The C/C++ Memory Model

Yossi Moalem

Check Point
SOFTWARE TECHNOLOGIES LTD

Agenda

- Part 1 (Boring): some theory
- Part 2 (Cool): consequences
- Part 3 (Important): applications
  - Part 3.14 (If time permits): a touch on atomics

Some Theory

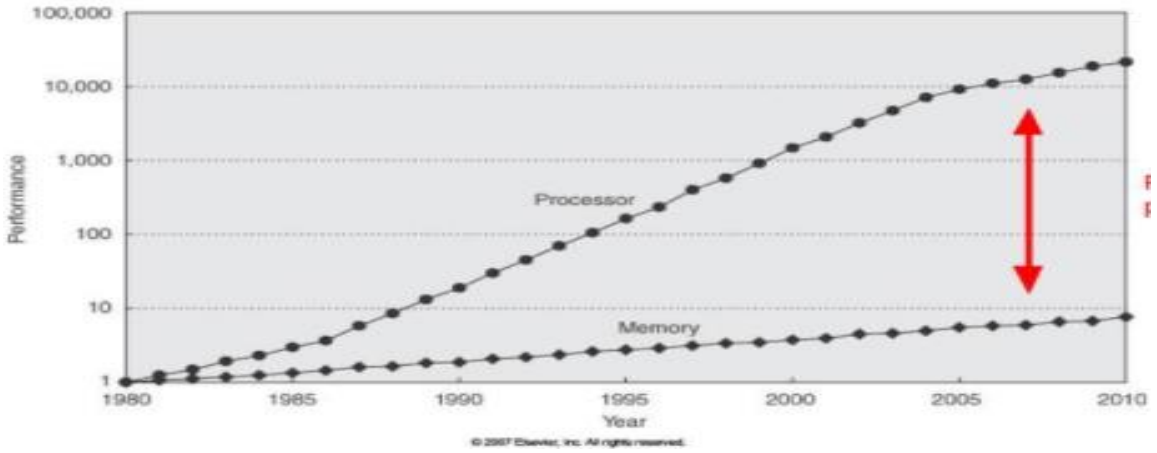
The computer we think we program for

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



