C++ now

C++ Memory Model

from C++11 to C++23

Alex Dathskovsky

2024

About Me:

SPEEDATA

alex.datskovsky@speedata.io

www.linkedin.com/in/alexdathskovsky

https://www.cppnext.com

Important Question

Does the processor executes the program as you wrote it?

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

If you were the compiler, what would you do?

```cpp
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 know in compile time

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

Algebraic Simplifications

(v + 1) + 3 ->

v + (1+3) ->

v + 4

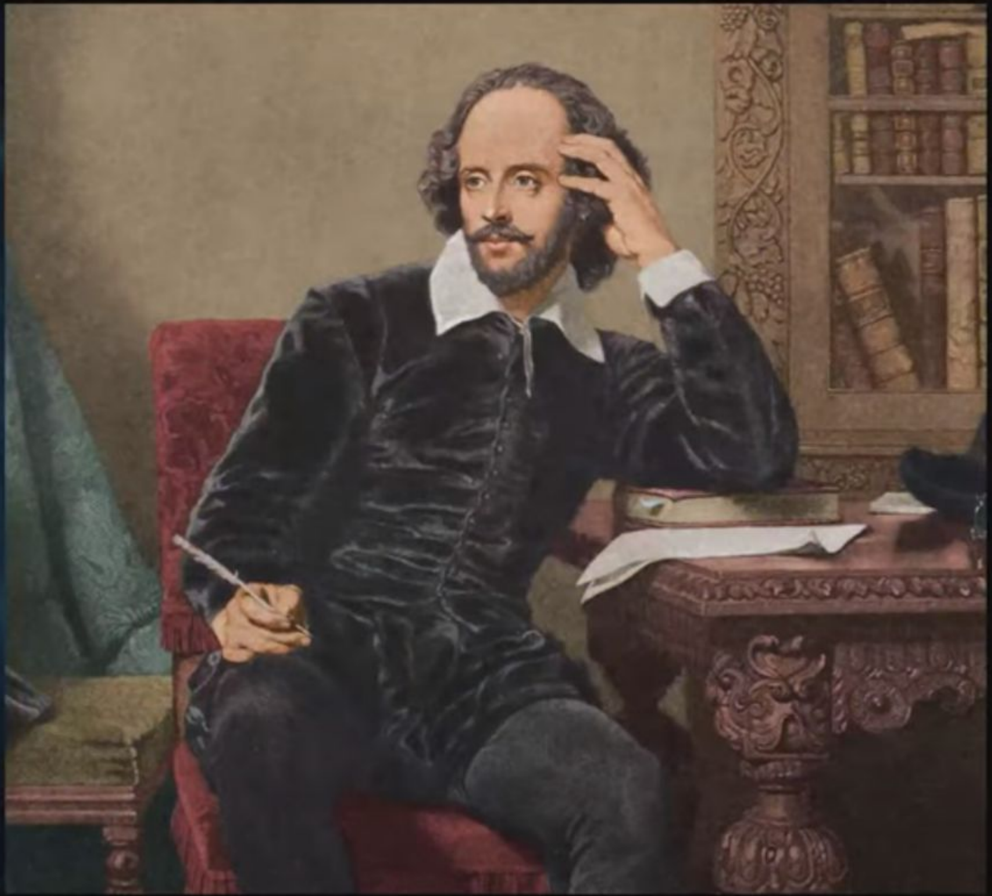
Strength Reduction

```
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 UINT

ALEX DATHSKOVSKY



Core C++
2023

Bloomberg
Engineering
millennium
think-cell

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;
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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

