NO SANE COMPILER WOULD OPTIMIZE ATOMICS

JF Bastien <@jfbastien>

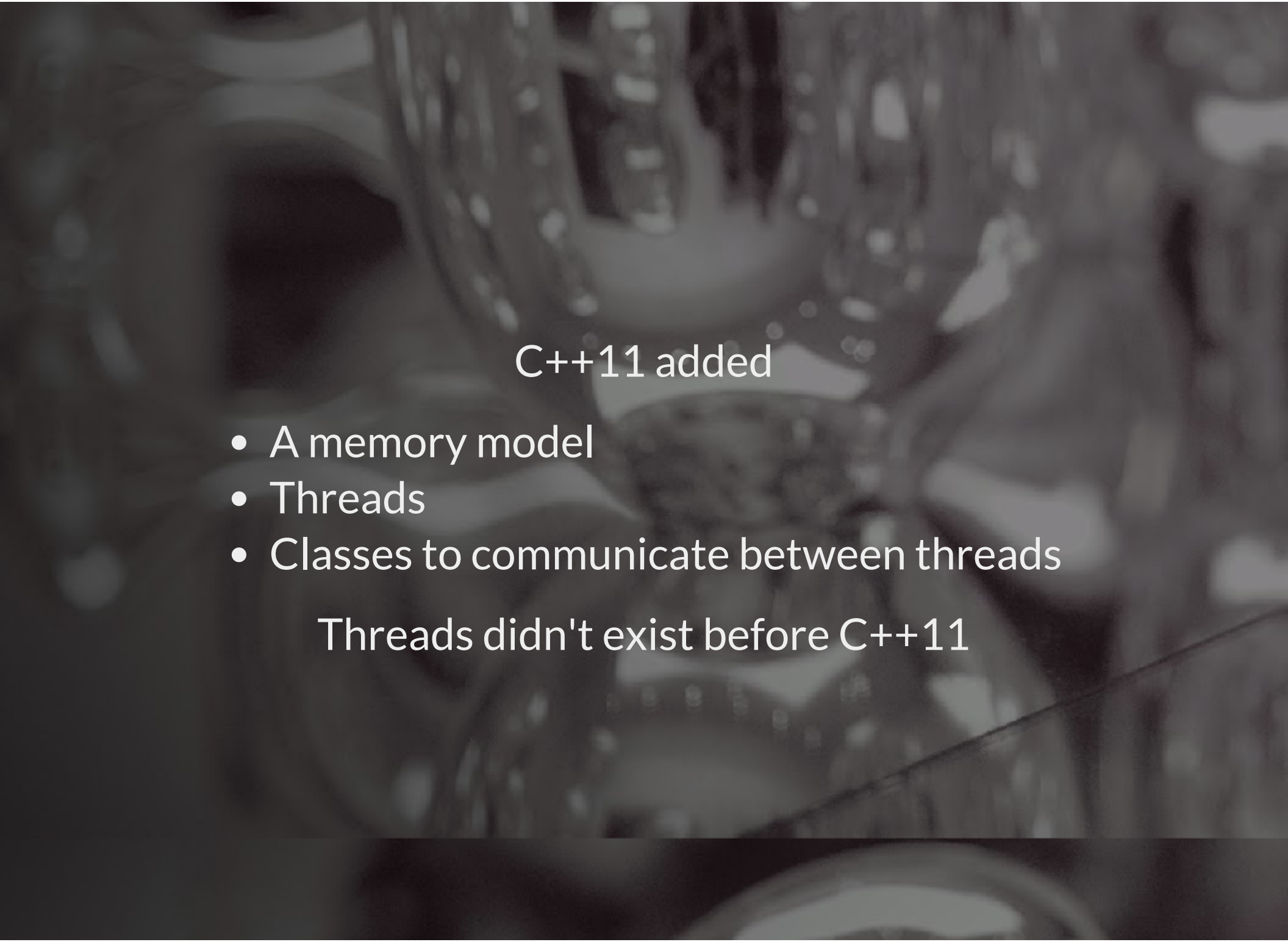
Compiler engineer on Chrome


false


DISCLAIMER


WHY DO I CARE?

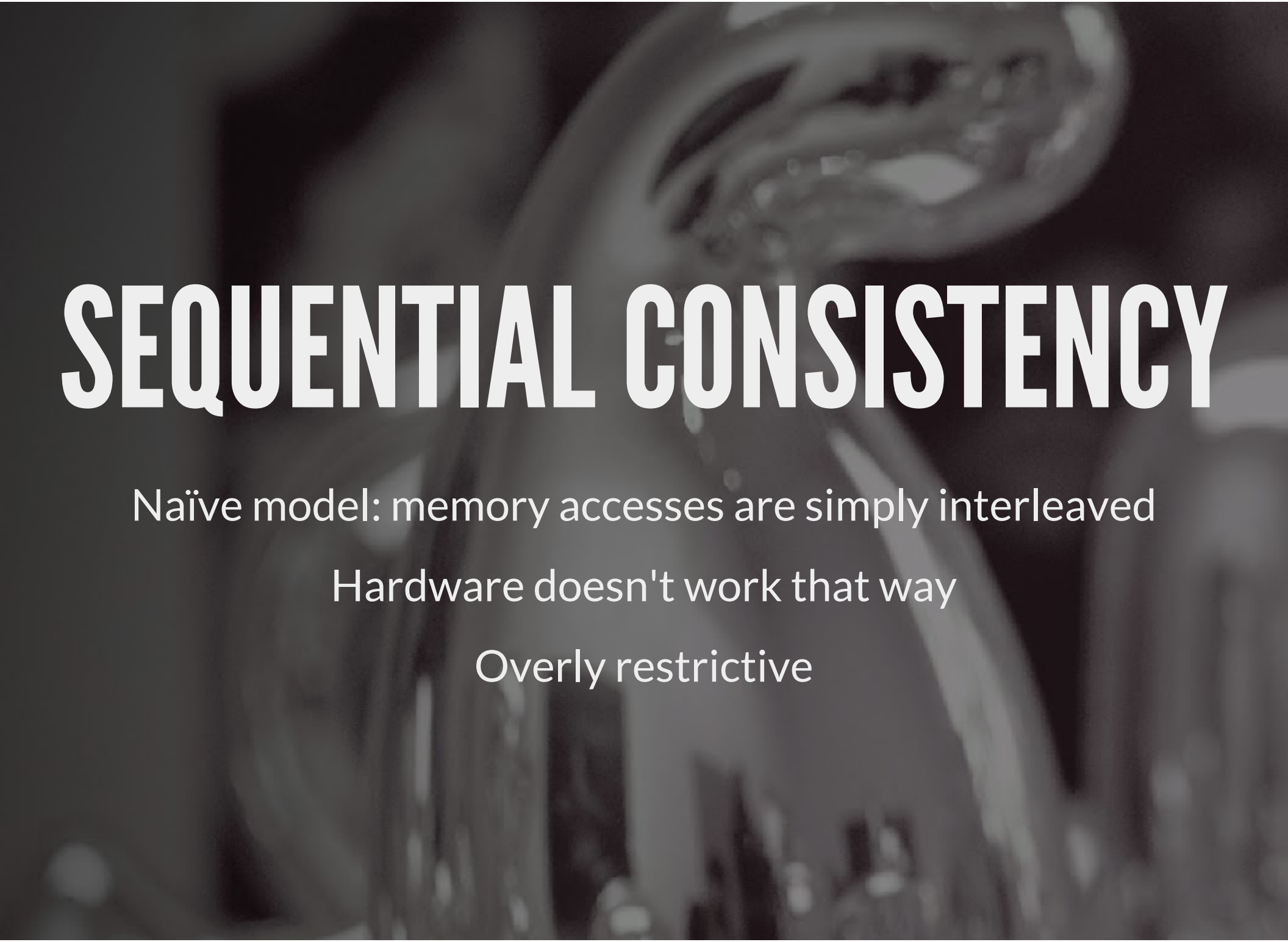

QUICK REVIEW



C++11 added

- A memory model
- Threads
- Classes to communicate between threads

Threads didn't exist before C++11



SEQUENTIAL CONSISTENCY

Naïve model: memory accesses are simply interleaved

Hardware doesn't work that way

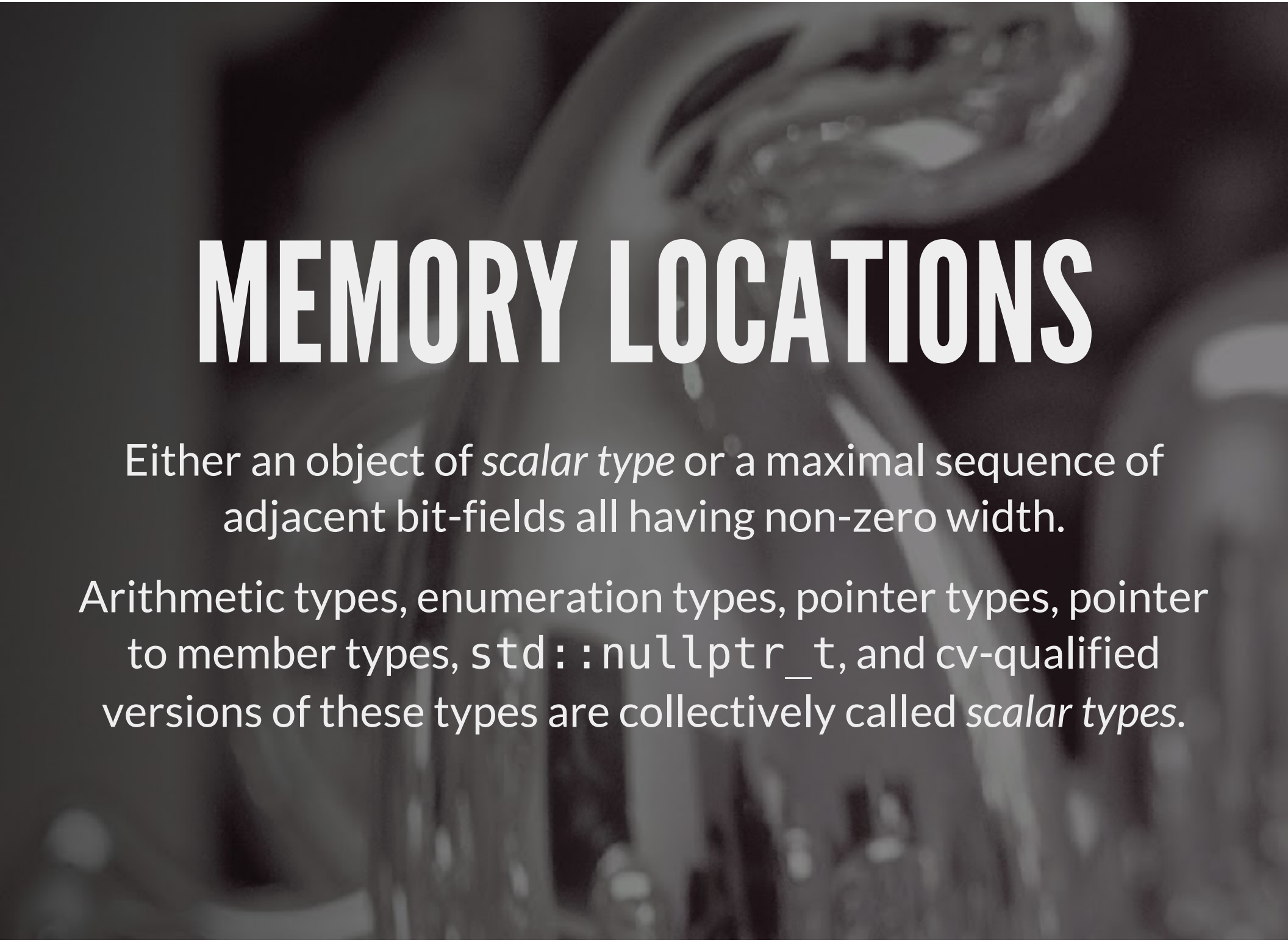
Overly restrictive

SC-DRF

Data race:

- Two accesses to the same *memory location* by different threads are *not ordered*
- At least one of them stores to the memory location
- At least one of them is not a synchronization action

Data races are UB



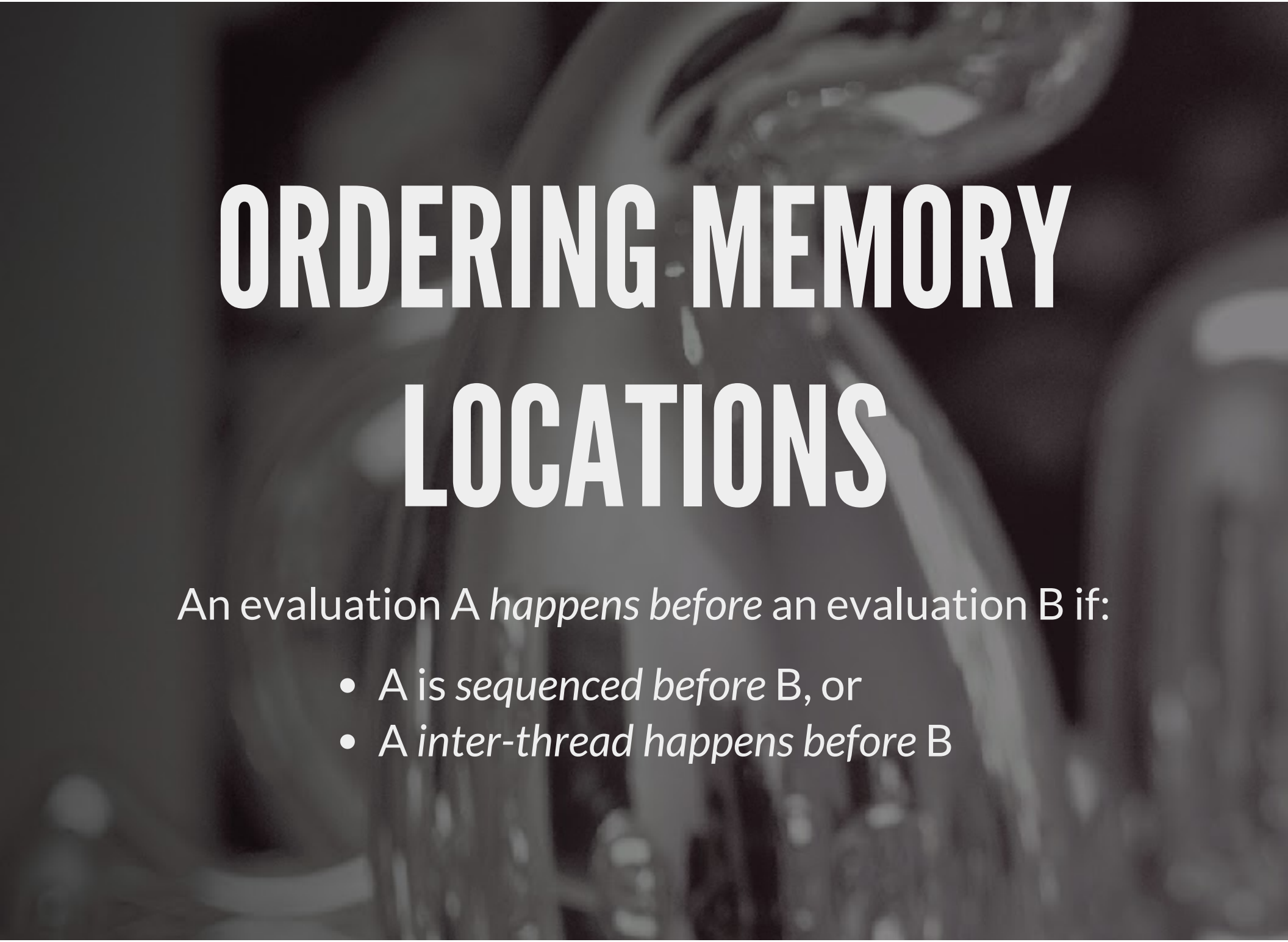
MEMORY LOCATIONS

Either an object of *scalar type* or a maximal sequence of adjacent bit-fields all having non-zero width.

Arithmetic types, enumeration types, pointer types, pointer to member types, `std::nullptr_t`, and cv-qualified versions of these types are collectively called *scalar types*.

