 //lock successfull  
238                 shared_val++;  
239                 count++;  
240                 flag.store(0, std::memory_order_relaxed);  
241             }  
242         }  
243     }
```

Relaxed

```
1 1 IncsharedXtimeNoAquire():                      # @IncsharedXtimeNoAquire()
2      xor    ecx, ecx
3      mov    rdx, qword ptr [rip + shared_val]
4      mov    esi, 1
5      jmp    .LBB0_1
6 2 .LBB0_3:                                     #   in Loop: Header=BB0_1 Depth=1
7      cmp    ecx, 100000000
8      jge    .LBB0_4
9 3 .LBB0_1:                                     # =>This Inner Loop Header: Depth=1
10     xor   eax, eax
11     lock   cmpxchg dword ptr [rip + flag], esi
12     jne    .LBB0_3
13     inc    rdx
14     mov    qword ptr [rip + shared_val], rdx
15     inc    ecx
16     mov    dword ptr [rip + flag], 0
17     jmp    .LBB0_3
18 4 .LBB0_4:
19     ret
```

Relaxed

```
1 1 IncsharedXtimeNoAquire():                      # @IncsharedXtimeNoAquire()
2      xor    ecx, ecx
3      mov    rdx, qword ptr [rip + shared_val]
4      mov    esi, 1
5      jmp    .LBB0_1
6 2 .LBB0_3:                                     #   in Loop: Header=BB0_1 Depth=1
7      cmp    ecx, 100000000
8      jge    .LBB0_4
9 3 .LBB0_1:                                     # =>This Inner Loop Header: Depth=1
10     xor   eax, eax
11     lock   cmpxchg dword ptr [rip + flag], esi
12     jne    .LBB0_3
13     inc    rdx
14     mov    qword ptr [rip + shared_val], rdx
15     inc    ecx
16     mov    dword ptr [rip + flag], 0
17     jmp    .LBB0_3
18 4 .LBB0_4:
19     ret
```

Relaxed

```
1  \ IncsharedXtimeNoAquire():                      # @IncsharedXtimeNoAquire()
2      xor    ecx, ecx
3      mov    rdx, qword ptr [rip + shared_val]
4      mov    esi, 1
5      jmp    .LBB0_1
6  \ .LBB0_3:                                         #   in Loop: Header=BB0_1 Depth=1
7      cmp    ecx, 10000000
8      jge    .LBB0_4
9  \ .LBB0_1:                                         # =>This Inner Loop Header: Depth=1
10     xor   eax, eax
11     lock   cmpxchg dword ptr [rip + flag], esi
12     jne    .LBB0_3
13     inc    rdx
14     mov    qword ptr [rip + shared_val], rdx
15     inc    ecx
16     mov    dword ptr [rip + flag], 0
17     jmp    .LBB0_3
18  \ .LBB0_4:
19      ret
```

Relaxed

```
1 1 IncsharedXtimeNoAquire():                      # @IncsharedXtimeNoAquire()
2      xor    ecx, ecx
3      mov    rdx, qword ptr [rip + shared_val]
4      mov    esi, 1
5      jmp    .LBB0_1
6 2 .LBB0_3:                                     #   in Loop: Header=BB0_1 Depth=1
7      cmp    ecx, 10000000
8      jge    .LBB0_4
9 3 .LBB0_1:                                     # =>This Inner Loop Header: Depth=1
10     xor   eax, eax
11     lock   cmpxchg dword ptr [rip + flag], esi
12     jne    .LBB0_3
13     inc    rdx
14     mov    qword ptr [rip + shared_val], rdx
15     inc    ecx
16     mov    dword ptr [rip + flag], 0
17     jmp    .LBB0_3
18 4 .LBB0_4:
19     ret
```

Relaxed surprise

```
run0 shared_val = 10266917
run1 shared_val = 10258969
run2 shared_val = 10174515
run3 shared_val = 10001845
```

Acquire