Cache Hierarchy, The numbers

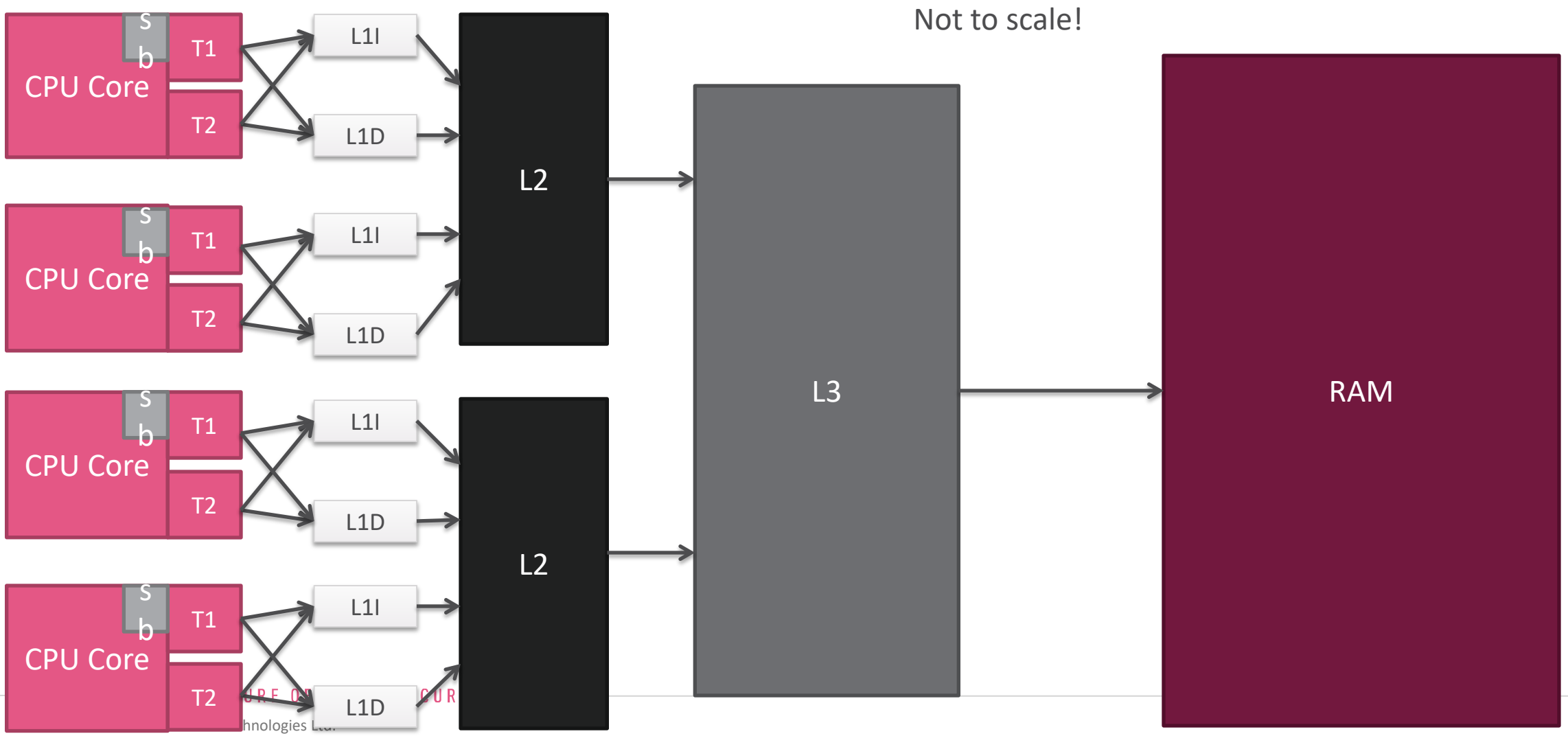
Faster

- 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

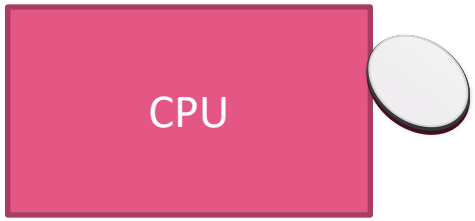
Check Point
SOFTWARE TECHNOLOGIES LTD

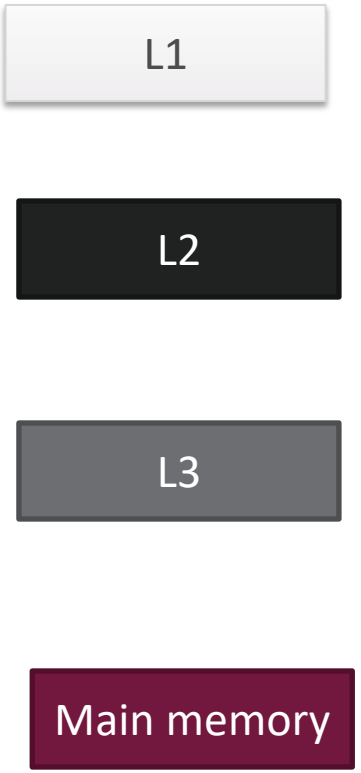
The computer we are actually programming for



One picture worth 1000 words
One animation worth 100 pictures

To scale!