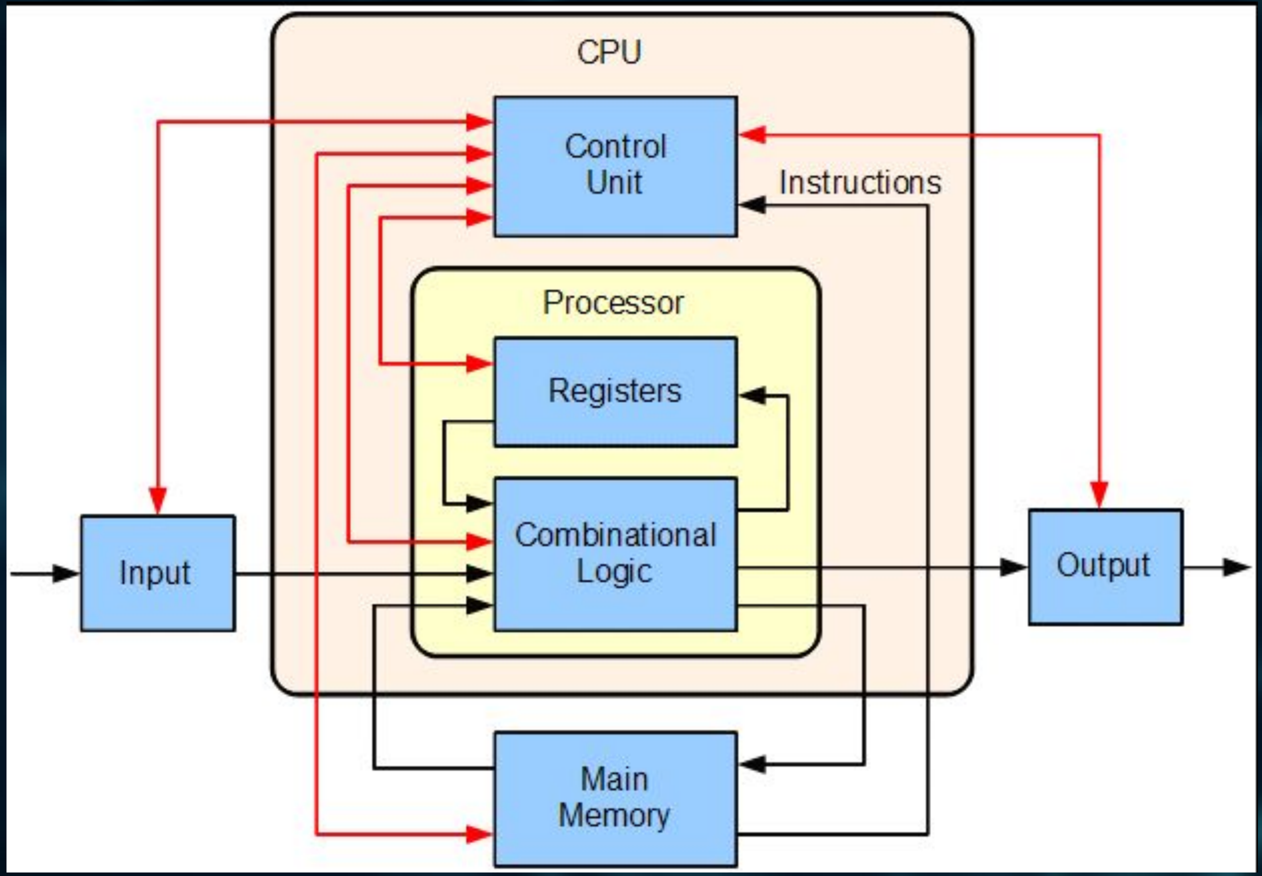
```
add(int, int):
        lea     eax, [rdi + rsi]
        ret
main:
        lea     eax, [2*rdi + 10]
        ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

CPU Optimisations

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

INO Execution



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

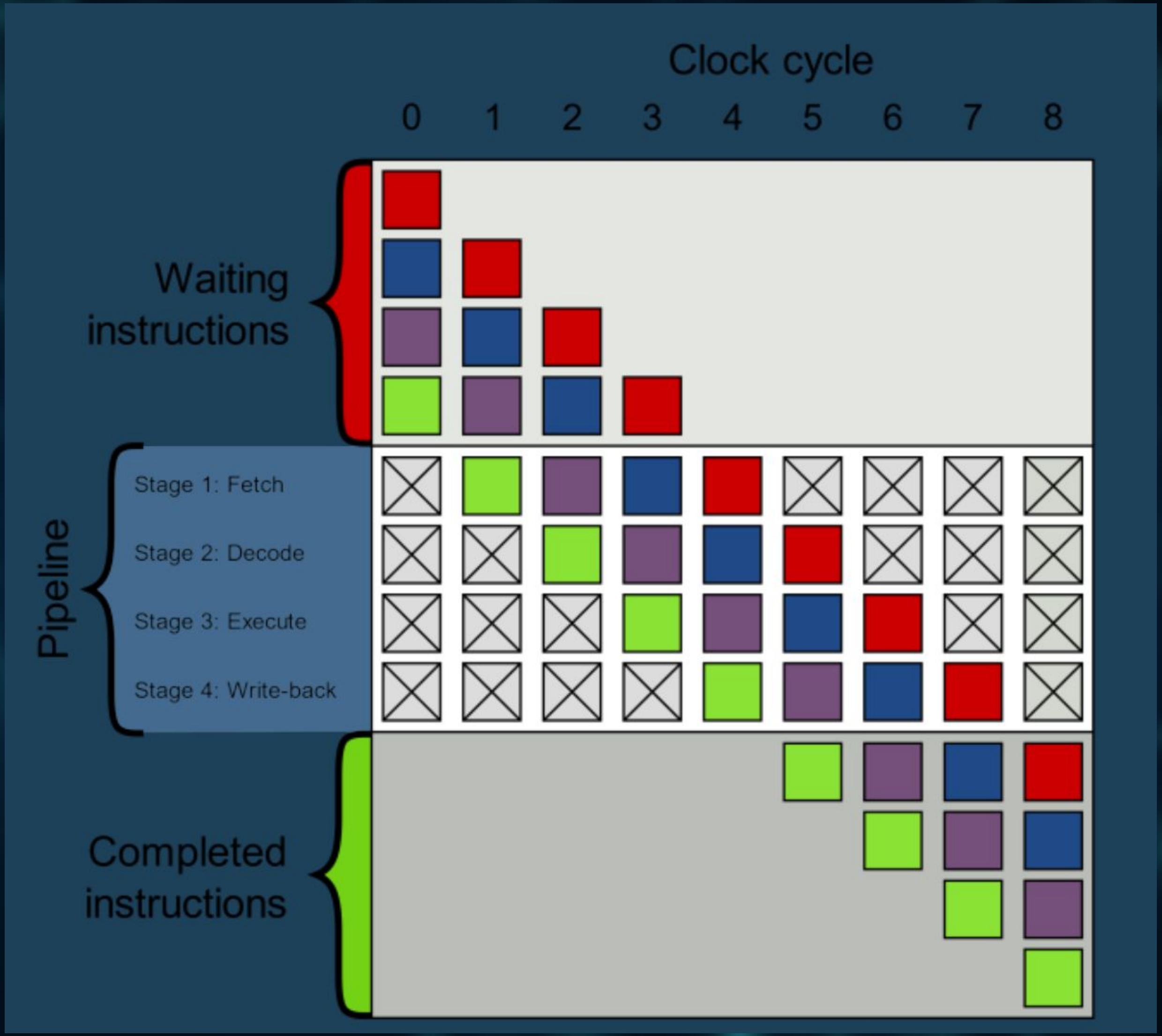
High Speed Query Execution with Accelerators and C++

ALEX DATHSKOVSKY

Cppcon
The C++ Conference

2022
September 12th-16th

INO Execution

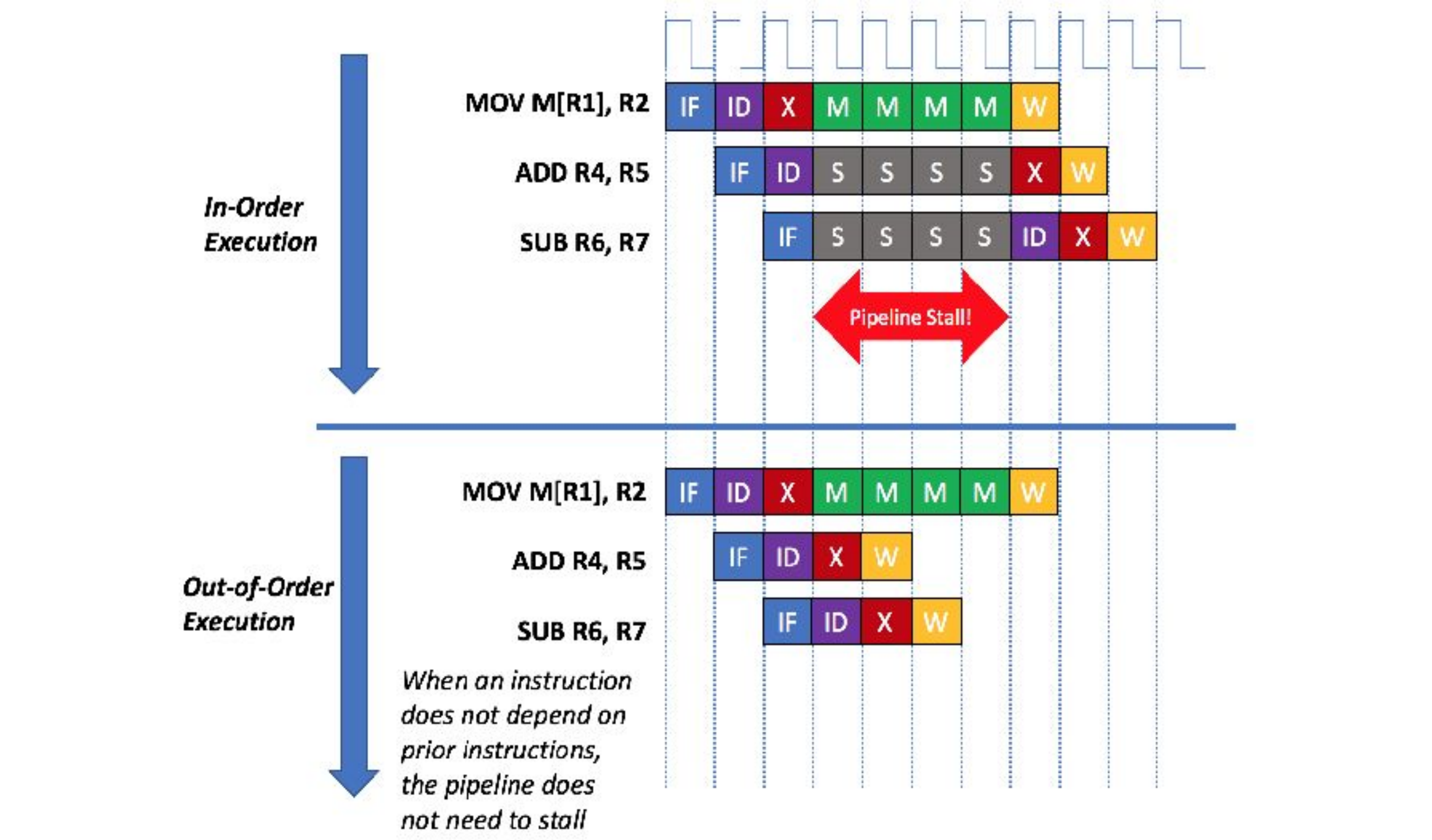


Source: https://en.wikipedia.org/wiki/Instruction_pipelining

INO Execution

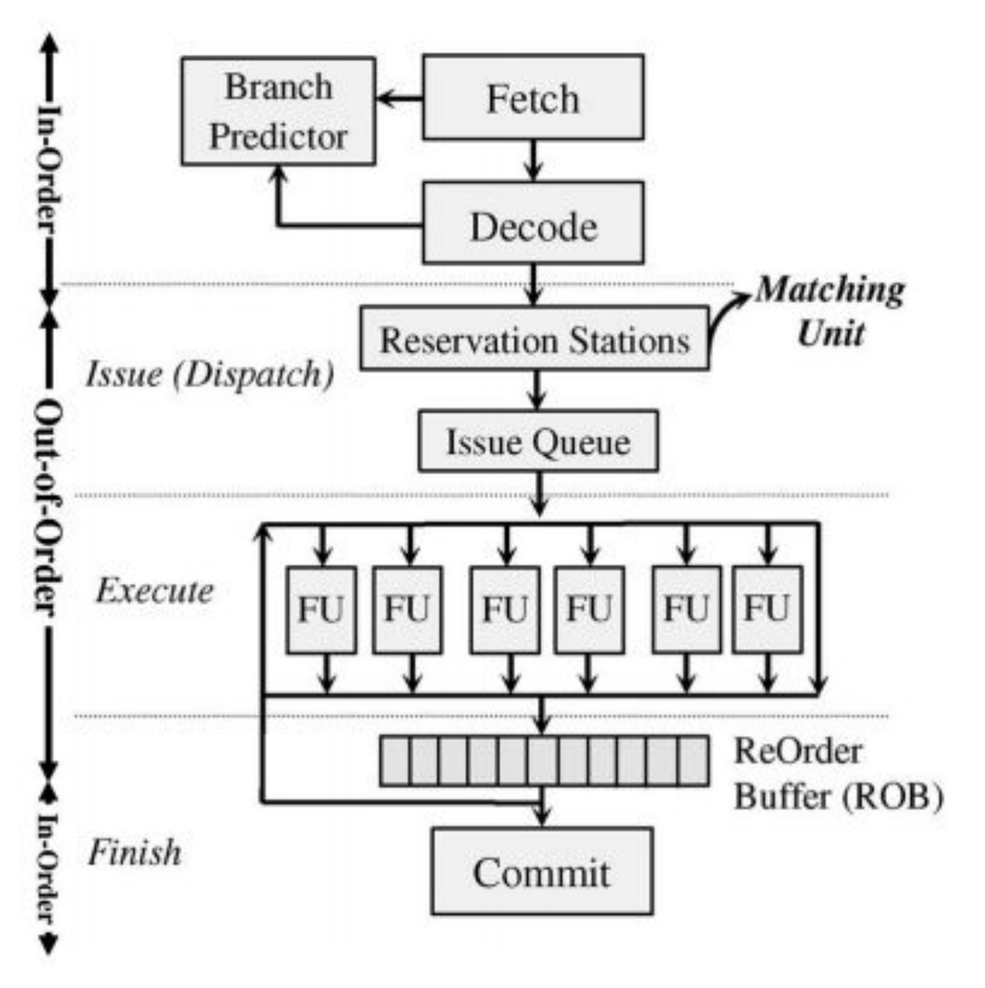
- instruction fetch
- if operands available execute it if not fetch them
- The instruction is executed by the functional unit
- The functional unit writers the result back to the register or memory

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 untils 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

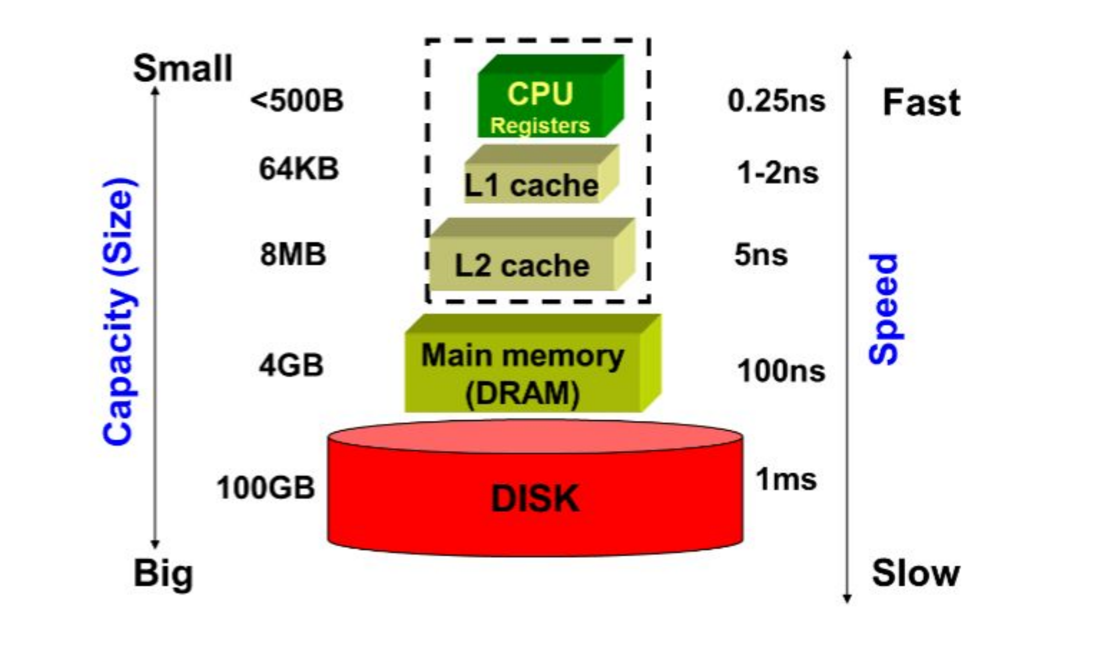
- **Check for data hazards**
  - an instruction can be executed when its operands have been calculated or loaded from memory