```
246 void IncsharedXtimeAquire(){
247     for (int count{0}; count < 10000000;) {
248         int expected{};
249         if (flag.compare_exchange_strong(expected, 1,
250             std::memory_order_acquire)){
251             //lock successfull
252             shared_val++;
253             count++;
254             flag.store(0, std::memory_order_release);
255         }
256     }
257 }
```

Acquire

```
20 IncsharedXtimeAquire():                      # @IncsharedXtimeAquire()
21     xor    ecx, ecx
22     mov    edx, 1
23     jmp    .LBB1_1
24 .LBB1_3:                                     #  in Loop: Header=BB1_1 Depth=1
25     cmp    ecx, 10000000
26     jge    .LBB1_4
27 .LBB1_1:                                     # =>This Inner Loop Header: Depth=1
28     xor    eax, eax
29     lock   cmpxchg dword ptr [rip + flag], edx
30     jne    .LBB1_3
31     inc    qword ptr [rip + shared_val]          # Boxed instruction
32     inc    ecx
33     mov    dword ptr [rip + flag], 0
34     jmp    .LBB1_3
35 .LBB1_4:
36     ret
```

Acquire No Surprises

```
run0 shared_val = 20000000  
run1 shared_val = 20000000  
run2 shared_val = 20000000  
run3 shared_val = 20000000
```

RISC V Disassembly

```
1  IncsharedXtimeNoAquire():
2      li      a4,9998336
3      addi    a4,a4,1664
4      lui     a3,%hi(flag)
5      lui     a2,%hi(shared_val)
6      li      a1,1
7  .L6:
8      addi    a0,a3,%lo(flag)
9      1: lr.w  a5,0(a0); bne a5,zero,1f; sc.w a6,a1,0(a0); bnez a6,1b; 1:
10     sext.w a5,a5
11     bnez   a5,.L6
12     ld     a5,%lo(shared_val)(a2)
13     addi   a5,a5,1
14     sd     a5,%lo(shared_val)(a2)
15     addi   a5,a3,%lo(flag)
16     amoswap.w zero,zero,0(a5)
17     addiw  a4,a4,-1
18     bnez   a4,.L6
19     ret
```

RISC V Disassembly

```
148      IncsharedXtimeAquire():
149          li      a4,9998336
150          addi    a4,a4,1664
151          lui     a3,%hi(flag)
152          lui     a2,%hi(shared_val)
153          li      a1,1
154      .L27:
155          addi    a0,a3,%lo(flag)
156          1: lr.w.aq a5,0(a0); bne a5,zero,1f; sc.w.aq a6,a1,0(a0); bnez a6,1b; 1:
157          sext.w  a5,a5
158          bnez    a5,.L27
159          ld      a5,%lo(shared_val)(a2)
160          addi    a5,a5,1
161          sd      a5,%lo(shared_val)(a2)
162          addi    a5,a3,%lo(flag)
163          fence   iorw,ow; amoswap.w zero,zero,0(a5)
164          addiw   a4,a4,-1
165          bnez    a4,.L27
166          ret
```

Other STL Concurrency Features

std::future and std::promise

- `std::future<T>` - value for which you may wait
- `std::promise<T>` - produces a future

```
80 int main(){
81     std::promise<int> p;
82     std::thread t1([&p](){
83         std::this_thread::sleep_for(1s);
84         p.set_value(42);
85     });
86
87     std::thread t2([](auto f){
88         fmt::print("The value is {}", f.get());}, p.get_future());
89
90     t1.join();
91     t2.join();
92 }
```

Futures and Promises

- Can help communication between threads
- Help build task-oriented utilities for executing work on different threads
- Future and promise are one shoot operation

Jthread

- Same as `std::thread` but is joinable by default
- `Jthread` is stoppable with `std::stop_source`
- Provides easier implementation where user don't have to think about joins

Synchronization Tools

- std::mutex, std::conditional_variable
- std::lock_guard – RAII helper for locking