ORDERING MEMORY LOCATIONS

Two memory operations are ordered if they cannot occur simultaneously



ORDERING MEMORY LOCATIONS

An evaluation A *happens before* an evaluation B if:

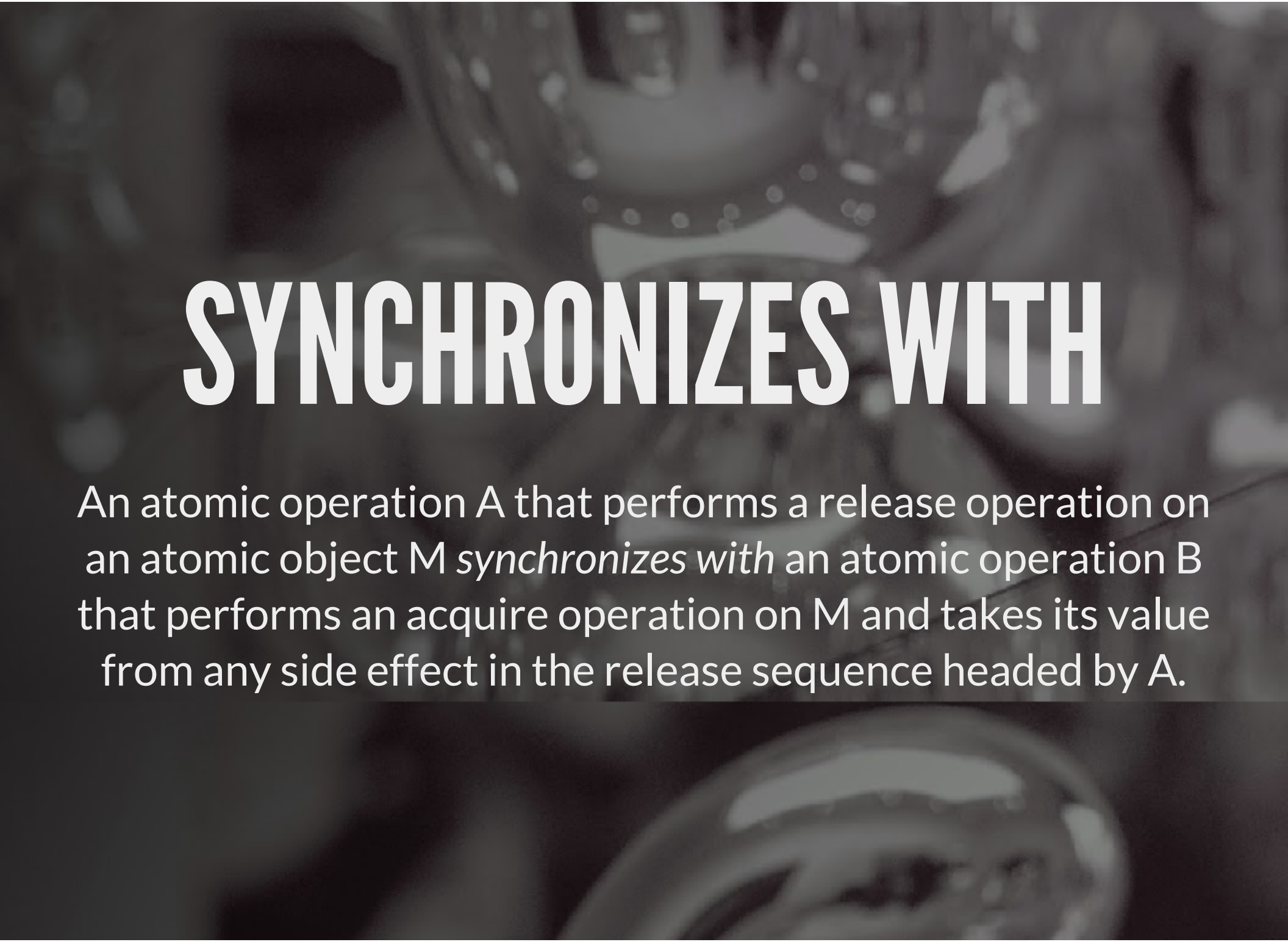
- A is *sequenced before* B, or
- A *inter-thread happens before* B

SEQUENCED BEFORE

An asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is *sequenced before* B, then the execution of A shall precede the execution of B. If A is not *sequenced before* B and B is not *sequenced before* A, then A and B are *unsequenced*.

INTER-THREAD HAPPENS BEFORE

- A *synchronizes with* B, or
- A is *dependency-ordered before* B, or
- for some evaluation X
  - A *synchronizes with* X and X is *sequenced before* B, or
  - A is *sequenced before* X and X *inter-thread happens before* B, or
  - A *inter-thread happens before* X and X *inter-thread happens before* B.



SYNCHRONIZES WITH

An atomic operation A that performs a release operation on an atomic object M *synchronizes with* an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

DEPENDENCY-ORDERED BEFORE

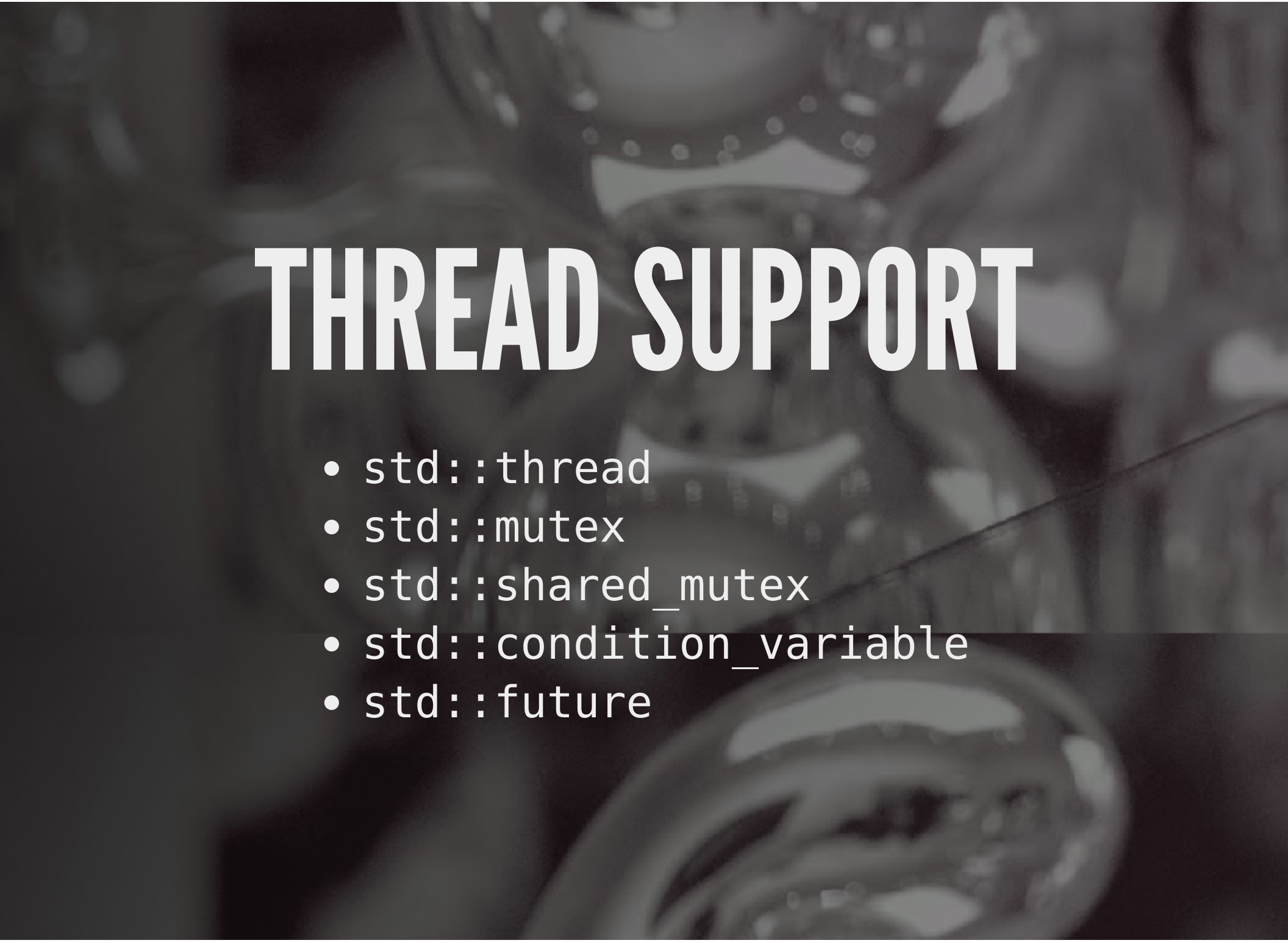
- A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
- for some evaluation X, A is *dependency-ordered before* X and X carries a dependency to B.

[Note: The relation “is dependency-ordered before” is analogous to “synchronizes with”, but uses release/consume in place of release/acquire. — end note]

SYNCHRONIZATION OPERATION

- On one or more memory locations: consume operation, an acquire operation, a release operation, or both an acquire and release operation.
- Without an associated memory location: acquire fence, release fence, or both an acquire and release fence.


[INTRO.MULTITHREAD]
1.10 Multi-threaded executions and data races

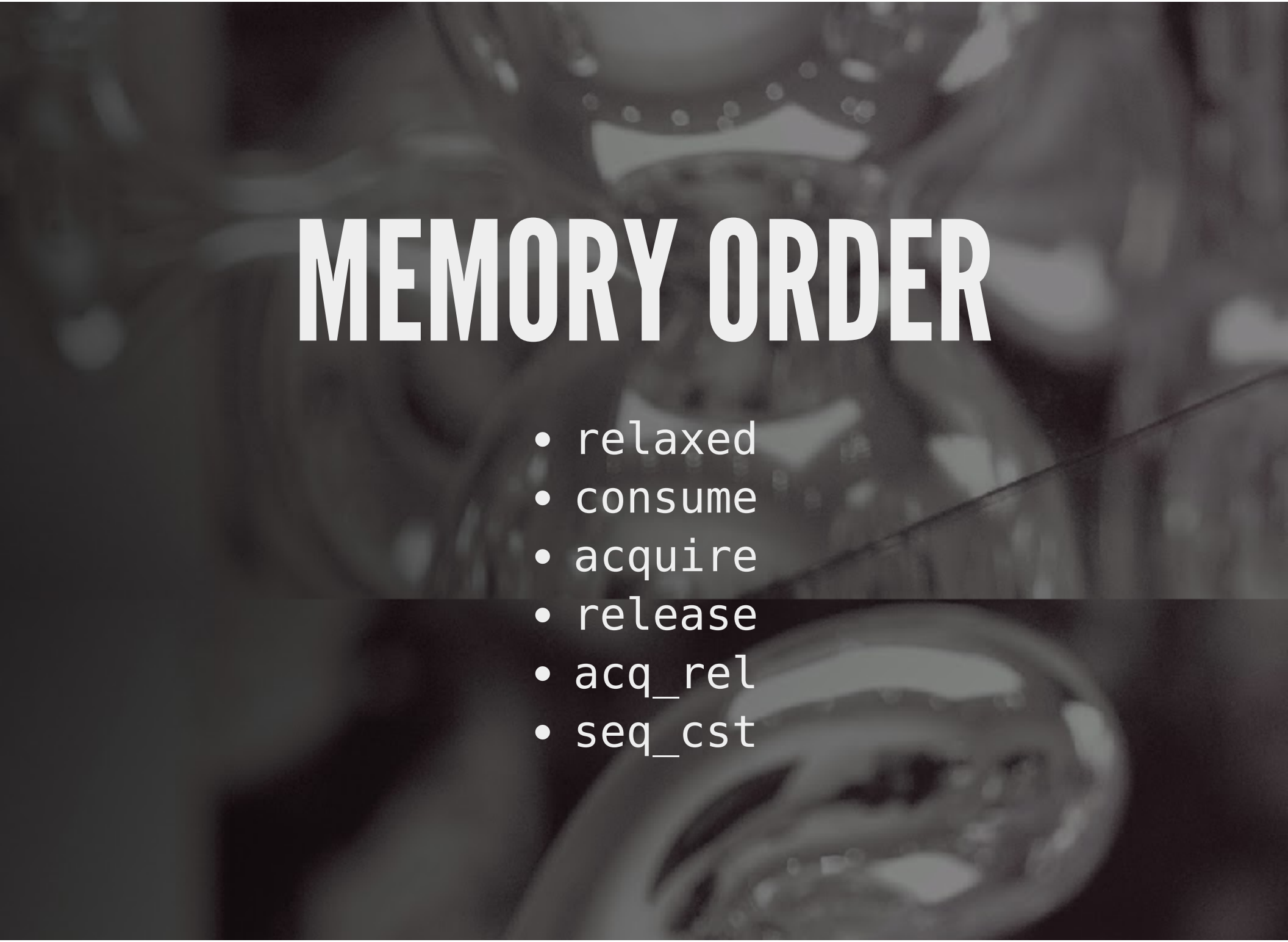


THREAD SUPPORT

- std::thread
- std::mutex
- std::shared_mutex
- std::condition_variable
- std::future

ATOMIC OPERATIONS

Atomic: indivisible with respect to all other atomic accesses to that object


MEMORY ORDER
• relaxed
• consume
• acquire
• release
• acq_rel
• seq_cst

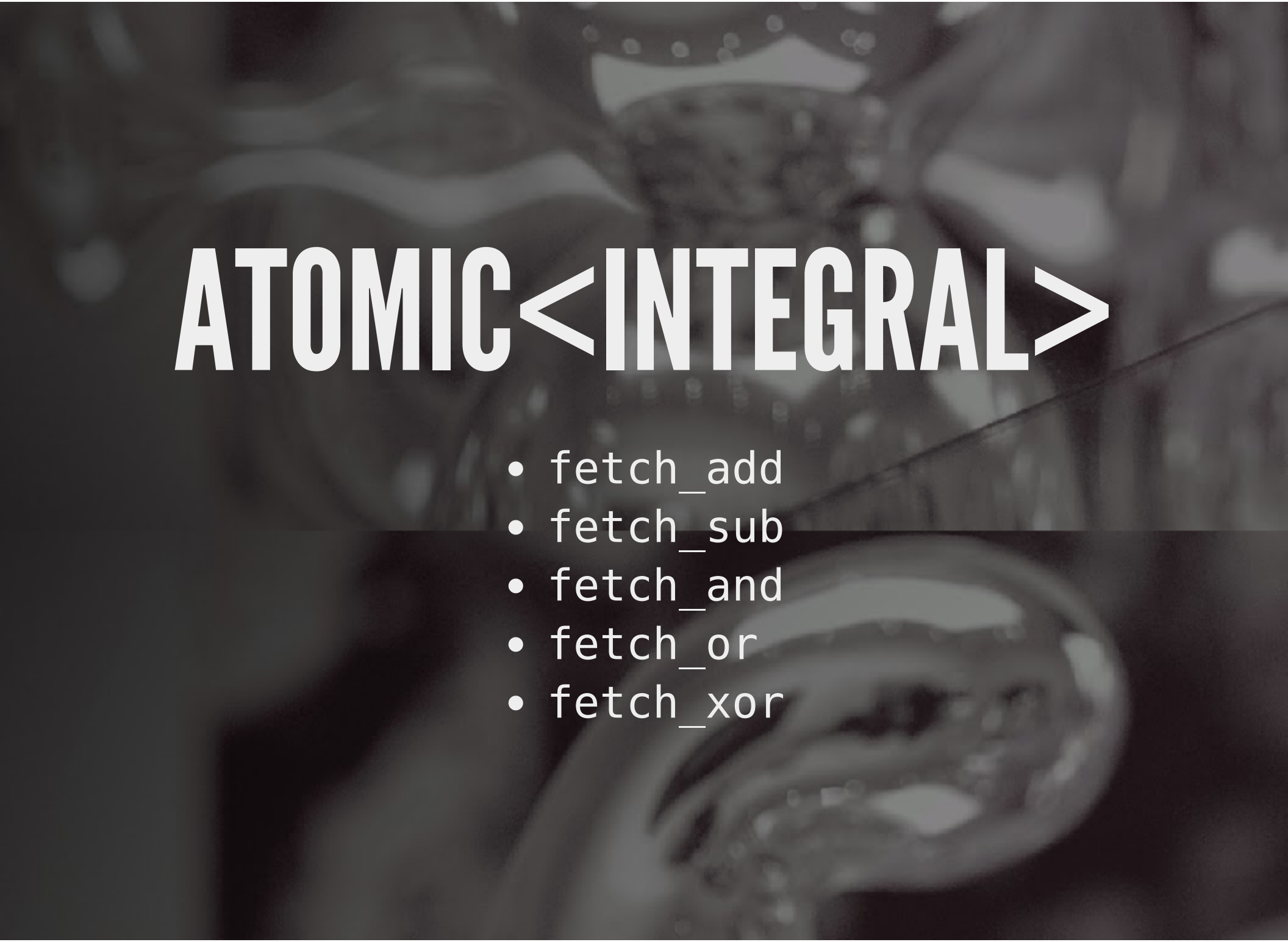

RELAXED
• racing accesses
• indivisible
• enforce cache coherency
• doesn't guarantee visibility

ATOMIC

- template<class T> struct atomic;
- template<> struct atomic<integral>;
- template<class T> struct atomic<T*>;
- template<class T> struct atomic<floating-point>; // future