  - an instruction stalls if operands are not available

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



CPU registers are the highest level of memory, part of the ISA

Reordering and C++

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Reordering Types

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

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

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Reordering between two threads

Thread 1	Thread 2
flag1 = 1;	flag2 = 1;
if (flag2 == 0){	if (flag1 == 0){
critical section}	critical section}

Sequential Consistency

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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	Thread 2
Foo(Flag)	
S(a)	S(a)
Bar(Flag)	Bar(Flag)

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Pop Quiz: can this happen?

Thread 1	Thread 2	Thread 3	Thread 4
x = 1	x = 2	y1 = x	y3 = x
		y2 = 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

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky


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

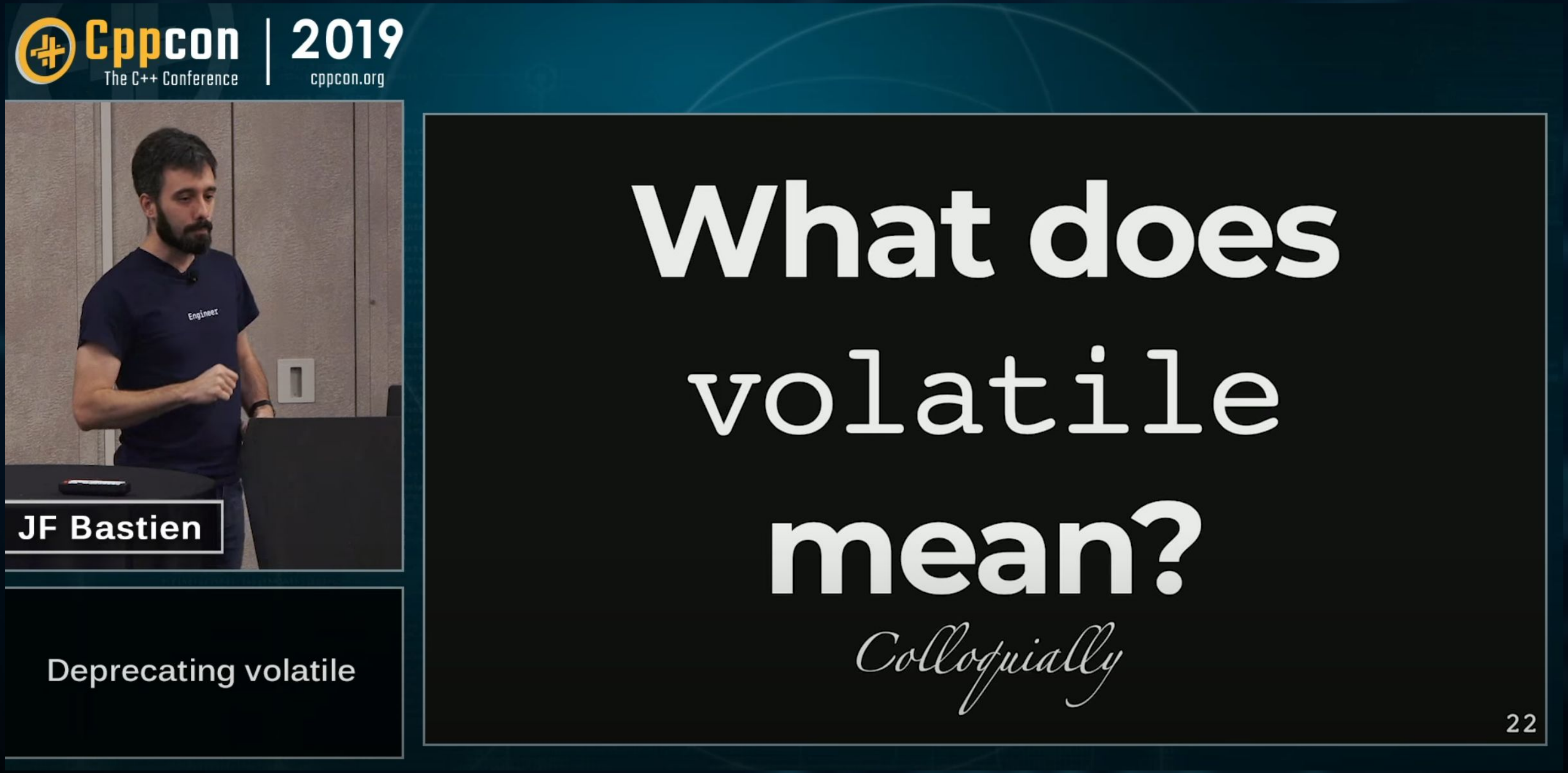
```
int main(){
    int y = 1;
    volatile int* x = &y;

