Cache consistency and the C++ memory model: Writing code for real hardware

Yossi Moalem

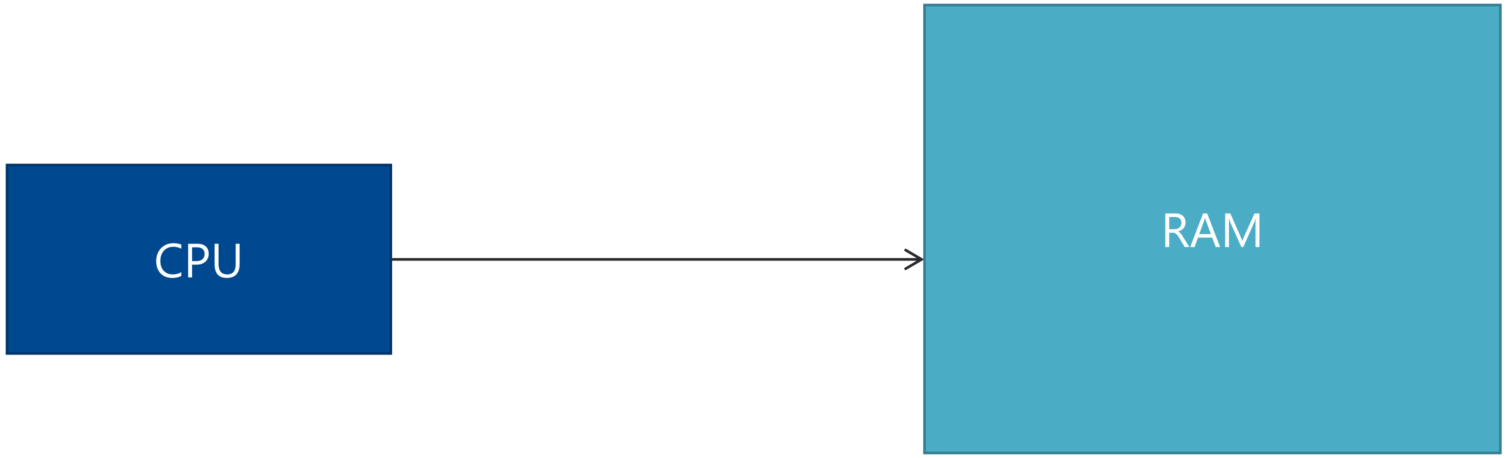
CYBERBIT
PROTECTING A NEW DIMENSION

About Me

- Team leader as Cyberbit
- 15 years as a C++ programmer
- Married with 3 children
- Obsessive cyclist

- E-mail: moalem.yossi@gmail.com

The computer we think we program for

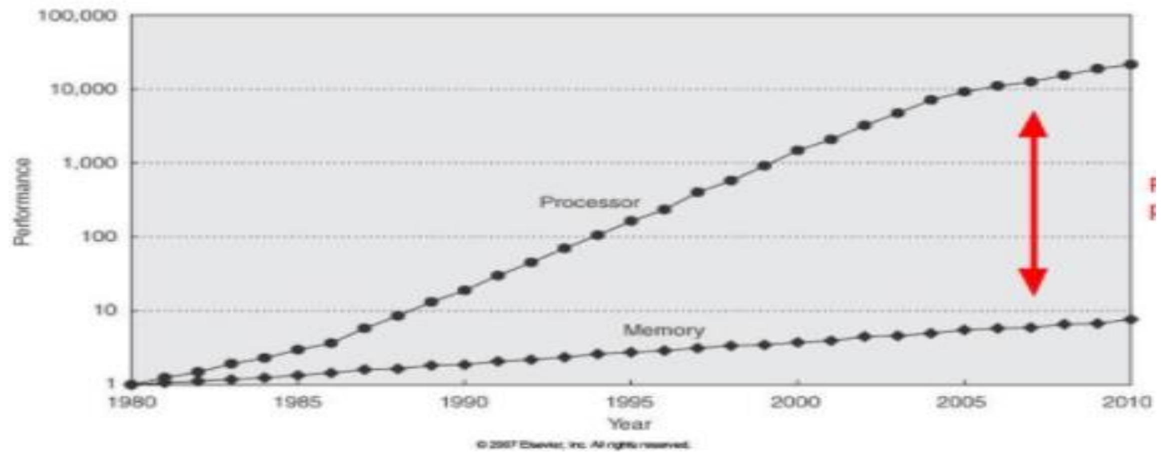


And then came Gordon Moore

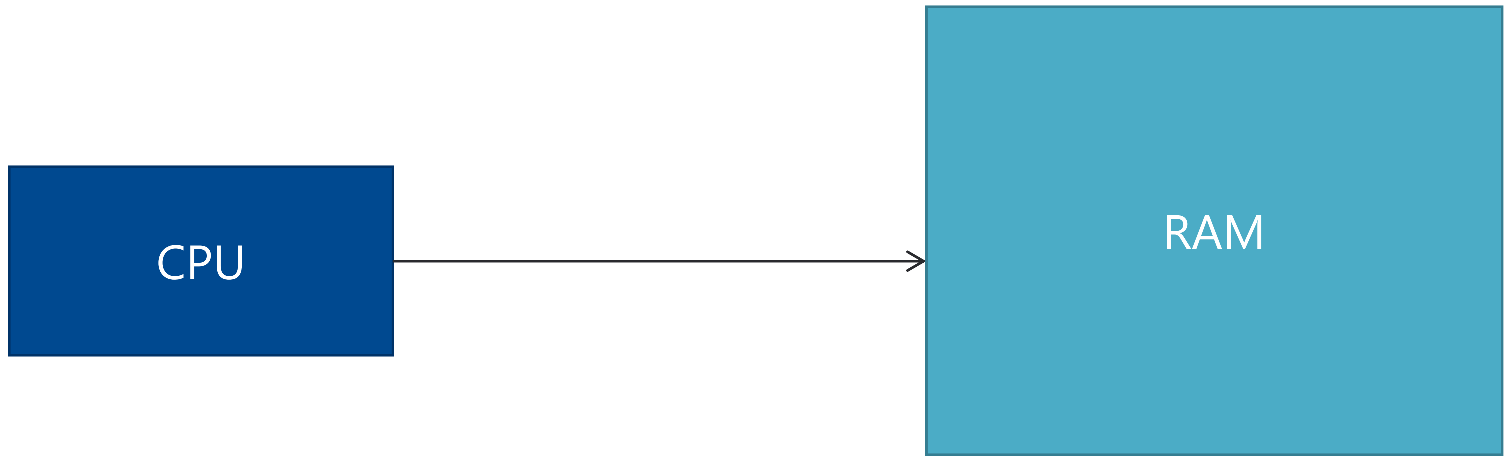
Moore's Law

The number of transistors in a dense integrated circuit doubles every two years

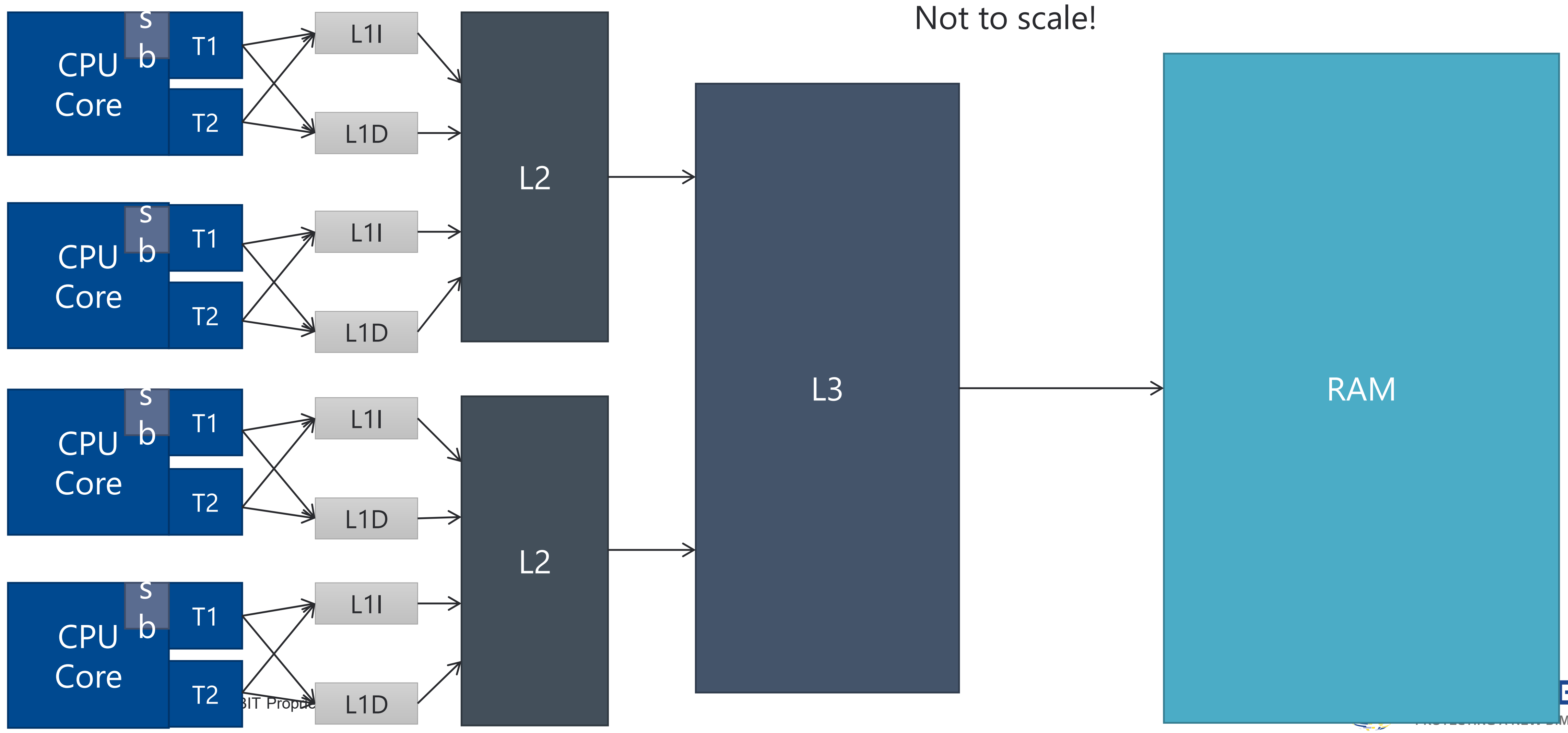
- CPU became much faster very fast
- Memory speed did not advance so fast