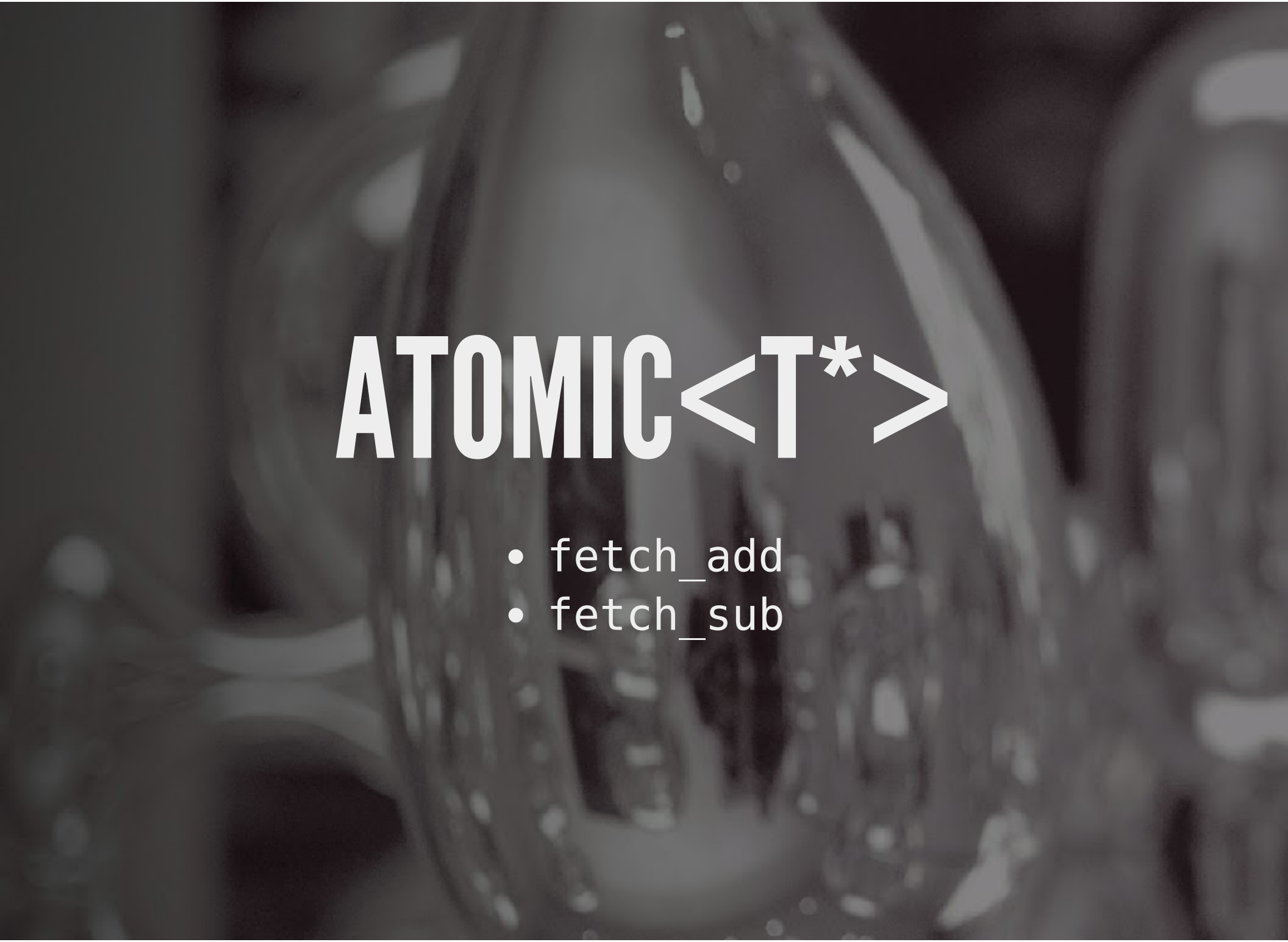
atomic<T>

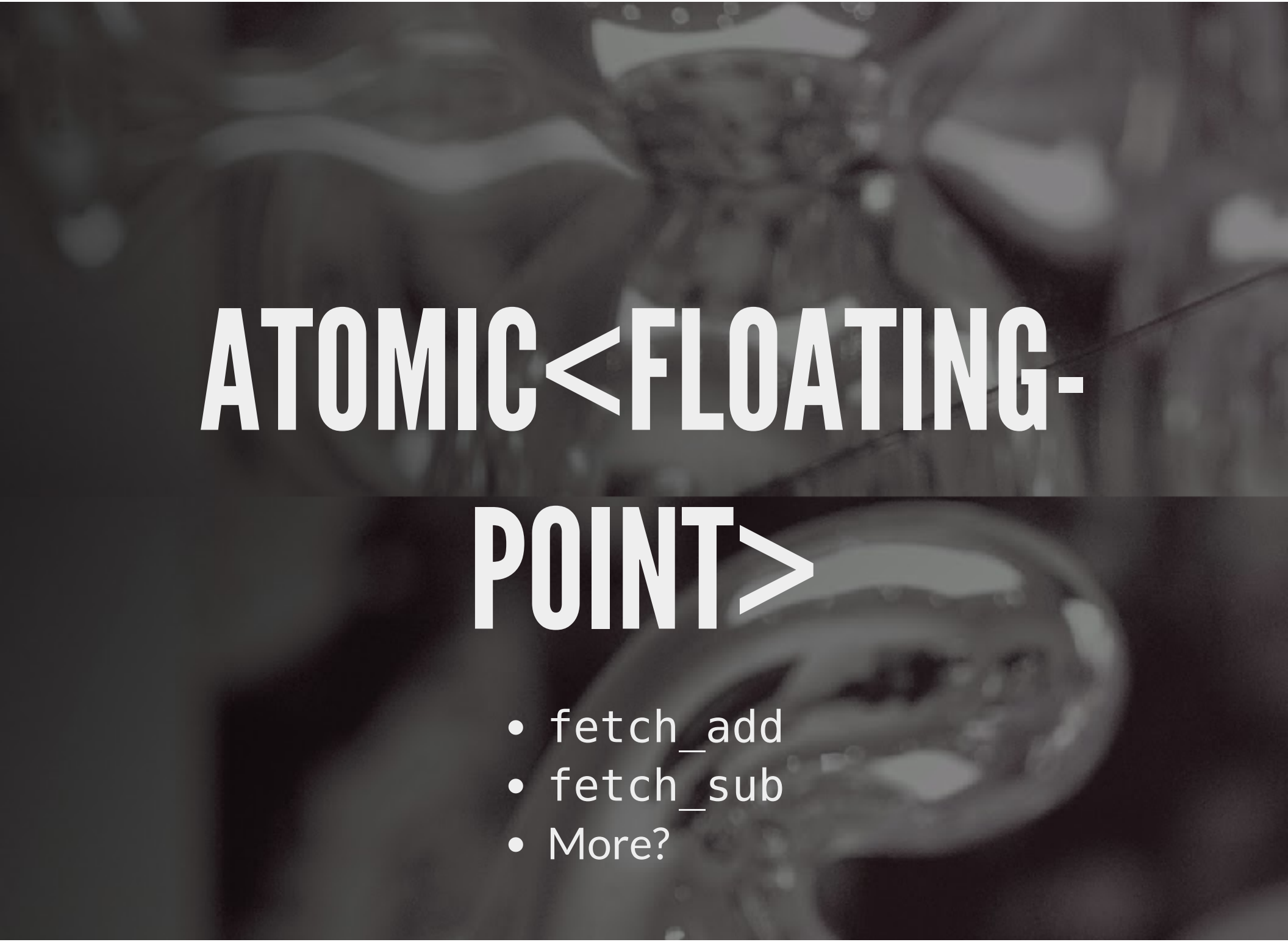
- load
- store
- exchange
- compare_exchange_{weak,strong}
- is_lock_free
- static constexpr is_always_lock_free // C++17



atomic<integral>

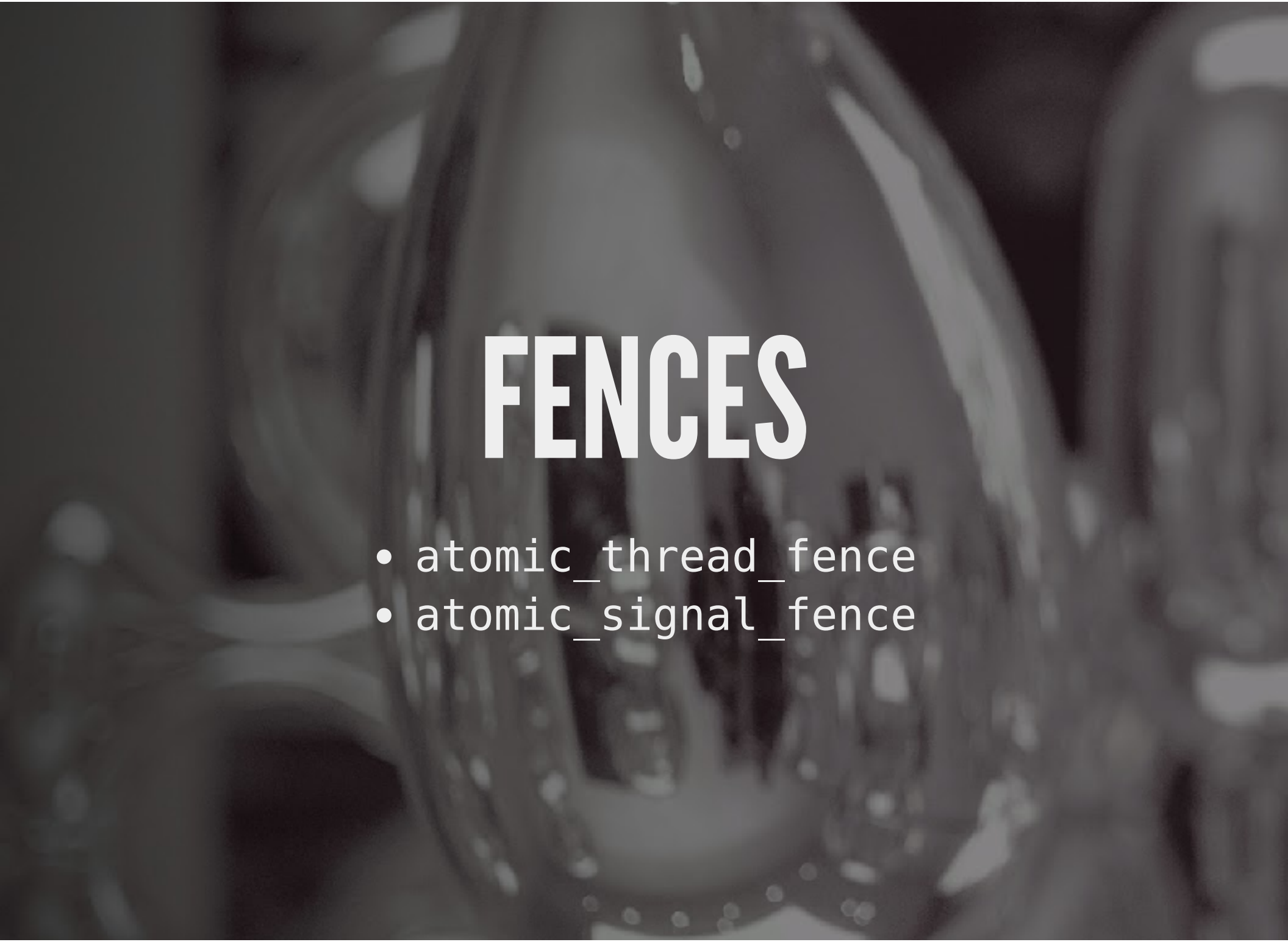
- `fetch_add`
- `fetch_sub`
- `fetch_and`
- `fetch_or`
- `fetch_xor`


ATOMIC<T*>
• fetch_add
• fetch_sub



ATOMIC<FLOATING-POINT>

- `fetch_add`
- `fetch_sub`
- More?



FENCES

- `atomic_thread_fence`
- `atomic_signal_fence`

LOCK-FREEDOM

Guaranteed for `std::atomic_flag`

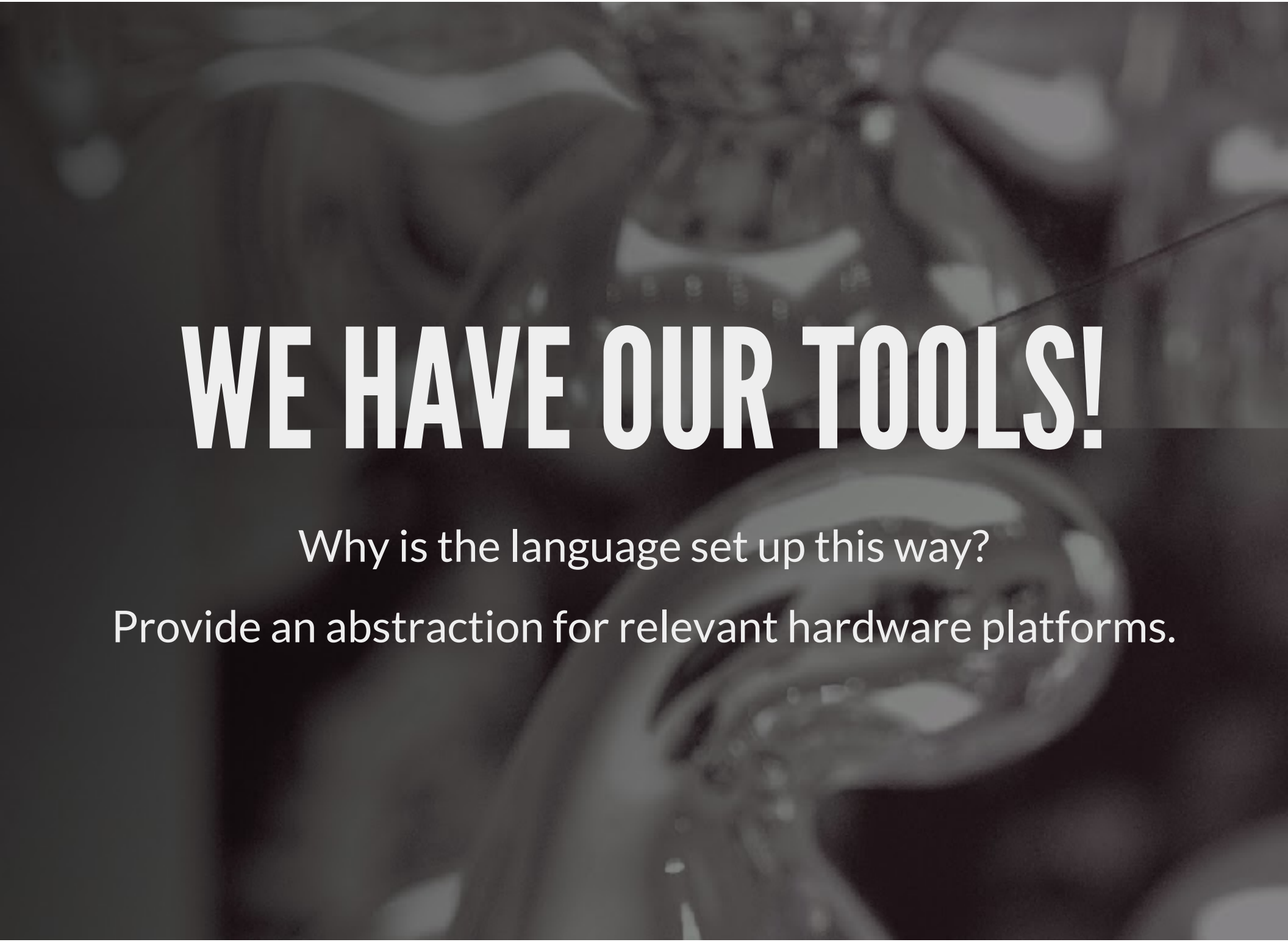
Roughtly: is there a compare-exchange for this size?

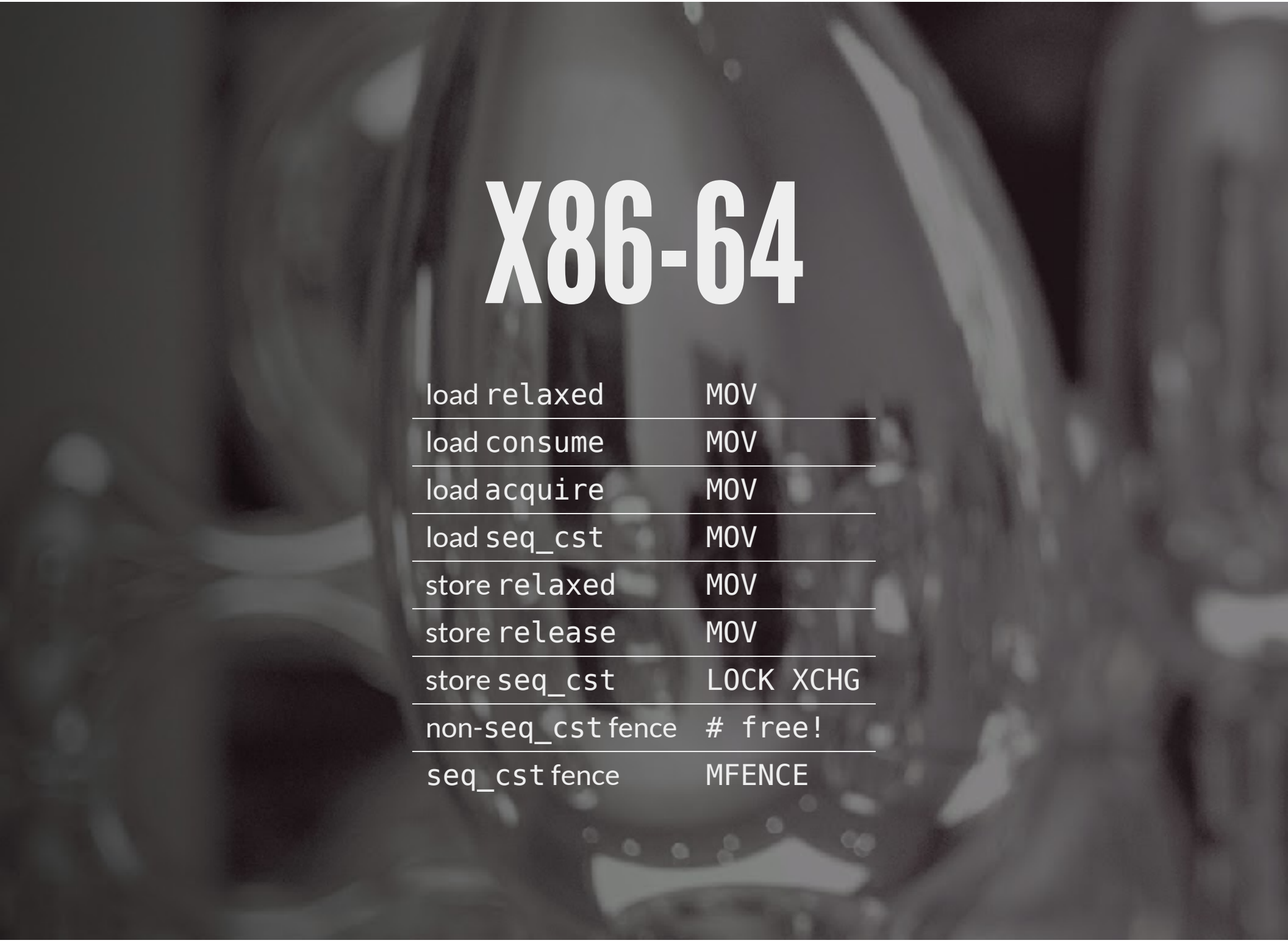
This is what I'm focusing on.