    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



Synchronization cont

Compiler Code Barriers

Prevent compiler from moving reads or writes across the barrier

Compiler Code Barriers

Prevent compi eads or writes across the bar

Thread 1

value = very_long_calc

asm volatile(“mfence” :

done = true

ead 2

le (not done){}

volatile(“mfence” ::: “memory”)

something

Locks and Atomics

Most Locks and atomic operation will act like barriers

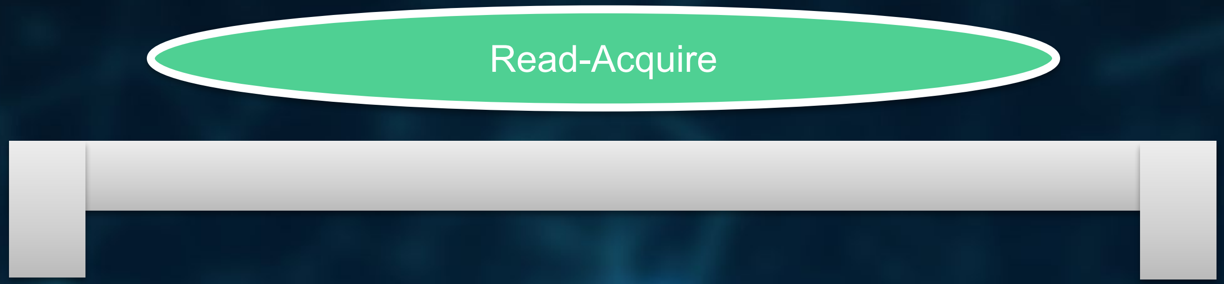
Understanding Memory Barriers / Memory Ordering

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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



All memory operations
stay below the line

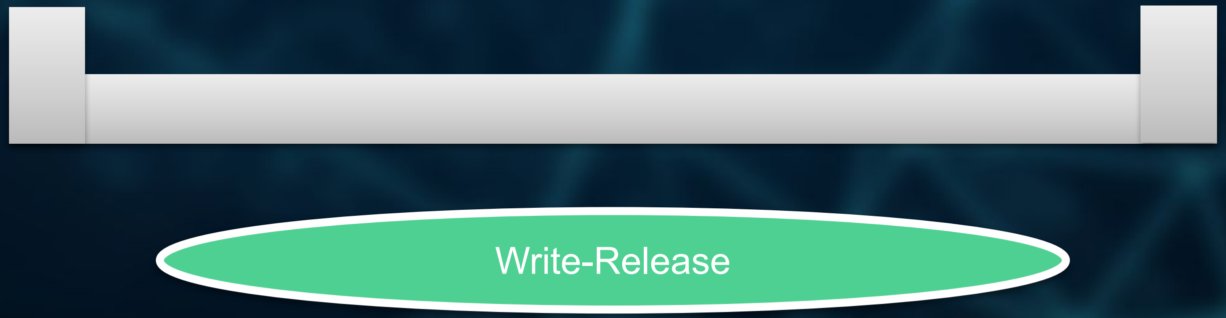
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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.*

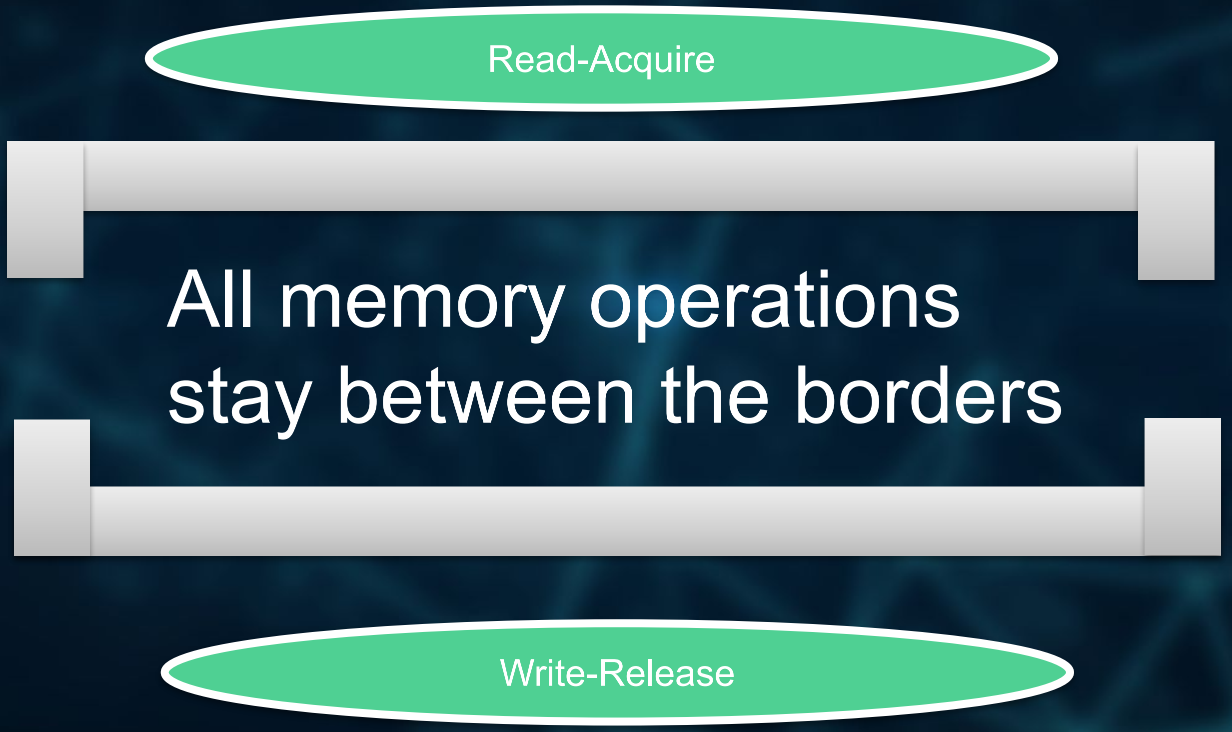
Release Semantics

All memory operations
stay above the line



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Memory Barrier / Acquire Release Order



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

What's The Difference?

Thread 1	Thread 2	Thread 3	Thread 4
x = sr(1)	y = sr(2)	y1 = la(x)	y3 = la(y)
		y2 = la(y)	y4 = la(x)

sr = store_release
la = load_acquire

T3: y1 = 1 y2 = 0
T4: y3 = 2 y4 = 0

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

What's The Difference?