The computer we think we program for



The computer we are actually programming for



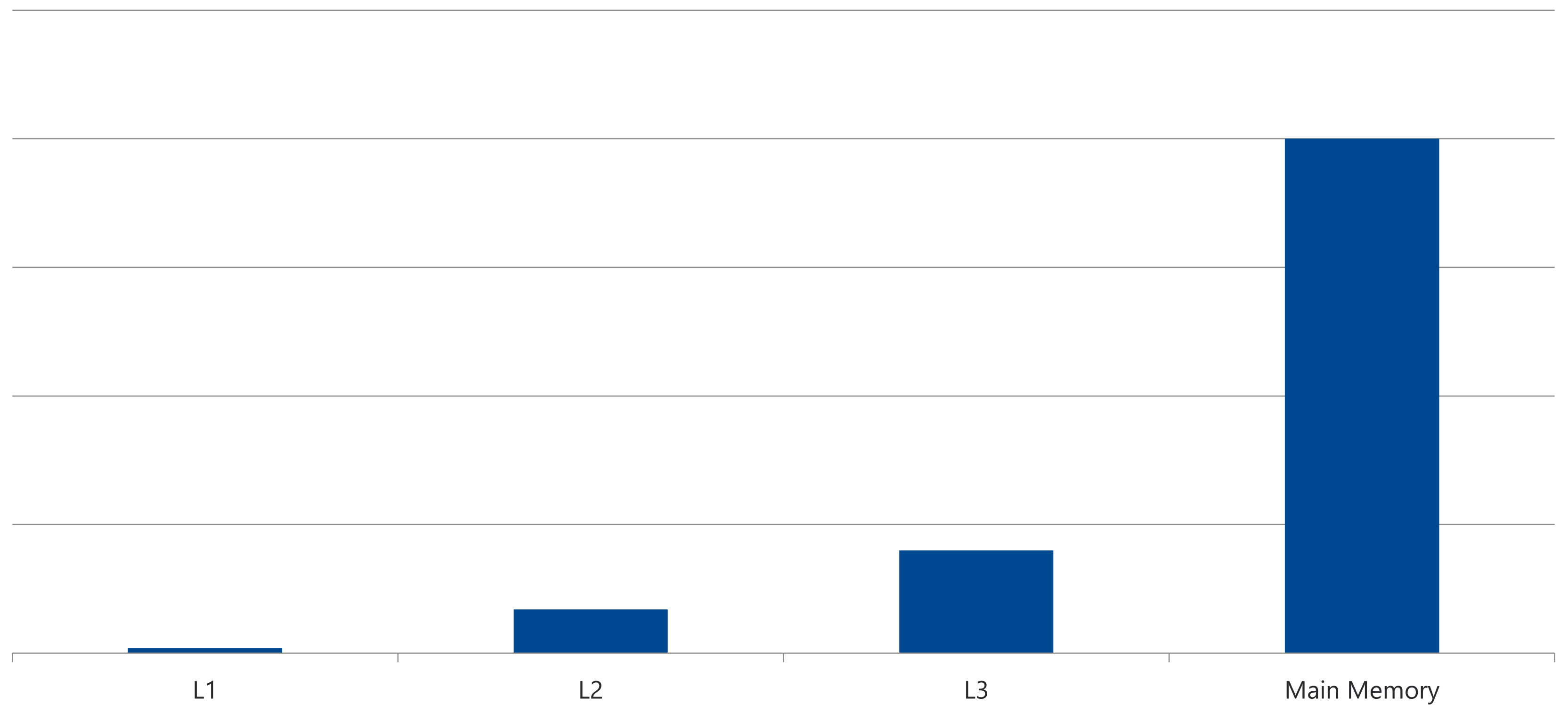
Cache Hierarchy, The numbers

Faster

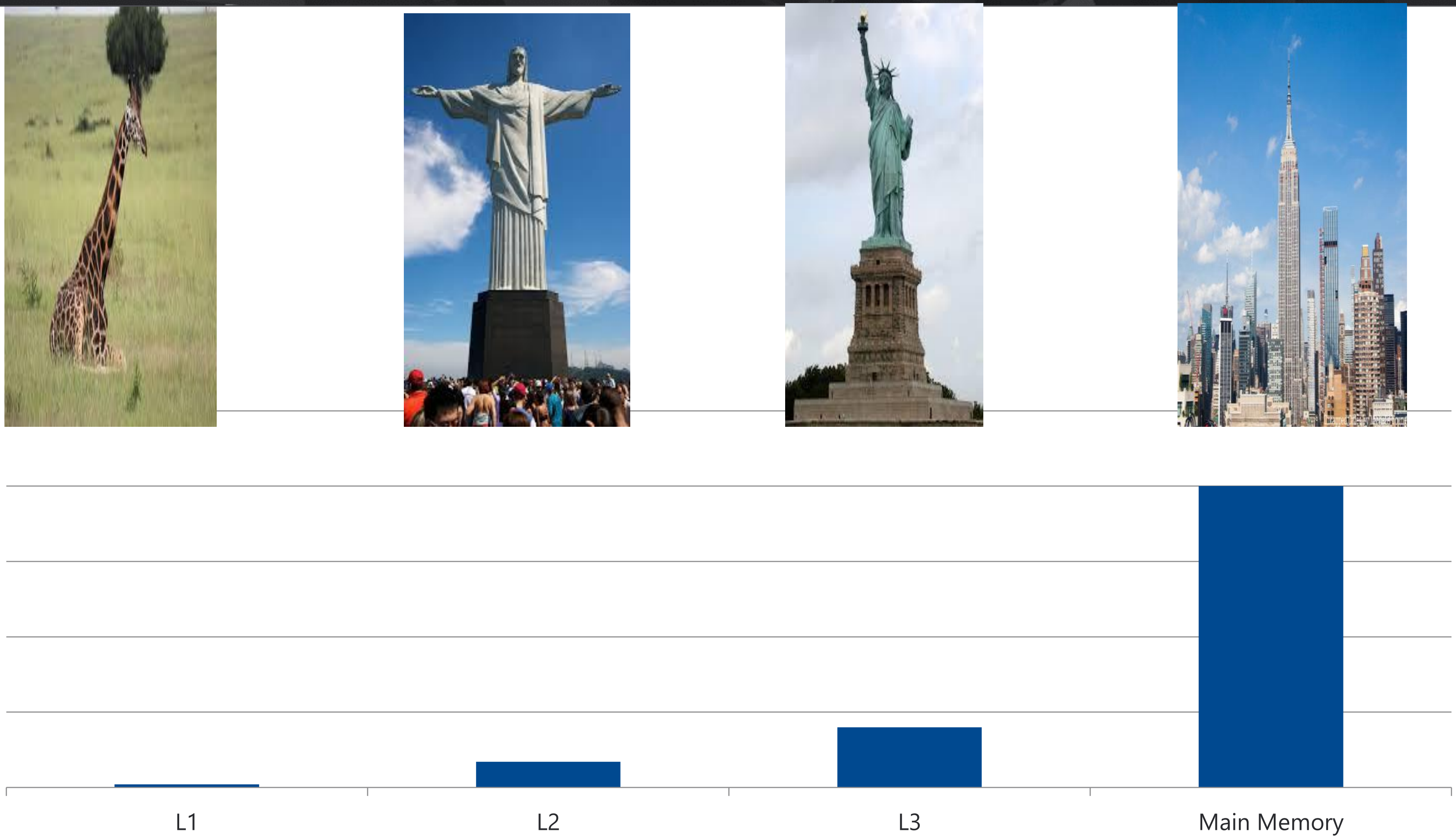
- L1 Cache:
  - 2-4 cycles
  - 32K, for instruction, 32K for data
  - Per core, shared between HW threads
- L2 Cache
  - 15-18 cycles
  - 256K, shared for instructions and data
  - Per CPU
- L3 Cache
  - 30-40 cycles
  - 32M, shared for instructions and data
  - Per machine
- Main Memory
  - Over 100 cycles (150-200 and maybe more!)

Larger

A graph is much better



One picture worth 1000 words

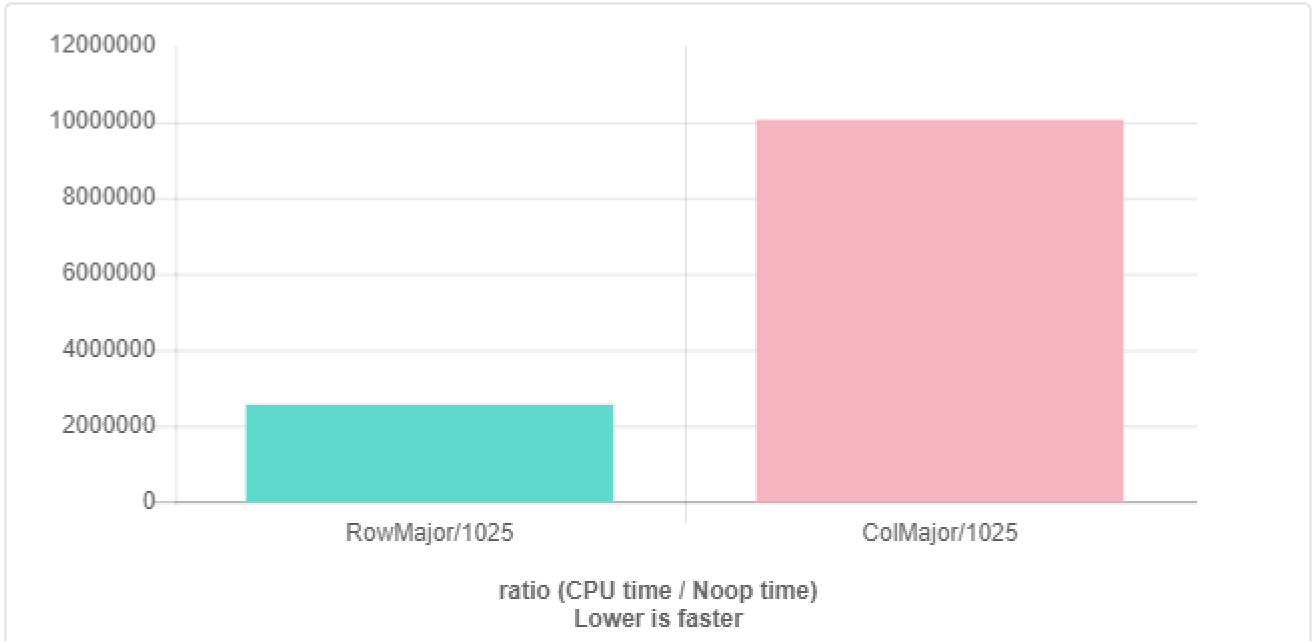


One animation worth 100 pictures

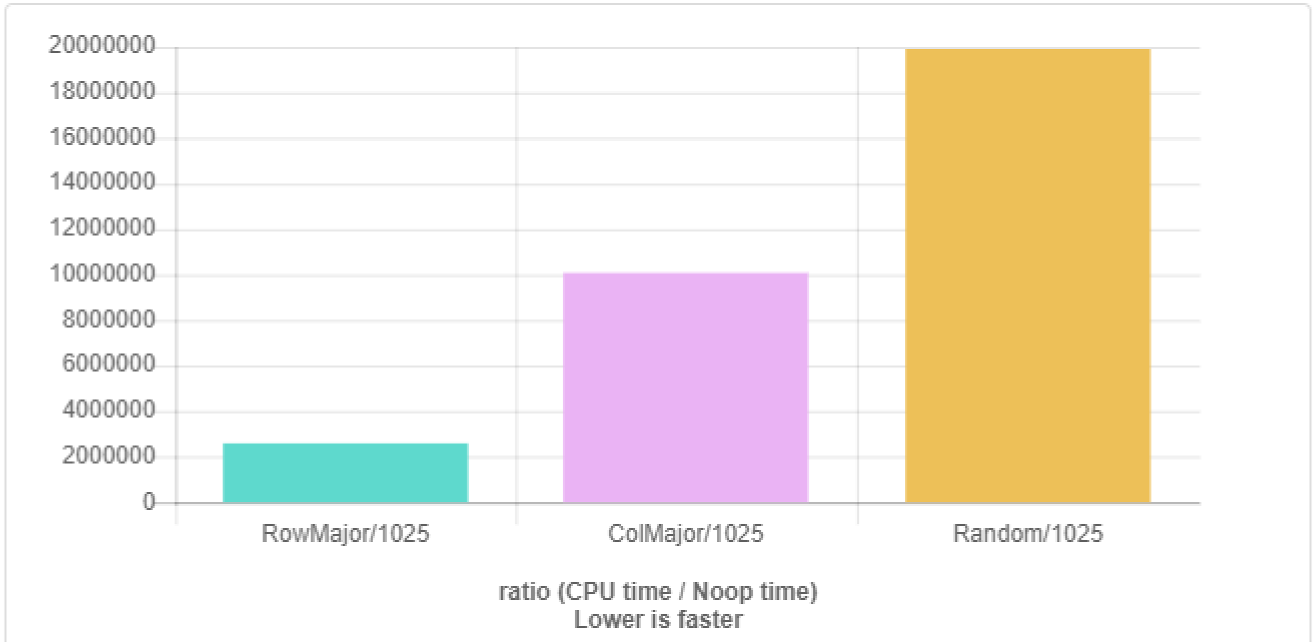
To scale!