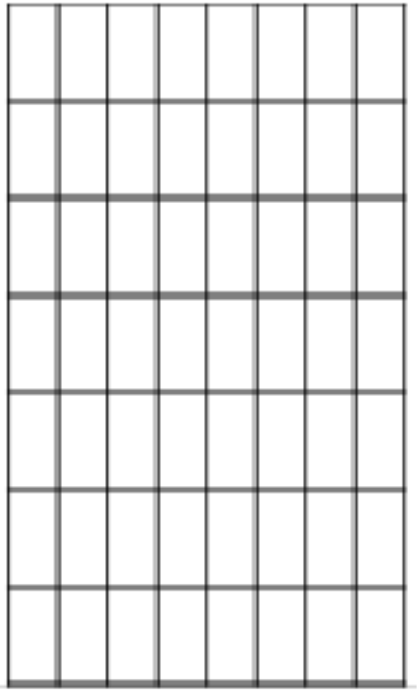
Check Point
SOFTWARE TECHNOLOGIES LTD

Cache: Requirements

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

Cache Line

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

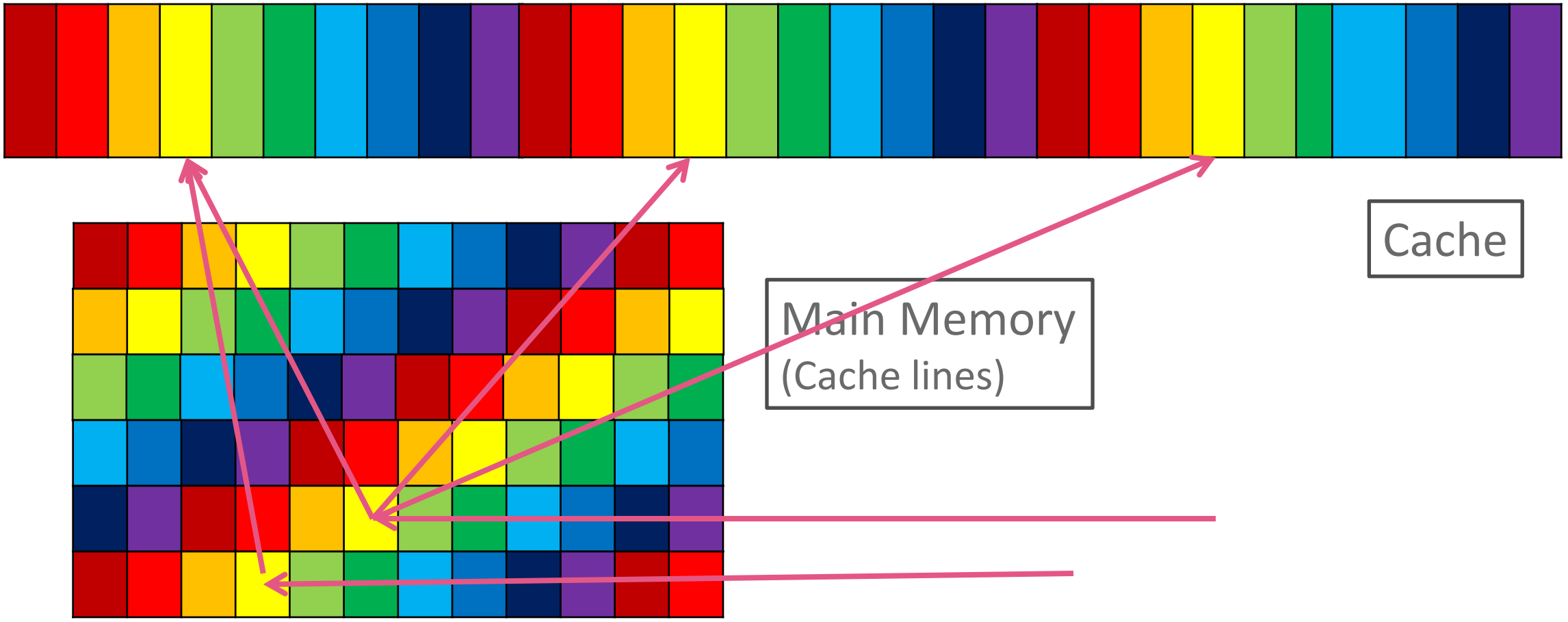


Cache

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

Main Memory

Cache lookup

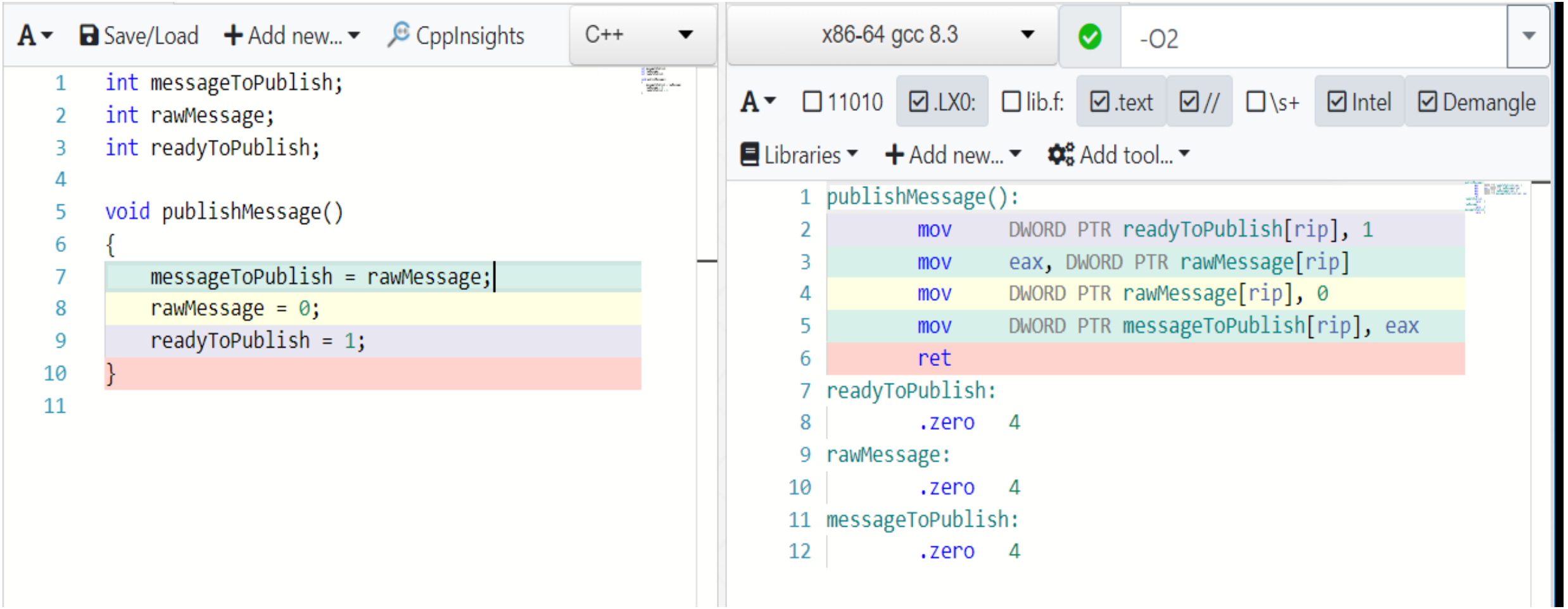


Compiler has freedom!

- Compiler can rearrange 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

Check Point
SOFTWARE TECHNOLOGIES LTD

Compiler reordering



Check Point
SOFTWARE TECHNOLOGIES LTD

Volatile

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

Compiler Barrier

```
7   void publishMessage()
8   {
9       messageToPublish = rawMessage;
10      rawMessage = 0;
11      atomic_signal_fence(memory_order_acq_rel);
12      readyToPublish = 1;
13  }
```

Libraries ▾ Add new... ▾ Add tool... ▾

```
1   publishMessage():