Thread 1	Thread 2	Thread 3	Thread 4
x = ss(1)	y = ss(2)	y1 = ls(x)	y3 = ls(y)
		y2 = ls(y)	y4 = ls(x)

ss = store_seq_cst
ls = load_seq_cst

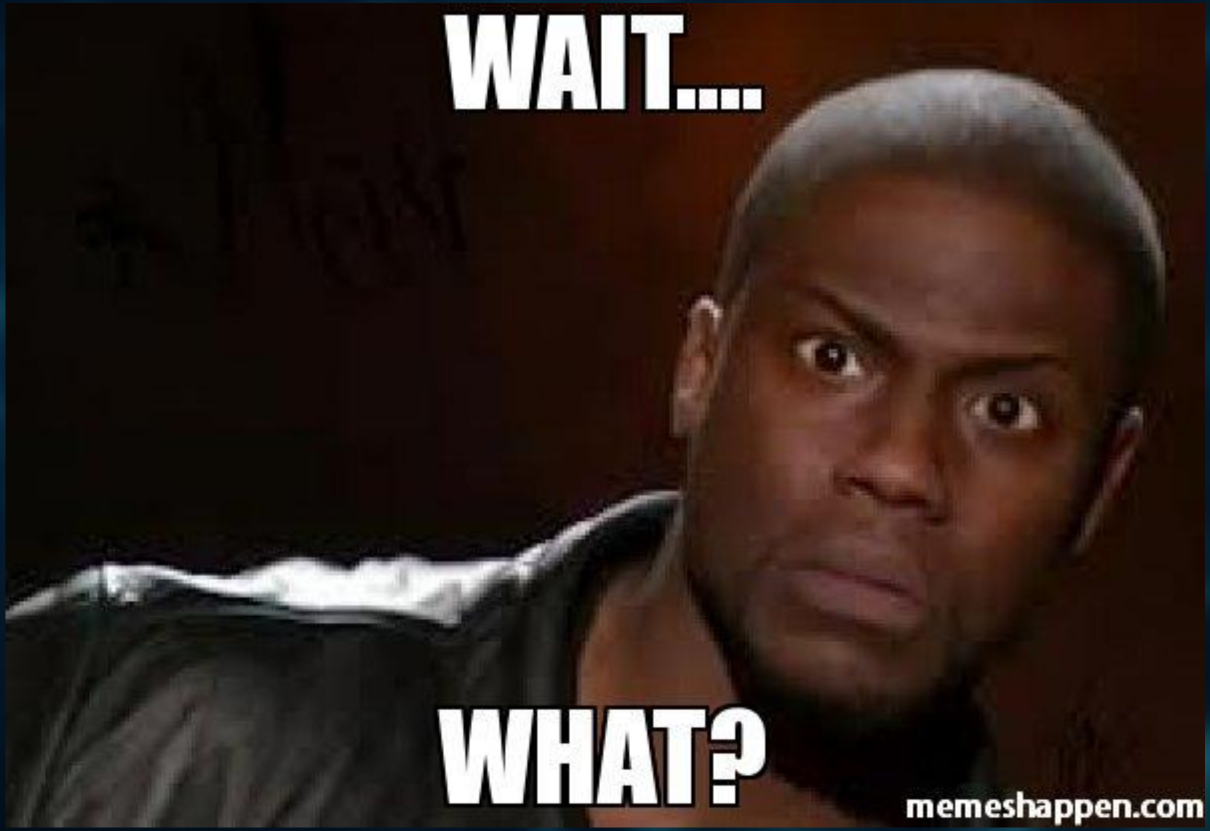
T3: y1 = 1 y2 = 2
T4: y1 = 1 y2 = 2

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]
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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{flase}
x = 0

Thread 1
x = 1
a.s(true)

Thread 2
z = a.lc()
if (z) y = x

FAIL

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Good Example

a = atomic_int*_value{null}

Thread 1

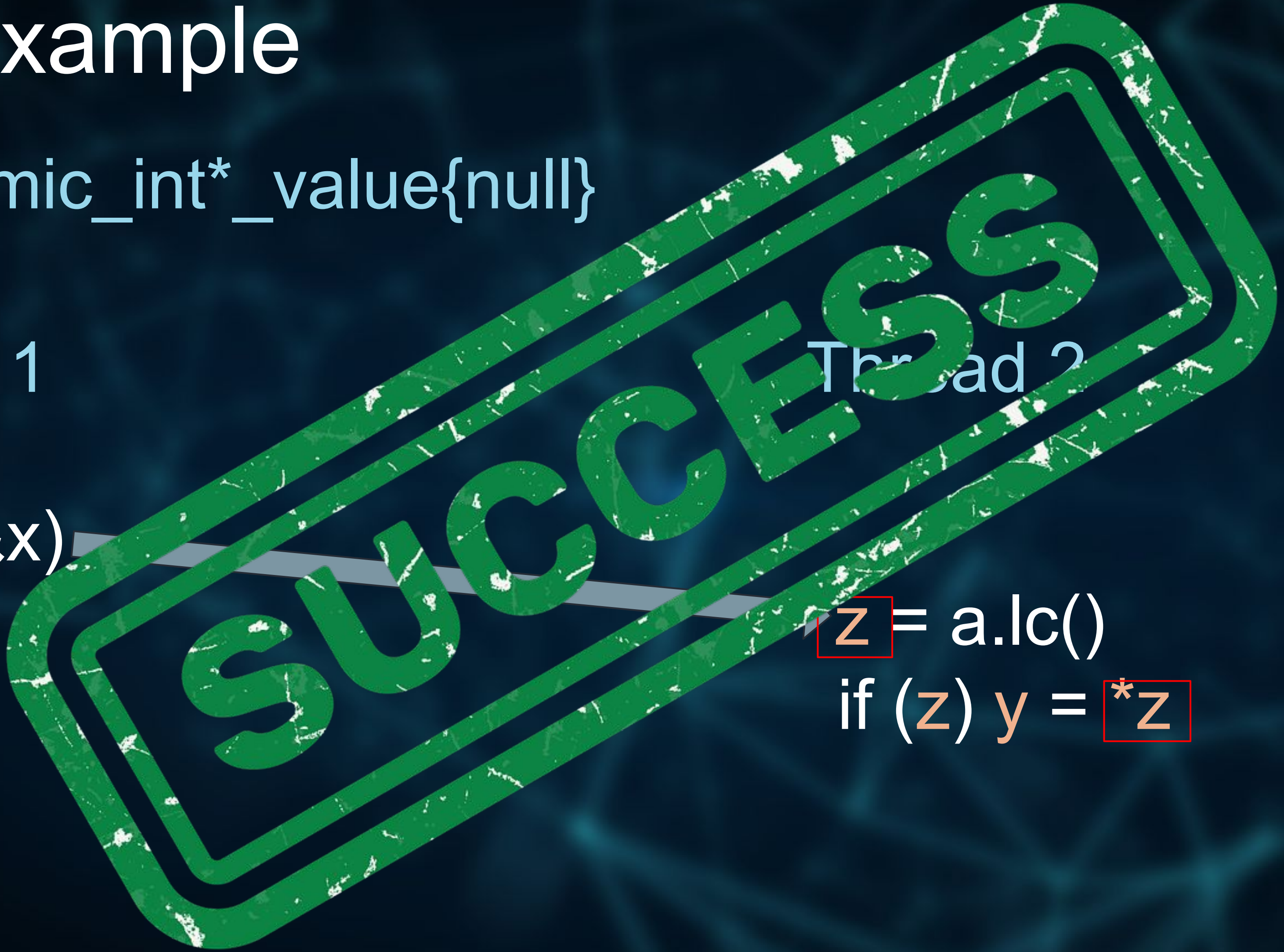
x = 1

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)