L1
L2
L3
CPU
Main memory

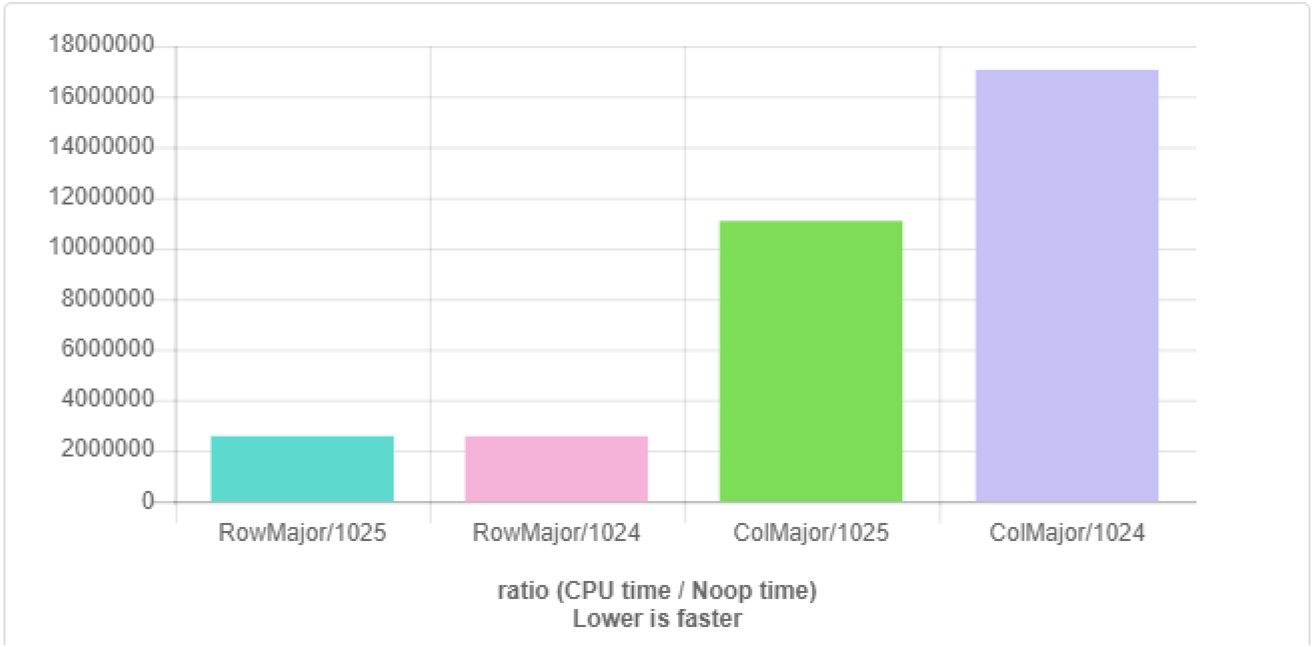
Row by row v’s col by col



Is col by col the worst?



Reducing matrix size

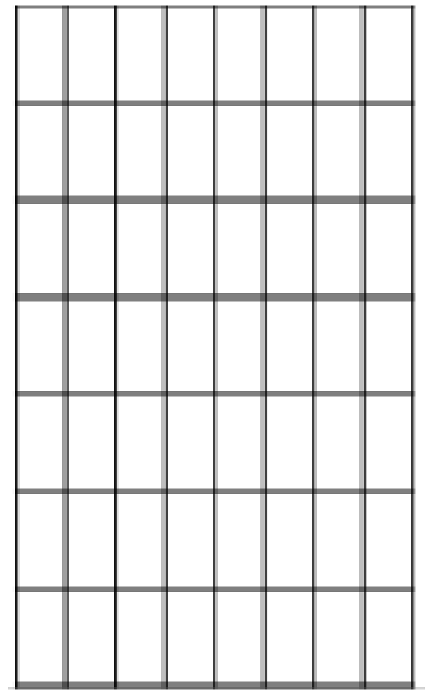


Cache: Requirements

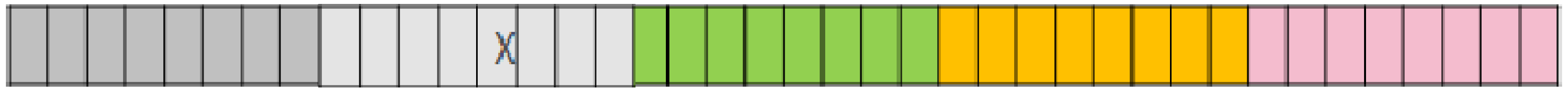
- Swapping in/out should be efficient
- Fast lookup
- Minimize maintenance
- Non-consecutive
- Locality

Cache Line

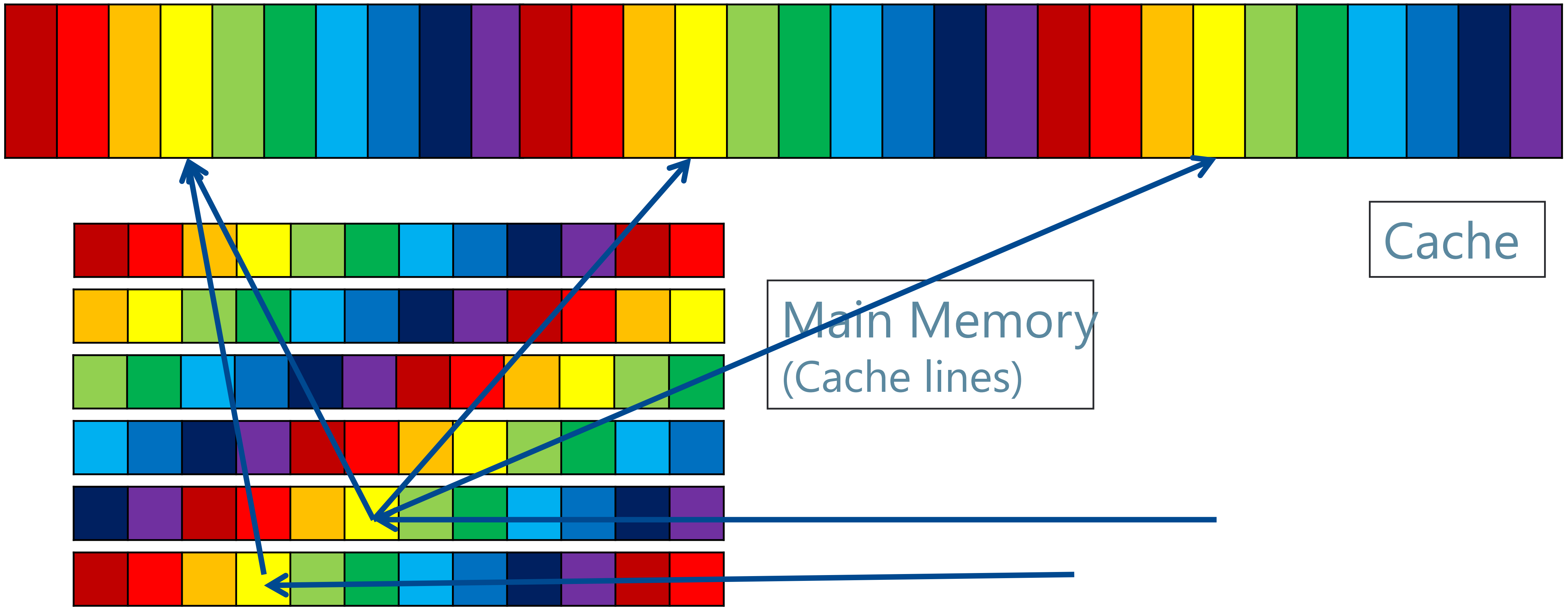
- Fixed size block of memory
- Smallest cache-able unit



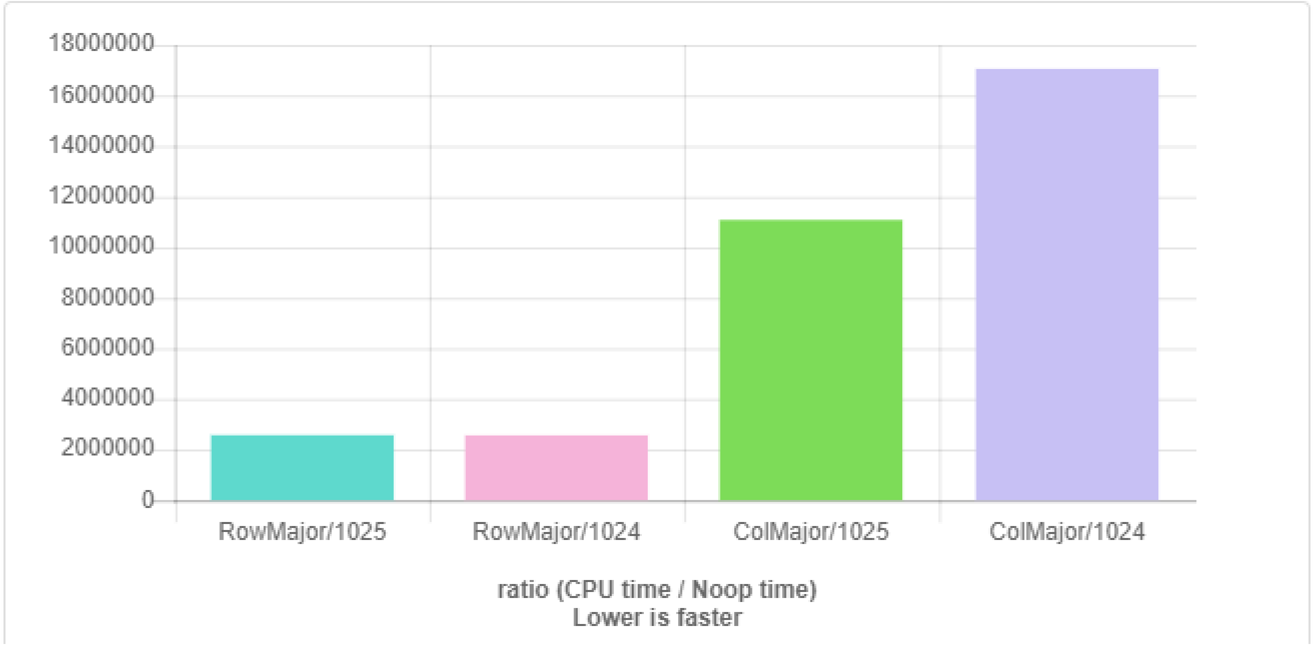
- When memory required, the whole cache line is swapped in



Cache lookup



Reducing matrix size



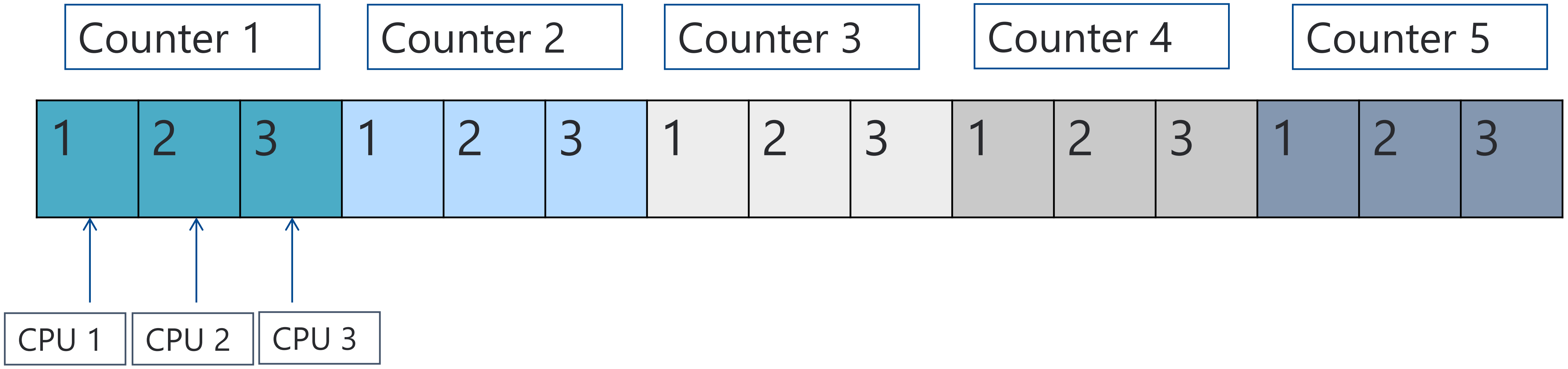
False Sharing

- Cores do not share data - they do share cache lines

- False sharing requires:
  - Several cores accessing the same cache line
  - Frequently
  - At least one is a writer

- Not only arrays:
  - Heap allocation, globals, statics
  - Even from different translation units

Lightweight Counters:

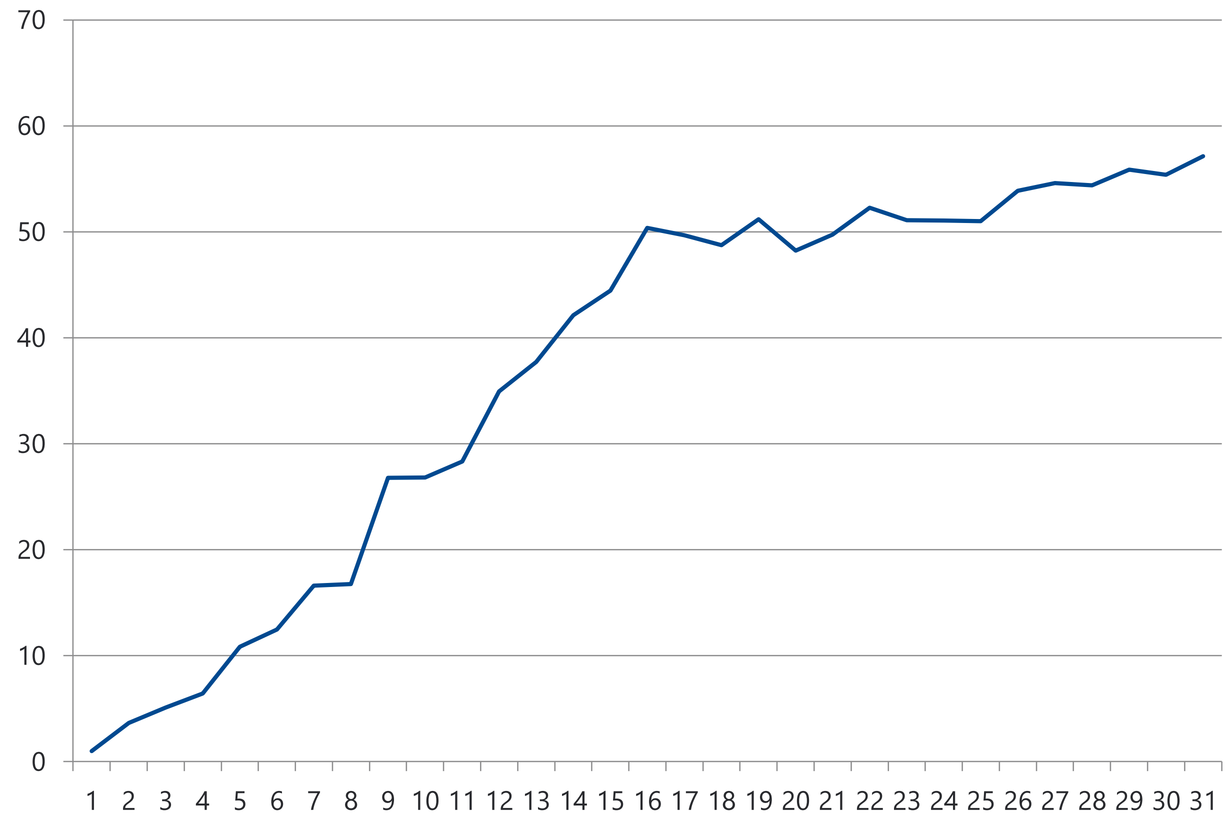


Measuring Strategy

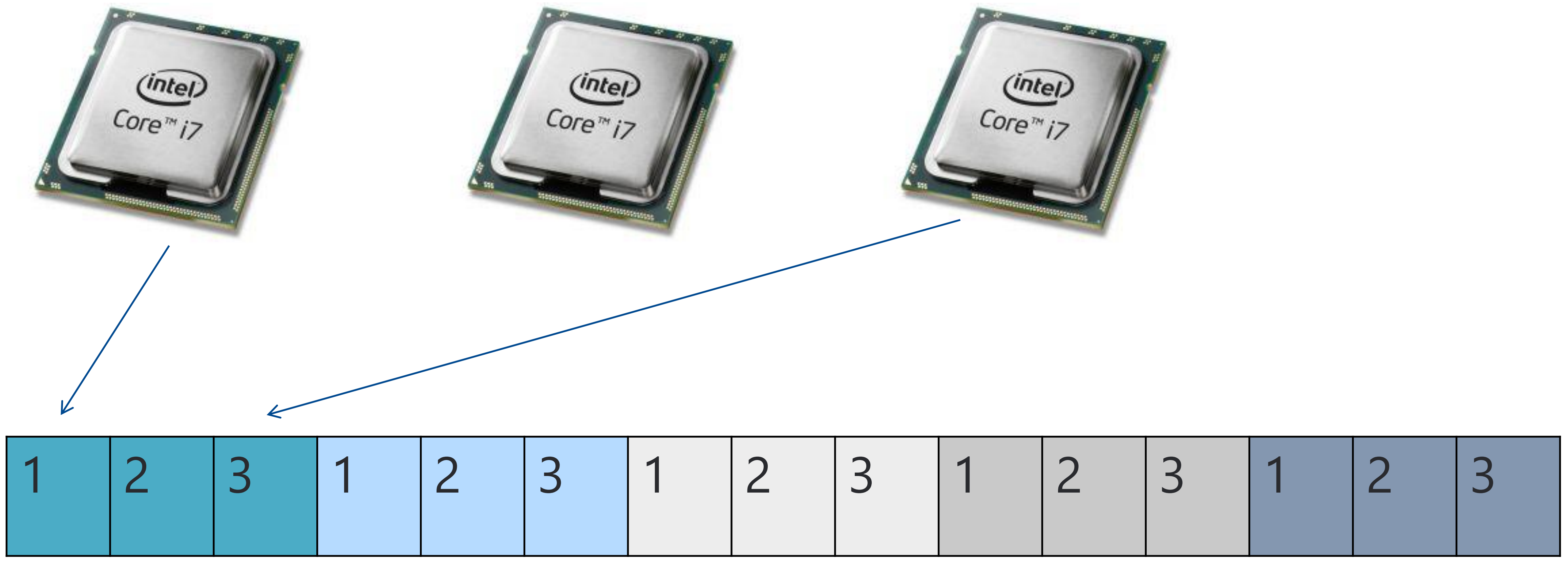
- Each thread will perform a pre-defined set of iterations
- So N threads will do N times more work than 1 thread
- Measure from 1 to 31 threads
- Calculate slowdown

- Expected:
  - Best case: slowdown of 1
  - Worst case: slowdown of N

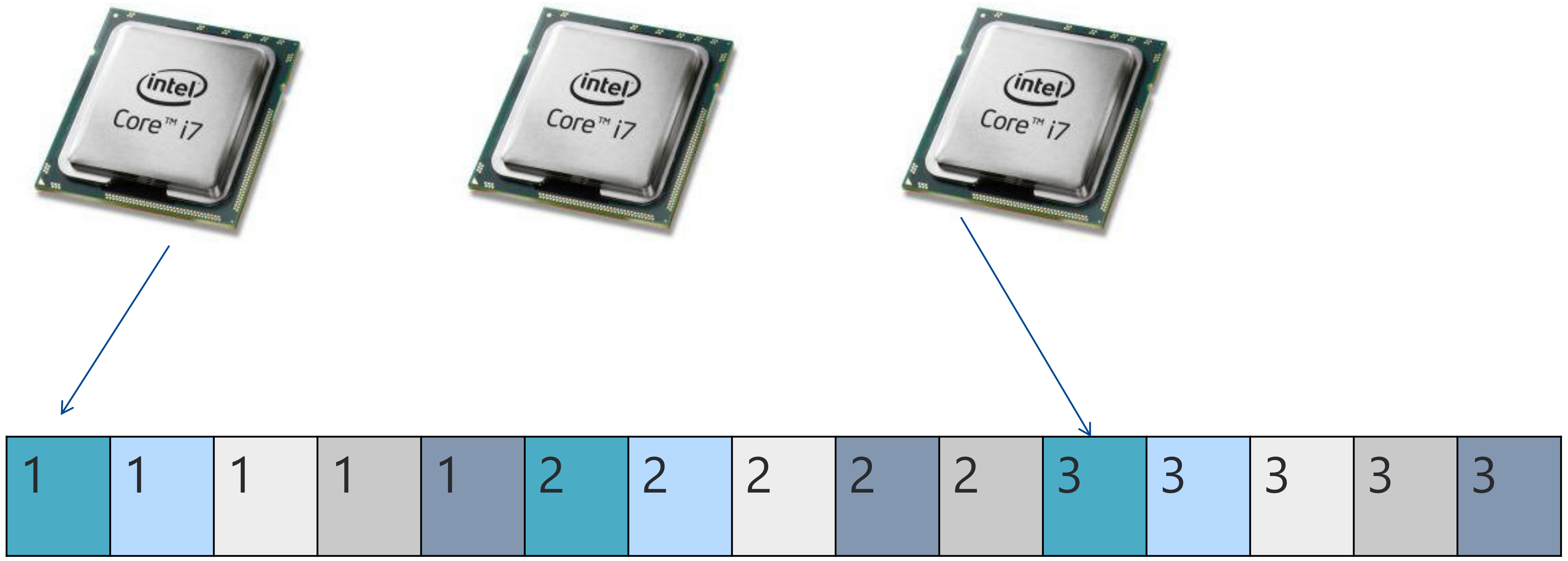
Naïve implementation results:



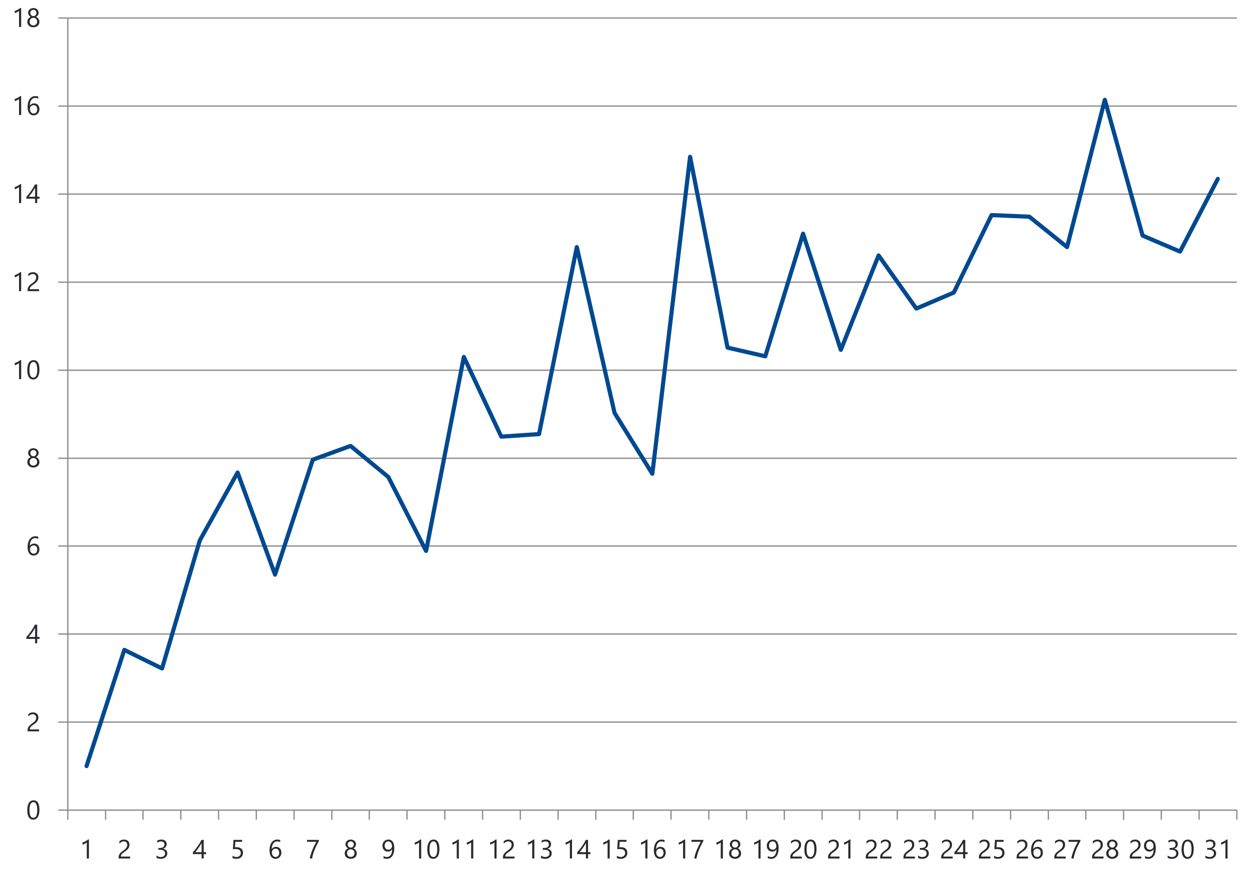
Naïve implementation – closer look



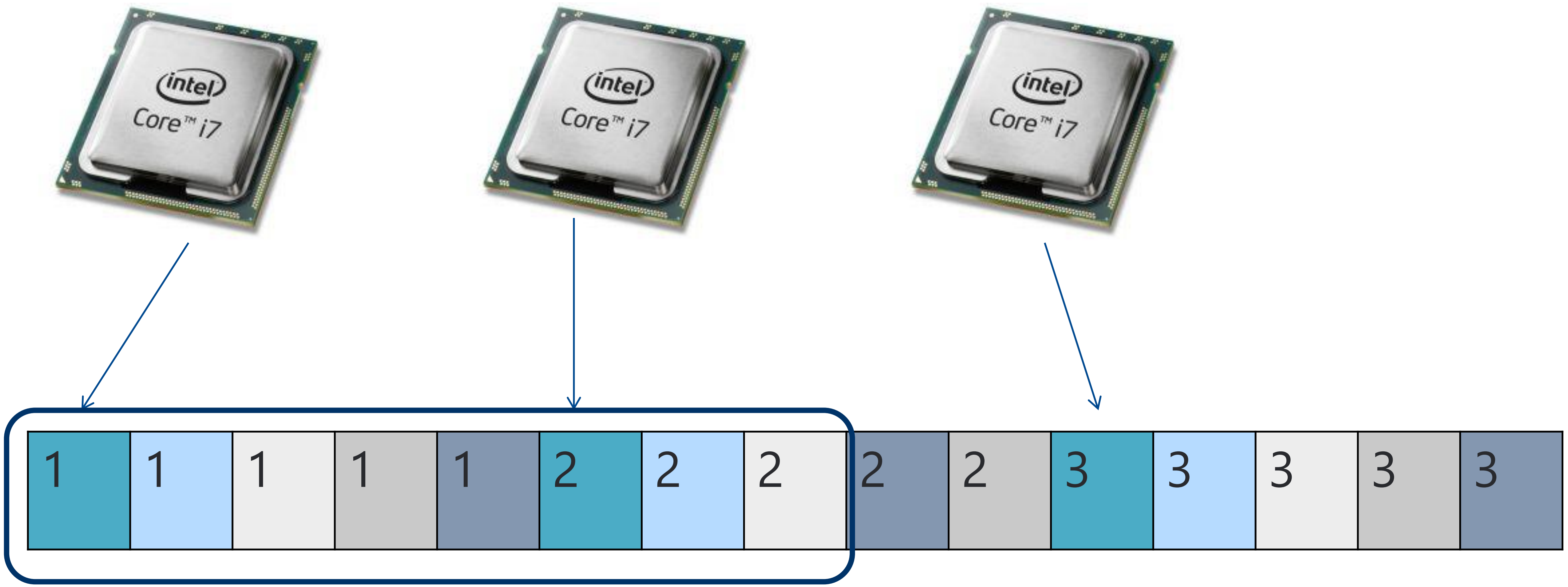
Group by CPU



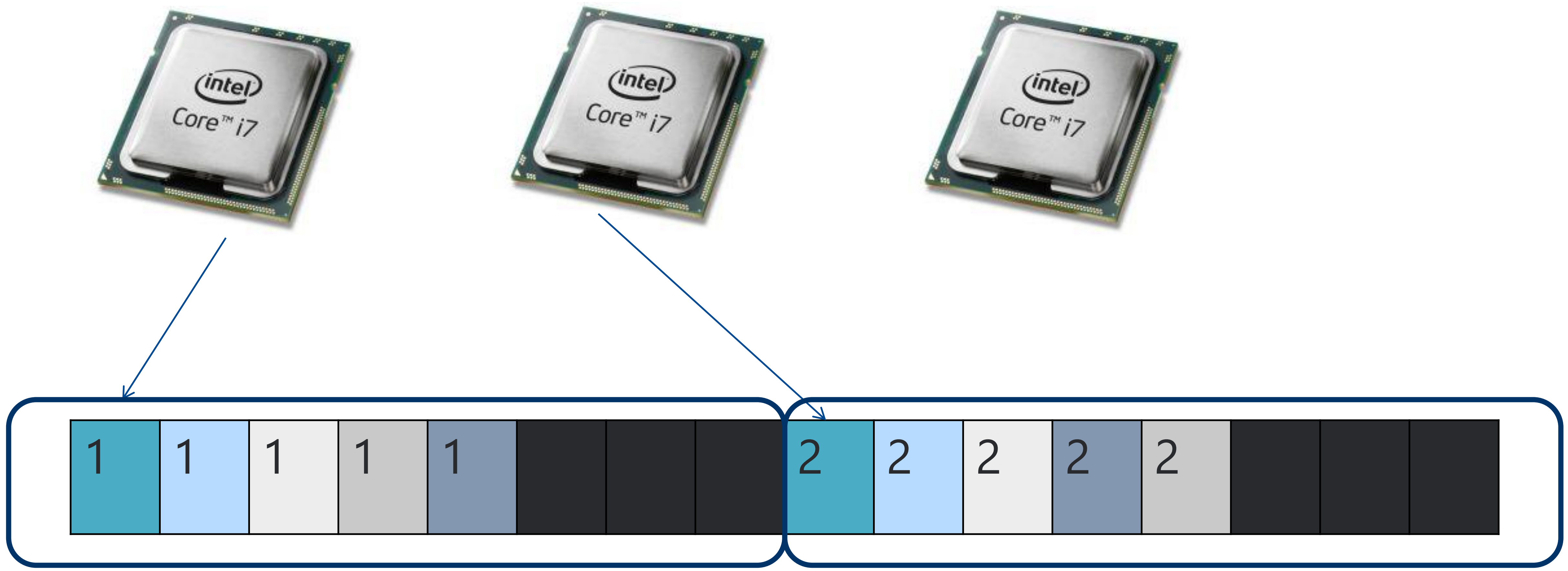
Group by CPU – Results



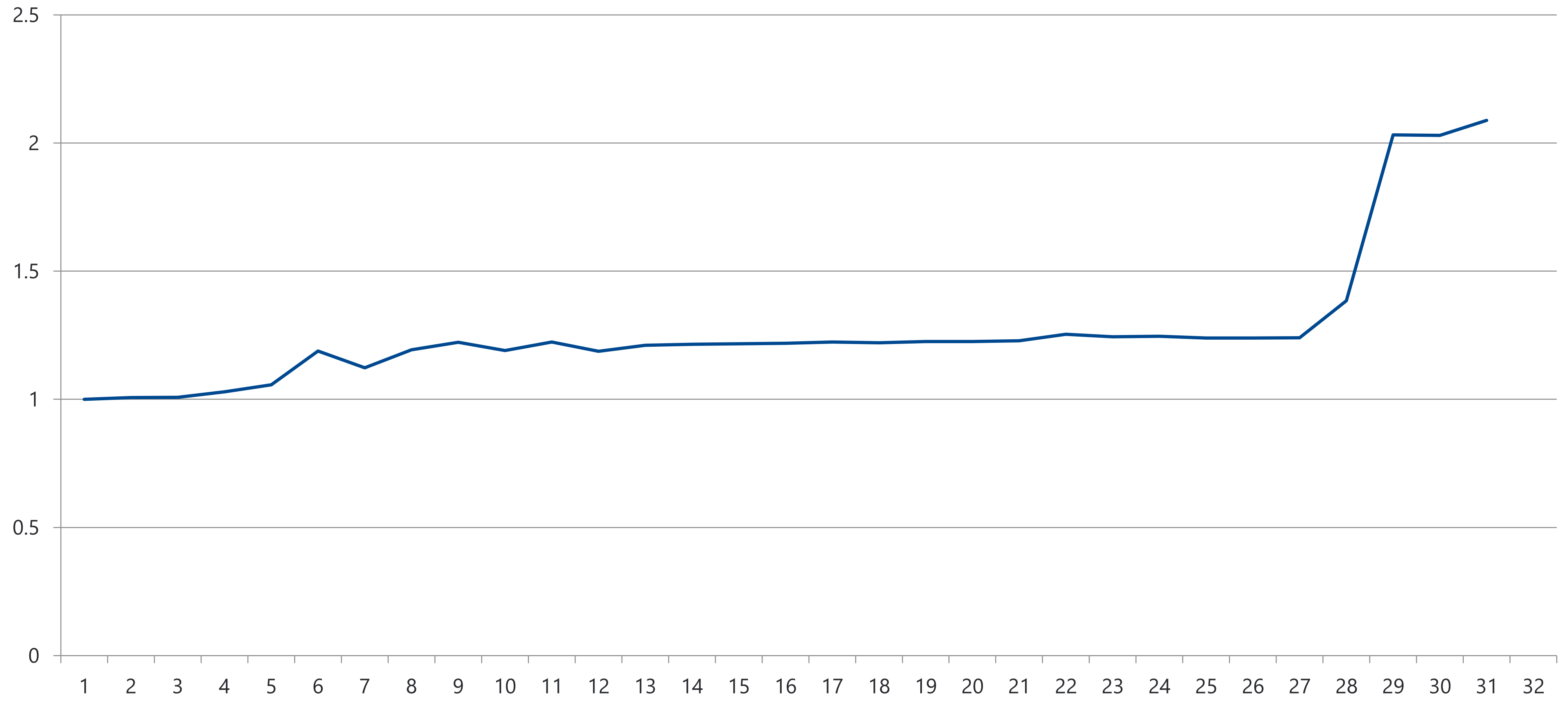
Group by CPU – closer look:



Align to cache line



Align to cache line – Results:



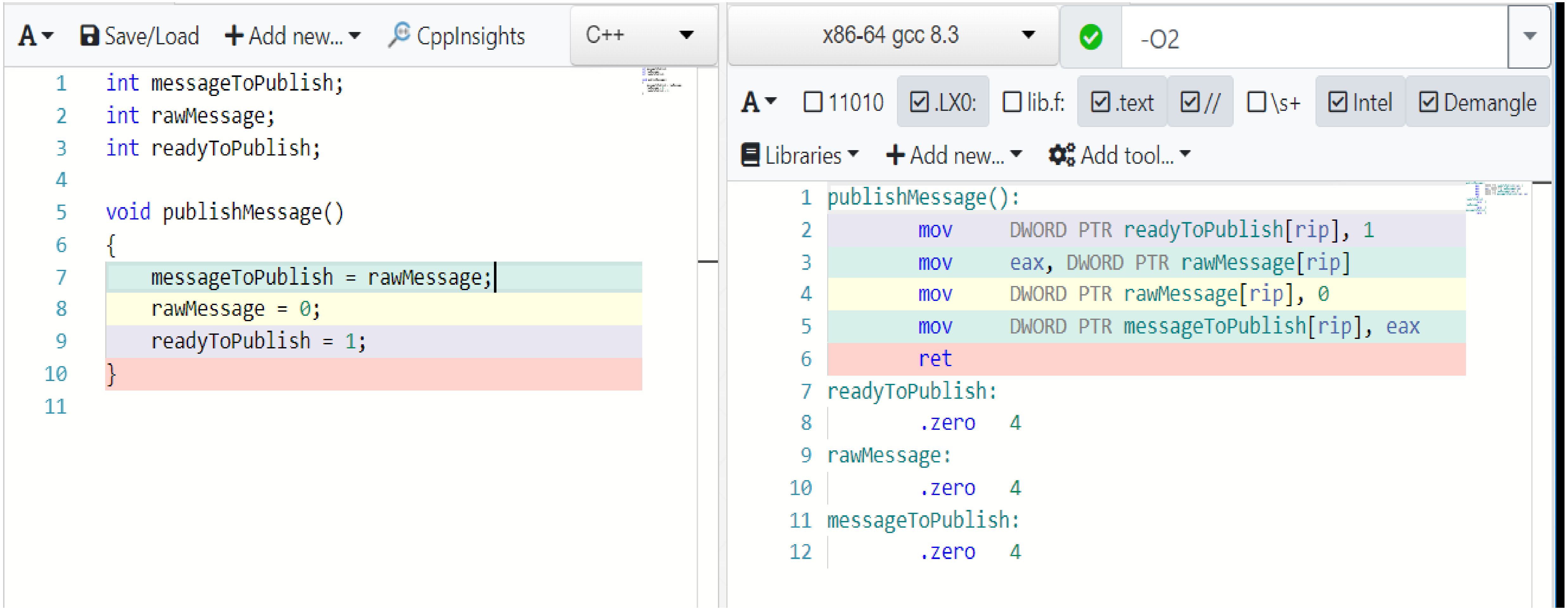
Compiler has freedom!