What does non-lock-free mean?

SIMPLE SINGLE PRODUCER / CONSUMER

```
struct Data { int spam, bacon, eggs; std::atomic<int> ready; };
void write(Data *d) {
  d->spam = 0xBEEF;
  d->bacon = 0xCAFE;
  d->eggs = 0x0000;
  d->ready.store(1, std::memory_order_release);
}
auto read(Data *d) {
  while (!d->ready.load(std::memory_order_acquire)) ;
  return std::make_tuple(d->spam, d->bacon, d->eggs);
}
```


WE HAVE OUR TOOLS!
Why is the language set up this way?
Provide an abstraction for relevant hardware platforms.



X86-64

load `relaxed`	`MOV`
load `consume`	`MOV`
load `acquire`	`MOV`
load `seq_cst`	`MOV`
store `relaxed`	`MOV`
store `release`	`MOV`
store `seq_cst`	`LOCK XCHG`
non-`seq_cst` fence	`# free!`
`seq_cst` fence	`MFENCE`

POWER

load `relaxed`	`ld`
load `consume`	`ld + dependencies`
load `acquire`	`ld; cmp; bc; isync`
load `seq_cst`	`hwsync; ld; cmp; bc; isync`
store `relaxed`	`st`
store `release`	`lwsync; st`
store `seq_cst`	`hwsync; st`
`cmpxchg relaxed`	`_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:`
non-`seq_cst` fence	`lwsync`
`seq_cst` fence	`hwsync`

ARMV7

load `relaxed`	`ldr`
load `consume`	`ldr + dependencies`
load `acquire`	`ldr; dmb`
load `seq_cst`	`lrd; dmb`
store `relaxed`	`str`
store `release`	`dmb; str`
store `seq_cst`	`dmb; str; dmb`
`cmpxchg relaxed`	`_loop: ldrex; mov 0; teq; strexeq; teq 0; bne _loop`
fences	`dmb`

ARMV8 A64

load `relaxed`	`ldr`
load `consume`	`ldr + dependencies`
load `acquire`	`ldar`
load `seq_cst`	`ldar`
store `relaxed`	`str`
store `release`	`stlr`
store `seq_cst`	`stlr`
`cmpxchg relaxed`	`_loop: ldxr; cmp; bne _exit; stxr; cbz _loop; _exit`
fences	`dmb`

ITANIUM

load relaxed	ld.acq
load consume	ld.acq
load acquire	ld.acq
load seq_cst	ld.acq
store relaxed	st.rel
store release	st.rel
store seq_cst	st.rel; mfB
cmpxchg acquire	cmpxchg.acq
non-seq_cst fence	# free!
seq_cst fence	mf

ALPHA

Oh, one of my favorite (NOT!) pieces of code in the kernel is the implementation of the `smp_read_barrier_depends()` macro, which on every single architecture except for one (alpha) is a no-op.

We have basically 30 or so empty definitions for it, and I think we have something like five uses of it. One of them, I think, is performance crticial, and the reason for that macro existing.

What does it do? The semantics is that it's a read barrier between two different reads that we want to happen in order wrt two writes on the writing side (the writing side also has to have a `smp_wmb()` to order those writes). But the reason it isn't a simple read barrier is that the reads are actually causally *dependent*, ie we have code like

```
first_read = read_pointer;
smp_read_barrier_depends();
second_read = *first_read;
```

and it turns out that on pretty much all architectures (except for alpha), the *data*dependency* will already guarantee that the CPU reads the thing in order. And because a read barrier can actually be quite expensive, we don't want to have a read barrier for this case.

But alpha? Its memory consistency is so broken that even the data dependency doesn't actually guarantee cache access order. It's strange, yes. No, it's not that alpha does some magic value prediction and can do the second read without having even done the first read first to get the address. What's actually going on is that the cache itself is unordered, and without the read barrier, you may get a stale version from the cache even if the writes were forced (by the write barrier in the writer) to happen in the right order.

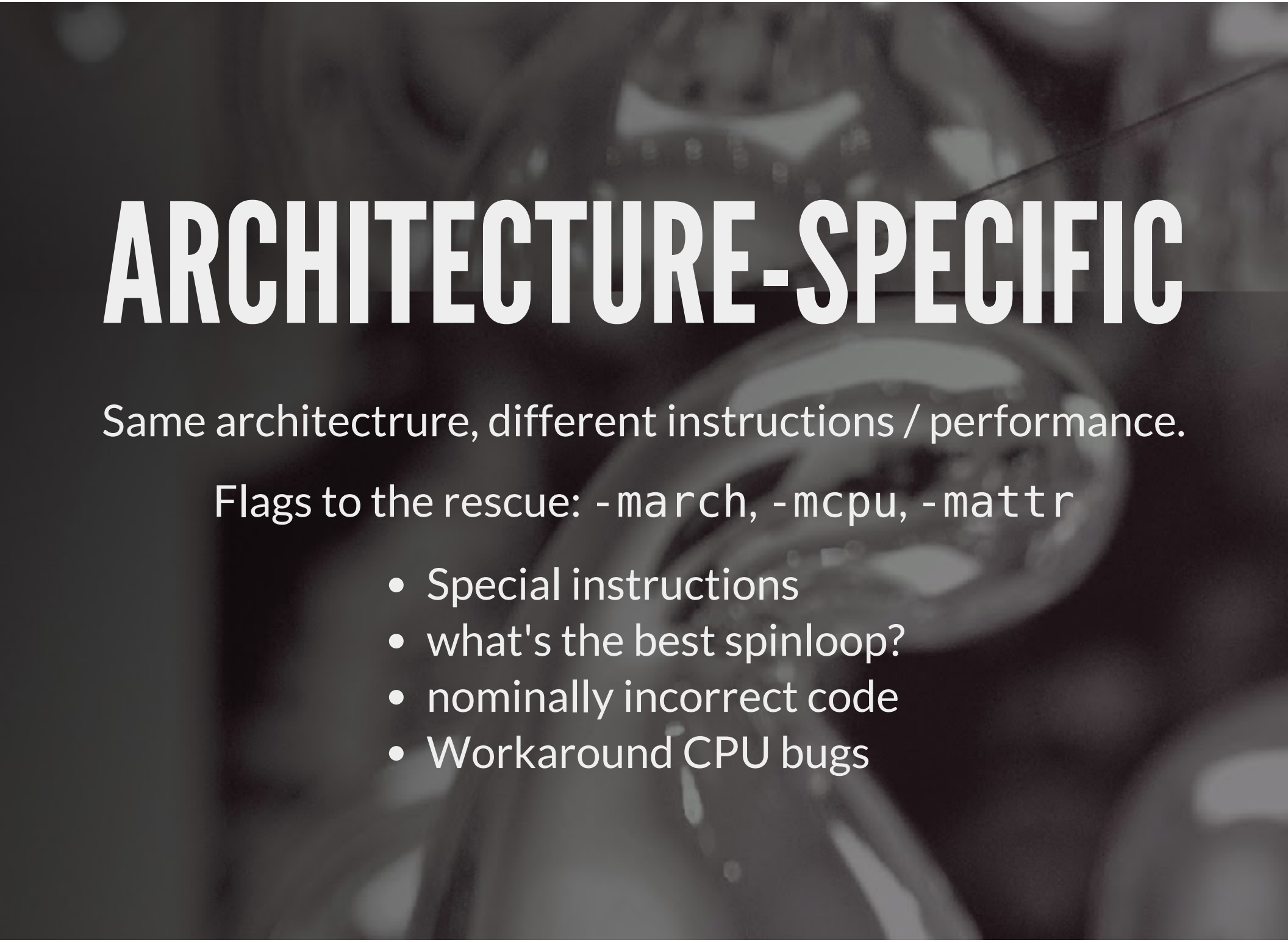
CONSUME FOR ARM

Address dependency rule

```
LDR r1, [r0]
LDR r2, [r1]
```

```
LDR r1, [r0]
AND r1, r1, #0
LDR r2, [r3, r1]
```

Look ma, no barrier!

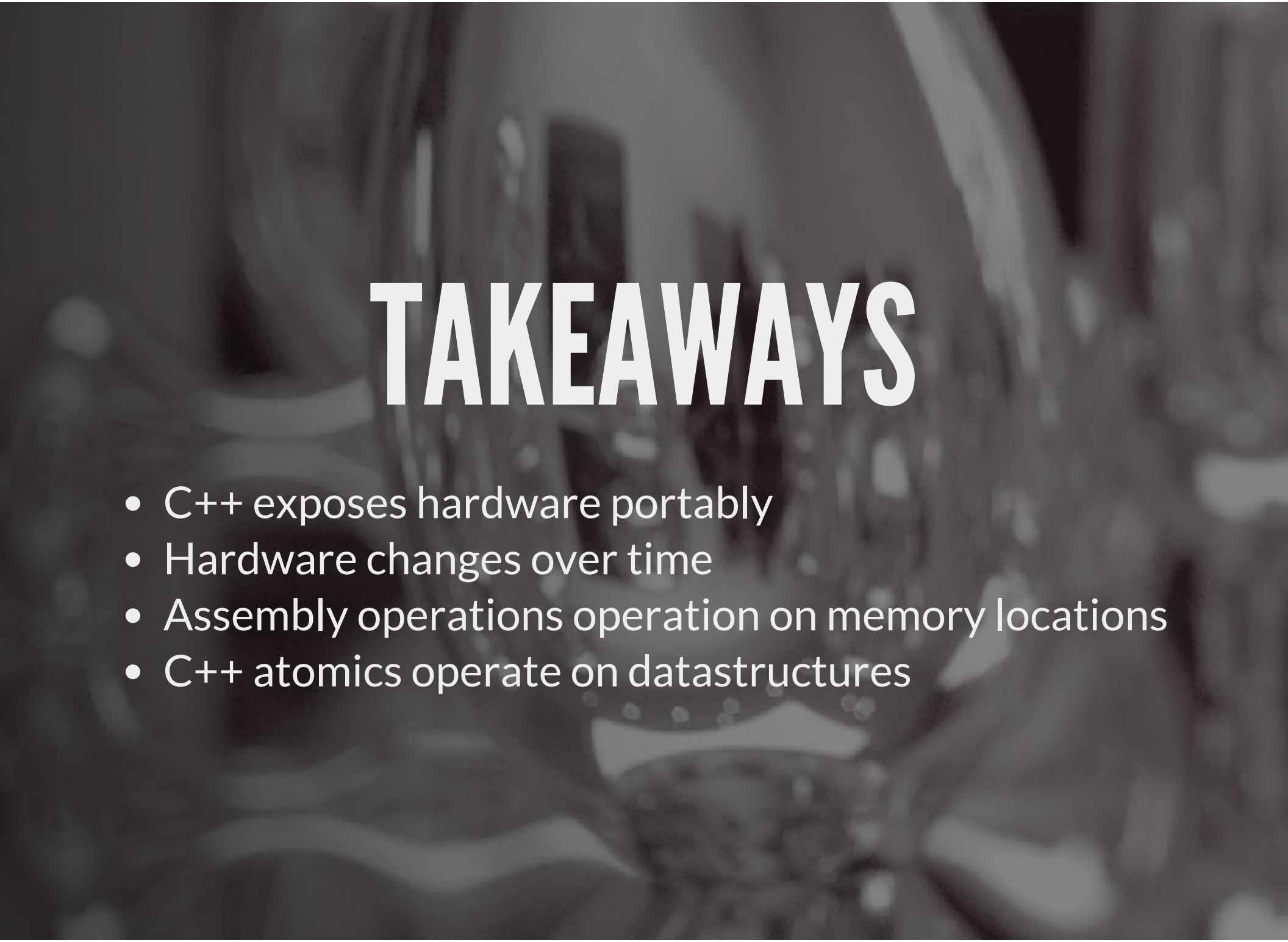


ARCHITECTURE-SPECIFIC

Same architectrure, different instructions / performance.

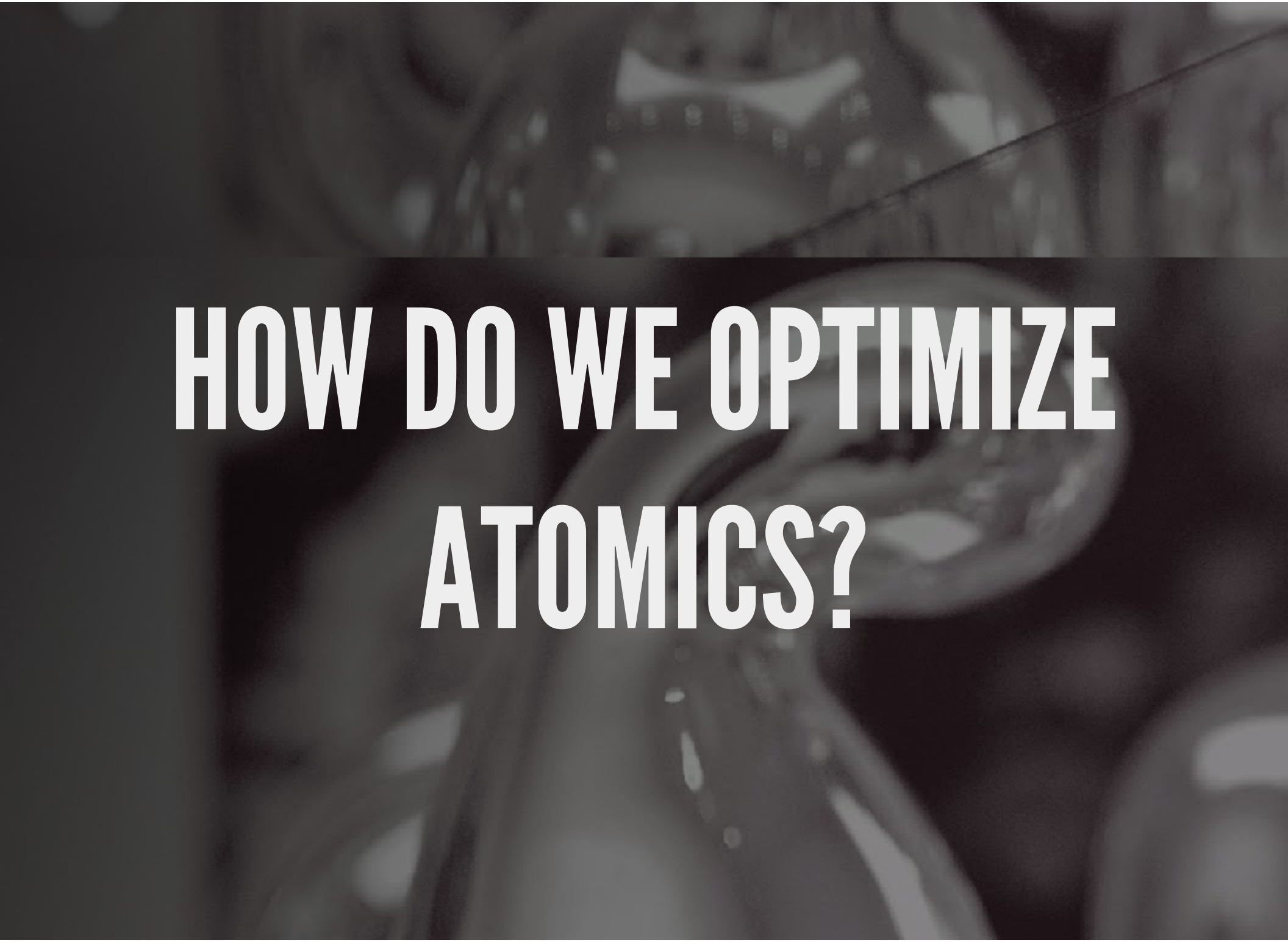
Flags to the rescue: `-march`, `-mcpu`, `-mattr`

- Special instructions
- what's the best spinloop?
- nominally incorrect code
- Workaround CPU bugs

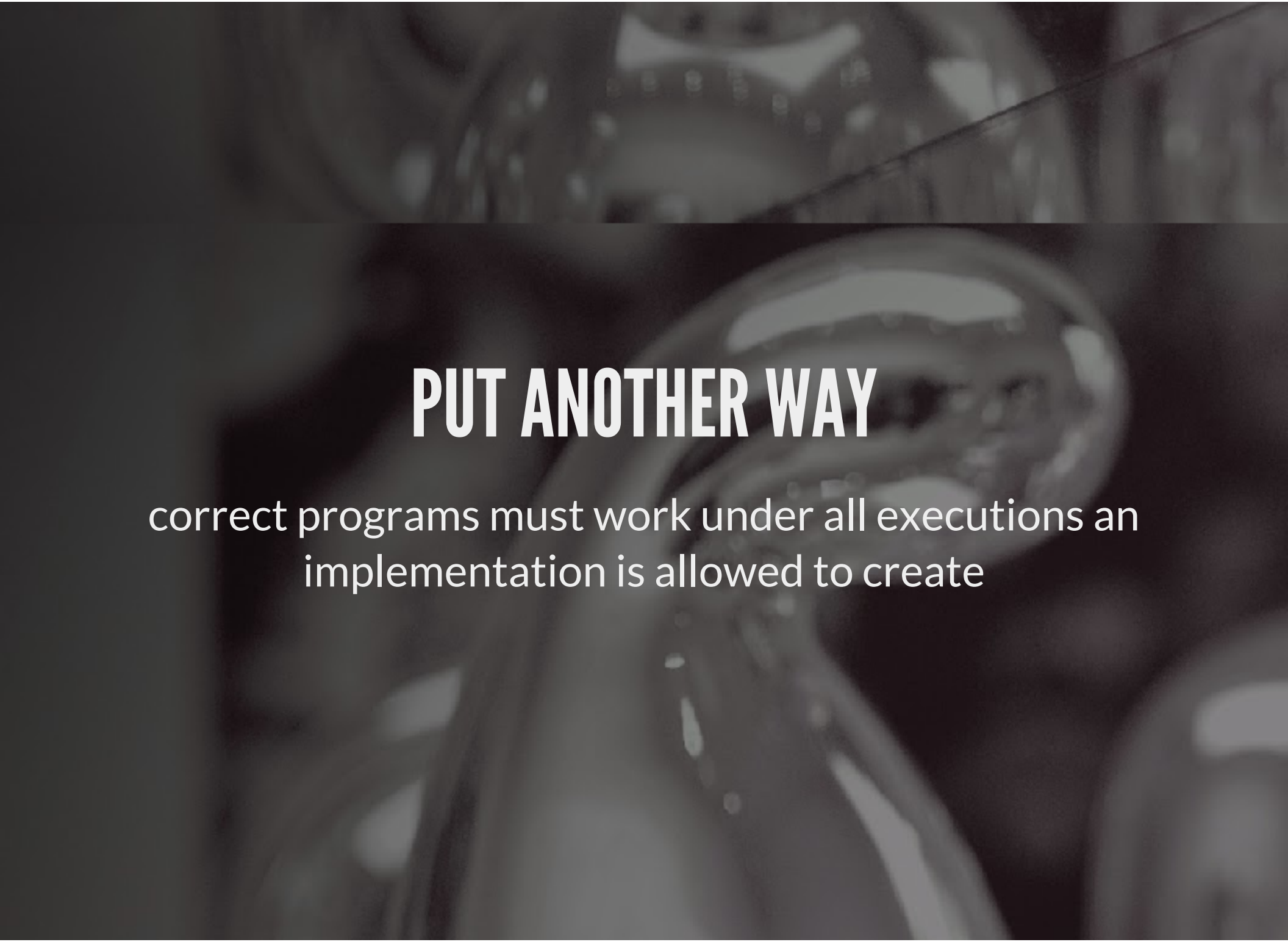


TAKEAWAYS

- C++ exposes hardware portably
- Hardware changes over time
- Assembly operations operation on memory locations
- C++ atomics operate on datastructures


HOW DO WE OPTIMIZE
ATOMICS?