What may happen here ?

```
Y = A2.load(acquire)

A1.store(4, release)
```

C++ Concurrency tools

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

std::thread

- Cross platform thread class

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

thread_local

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

```cpp
thread_local uint64_t inc = 0;

void do_count(std::string_view name, int count, uint64_t& ret){
    for
        
    }

    ret
}

int mai
    uin
    std
    std
    a.join(); b.join();
    fmt::print("ret a: {}\nret b: {}\nmain inc: {}\n", ra, rb, inc);
}
```

```
ret a: 10
ret b: 20
main inc: 0
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

thread_local

- Please don't abuse it

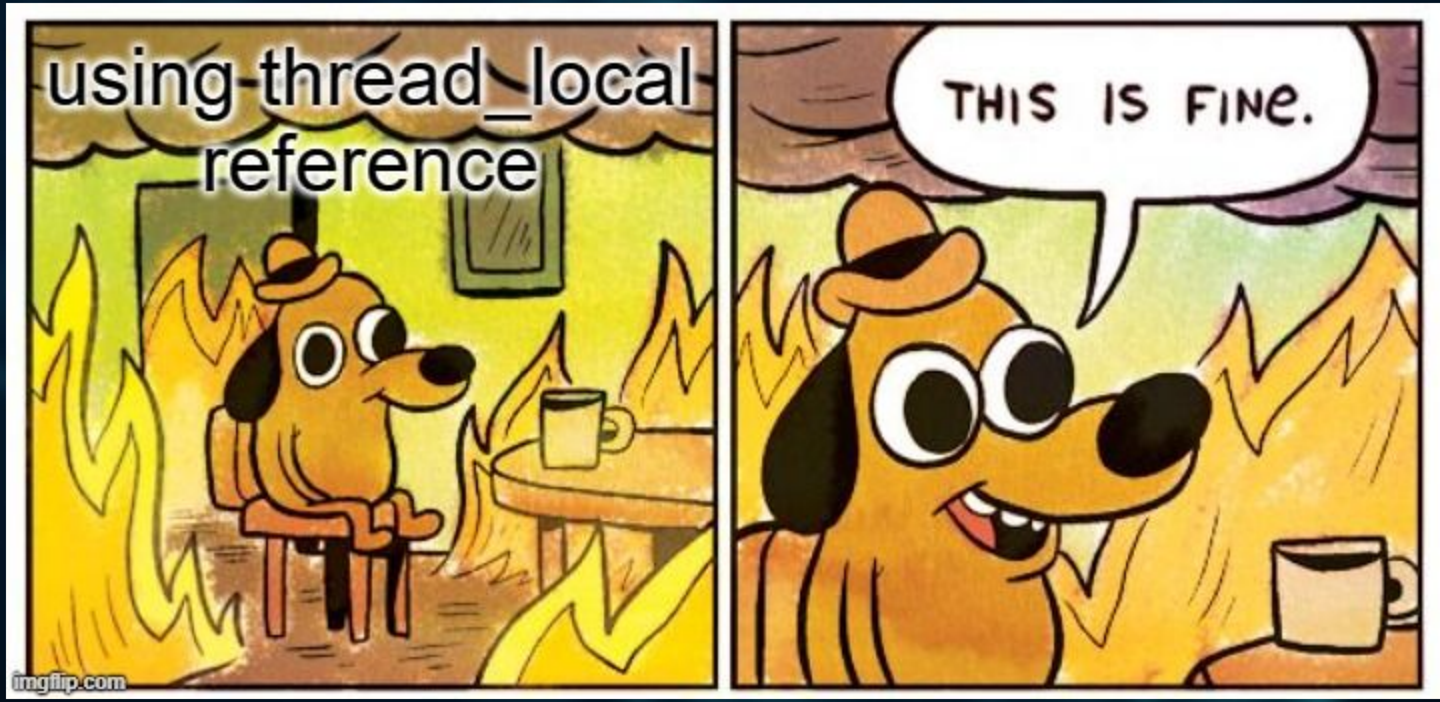
```cpp
thread_local uint64_t inc = 0;

void do_count(std::string_view name, int count,
              uint64_t& ret, uint64_t* val){
    for
    
    }

    ret
}

int mai
    uin
    std                                                      c);
    std::thread b(do_count, "A", 20, std::ref(rb), &inc);
    a.join(); b.join();
    fmt::print("ret a: {}\nret b: {}\nmain inc: {}\n", ra, rb, inc);
}
```

```
ret a: 30
ret b: 20
main inc: 30
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

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

```cpp
void spin_bool_foo(std::atomic<bool>& flag){
    while(flag.exchange(true)){
        //spin;
    }
    fmt::print("Worker Spin lock opened doing stuff \n");
    //doing stuff
    fmt::print("Worker releasing lock \n");
    flag = false;
}
```

std::atomic Spinlock Example Cont

```
int main(){
    std::atomic<bool> flag{true};
    auto holder = std::thread([&flag](){
        std::this_thread::sleep_for(1s);
        fmt::print("Holder releasing lock \n");
        flag = false;});

    auto worker = std::thread(spin_bool_foo, std::ref(flag));

    holder.join();
    worker.join();
}
```

std::atomic Spinlock Example

```
Holder releasing lock
Worker Spin lock opened doing stuff
Worker releasing lock
```

std::atomic<std::shared_ptr>

Atomic Builtins

Atomic Builtins

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

Volatile vs Atomics

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Volatile vs Atomics

```
uint64_t sink;

uint64_t func_vol() {
    constexpr volatile uint64_t *ptr = &sink;
    return *ptr;
}

uint64_t func_atom() {
    return __atomic_load_n(&sink, __ATOMIC_RELAXED);
}
```

X86-64 disassembly

```
func_vol():                             # @func_vol()
        mov     rax, qword ptr [rip + sink]
        ret
func_atom():                            # @func_atom()
        mov     rax, qword ptr [rip + sink]
        ret
sink:
        .quad   0                                  # 0x0
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

X86-64 32 bits disassembly

```
func_vol():                                  # @func_vol()
        call    .L0$pb
.L0$pb:
        pop     eax
.Ltmp0:
        add     eax, offset _GLOBAL_OFFSET_TABLE_+(.Ltmp0-.L0$pb)
        mov     edx, dword ptr [eax + sink@GOTOFF+4]
        mov     eax, dword ptr [eax + sink@GOTOFF]
        ret
```

X86-64 32 bits disassembly

```
func_atom():                                    # @func_atom()
        call    .L1$pb
.L1$pb:
        pop     eax
.Ltmp3:
        add     eax, offset _GLOBAL_OFFSET_TABLE_+(.Ltmp3-.L1$pb)
        movq    xmm0, qword ptr [eax + sink@GOTOFF] # xmm0 = mem[0],zero
        movd    eax, xmm0
        pshufd  xmm0, xmm0, 85                  # xmm0 = xmm0[1,1,1,1]
        movd    edx, xmm0
        ret
sink:
        .quad   0                               # 0x0
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

RISC V-64 disassembly

```
func_vol():                                     # @func_vol()
.Lpcrel_hi0:
        auipc   a0, %pcrel_hi(sink)
        addi    a0, a0, %pcrel_lo(.Lpcrel_hi0)
        ld      a0, 0(a0)
        ret
func_atom():                                    # @func_atom()
.Lpcrel_hi1:
        auipc   a0, %pcrel_hi(sink)
        ld      a0, %pcrel_lo(.Lpcrel_hi1)(a0)
        ret
sink:
        .quad   0                                         # 0x0
```

RISC V-32 disassembly

```
func_vol():                            # @func_vol()
        lui     a0, %hi(sink)
        lw      a1, %lo(sink+4)(a0)
        lw      a0, %lo(sink)(a0)
        ret
func_atom():                           # @func_atom()
        addi    sp, sp, -16
        sw      ra, 12(sp)                       # 4-byte Folded Spill
        lui     a0, %hi(sink)
        addi    a0, a0, %lo(sink)
        li      a1, 0
        call    __atomic_load_8
        lw      ra, 12(sp)                       # 4-byte Folded Reload
        addi    sp, sp, 16
        ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Fence Example

Relaxed