- The compiler can change the code
    - Rearrange/remove memory access
    - Reuse location

- To use the HW better:
    - Provide better locality
    - Reduce stale cycles

- Maintain observable state, from the **same thread** POV

Compiler reordering



```cpp
int messageToPublish;
int rawMessage;
int readyToPublish;

void publishMessage()
{
    messageToPublish = rawMessage;
    rawMessage = 0;
    readyToPublish = 1;
}
```

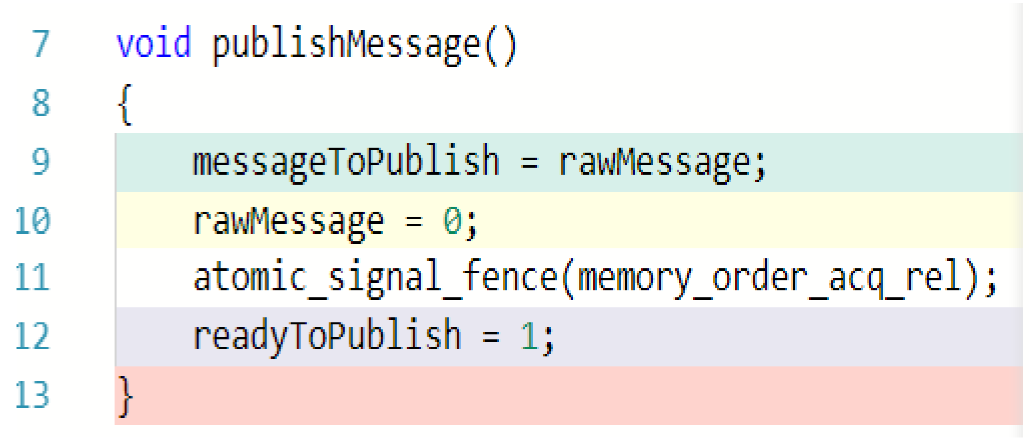
```asm
publishMessage():
        mov     DWORD PTR readyToPublish[rip], 1
        mov     eax, DWORD PTR rawMessage[rip]
        mov     DWORD PTR rawMessage[rip], 0
        mov     DWORD PTR messageToPublish[rip], eax
        ret
readyToPublish:
        .zero   4
rawMessage:
        .zero   4
messageToPublish:
        .zero   4
```

CYBERBIT PROTECTING A NEW DIMENSION

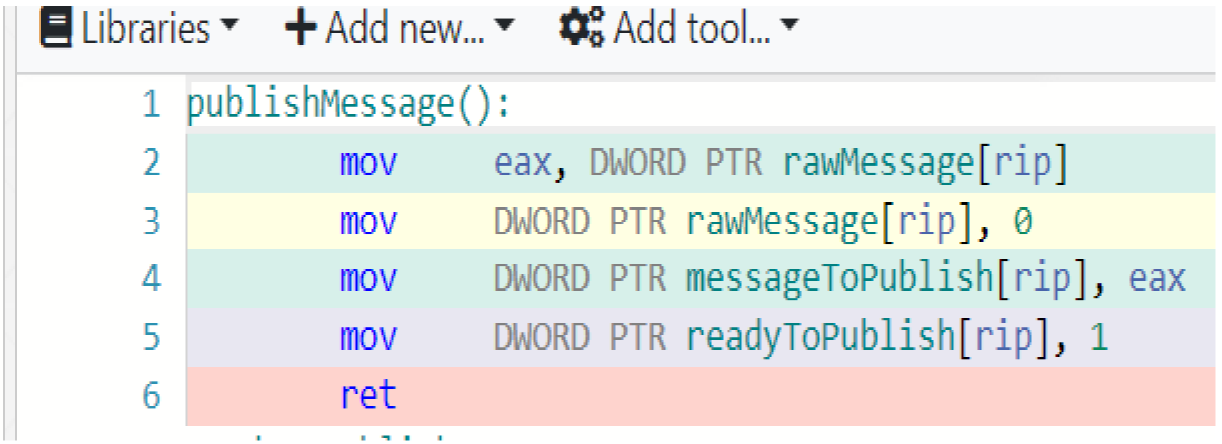
Volatile

- Introduced for MMIO
- Reordering non-volatile and volatile is permitted
- Every access to the variable will be restricted

Compiler Barrier



```
void publishMessage()
{
    messageToPublish = rawMessage;
    rawMessage = 0;
    atomic_signal_fence(memory_order_acq_rel);
    readyToPublish = 1;
}
```



```
publishMessage():
        mov     eax, DWORD PTR rawMessage[rip]
        mov     DWORD PTR rawMessage[rip], 0
        mov     DWORD PTR messageToPublish[rip], eax
        mov     DWORD PTR readyToPublish[rip], 1
        ret
```

HW has freedom too!

- CPU may reorder instructions as well
- Several functional units
- Execute instruction while waiting for previous instructions operand
- Data availability order, rather then instruction order.
- Reduce memory access wait time

CPU Reordering

Assume finished and answer initialized to 0

```
<Core #1>
answer = 42;
finished = 1;
```

May reorder

```
<Core #2 >
 while (! finished) { ; }
 cout << answer;
```

May reorder

- Threads need to communicate
- Reads and writes order reasoning
- With respect to memory access

Memory Model

- The set of allowed reorders
  - LoadLoad, LoadStore, StoreStore, StoreLoad
- And how to limit that
  - Toolchain issue fence for the architecture
    - If such fence is unavailable, a stronger fence is issued
      - Full fence
      - One-way fence
- C++ did not define memory model until C++ 11
- Synchronization mechanisms include the required fence

SC-DRF

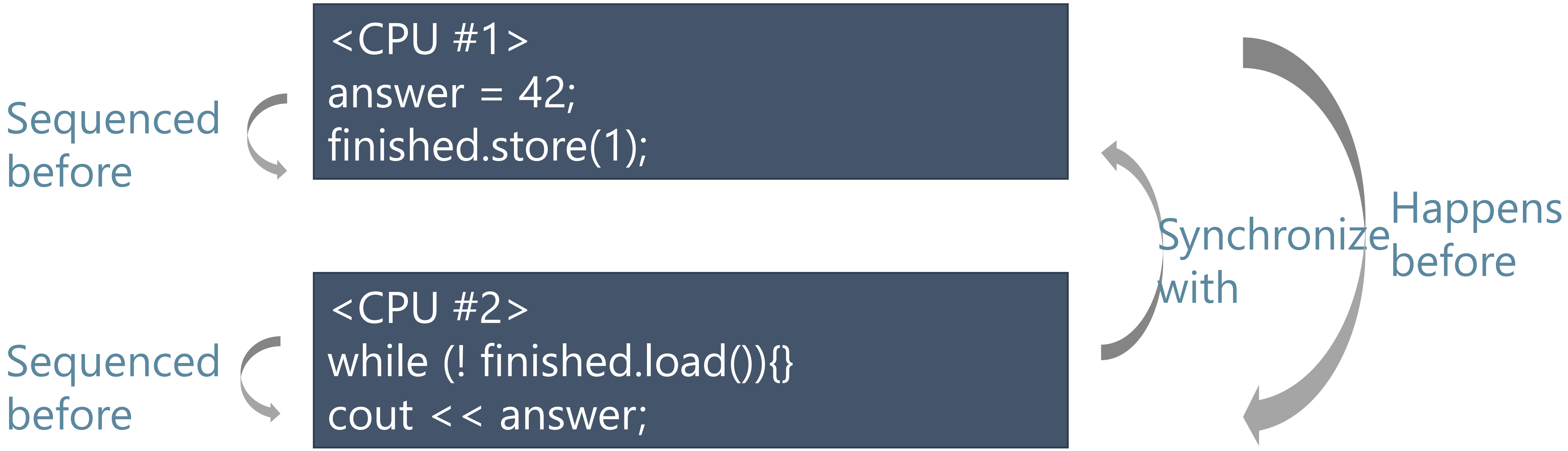
- **Sequential consistency (SC)**:

"the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program"

- **Data race**: simultaneously accessing an object by two threads, and at least one thread is a writer.
- **Simultaneously:** without happens-before ordering.

It appears to execute the program you wrote, as long as you didn't write a data race.

CPU Reordering, Example Revised



```
<CPU #1>
answer = 42;
finished.store(1);
```

```
<CPU #2>
while (! finished.load()){}
cout << answer;
```

Transitivity

Thread #1

```
data = 42;
t1.store (1);
```

Thread #2

```
while(! t1.load() ) { ; }
    t2.store(1);
```

Thread #3

```
while(! t2.load() ) { ; }
assert (data == 42);
```

Instruction reordering in action

First seen at http://preshing.com

x, y, rX and rY initialized to zero

```
<Thread 1>
 x = 1;
// compiler barrier here
rY = y;
```

```
<Thread 2>
 y = 1;
// compiler barrier here
 rX = x;
```

The Idea, cont'd

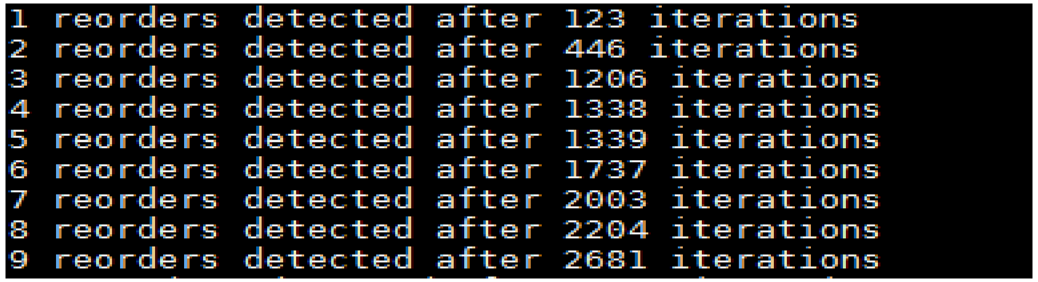
```
<Main thread>
Initialize all semaphores
Spawns two threads
Do forever:
    Initialize to zero
    Post on start semaphores
    Wait on end semaphore
    Wait on end semaphore
    Check if both, rX and rY are zero
```

```
<Thread 1>
Do forever:
    Wait on start semaphore #1
     x = 1;
    Compiler Barrier
     rY = y;
  Post on end semaphore
```

```
<Thread 2>
Do forever:
    Wait on start semaphore #2
     y= 1;
    Compiler Barrier
     rX = x;
  Post on end semaphore
```

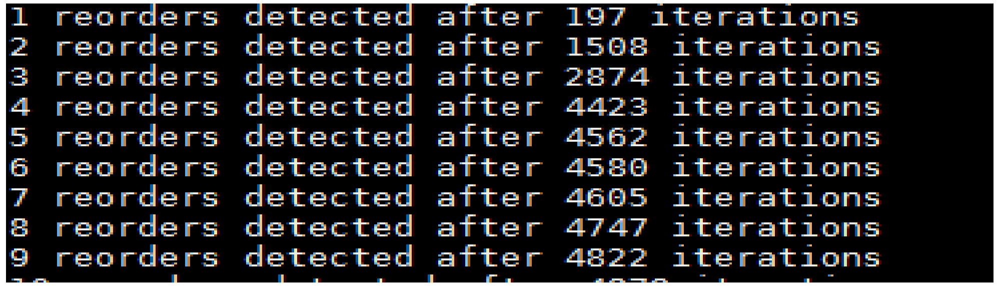
Results:

Compiled with no optimization:



```
1 reorders detected after 123 iterations
2 reorders detected after 446 iterations
3 reorders detected after 1206 iterations
4 reorders detected after 1338 iterations
5 reorders detected after 1339 iterations
6 reorders detected after 1737 iterations
7 reorders detected after 2003 iterations
8 reorders detected after 2204 iterations
9 reorders detected after 2681 iterations
```

Compiled with O3



```
1 reorders detected after 197 iterations
2 reorders detected after 1508 iterations
3 reorders detected after 2874 iterations
4 reorders detected after 4423 iterations
5 reorders detected after 4562 iterations
6 reorders detected after 4580 iterations
7 reorders detected after 4605 iterations
8 reorders detected after 4747 iterations
9 reorders detected after 4822 iterations
```

Solution

- Replacing the compiler barrier with a memory fence

```
 x = 1;
atomic_thread_fence(memory_order_seq_cst);
rY = y;
```

```
 y = 1;
atomic_thread_fence(memory_order_seq_cst);
rX = x;
```

Solution

- Demonstrating this would be kind’a boring….