2           mov     eax, DWORD PTR rawMessage[rip]
3           mov     DWORD PTR rawMessage[rip], 0
4           mov     DWORD PTR messageToPublish[rip], eax
5           mov     DWORD PTR readyToPublish[rip], 1
6           ret
```

HW has freedom too!

- Fetch instruction
- Push to instruction queue
- Wait for the operands
- Dispatched (even before earlier instructions)
- Executes
- Write result to queue
- When its time comes (older results have been written), write result to register file

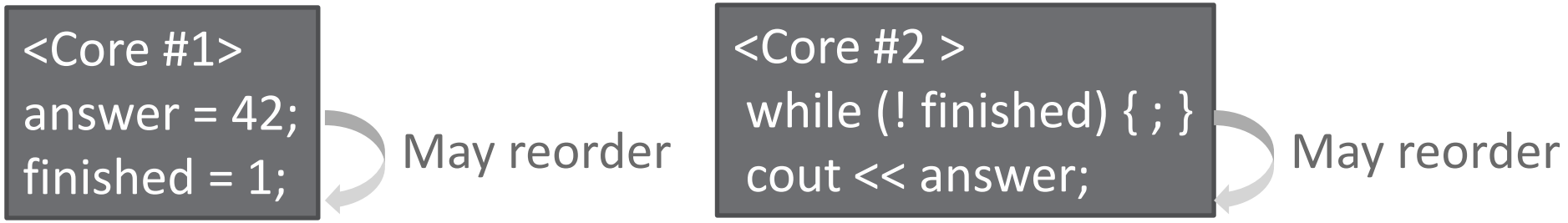
Check Point
SOFTWARE TECHNOLOGIES LTD

Out of Order Execution

- Data availability order, rather then instruction order.
- Reduce memory access wait time
- Several functional units

CPU Reordering

Assume finished and answer initialized to 0



Check Point
SOFTWARE TECHNOLOGIES LTD

Instruction Reordering

- Several instructions execute in parallel
- Threads needs to communicate
- Reads and writes order reasoning