```
std::atomic<int> flag{0};
uint64_t shared_val{};
```

```
void IncsharedXtimeNoAquire(){
    for (int count{0}; count < 10000000;) {
        int expected{};
        if (flag.compare_exchange_strong(expected, 1,
        std::memory_order_relaxed)){
            //lock successfull
            shared_val++;
            count++;
            flag.store(0, std::memory_order_relaxed);
        }
    }
}
```

Relaxed

```
IncsharedXtimeNoAquire():                       # @IncsharedXtimeNoAquire()
        xor     ecx, ecx
        mov     rdx, qword ptr [rip + shared_val]
        mov     esi, 1
        jmp     .LBB0_1
.LBB0_3:                                        #   in Loop: Header=BB0_1 Depth=1
        cmp     ecx, 10000000
        jge     .LBB0_4
.LBB0_1:                                        # =>This Inner Loop Header: Depth=1
        xor     eax, eax
        lock            cmpxchg dword ptr [rip + flag], esi
        jne     .LBB0_3
        inc     rdx
        mov     qword ptr [rip + shared_val], rdx
        inc     ecx
        mov     dword ptr [rip + flag], 0
        jmp     .LBB0_3
.LBB0_4:
        ret
```

Relaxed

```
IncsharedXtimeNoAquire():                # @IncsharedXtimeNoAquire()
        xor     ecx, ecx
        mov     rdx, qword ptr [rip + shared_val]
        mov     esi, 1
        jmp     .LBB0_1
.LBB0_3:                                 #   in Loop: Header=BB0_1 Depth=1
        cmp     ecx, 10000000
        jge     .LBB0_4
.LBB0_1:                                 # =>This Inner Loop Header: Depth=1
        xor     eax, eax
        lock            cmpxchg dword ptr [rip + flag], esi
        jne     .LBB0_3
        inc     rdx
        mov     qword ptr [rip + shared_val], rdx
        inc     ecx
        mov     dword ptr [rip + flag], 0
        jmp     .LBB0_3
.LBB0_4:
        ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Relaxed

```
IncsharedXtimeNoAquire():                # @IncsharedXtimeNoAquire()
        xor     ecx, ecx
        mov     rdx, qword ptr [rip + shared_val]
        mov     esi, 1
        jmp     .LBB0_1
.LBB0_3:                                #   in Loop: Header=BB0_1 Depth=1
        cmp     ecx, 10000000
        jge     .LBB0_4
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        xor     eax, eax
        lock            cmpxchg dword ptr [rip + flag], esi
        jne     .LBB0_3
        inc     rdx
        mov     qword ptr [rip + shared_val], rdx
        inc     ecx
        mov     dword ptr [rip + flag], 0
        jmp     .LBB0_3
.LBB0_4:
        ret
```

Relaxed

```
IncsharedXtimeNoAquire():                 # @IncsharedXtimeNoAquire()
        xor     ecx, ecx
        mov     rdx, qword ptr [rip + shared_val]
        mov     esi, 1
        jmp     .LBB0_1
.LBB0_3:                                    #   in Loop: Header=BB0_1 Depth=1
        cmp     ecx, 10000000
        jge     .LBB0_4
.LBB0_1:                                    # =>This Inner Loop Header: Depth=1
        xor     eax, eax
        lock            cmpxchg dword ptr [rip + flag], esi
        jne     .LBB0_3
        inc     rdx
        mov     qword ptr [rip + shared_val], rdx
        inc     ecx
        mov     dword ptr [rip + flag], 0
        jmp     .LBB0_3
.LBB0_4:
        ret
```

Alex Dathshkovsky | alex.dathshkovsky@speedata.io | www.linkedin.com/in/alexdathshkovsky

Relaxed surprise

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

Acquire

```
void IncsharedXtimeAquire(){
    for (int count{0}; count < 10000000;) {
        int expected{};
        if (flag.compare_exchange_strong(expected, 1,
        std::memory_order_acquire)){
            //lock successfull
            shared_val++;
            count++;
            flag.store(0, std::memory_order_release);
        }
    }
}
```

Acquire

```
IncsharedXtimeAquire():                       # @IncsharedXtimeAquire()
        xor     ecx, ecx
        mov     edx, 1
        jmp     .LBB1_1
.LBB1_3:                                         #   in Loop: Header=BB1_1 Depth=1
        cmp     ecx, 10000000
        jge     .LBB1_4
.LBB1_1:                                         # =>This Inner Loop Header: Depth=1
        xor     eax, eax
        lock            cmpxchg dword ptr [rip + flag], edx
        jne     .LBB1_3
        inc     qword ptr [rip + shared_val]
        inc     ecx
        mov     dword ptr [rip + flag], 0
        jmp     .LBB1_3
.LBB1_4:
        ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Acquire No Surprises

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

RISC V Disassembly

```
IncsharedXtimeNoAquire():
        li      a4,9998336
        addi    a4,a4,1664
        lui     a3,%hi(flag)
        lui     a2,%hi(shared_val)
        li      a1,1
.L6:
        addi    a0,a3,%lo(flag)
         1: lr.w a5,0(a0); bne a5,zero,1f; sc.w a6,a1,0(a0); bnez a6,1b; 1:
        sext.w  a5,a5
        bnez    a5,.L6
        ld      a5,%lo(shared_val)(a2)
        addi    a5,a5,1
        sd      a5,%lo(shared_val)(a2)
        addi    a5,a3,%lo(flag)
        amoswap.w zero,zero,0(a5)
        addiw   a4,a4,-1
        bnez    a4,.L6
        ret
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

RISC V Disassembly

```
IncsharedXtimeAquire():
        li      a4,9998336
        addi    a4,a4,1664
        lui     a3,%hi(flag)
        lui     a2,%hi(shared_val)
        li      a1,1
.L27:
        addi    a0,a3,%lo(flag)
         1: lr.w.aq a5,0(a0); bne a5,zero,1f; sc.w.aq a6,a1,0(a0); bnez a6,1b; 1:
        sext.w  a5,a5
        bnez    a5,.L27
        ld      a5,%lo(shared_val)(a2)
        addi    a5,a5,1
        sd      a5,%lo(shared_val)(a2)
        addi    a5,a3,%lo(flag)
        fence iorw,ow; amoswap.w zero,zero,0(a5)
        addiw   a4,a4,-1
        bnez    a4,.L27
        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

```
int main(){
    std::promise<int> p;
    std::thread t1([&p](){
        std::this_thread::sleep_for(1s);
        p.set_value(42);
    });

    std::thread t2([](auto f){
        fmt::print("The value is {}", f.get());}, p.get_future());

    t1.join();
    t2.join();
}
```

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

```
int main(){
    std::mutex m;
    auto doer = [&m](int idx){
        std::lock_guard l(m);
        fmt::print("doer{} thread doing stuff\n", idx);
    };
    std::jthread t1(doer, 0);
    std::jthread t2(doer, 1);
}
```

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

```
std::binary_semaphore start_speaking{0}, person1_spoke{0},
                      person2_spoke{0};


void person_speak(std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r){
    w.acquire();
    fmt::print("{}", te
    std::this_thread::s
    r.release();
}


int main(){
    std::jthread p1(per                                    ref(start_speaking), std::ref(person1_spoke));
    std::jthread p2(per                                    ref(person1_spoke), std::ref(person2_spoke));
    start_speaking.release();
    person2_spoke.acquire();

    fmt::print("GoodBye!");

    return 0;
}
```

```
Hello Person2
Hello person1
GoodBye!
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Synchronization Tools : stop_source

```cpp
std::binary_semaphore start_speaking{0}, p1_done{0}, p2_done{0};

void person_speak(std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r, std::stop_token s){
    w.acquire();
    while (not s.stop_request
        fmt::print("{}", text
        std::this_thread::sle
    }
    r.release();
}

void stop_speaking(std::array
    for (auto i = 0u; i < ssr
        std::this_thread::sle
        ssrc[i].request_stop(
    }
}


int main() {
    std::array ssrcs{std::sto
    std::jthread p1(person_sp
                std::ref(                                                  ());
    std::jthread p2(person_sp
                std::ref(p1_done), std::ref(p2_done), ssrcs[1].get_token());
    start_speaking.release();
    std::jthread sched(stop_speaking, ssrcs);
    p2_done.acquire();
    fmt::print("Goodby!");
}
```