Non sequentially consistent atomics

Relaxed atomic

- Atomicity
- RMW will always see the latest value

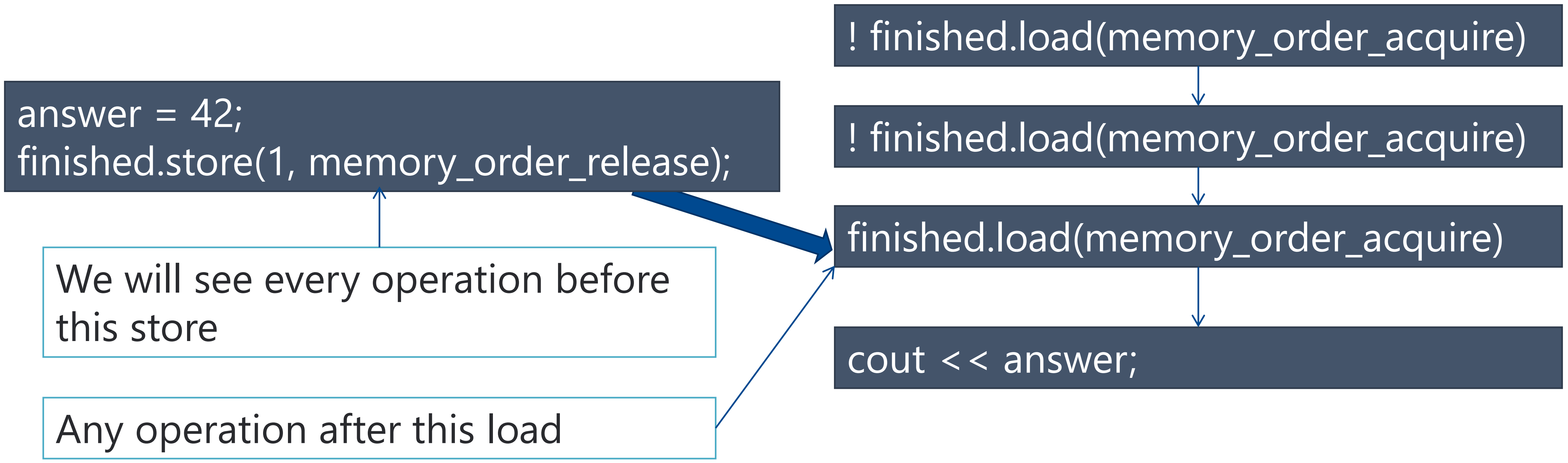
- No ordering!
  - One thread can see operation A before B, while another thread may see it the other way!

Non sequentially consistent atomics

Acquire - Release

- Must be in tandem
- Operations sequenced before “release”, will be visible to operation sequenced after “acquire” that synchronize with it
- Does not provide global ordering
- Better shown with an example:

Acquire/Release – another look



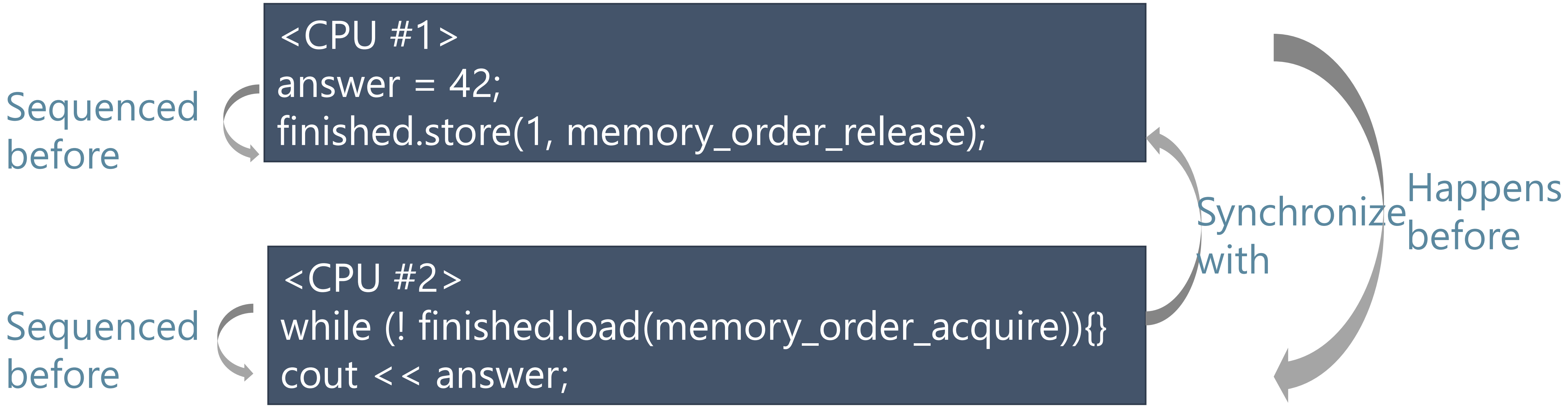
```
answer = 42;
finished.store(1, memory_order_release);
```

We will see every operation before this store

Any operation after this load

```
! finished.load(memory_order_acquire)
! finished.load(memory_order_acquire)
finished.load(memory_order_acquire)
cout << answer;
```

Acquire/Release



```
<CPU #1>
answer = 42;
finished.store(1, memory_order_release);
```

```
<CPU #2>
while (! finished.load(memory_order_acquire)){}
cout << answer;
```

What about this:

```
atomic <bool>  x (false);
atomic <bool>  y (false);
atomic <bool> bothSet(false);

thread reader1 ([&]() { while (!x.load(memory_order_XXX);
            if (y) bothSet= true;  } ) ;
thread reader2 ([&]() { while (!y.load(memory_order_XXX));
           if (x) bothSet= true;  } ) ;
thread writer1 ([&]() { x.store(true, memory_order_XXX); } ) ;
thread writer2 ([&]() { y.store(true, memory_order_XXX ); } ) ;

reader1.join(); reader2.join(); writer1.join(); writer2.join ();
assert ( bothSet );
```

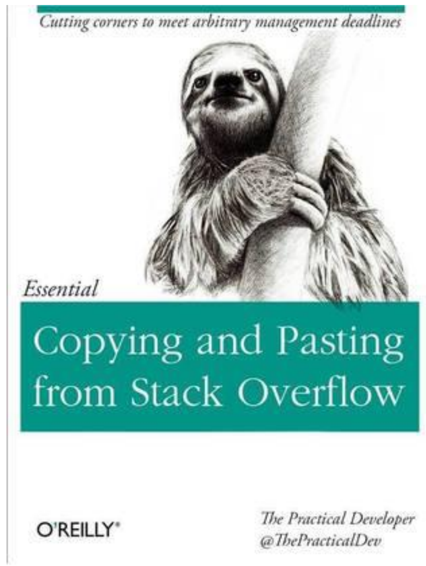
CYBERBIT
PROTECTING A NEW DIMENSION

So…

- SC mandates a single total order of events of all operations tagged as such
- With relaxed atomics, threads do not have to agree on the order of events
- They only have to agree on the modification order of each individual item

Should you use non-sequentially consistent

- Does every micro-second count?
  - Is this a hot path?
  - Is this a problematic spot?
- Will it have massive testing and reviews?
- Is this a known pattern?


Cutting corners to meet arbitrary management deadlines
Essential
Copying and Pasting from Stack Overflow
O'REILLY
The Practical Developer
@ThePracticalDev

Data Oriented Design

Consider the following class:

```
class Object {
   int _pos[2];
   int _speed;
   Model  _model;
   const char  _name[NAME_SIZE];
   ....
   int _foo;
};
```

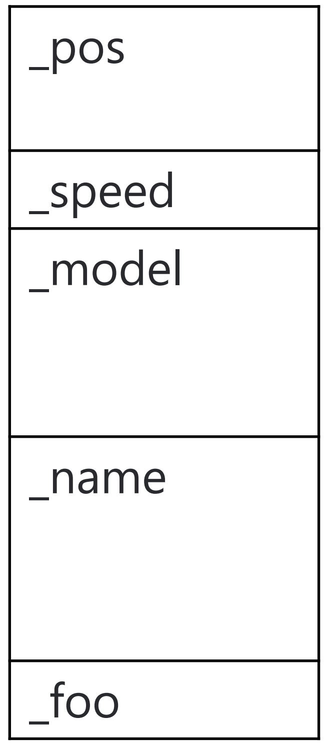
What is the most expensive operation?

```
void Object::update( int time)
{
  float f = sqrt(
  _pos[0] + (time* _speed) +
  _pos[1] + (time * _speed ) ) ;
  _foo +=  f;
}
```

1. Load _pos, cache miss, 200 cycles
2. Load speed, same cache line, 3 cycles
3. Multiply and add, twice, 5 cycles twice
4. Square root, 30 cycles
5. Load _foo, cache miss, 200 cycles
6. Add result to _foo, 1 cycle

Total: 450 cycles

Memory



Can the compiler help?

- 50 out of 450 cycles are real work

Compiler’s domain is the 50 cycles

Not very much…

Back to the Example

```
class Object {
   int _pos[2];
   int _speed;
   int _foo;
   Model  _model;
   const char  _name[NAME_SIZE];
   ....
};
```

The new cost:

```
void Object::update( int time)
{
  float f = sqrt(
  _pos[0] + (time* _speed) +
  _pos[1] + (time * _speed ) ) ;
  _foo +=  f;
}
```

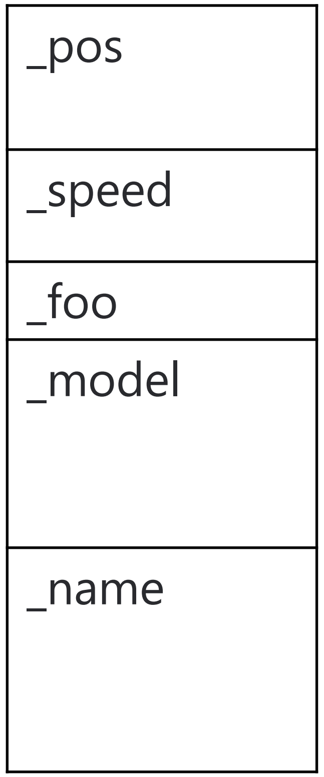
1. Load _pos, cache miss, 200 cycles
2. Load speed, same cache line, 3 cycles
3. Multiply and add, twice, 5 cycles, twice
4. Square root, 30 cycles

~~5. Load _foo, cache miss, 200 cycles~~

5. Load _foo, 5 cycles
6. Add result to _foo, 1 cycle

Total: 250 cycles

Memory



Can we do better?

```
for ( auto & object : objects ) {
        object.update(time);
}
```

Regroup the data

```
class Object {
   Model  _model;
   const char    _name[NAME_SIZE];
   ....
};
```

```
class ObjectPosition{
   float _pos[2];
   float _speed;
   int _foo;
};
```

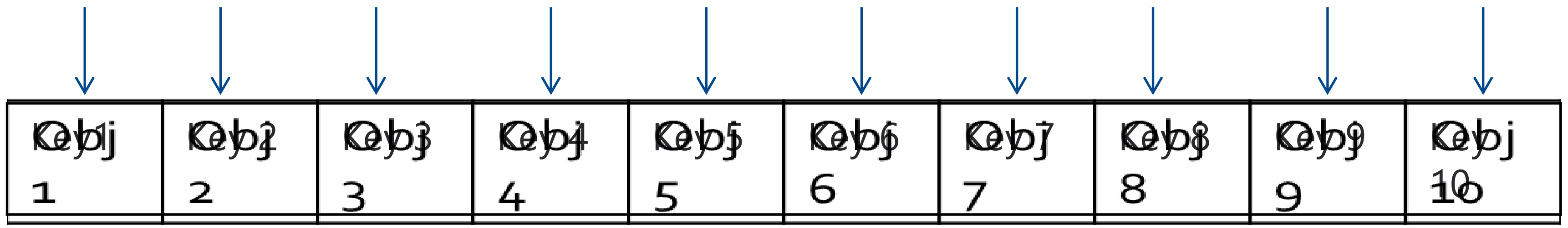
The new cost

- Single cache line, multiple objects
- Shared cost
- On average, fetching will cost us about 40 cycles
- Total cost ~90 cycles

DoD in one sentence

- Make continuous, tightly packed, chunks of memory that will be used consecutively
- Re-group fields according to their usage
- When it is needed, and the transformation that's needed

Container Searching for a key/condition



- Load the key
- Load the object selectively

Polymorphism

```
for (Shape * currentShape : shapes) {
        currentShape->draw();
}
```

- shapes is likely to contain:

    square, circle, polygon, square, text, rectangle....