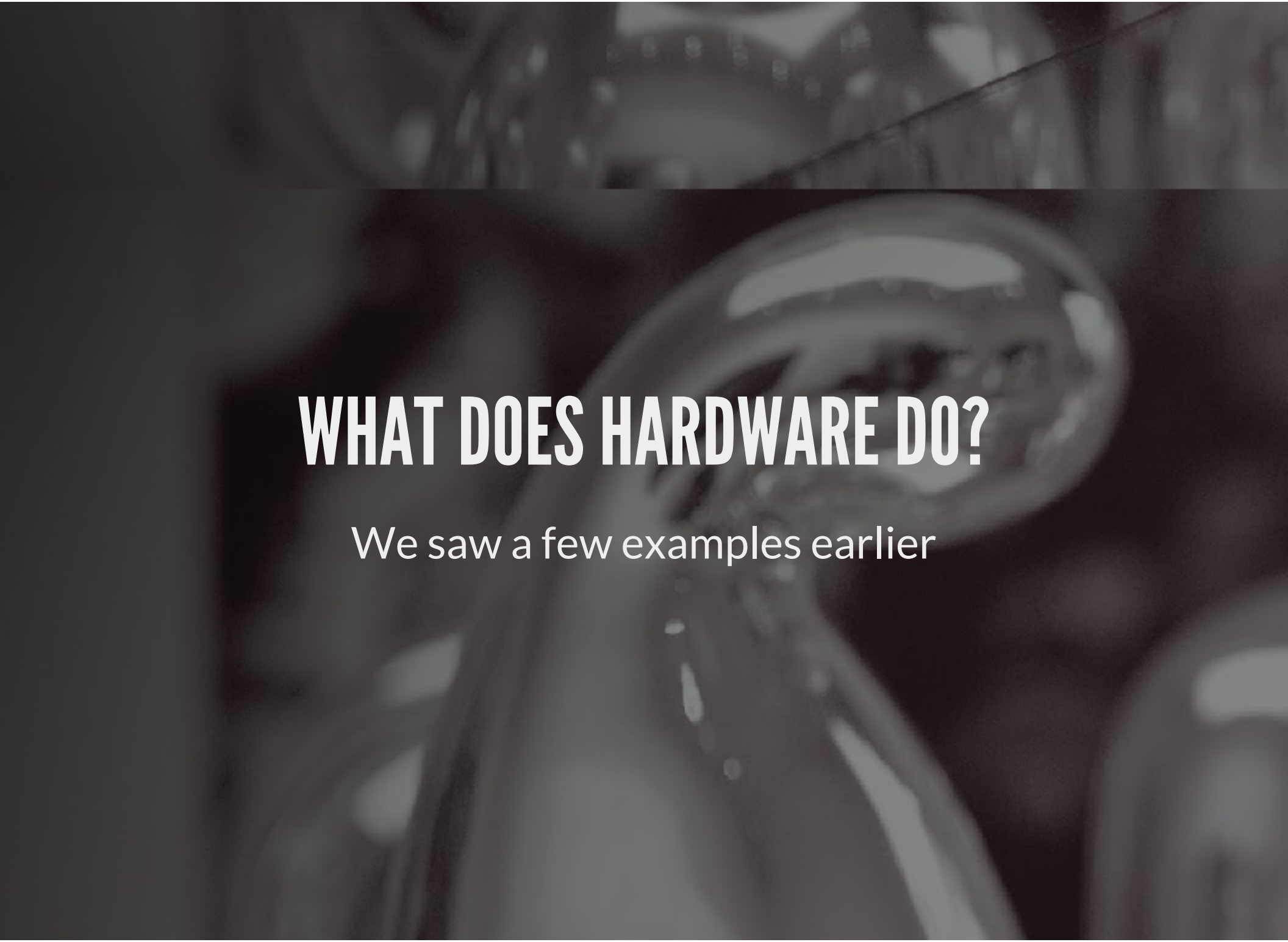

AS-IF
more atomic: don’t violate forward progress
less atomic: don’t add non-benign race which weren’t already present



PUT ANOTHER WAY

correct programs must work under all executions an implementation is allowed to create


WHAT CAN THE COMPILER DO?
At least as much as the hardware could do
and maybe more ☺


WHAT DOES HARDWARE DO?
We saw a few examples earlier

SIMPLE EXAMPLE

```
void inc(std::atomic<int> *y) {
  *y += 1;
}
std::atomic<int> x;
void two() {
  inc(&x);
  inc(&x);
}
```

```
std::atomic<int> x;
void two() {
  x += 2;
}
```

SIMILAR EXAMPLE

```
std::atomic<int> x;
void inc(int val) {
  x += 1;
  x += val;
}
```

```
_Z3inci:
  lock incl x(%rip)
  lock addl %edi, x(%rip)
```

OPPORTUNITIES THROUGH INLINING

```
template<typename T>
bool silly(std::atomic<T> *x, T expected, T desired) {
  x->compare_exchange_strong(expected, desired); // Inlined.
  return expected == desired;
}
```

```
template<typename T>
bool silly(std::atomic<T> *x, T expected, T desired) {
  return x->compare_exchange_strong(expected, desired);
}
```

A TRICKY ONE

Works for any memory order but `release` and `acq_rel`

```
template<typename T>
bool optme(std::atomic<T> *x, T desired) {
  T expected = desired;
  return x->compare_exchange_strong(expected, desired
    std::memory_order_seq_cst, std::memory_order_relaxed);
}
```

```
template<typename T>
bool optme(std::atomic<T> *x, T desired) {
  return x->load(std::memory_order_seq_cst) == desired; // †
}
```

† compiler: mark transformed load as *release sequence* ‡

‡ as defined in section 1.10 of the C++ standard

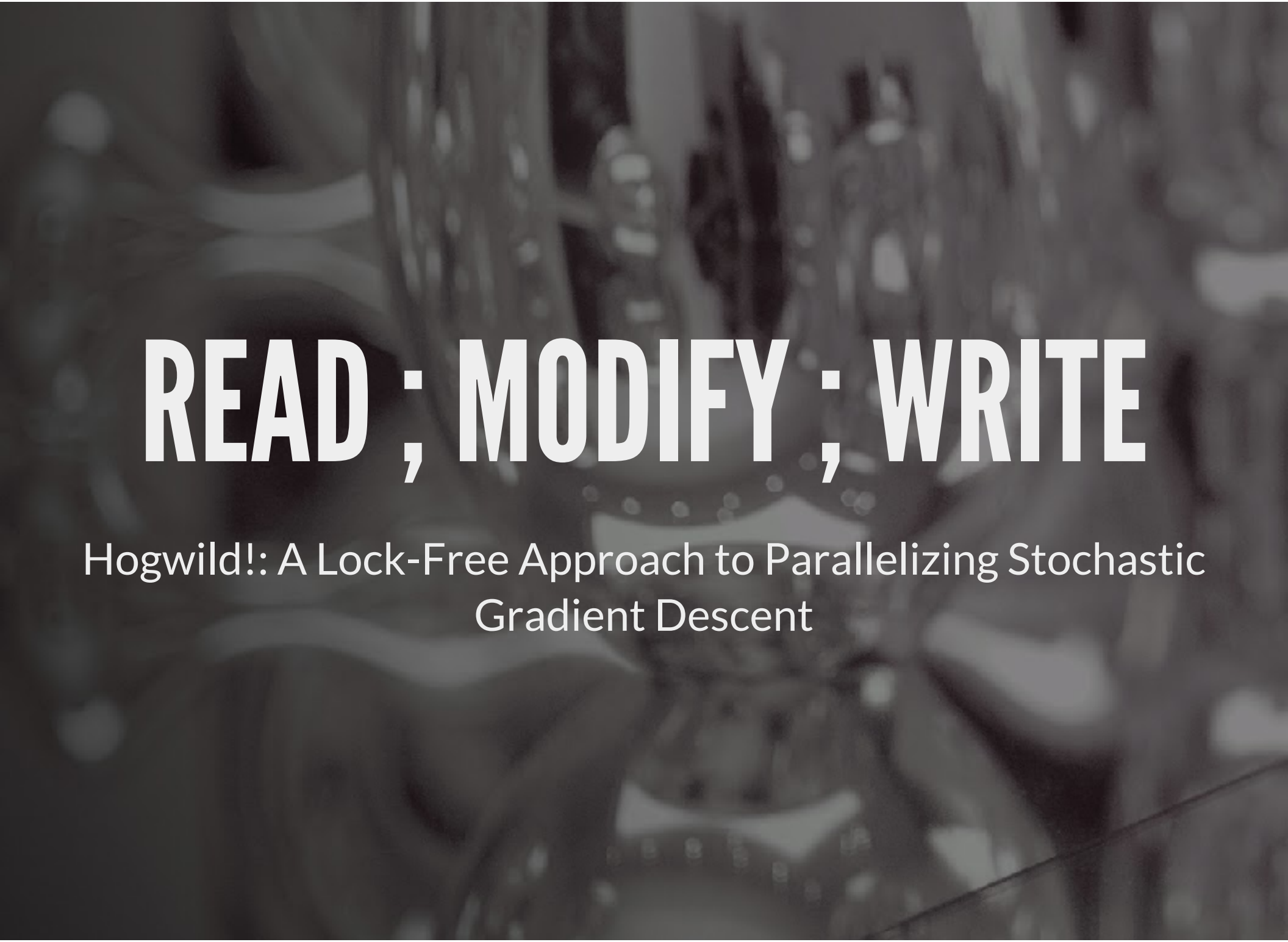
STRONGER, FASTER

```
template<typename T>
T optmetoo(std::atomic<T> *x, T y) {
  T z = x->load();
  x->store(y);
  return z;
}
```

```
template<typename T>
T optmetoo(std::atomic<T> *x, T y) {
  return x->exchange(y);
}
```

BETTER?

INTERESTING PROBLEM...


READ ; MODIFY ; WRITE
Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

RACY

```
float y;
void g(float a) { y += a; }
```

```
        addss   y(%rip), %xmm0
        movss   %xmm0, y(%rip)
```

VOLATILE

```
volatile float yv;
void gv(float a) { yv += a; }
```

```
        addss   yv(%rip), %xmm0
        movss   %xmm0, yv(%rip)
```

CORRECT?

```
std::atomic<float> x;
void f(float a) {
  x.store(x.load(std::memory_order_relaxed) + a,
          std::memory_order_relaxed);
}
```

```
        movl    x(%rip), %eax
        movd    %eax, %xmm1
        addss   %xmm0, %xmm1
        movd    %xmm1, %eax
        movl    %eax, x(%rip)
```

```
        addss   x(%rip), %xmm0
        movss   %xmm0, x(%rip)
```

Optimization FTW!

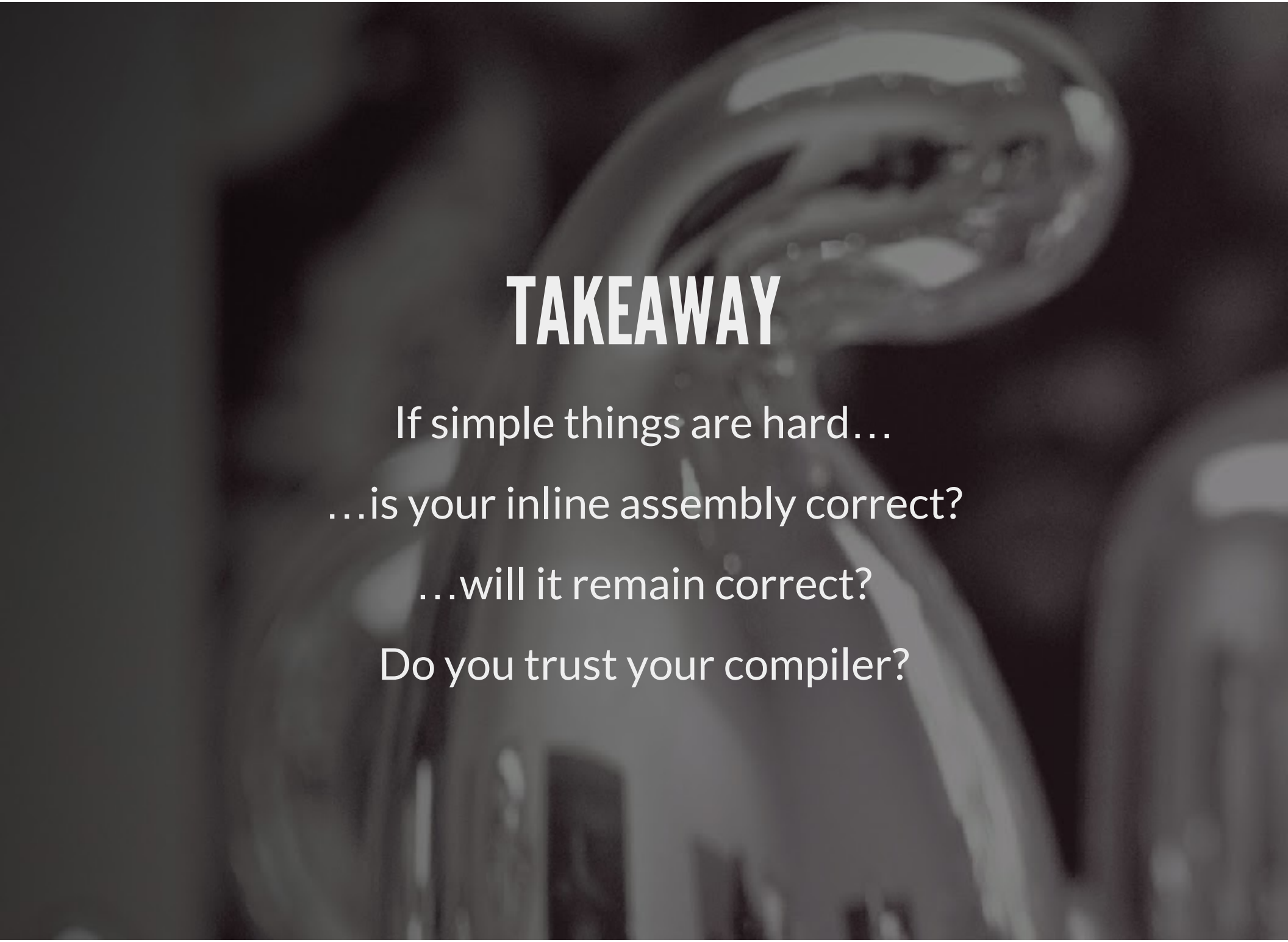
MOAR!