Check Point
SOFTWARE TECHNOLOGIES LTD

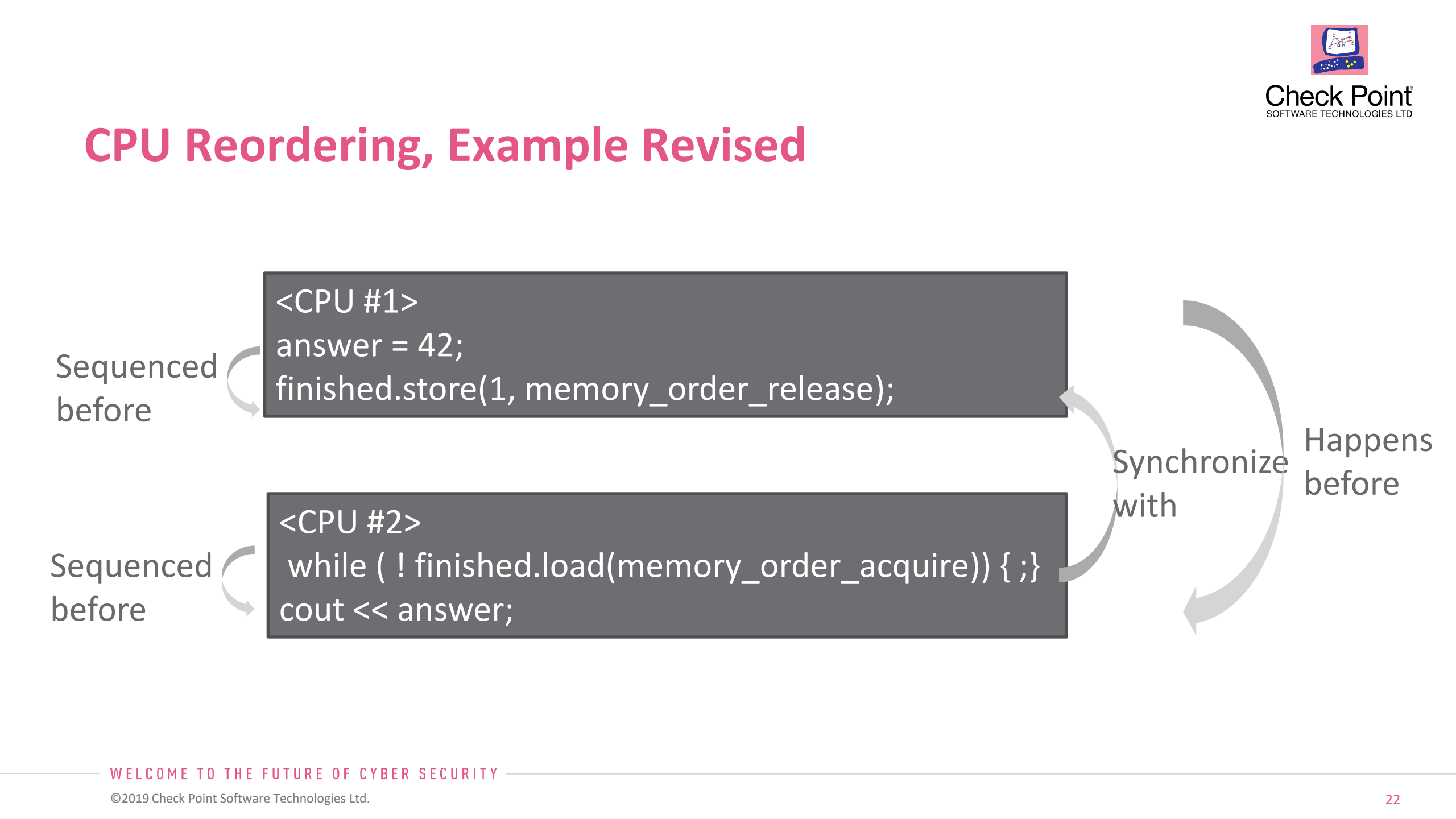
Language Memory Model

- The set of allowed reorders
  - LoadLoad
  - LoadStore
  - StoreStore
  - StoreLoad
- C++ did not define memory model until C++ 11

Check Point
SOFTWARE TECHNOLOGIES LTD

Memory Fence (memory barrier, machine barrier)

- Ordering with respect to memory access
- The toolchain will issue the correct fence for this architecture
- If such fence is unavailable, a stronger fence is issued
  - Full fence
  - One way fence
- Synchronization mechanisms include the required fence



Check Point
SOFTWARE TECHNOLOGIES LTD

SC-DRF

- **Sequential consistency (SC)**: Defined in 1979 by Leslie Lamport: *"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 object by two threads, and at least one thread is a writer.
- **Simultaneously:** without happens-before ordering.