```
126 int main(){
127     std::mutex m;
128     auto doer = [&m](int idx){
129         std::lock_guard l(m);
130         fmt::print("doer{} thread doing stuff\n", idx);
131     };
132     std::jthread t1(doer, 0);
133     std::jthread t2(doer, 1);
134 }
```

Synchronization Tools

- `std::unique_lock` – same as lock guard but unique and can be deferred.
- `std::scoped_lock` – takes ownership of multiple locks at once with RAII with deadlock avoidance algorithm

Synchronization Tools

- `std::counting_semaphore` – semaphore that can be set to an arbitrary number.
- `Std::binary_semaphore` – same as `counting_semaphore` but it's set to 1.

Synchronization Tools

Synchronization Tools : stop_source

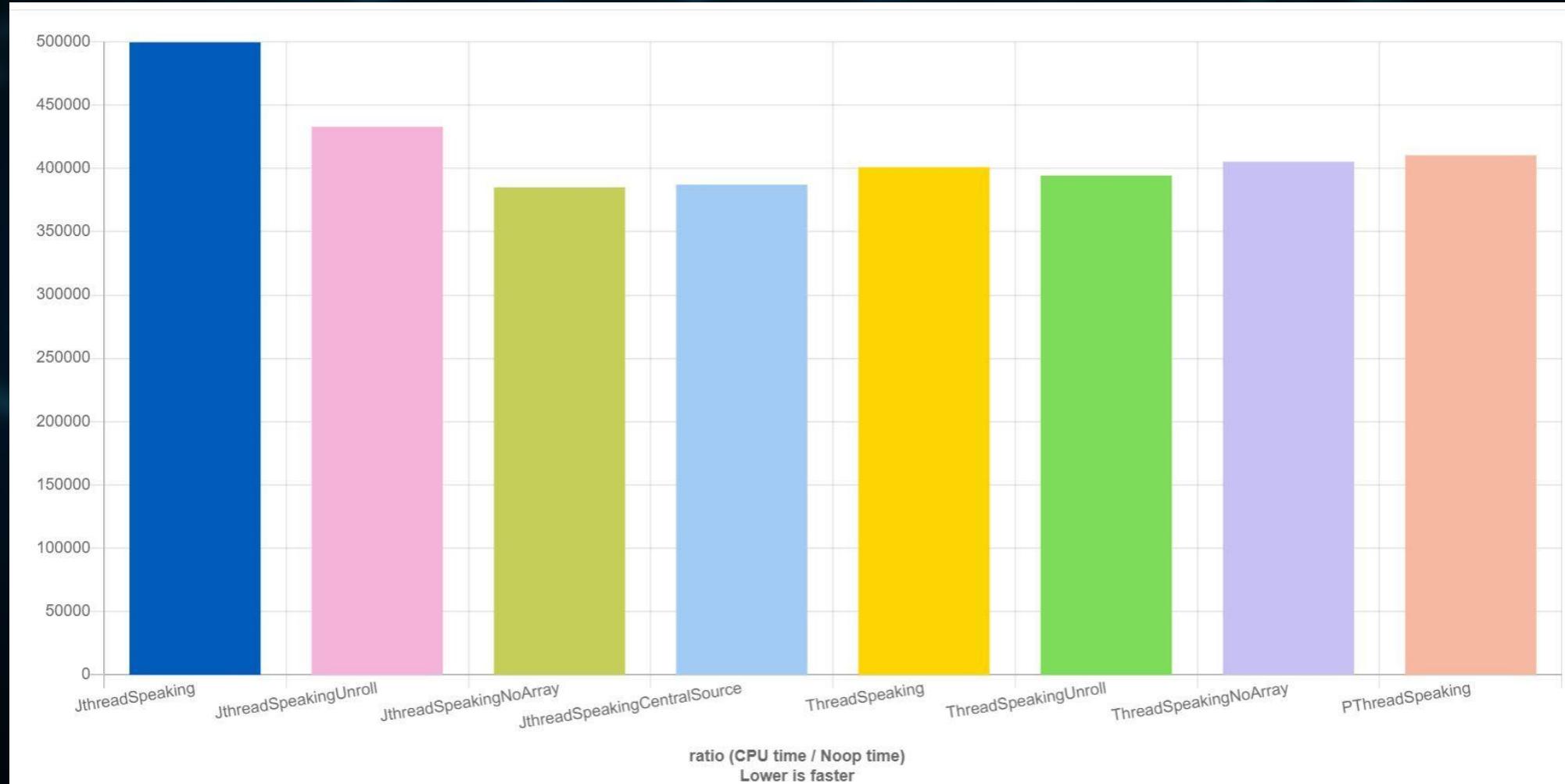
```
8  std::binary_semaphore start_speaking{0}, p1_done{0}, p2_done{0};
9
10 void person_speak(std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r, std::stop_token s){
11     w.acquire();
12     while (not s.stop_request) {
13         fmt::print("{}\n", text);
14         std::this_thread::sleep_for(std::chrono::milliseconds(100));
15     }
16     r.release();
17 }
18
19 void stop_speaking(std::array<std::stop_token> ssr)
20 {
21     for (auto i = 0u; i < ssr.size(); ++i)
22         std::this_thread::sleep_for(std::chrono::milliseconds(100));
23         ssr[i].request_stop();
24 }
25
26
27 int main() {
28     std::array ssrcs{std::stop_token{}};
29     std::jthread p1(person_speak, std::ref(ssrcs[0]));
30     std::jthread p2(person_speak, std::ref(ssrcs[1]));
31     std::jthread stop_speaking(stop_speaking, ssrcs);
32     std::ref(p1_done), std::ref(p2_done), ssrcs[1].get_token());
33     start_speaking.release();
34     std::jthread sched(stop_speaking, ssrcs);
35     p2_done.acquire();
36     fmt::print("Goodby!\n");
37 }
```

Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson1 im Person2
Hello Peson1 im Person2
Hello Peson1 im Person2
Hello Peson1 im Person2
Goodby!

Synchronization Tools : stop_source