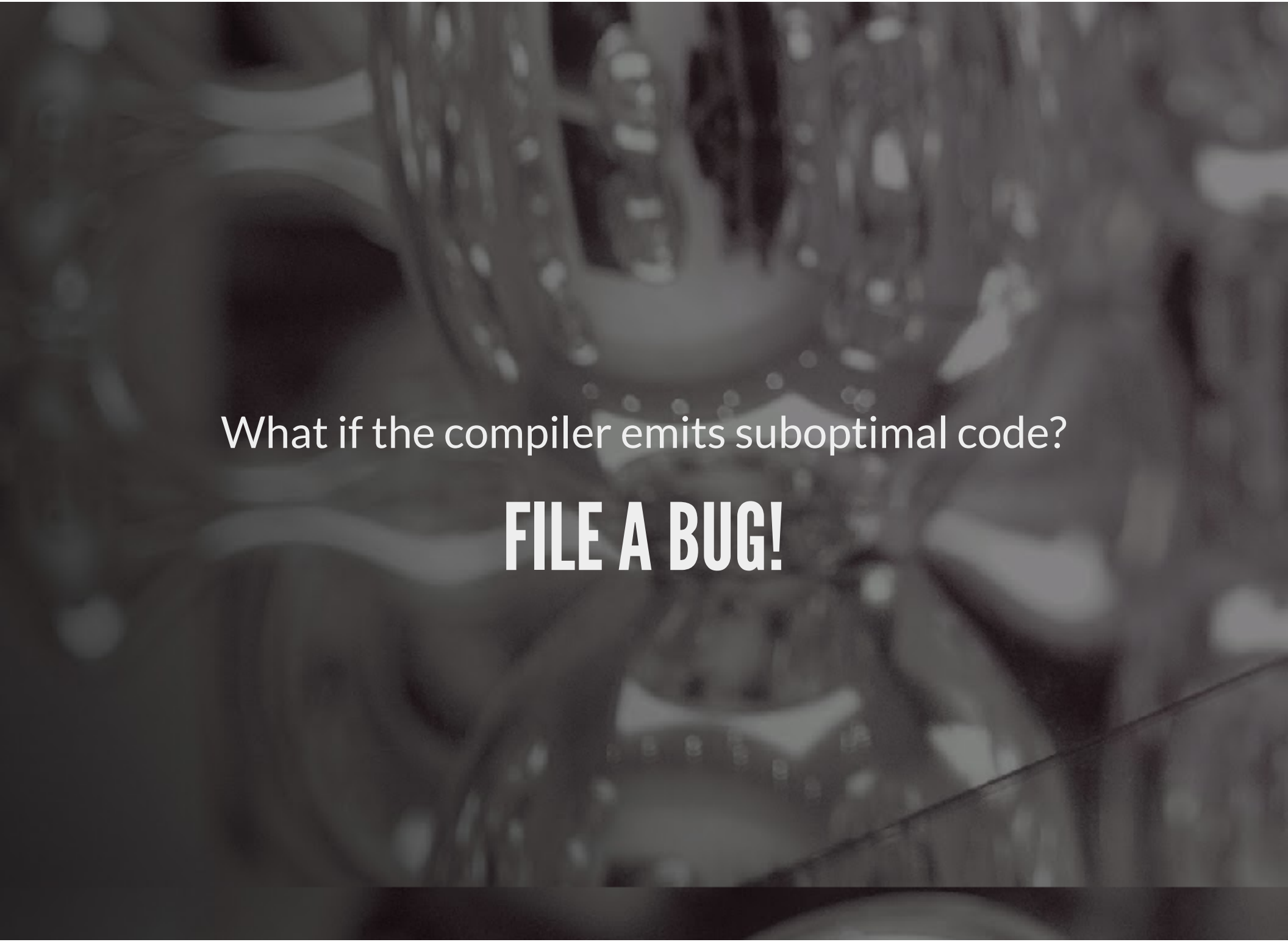
Inlining and constant propagation

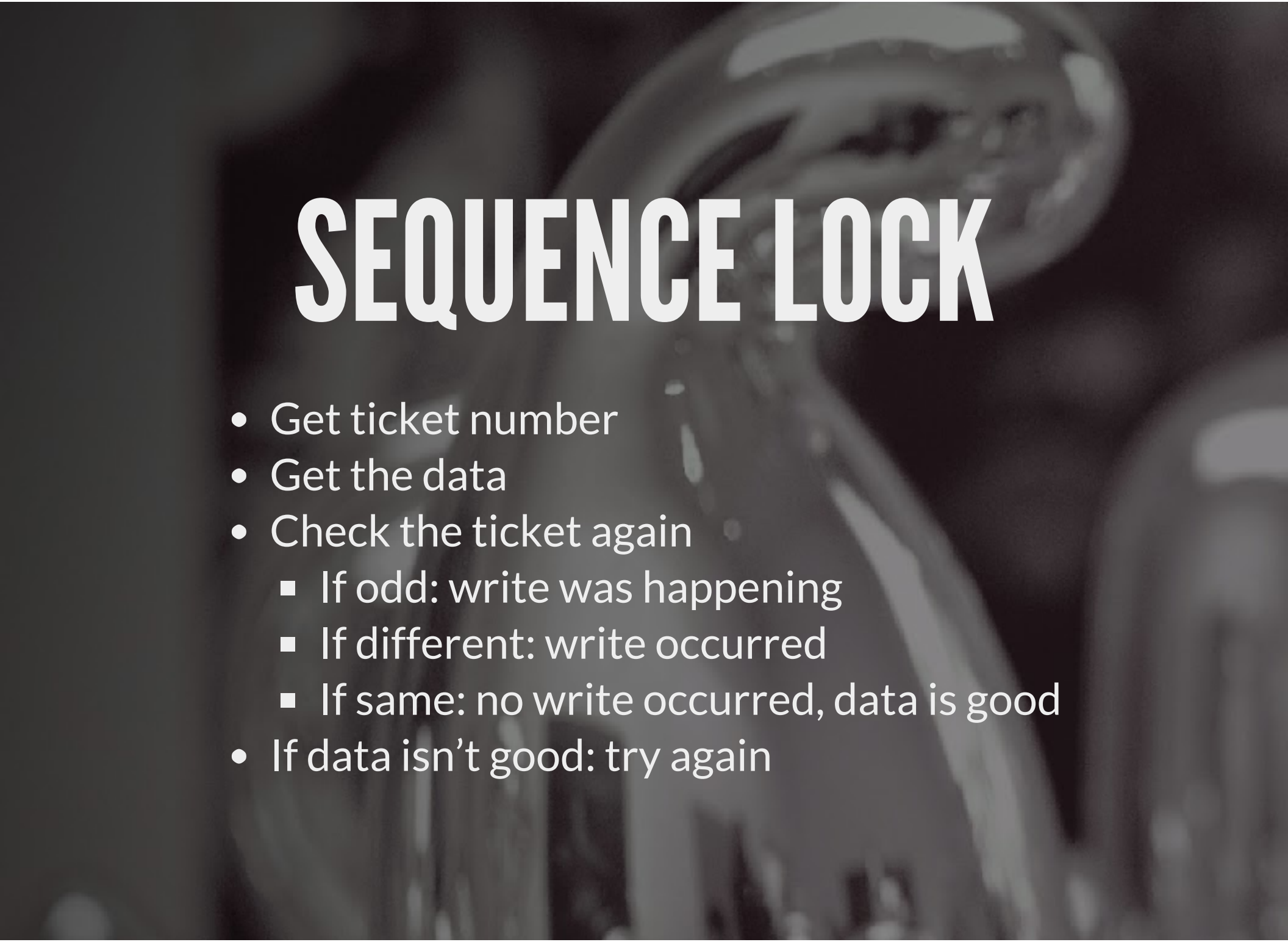
`atomic<T>::fetch_and(~(T)0)` → `atomic<T>::load()`

Same for `fetch_or(0)` and `fetch_xor(0)`

`atomic<T>::fetch_and(0)` → `atomic<T>::store(0)`


TAKEAWAY
If simple things are hard…
…is your inline assembly correct?
…will it remain correct?
Do you trust your compiler?


What if the compiler emits suboptimal code?
FILE A BUG!



SEQUENCE LOCK

- Get ticket number
- Get the data
- Check the ticket again
  - If odd: write was happening
  - If different: write occurred
  - If same: no write occurred, data is good
- If data isn’t good: try again

```
std::tuple<T, T> reader() {
  T d1, d2; unsigned seq0, seq1;
  do {
    seq0 = seq.load(std::memory_order_acquire);
    d1 = data1.load(std::memory_order_relaxed);
    d2 = data2.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    seq1 = seq.load(std::memory_order_relaxed);
  } while (seq0 != seq1 || seq0 & 1);
  return {d1, d2};
}
```

```
void writer(T d1, T d2) {
  unsigned seq0 = seq.load(std::memory_order_relaxed);
  seq.store(seq0 + 1, std::memory_order_relaxed);
  data1.store(d1, std::memory_order_release);
  data2.store(d2, std::memory_order_release);
  seq.store(seq0 + 2, std::memory_order_release);
}
```


We *really* want a release store
(there is no release store)

READ-DON'T-MODIFY-WRITE

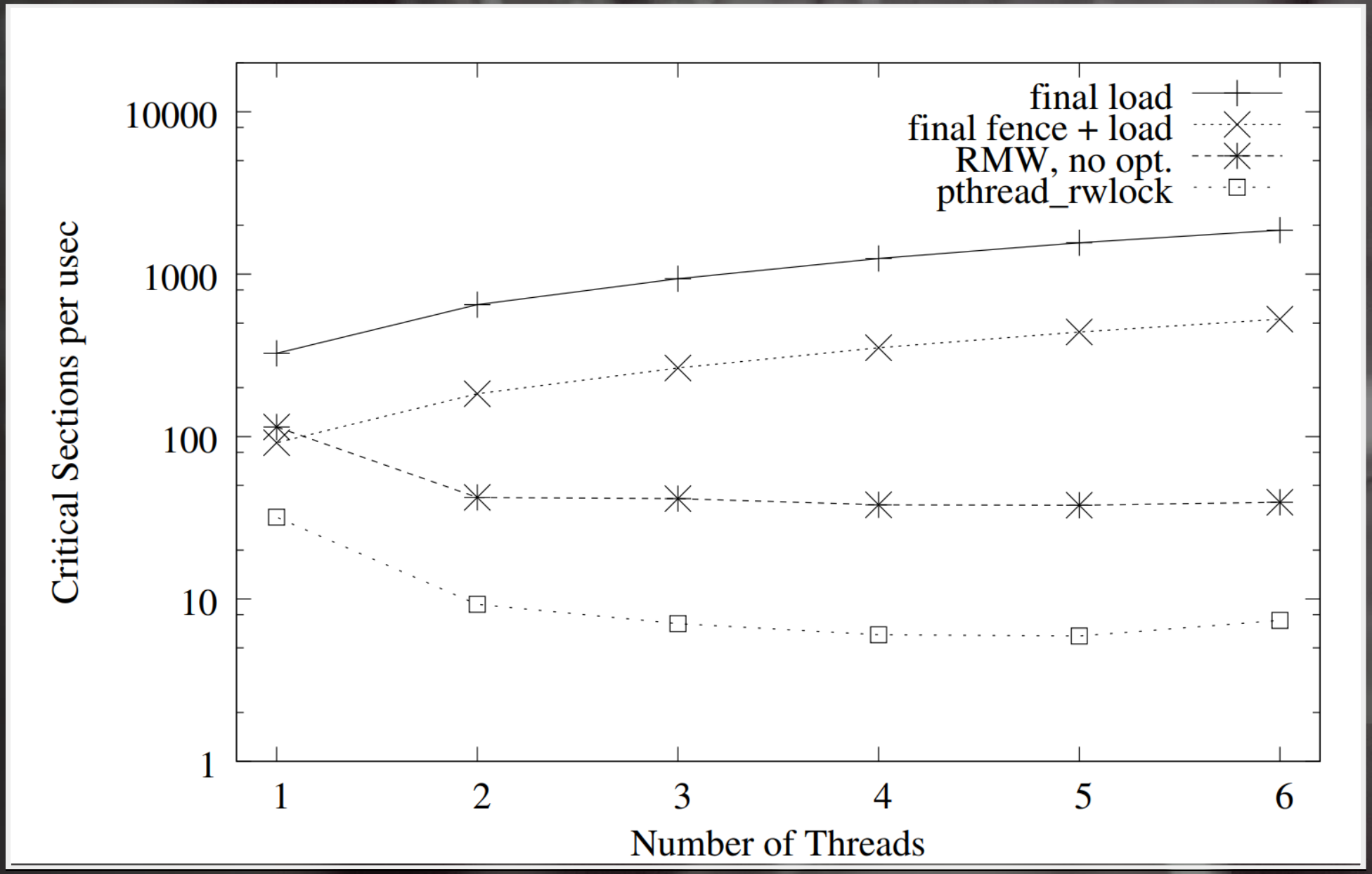
```
T d1, d2;
unsigned seq0, seq1;
do {
  seq0 = seq.load(std::memory_order_acquire);
  d1 = data1.load(std::memory_order_relaxed);
  d2 = data2.load(std::memory_order_relaxed);
  seq1 = seq.fetch_add(0, std::memory_order_release);
} while (seq0 != seq1 || seq0 & 1);
```

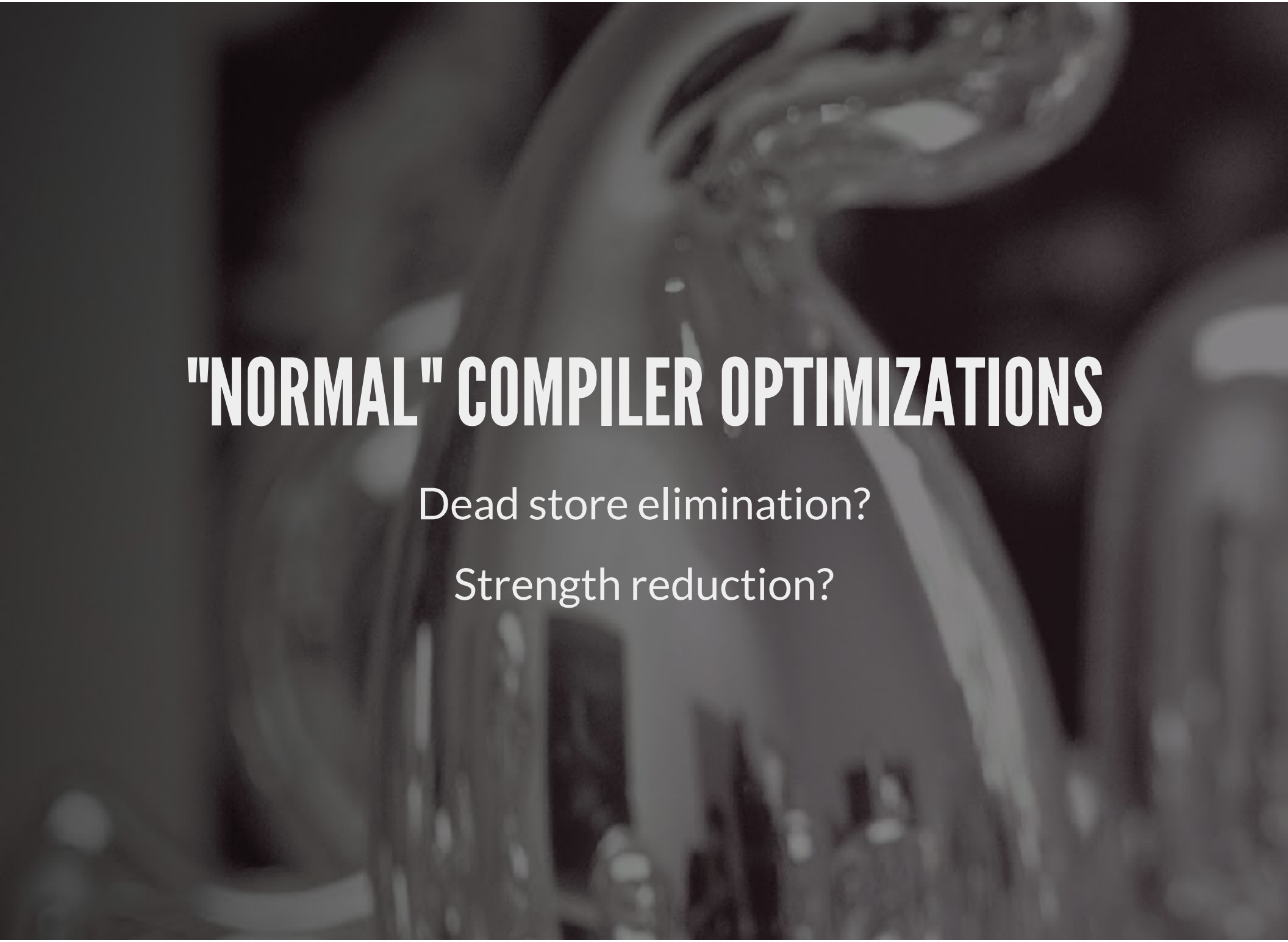
```
 mov    0x200a76(%rip),%edx        # seq
 mov    0x200a74(%rip),%eax        # data1
 mov    0x200a72(%rip),%ecx        # data2
# acquire fence
 mov    0x200a64(%rip),%esi        # seq
```

```
 mov    0x2004c6(%rip),%ecx        # seq
 mov    0x2004bc(%rip),%esi        # data1
 xor    %edx,%edx
 mov    0x2004af(%rip),%r8d        # data2
 lock xadd %edx,0x2004af(%rip)     # seq RdMW
```

```
 mov    0x200a46(%rip),%edx        # seq
 mov    0x200a44(%rip),%eax        # data1
 mov    0x200a42(%rip),%ecx        # data2
 mfence                            # optimized RdMW!
 mov    0x200a31(%rip),%esi        # seq
```

WHAT'S BETTER?




"NORMAL" COMPILER OPTIMIZATIONS
Dead store elimination?
Strength reduction?