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

Check Point
SOFTWARE TECHNOLOGIES LTD

Transitivity

Thread #1

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

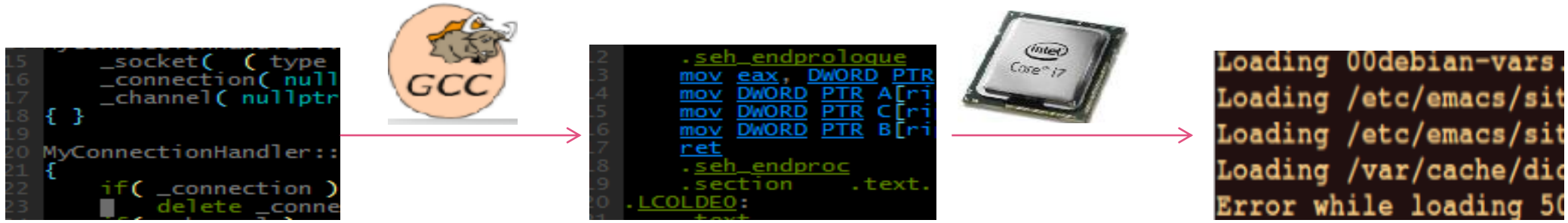
Thread #2

```
if (t1.load() )
    t2.store(1);
```

Thread #3

```
if (t2.load() )
// data == 42 ??
```

Bottom line:



- Memory is divided into Cache Lines
- This is the smallest cacheable unit
- Cache line can be placed in a restricted number of “cache slots”

- Compiler and CPU may reorder instructions
- SC-DRF
- Fences restricts the reordering

Consequences

The Idea

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

Check Point
SOFTWARE TECHNOLOGIES LTD

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 memory fence

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

- Demonstrating this would be kind’a boring….

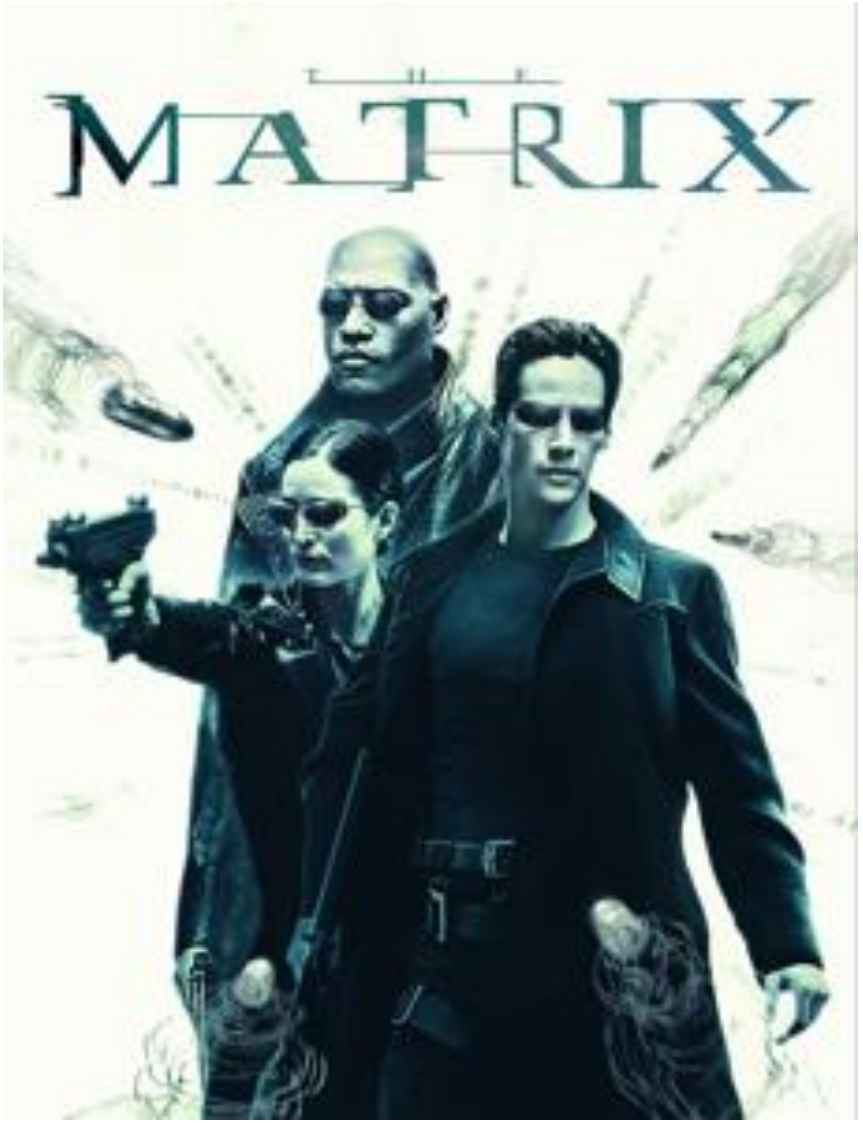
Another approach: CPU affinity

- CPU never sees its own data out of order
- Setting affinity just for this may be a huge overkill