```
10    std::binary_semaphore startSpeaking{0}, p1_done{0}, p2_done{0};
11
12    void person_speak(std::stop_token s, std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r){
13        w.acquire();
14        while (not s.stop_requested()){
15            fmt::print("{}\n", text);
16            std::this_thread::sleep_for(1s);
17        }
18        r.release();
19    }
20
21    void stopSpeaking(std::array<std::stop_source, 2> ssrc){
22        for (auto i = 0u; i < ssrc.size(); i++){
23            std::this_thread::sleep_for(2s);
24            ssrc[i].request_stop();
25        }
26    }
27
28
29    int main() {
30        std::jthread p1(person_speak, "Hello Person2 im Person1\n",
31                      std::ref(startSpeaking), std::ref(p1_done));
32        std::jthread p2(person_speak, "Hello Person1 im Person2\n",
33                      std::ref(p1_done), std::ref(p2_done));
34        startSpeaking.release();
35        std::jthread sched(stopSpeaking, std::array{p1.get_stop_source(), p2.get_stop_source()});
36        p2_done.acquire();
37        fmt::print("Goodby!\n");
38    }
```

Std::threads : Micro Benchmark



Synchronization Tools - cont

- `Std::latch`– threads may block on latch until its value is zero, latch is one shot.
- `std::barrier`– very similar to latch but can be used multiple times .

Synchronization Tools – Latch Example

```
139 int main(){
140     std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
141     std::vector<std::jthread> workers;
142     std::latch jobs{ppl.size()}, go_home{1};
143     auto job = [&jobs, &go_home](std::string_view name){
144         fmt::print("{} is doing a job\n", name);
145         jobs.count_down();
146         go_home.wait();
147         fmt::print("{} is going home\n", name);
148     };
149
150     for (const auto& p : ppl) {
151         workers.push_back(std::jthread(job, p));
152     }
153     jobs.wait();
154     fmt::print("Go home!\n");
155     go_home.count_down();
156 }
```

Synchronization Tools – Latch Example

```
Alex is doing a job  
Dani is doing a job  
Guy is doing a job  
Benny is doing a job  
Go home!  
  