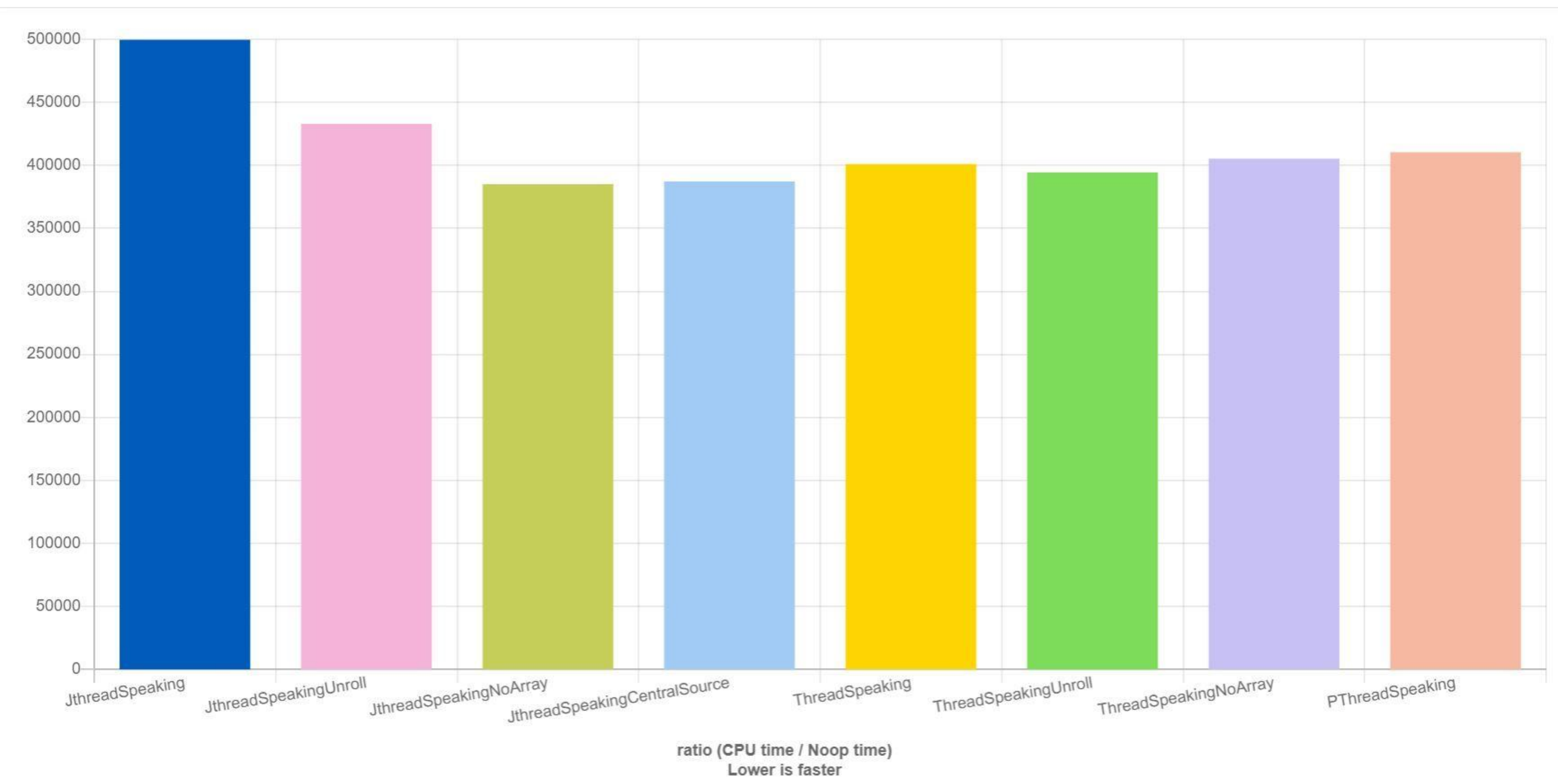
```
Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson2 im Person1
Hello Peson1 im Person2
Hello Peson1 im Person2
Hello Peson1 im Person2
Hello Peson1 im Person2
Goodby!
```

Synchronization Tools : stop_source

```cpp
std::binary_semaphore start_speaking{0}, p1_done{0}, p2_done{0};

void person_speak(std::stop_token s, std::string_view text, std::binary_semaphore& w, std::binary_semaphore& r){
    w.acquire();
    while (not s.stop_requested()){
        fmt::print("{}", text);
        std::this_thread::sleep_for(1s);
    }
    r.release();
}

void stop_speaking(std::array<std::stop_source, 2> ssrc){
    for (auto i = 0u; i < ssrc.size(); i++){
        std::this_thread::sleep_for(2s);
        ssrc[i].request_stop();
    }
}


int main() {
    std::jthread p1(person_speak, "Hello Peson2 im Person1\n",
                    std::ref(start_speaking), std::ref(p1_done));
    std::jthread p2(person_speak, "Hello Peson1 im Person2\n",
                    std::ref(p1_done), std::ref(p2_done));
    start_speaking.release();
    std::jthread sched(stop_speaking, std::array{p1.get_stop_source(), p2.get_stop_source()});
    p2_done.acquire();
    fmt::print("Goodby!");
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Std::threads : Micro Benchmark



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

Synchronization Tools - cont

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

Synchronization Tools – Latch Example

```
int main(){
    std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
    std::vector<std::jthread> workers;
    std::latch jobs{ppl.size()}, go_home{1};
    auto job = [&jobs, &go_home](std::string_view name){
        fmt::print("{} is doing a job\n", name);
        jobs.count_down();
        go_home.wait();
        fmt::print("{} is going home\n", name);
    };

    for (const auto& p : ppl) {
        workers.push_back(std::jthread(job, p));
    }
    jobs.wait();
    fmt::print("Go home!\n");
    go_home.count_down();
}
```

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

```cpp
int main(){
    std::vector<std::string_view> ppl{"Alex", "Dani", "Benny", "Guy"};
    std::vector<std::jthread> workers;
    auto on_complete = [](){
        static int x = 0;
        not x++ ? fmt::print("Go Home!\n") : fmt::print("Done!\n"); };
    std::barrier sync{ppl.size(), on_complete};
    auto job = [&sync](std::string_view name){
        fmt::print("{} is doing a job\n", name);
        sync.arrive_and_wait();
        fmt::print("{} is going home\n", name);
        sync.arrive_and_wait();
    };

    for (const auto& p : ppl) {
        workers.push_back(std::jthread(job, p));
    }
}
```

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

```
int val = 10;
int main(){
    std::shared_timed_mutex stm;
    auto access_and_print = [&stm](int v){
        std::shared_lock l(stm, 1s);
        int my_val = val + v;
        fmt::print("val {}\n", v);
    };

    std::jthread t1(access_and_print, 1);
    std::jthread t2(access_and_print, 2);
}
```

Synchronization Tools

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

```cpp
struct X{
    int x;
};
```

```asm
get_once(int):                                   # @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
```

```cpp
        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.

```cpp
int main(){
    std::once_flag flag;
    auto f = [&flag](){
        static int x{10};
        std::call_once(flag, [](){x++;});
        fmt::print("{}\n", x);
    };

    std::jthread t1(f);
    std::jthread t2(f);
}
```

```
11
11
```

Synchronization Tools

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

```
int main(){
    std::packaged_task sum_task([](int a, int b){
        return a+b; });

    std::packaged_task mul_task([](int a, int b){ return a*b; });

    mul_task(10, 2);
    sum_task(10, mul_task.get_future().get());

    fmt::print("tot: {}", sum_task.get_future().get());

    return 0;
}
```

QUESTIONS

?

THANK YOU FOR LISTENING

ALEX DATSKOVSKY

+97254-7685001

ALEX.DATSKOVSKY@SPEEDATA.IO

WWW.LINKEDIN.COM/IN/ALEXDATHSKOVSKY

https://www.cppnext.com