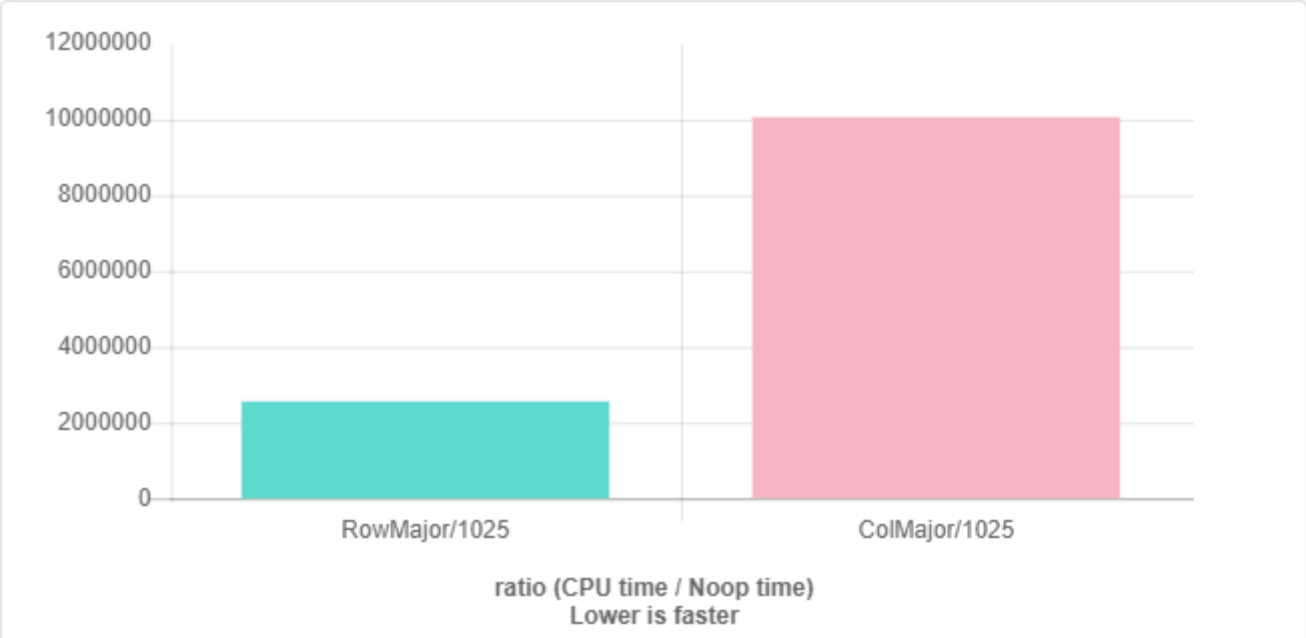
Matrix traversal

Count the amount of odds in a matrix

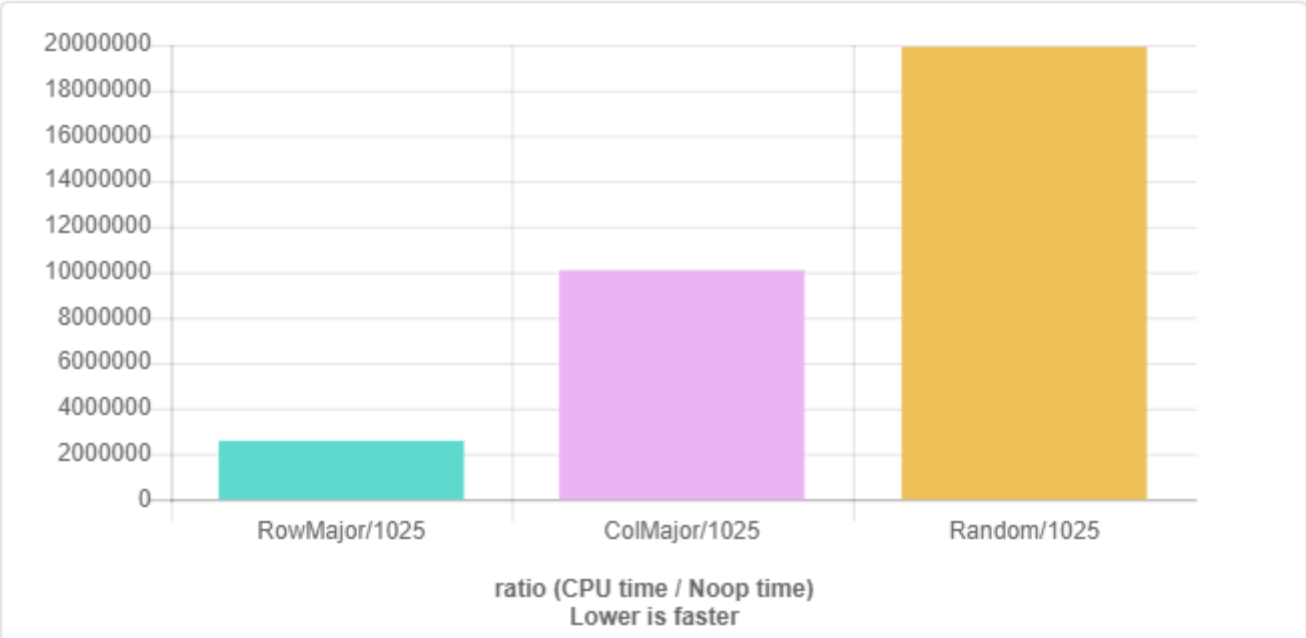