Alex is going home  
Dani is going home  
Guy is going home  
Benny is going home
```

Synchronization Tools – Barrier Example

```
162 int main(){
163     std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
164     std::vector<std::jthread> workers;
165     auto on_complete = [](){
166         static int x = 0;
167         not x++ ? fmt::print("Go Home!\n") : fmt::print("Done!\n"); };
168     std::barrier sync{ppl.size(), on_complete};
169     auto job = [&sync](std::string_view name){
170         fmt::print("{} is doing a job\n", name);
171         sync.arrive_and_wait();
172         fmt::print("{} is going home\n", name);
173         sync.arrive_and_wait();
174     };
175
176     for (const auto& p : ppl) {
177         workers.push_back(std::jthread(job, p));
178     }
179 }
```

Synchronization Tools – Barrier Example

Dani is doing a job

Guy is doing a job

Benny is doing a job

Alex is doing a job

Go Home!

Benny is going home

Alex is going home

Dani is going home

Guy is going home

Done!

Synchronization Tools

- `std::shared_mutex` – may be used by couple of threads or be exclusive.
- `std::timed_mutex` – same as mutex but has a claim time out.
- `std::shared_timed_mutex` – combination of shared and timed mutexes.
- `std::shared_lock` – RAII wrapper for timed and shared mutexes

Synchronization Tools

```
179 int val = 10;
180 int main(){
181     std::shared_timed_mutex stm;
182     auto access_and_print = [&stm](int v){
183         std::shared_lock l(stm, 1s);
184         int my_val = val + v;
185         fmt::print("val {}{}\n", v);
186     };
187
188     std::jthread t1(access_and_print, 1);
189     std::jthread t2(access_and_print, 2);
190 }
```

Synchronization Tools

- Static: from C++11 static variables initialization is magic

```
struct X{
    int x;
};

get_once(int):
    movzx   eax, byte ptr [rip + guard variable for get_once(int)::single]
    test    al, al
    je     .LBB3_1
    lea     rax, [rip + get_once(int)::single]
    ret

    std::jthread t1([](){auto& x = get_once(10);}),
    t2([](){auto& x = get_once(11);});
}
```

Synchronization Tools

- `std::call_once` and `std::once_flag` – help to do something just once.

```
200 int main(){
201     std::once_flag flag;
202     auto f = [&flag](){
203         static int x{10};
204         std::call_once(flag, [](){x++;});
205         fmt::print("{}\n", x);
206     };
207
208     std::jthread t1(f);
209     std::jthread t2(f);
210 }
```

11

11

Synchronization Tools

- `Std::packaged_task` – wraps a callable into a thread and returns a future

```
212 int main(){
213     std::packaged_task<sum_task> sum_task([](int a, int b){
214         return a+b; });
215
216     std::packaged_task<mul_task> mul_task([](int a, int b){ return a*b; });
217
218     mul_task(10, 2);
219     sum_task(10, mul_task.get_future().get());
220
221     fmt::print("tot: {}", sum_task.get_future().get());
222
223     return 0;
224 }
```

QUESTIONS



THANK YOU FOR LISTENING

ALEX DATSKOVSKY

+97254-7685001

ALEX.DATSKOVSKY@SPEEDATA.IO

[WWW.LINKEDIN.COM/IN/ALEXDATHSKOVSKY](https://www.linkedin.com/in/alexdatshovsky)

<https://www.cppnext.com>