RELAXED OPTIMIZATION

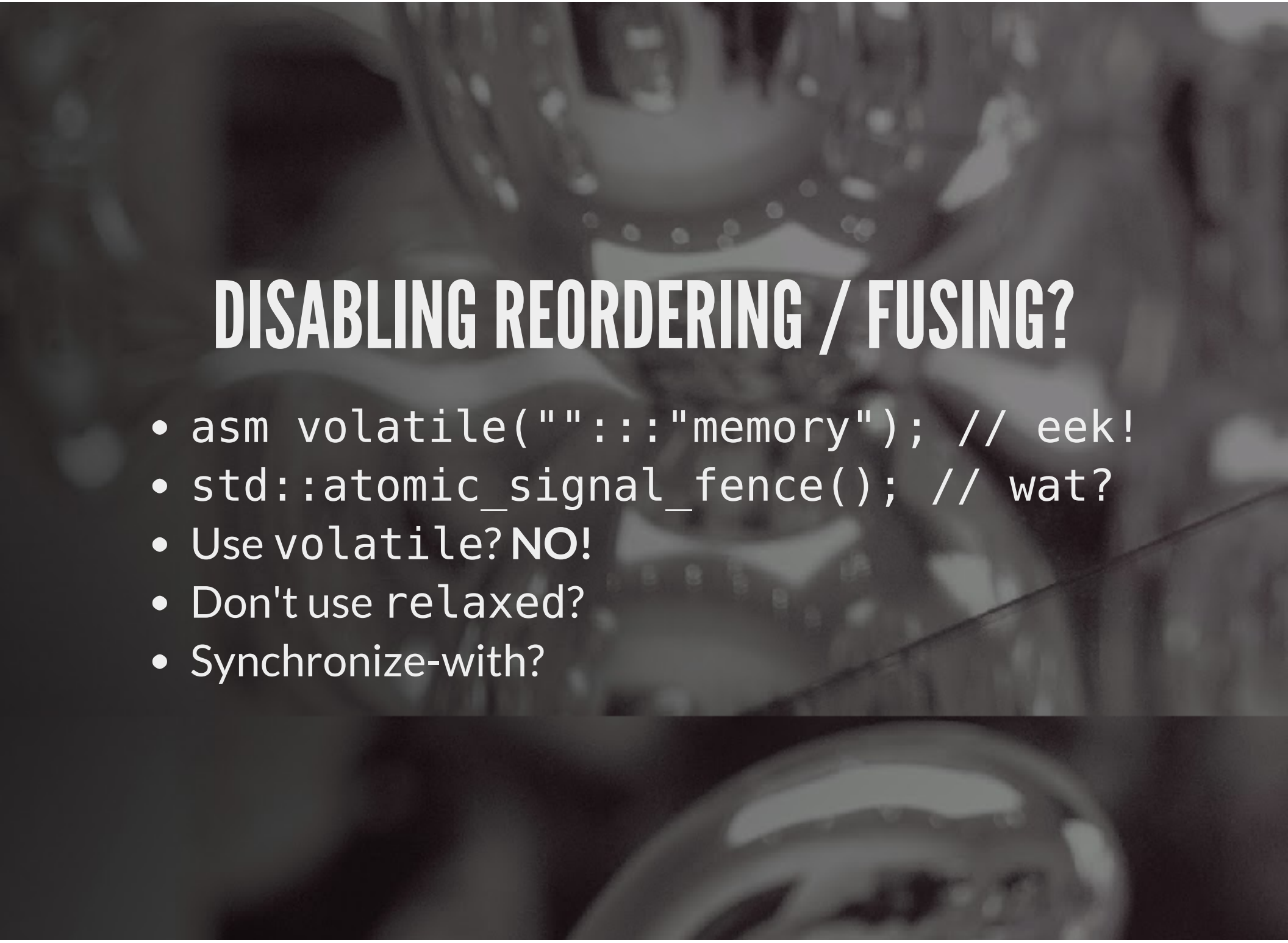
```
std::atomic<int> x, y;
void relaxed() {
  x.fetch_add(1, std::memory_order_relaxed);
  y.fetch_add(1, std::memory_order_relaxed);
  x.fetch_add(1, std::memory_order_relaxed);
  y.fetch_add(1, std::memory_order_relaxed);
}
```

```
std::atomic<int> x, y;
void relaxed() {
  x.fetch_add(2, std::memory_order_relaxed);
  y.fetch_add(2, std::memory_order_relaxed);
}
```

PROGRESS BAR

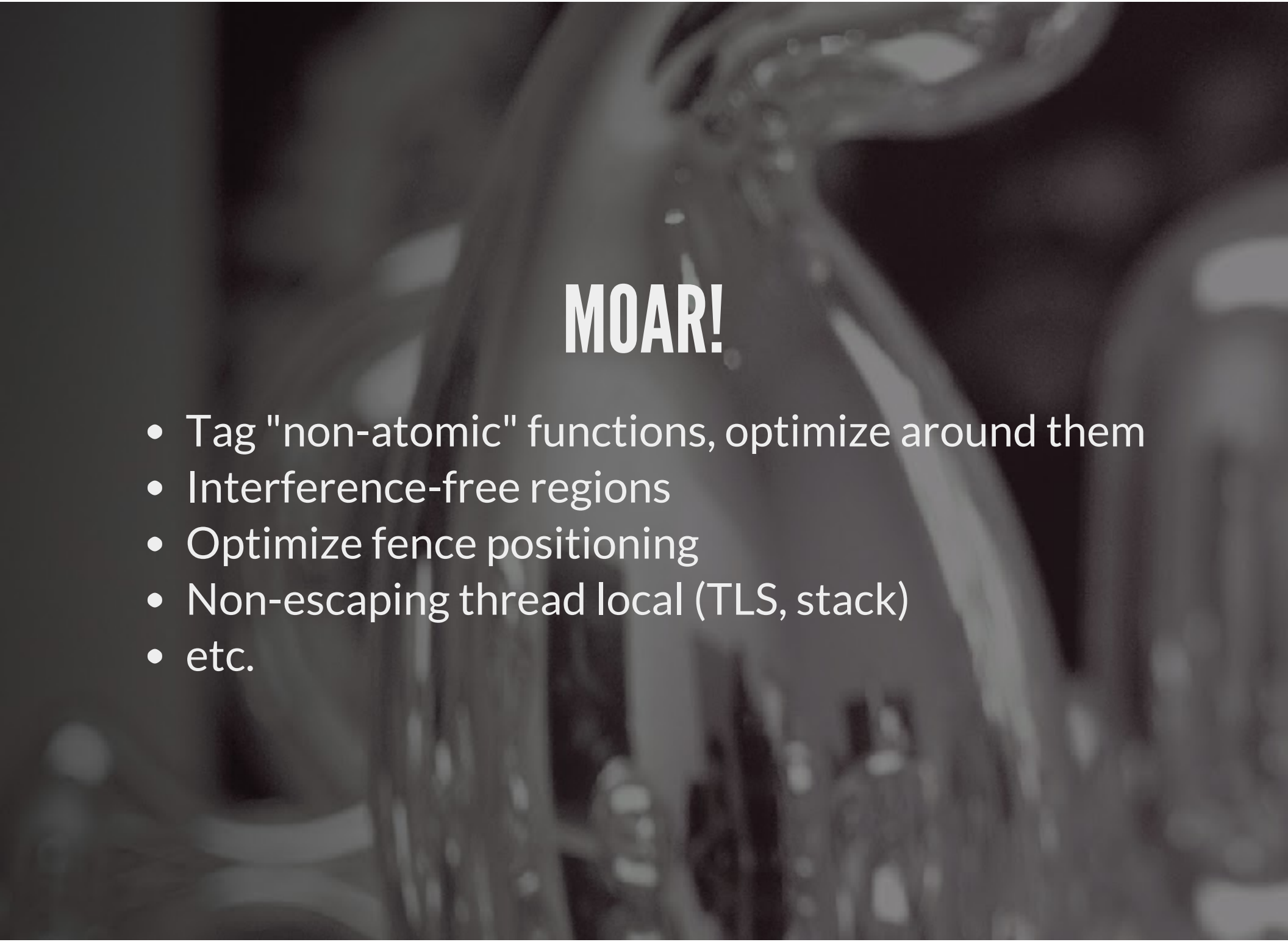
```
atomic<int> progress(0);
f() {
    for (i = 0; i < 1000000; ++i) {
        // Heavy work, no shared memory...
        ++progress;
    }
}
```

```
atomic<int> progress(0);
f() {
    int my_progress = 0; // register allocated
    for (i = 0; i < 1000000; ++i) {
        // Heavy work, no shared memory...
        ++my_progress;
    }
    progress += my_progress;
}
```



DISABLING REORDERING / FUSING?

- `asm volatile("":::"memory"); // eek!`
- `std::atomic_signal_fence(); // wat?`
- Use `volatile`? **NO!**
- Don't use `relaxed`?
- Synchronize-with?



MOAR!

- Tag "non-atomic" functions, optimize around them
- Interference-free regions
- Optimize fence positioning
- Non-escaping thread local (TLS, stack)
- etc.

STD::MUTEX FOR SANITY

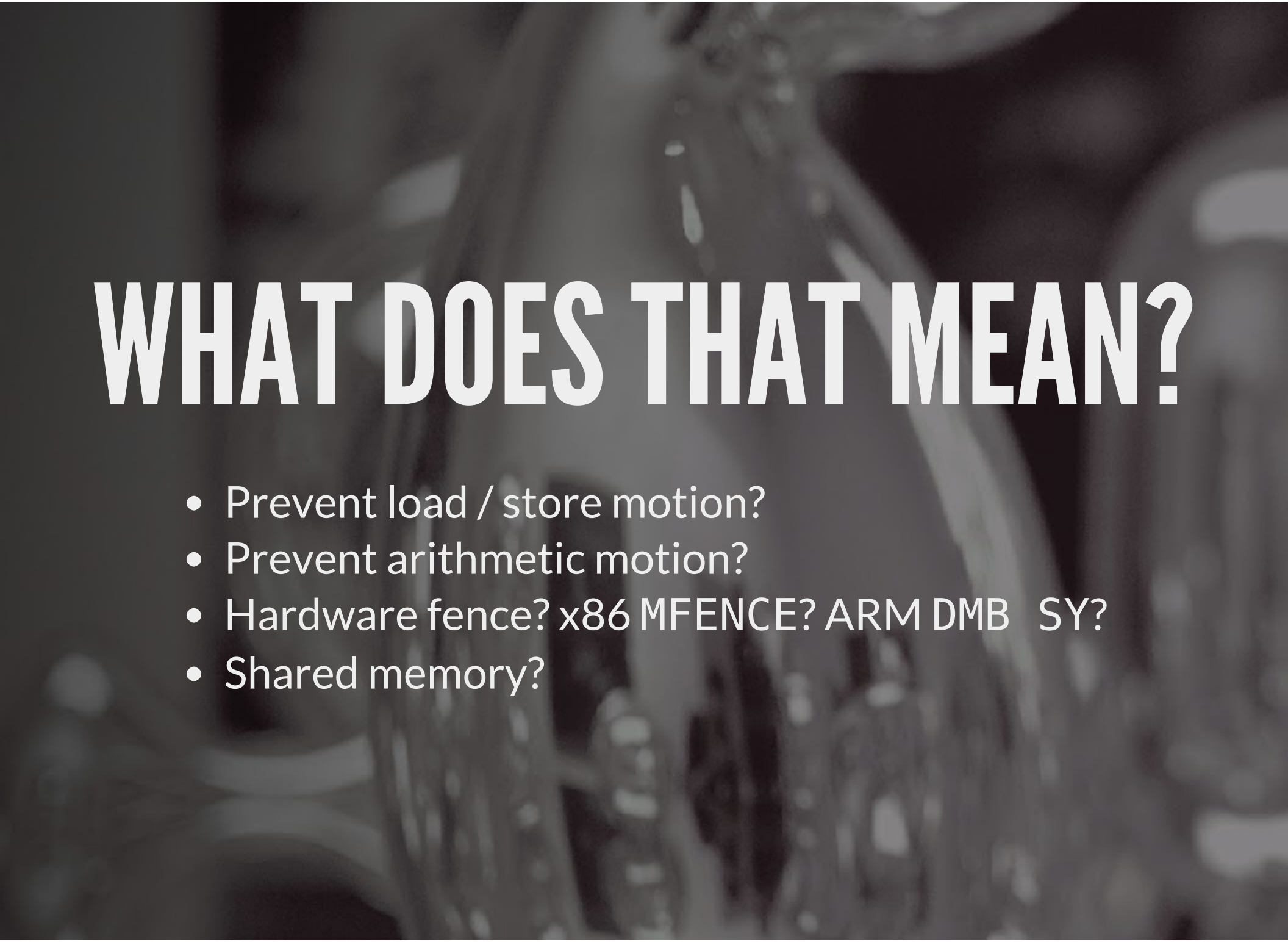
- easier to use correctly
- in theory...
  - it could still get optimized
  - lock: `acquire`; unlock: `release`
- pthread and kernel call makes this hard
- `std::synchronic` to the rescue!


STD::MUTEX
Knows better

VOLATILE

Initial motivation: `getChar()` on PDP-11, reading memory-mapped KBD_STAT I/O register

K&R: The purpose of `volatile` is to force an implementation to suppress optimization that could otherwise occur



WHAT DOES THAT MEAN?

- Prevent load / store motion?
- Prevent arithmetic motion?
- Hardware fence? x86 `MFENCE`? ARM `DMB SY`?
- Shared memory?


IT GETS BETTER

XTENSA GCC OPTIONS

`-mserialize-volatile`

`-mnoserialize-volatile`

When this option is enabled, GCC inserts `MEMW` instructions before `volatile` memory references to guarantee sequential consistency.

MSVC

You can use the `/volatile` compiler switch to modify how the compiler interprets this keyword. If a struct member is marked as `volatile`, then `volatile` is propagated to the whole structure. If a structure does not have a length that can be copied on the current architecture by using one instruction, `volatile` may be completely lost on that structure.

clang emulates this with `-fms-volatile`

WHAT DOES IT MEAN?

¯_(ツ)_/¯

- can tear
- each byte touched exactly once
- no machine-level ordering
- enforces abstract machine ordering with other `volatile`
- can't duplicate
- can't elide

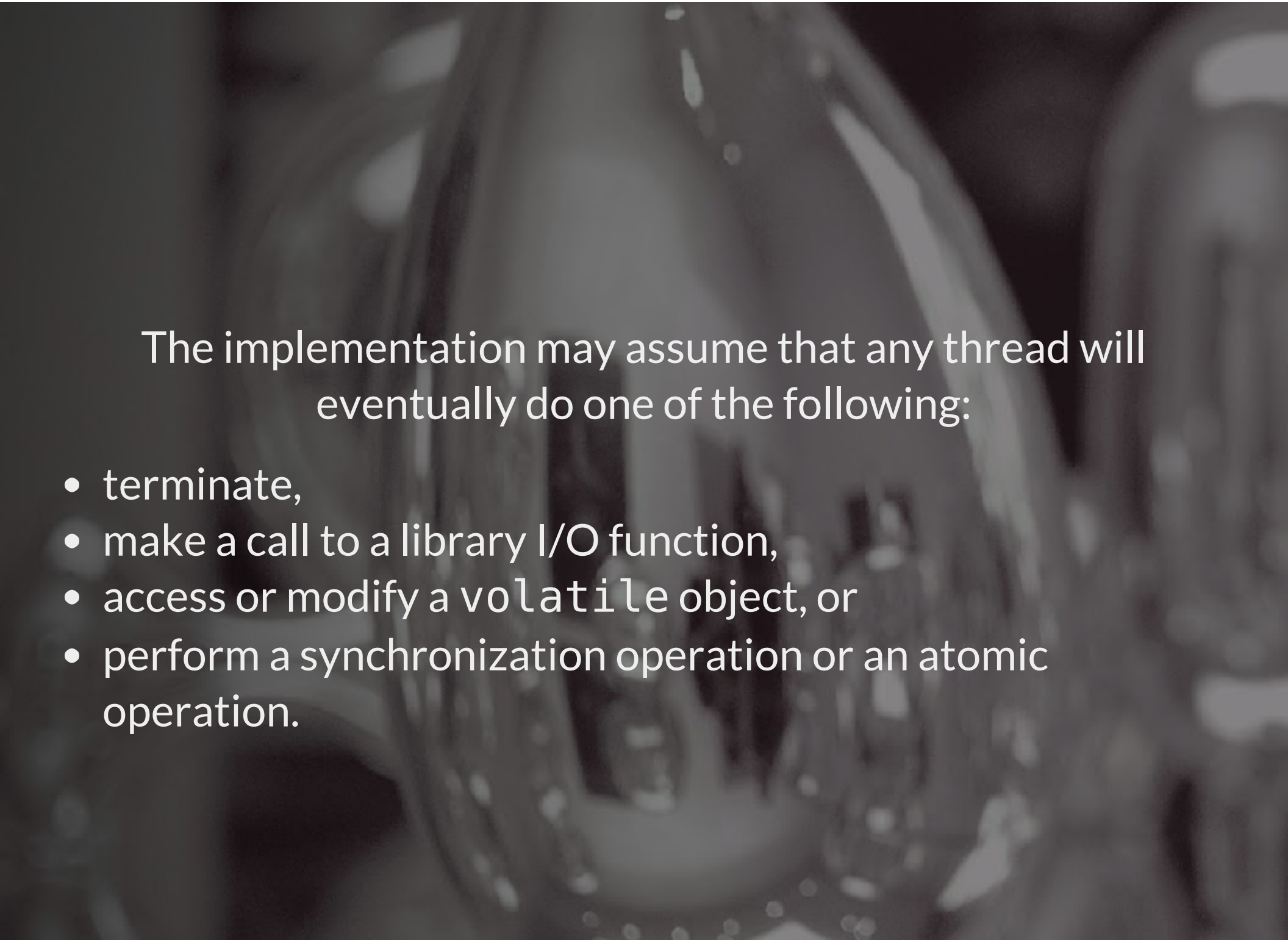

CAN VOLATILE BE
LOCK-FREE?
¯_(ツ)_/¯


CAN VOLATILE DO
RMW?
No

FUN VOLATILE FACT

Is this undefined behavior?

```
volatile int foo;
void bar() { int baz; while (1) baz = foo; }
```