Polymorphism, Resolution

```
for (Circle* currentCircle : circles) {
        currentCircle->draw();
  }

for (Square* currentSquare : squares) {
        currentSquare->draw();
 }
```

Cache Oblivious Algorithms

- Characteristics, not existence
- Attempt to maximize cache hits
- All levels of cache hierarchy
- Can be out-performed by cache aware

Analyzing Memory Utilization

- Analyzing using big O notation
- Idealized cache model
  - Ignore cache hierarchy
  - Ignore replacing policies
  - Ignore associativity

Example: Search

Search a sequence of numbers for the highest number, which is less than X

- Data is searched many times
- Ignore preparation time

How should we store the sequence??

Attempt #1: Binary Search

- log(n) comparisons
- Given the distance, assume that they will require memory access

- log(n)-log(B) memory accesses are required

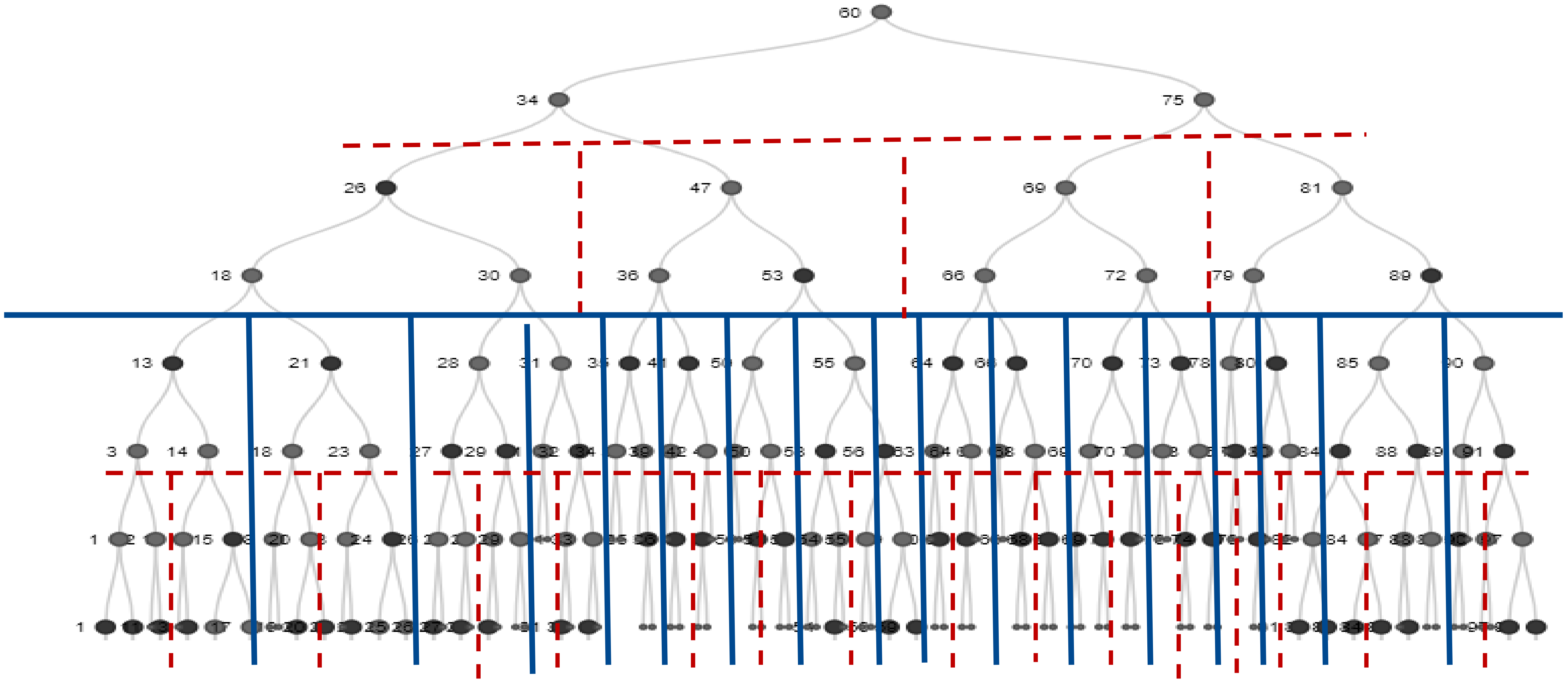
Attempt #2: B-Tree

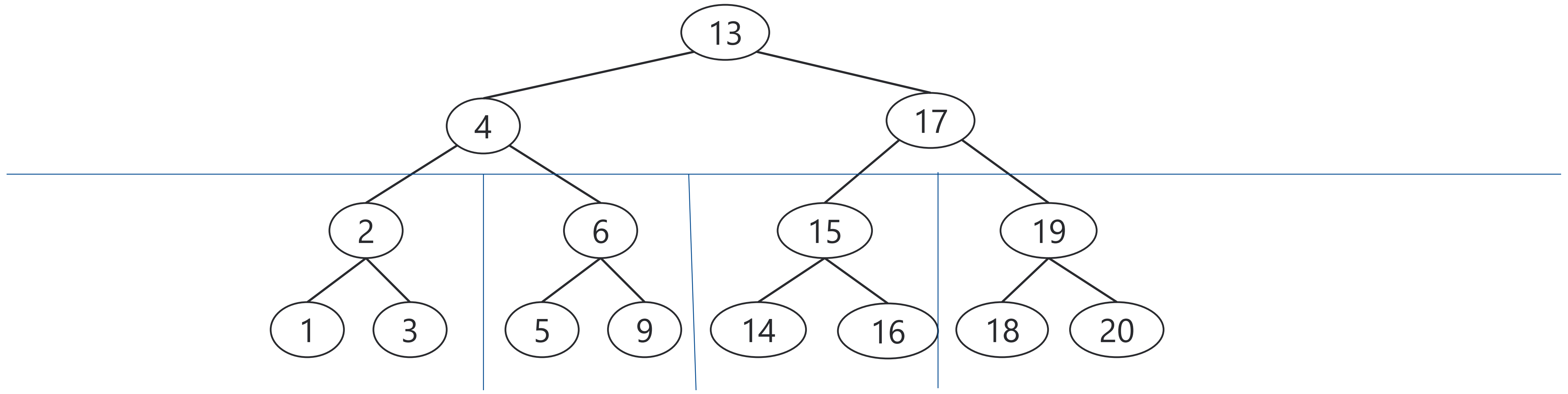
- Set B to our cache-line size
- We will require $\log_B(N)$ steps
- Each node will be loaded in one cache line
- O(log (N) / log(B)) memory accesses

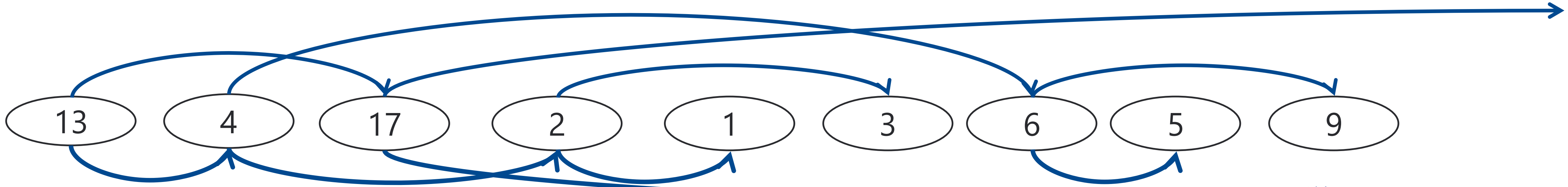
- But….
  - We need to know B

Van Emde Boas

- Full description and analysis is outside the scope
- Set a fully balanced tree
- Recursively divide it to sub-trees
- Each sub-tree is copied to sequential memory
- Use this to search


60
34
75
26
47
69
81
18
30
36
53
66
72
79
89
13
21
28
85


13
4
17
2
6
15
19
1
3
5
9
14
16
18
20


13
4
17
2
1
3
6
5
9

Van Emde Boas, intuitive analysis

- Each section is of size B or less
  - 2 memory accesses per section
- Section height between log(B) to log(B)/2
- Tree height is log(n)
- Max sections we will visit is log(N)/(log(B)/2)
- This will require 4(log(N)/log(B)) memory accesses

Thank You

CYBERBIT
PROTECTING A NEW DIMENSION

The Pattern

```
Singleton* Singleton ::instance () {
  if ( _instance  == nullptr ) {
       std::lock_guard<std::mutex>
lock(_mutex);
       if (_instance == nullptr) {
              _instance = new Singleton();
         }
    }
  return _instance;
```

Allocate memory
Call C'tor
Assign

Attempt #1: Adding temporary

```
Singleton * Singleton ::instance () {
  if ( _instance == nullptr ) {
        std::lock_guard<std::mutex> lock(_mute
         if (_instance == nullptr) {
                Singleton * tmp = new Singleton();
                 _instance = tmp;
          }
    }
  return _instance;
}
```

Optimize out the temporary.
Back to square 1.

Attempt #2: Outsmart the compiler

- Change tmp to larger scope, say static
  - Compiler can still detect this
- Define tmp as extern
  - Can still detect this
  - Or, place construction after both
- Define helper on other translation unit
  - Compiler must assume it can throw
    - No inlining
  - Link-time inlining kills this attempt

Attempt #3: Volatile

- Qualify tmp and _instance as volatile
  - All side effects of one volatile must be completed before addressing the other

```
Singleton * Singleton ::instance () {
  if ( _instance == nullptr ) {
      std::lock_guard<std::mutex> lock(_mutex);
      if ( _instance == nullptr ) {
              Singleton * volatile tmp = new Singleton();
              _instance = tmp;  // static Singleton * volatile
          }
    }
  return _instance;
}
```

Attempt #3: Volatile, cont'd

Lets inline a constructor:

```
Singleton * Singleton ::instance () {
  if ( _instance == nullptr ) {
      std::lock_guard<std::mutex> lock(_mutex);
       if ( _instance == nullptr ) {
              Singleton * volatile tmp = new Singleton();
               tmp->x = 4 //from the c'tor
               _instance = tmp;
         }
    }
  return _instance;
}
```

This new instruction may be reordered

Conclusion

Trying to outsmart the compiler is a bad idea

Attempt #4: Compiler barrier

```
Singleton * Singleton::instance () {
  if (_instance == nullptr) {
        std::lock_guard<std::mutex> lock(_mutex);
        if (_instance == nullptr) {
                Singleton * tmp = new Singleton();
                 // Compiler Barrier here
                 _instance = tmp;
        }
    }
  return _instance;
}
```

What about CPU Re-Ordering

Game Over!

Attempt #5: Memory Barrier

```
Singleton * Singleton::instance() {
    if (_instance == nullptr) {
        std::lock_guard<std::mutex> lock(_mutex);
        if (_instance == nullptr) {
            Singleton * tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_seq_sct);
            _instance = tmp;
        }
    }
    return _instance ;
}
```

Non atomic assignment

Attempt #5: atomic

```
Singleton * Singleton ::instance() {
      Singleton * tmp = _instance.load();
       if (tmp == nullptr) {
             std::lock_guard<std::mutex> lock(_mutex);
             tmp = _instance.load();
             if (tmp == nullptr) {
               tmp = new Singleton;
               _instance = tmp;
             }
        }
       return tmp ;
}
```

This works!

- But uses sequential consistency
- Can be expensive

- Can we do better?

Attempt #6: acquire-release

```
Singleton * Singleton ::instance() {
        Singleton * tmp = _instance.load(std::memory_order_acquire);
         if (tmp == nullptr) {
              std::lock_guard<std::mutex> lock(_mutex);
              tmp = _instance.load(memory_order_relaxed);
              if (tmp == nullptr) {
                    tmp = new Singleton ;
                    _instance.store(tmp, memory_order_release);
               }
          }
          return tmp;
}
```

Attempt #7: do we need the lock?

```
Singleton * Singleton::instance() {
    Singleton* tmp = _instance.load(memory_order_relaxed);
     if (tmp == nullptr) {
         Singleton * newInstance = new Singleton ;
          if (! (_instance.compare_exchange_strong( tmp, newInstance,
                                                              memory_order_relaxed) ) ) {
                   delete newInstance;
          }
     }
     return _instance.load(memory_order_relaxed);
}
```

Back to the sketching board

C++ 11 states:

> *If control enters the declaration concurrently while the variable is being initialized, the concurrent execution will wait for completion of the initialization.*

So, the final answer…

```
Singleton & Singleton::instance() {
    static Singleton instance;
    return instance;
}
```