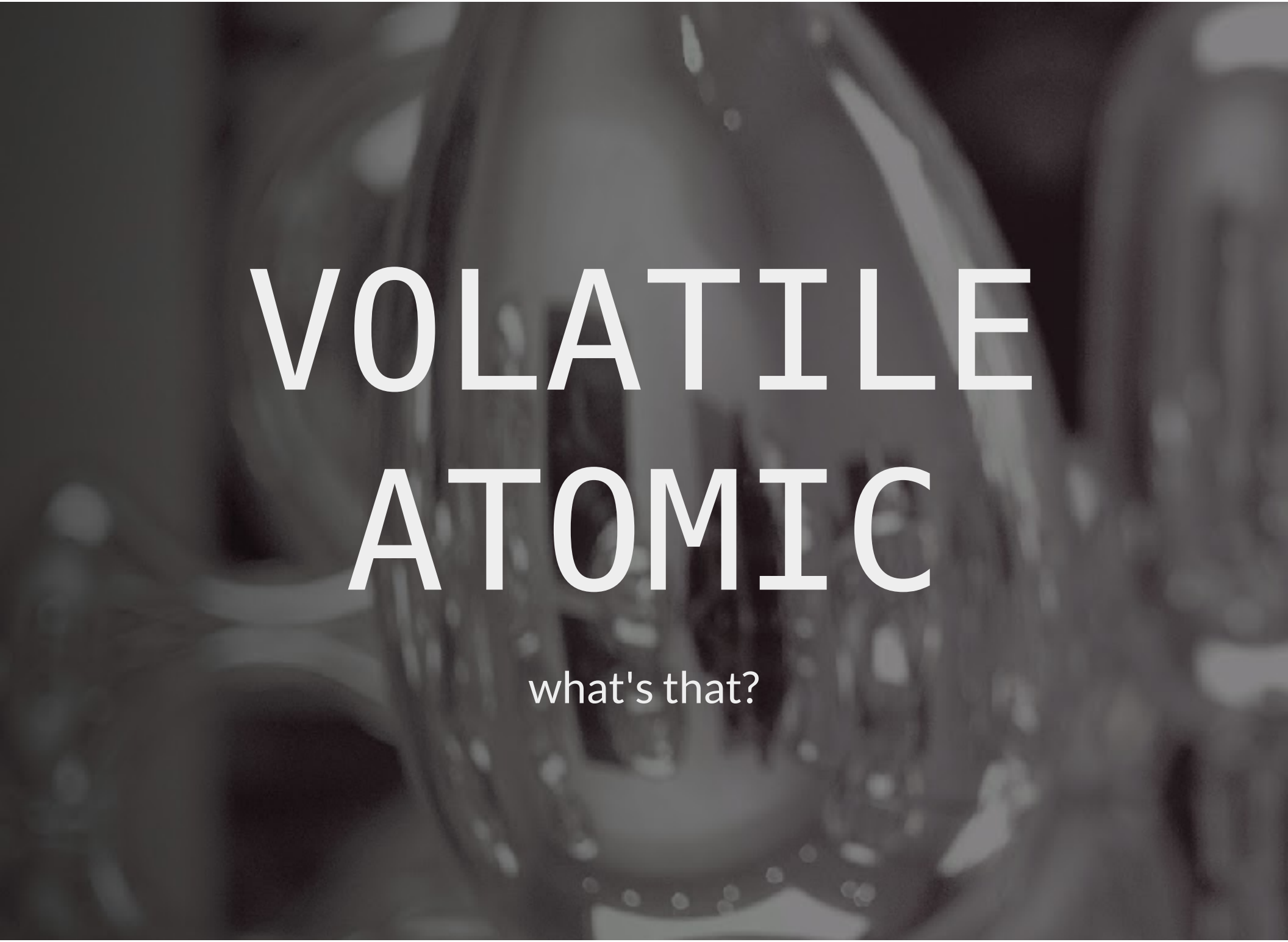


The implementation may assume that any thread will eventually do one of the following:

- terminate,
- make a call to a library I/O function,
- access or modify a `volatile` object, or
- perform a synchronization operation or an atomic operation.

ARE THERE PORTABLE `VOLATILE` USES?

- mark local variables in the same scope as a `setjmp`, preserve value across `longjmp`
- **externally modified**: physical memory mapped in multiple virtual addresses
- `volatile sigatomic_t` may be used to communicate with a signal handler in the same thread


VOLATILE
ATOMIC
what's that?

NON-TEMPORALS

“nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope nope n” — Steve Canon


WHAT DO COMPILERS DO?
by compilers I mean clang


STANDARD LIBRARY
C's _Atomic ↔ C++'s std::atomic
_sync_* builtins
_atomic_* builtins


FRONT-END
does little besides inlining constant memory_order_*

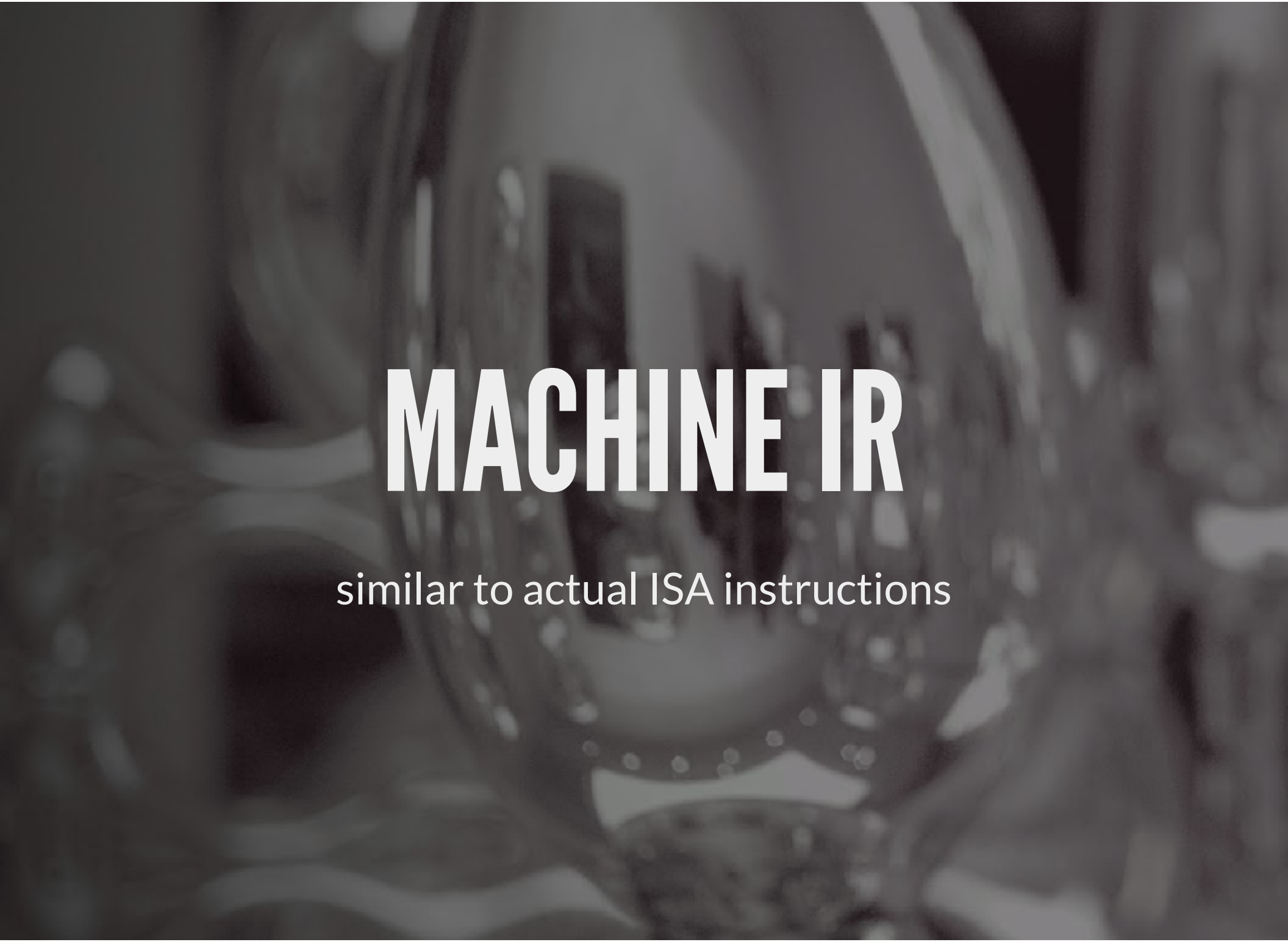
MIDDLE-END

- load atomic [volatile] <ty>, <ty>* <pointer> [singlethread] <ordering>, align <alignment>
- store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread] <ordering>, align <alignment>

MIDDLE-END

- atomicrmw [volatile] <operation> <ty>* <pointer>, <ty> <value> [singlethread] <ordering>
- cmpxchg [weak] [volatile] <ty>* <pointer>, <ty> <cmp>, <ty> <new> [singlethread] <success ordering> <failure ordering>
- fence [singlethread] <ordering>


IR
isSimple, isVolatile, isAtomic: steer clear


MACHINE IR
similar to actual ISA instructions

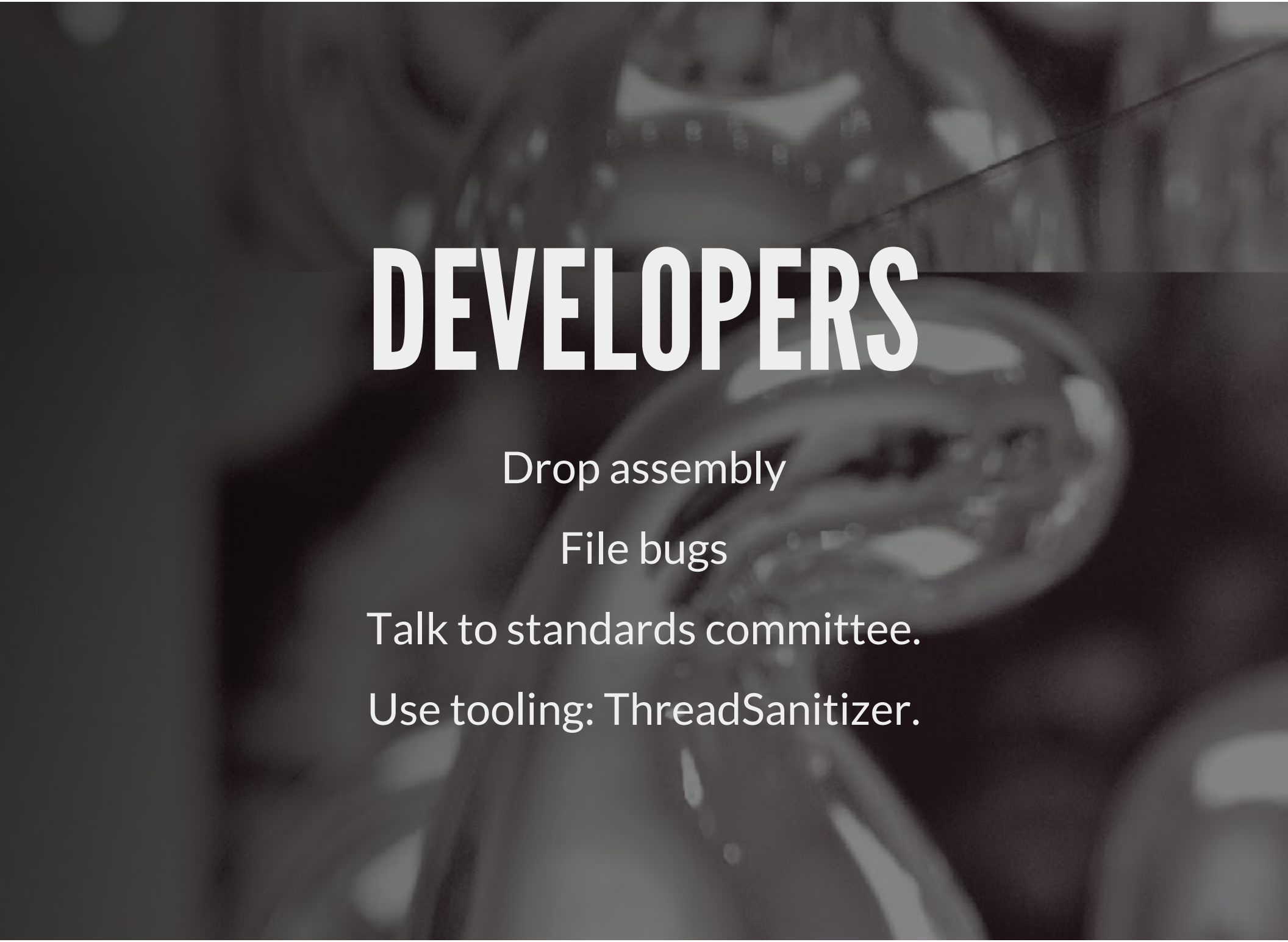

NOW WHAT?

STANDARDS COMMITTEE

Assume optimizations occur, encourage them.

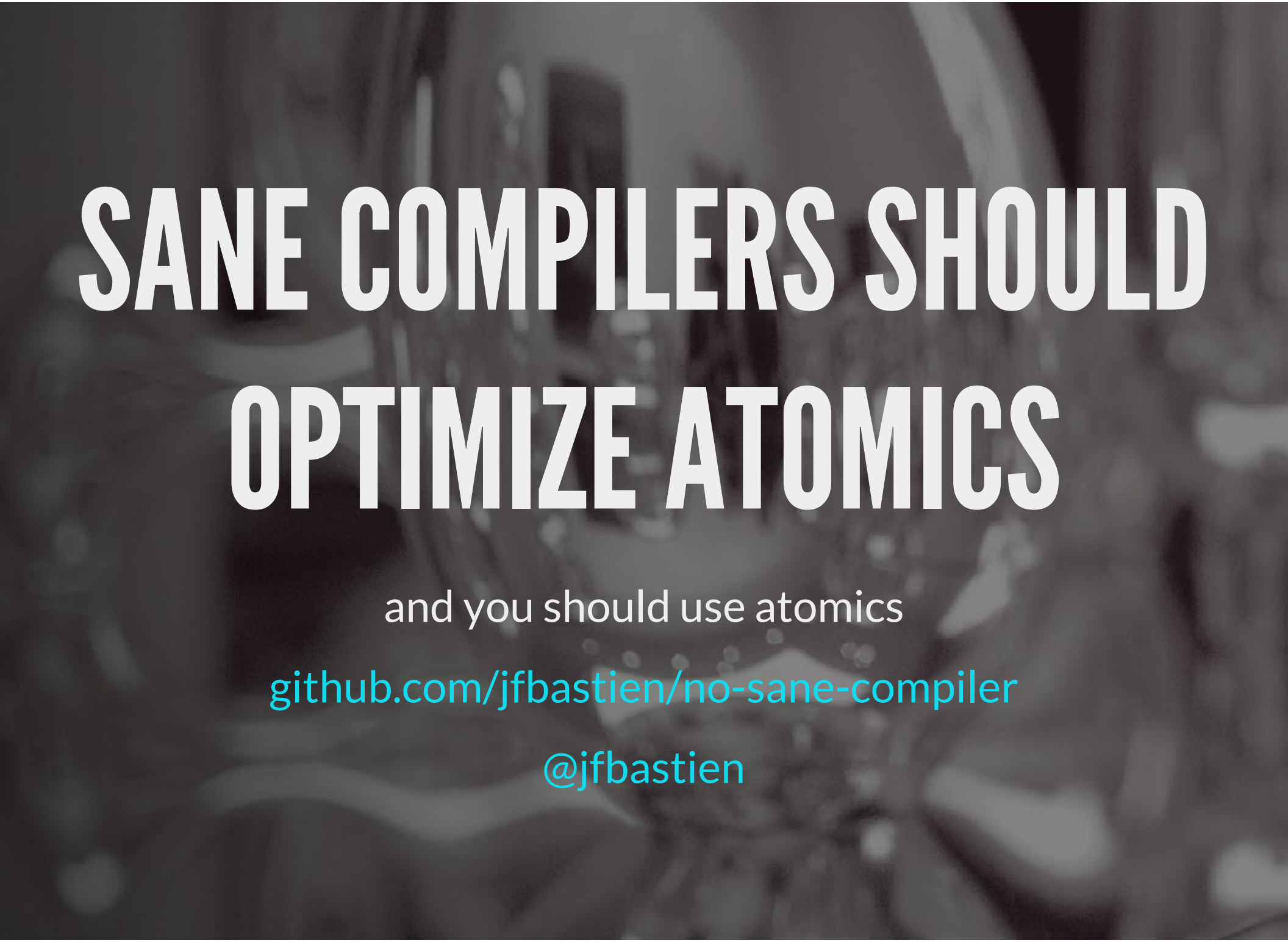
Standardize common practice, enable to-the-metal optimizations.

More libraries: easy to use concurrency & parallelism; hard to get it wrong.


DEVELOPERS
Drop assembly
File bugs
Talk to standards committee.
Use tooling: ThreadSanitizer.


HARDWARE VENDORS
Showcase your hardware’s strengths.


COMPILER WRITERS
Get back to work!


SANE COMPILERS SHOULD
OPTIMIZE ATOMICS
and you should use atomics
github.com/jfbastien/no-sane-compiler
@jfbastien