How hard can it be…



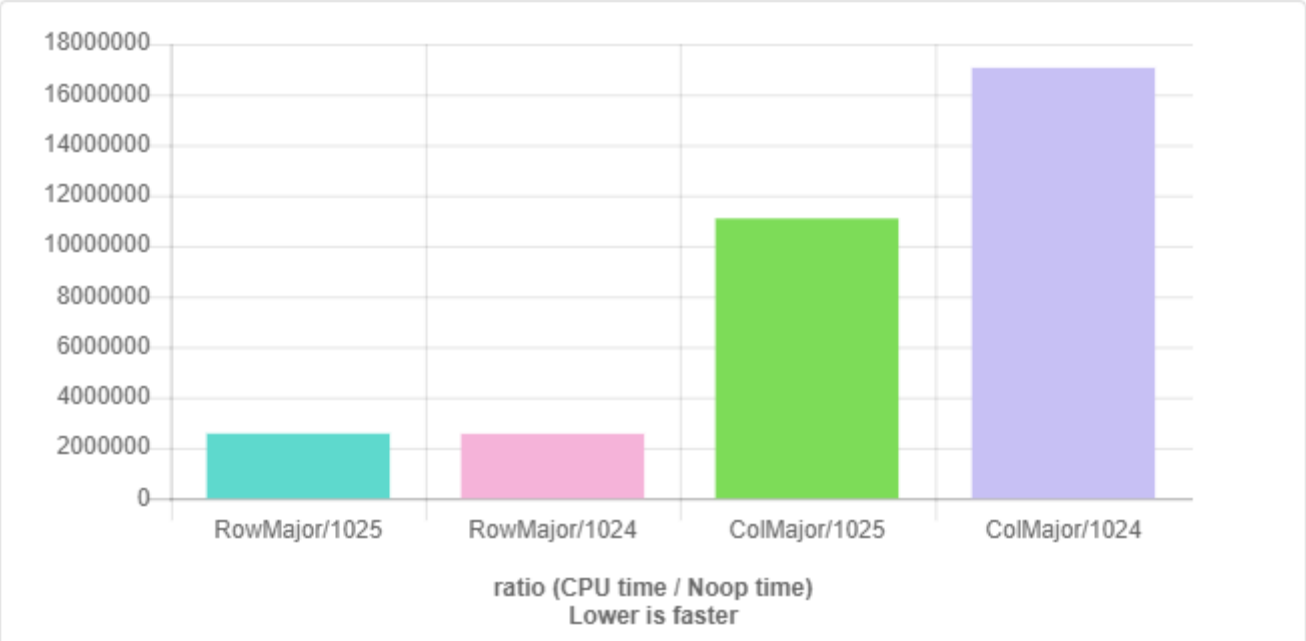
Row by row Vs. col by col



Is Col by col the worst?



Reducing matrix size



Multi-Core

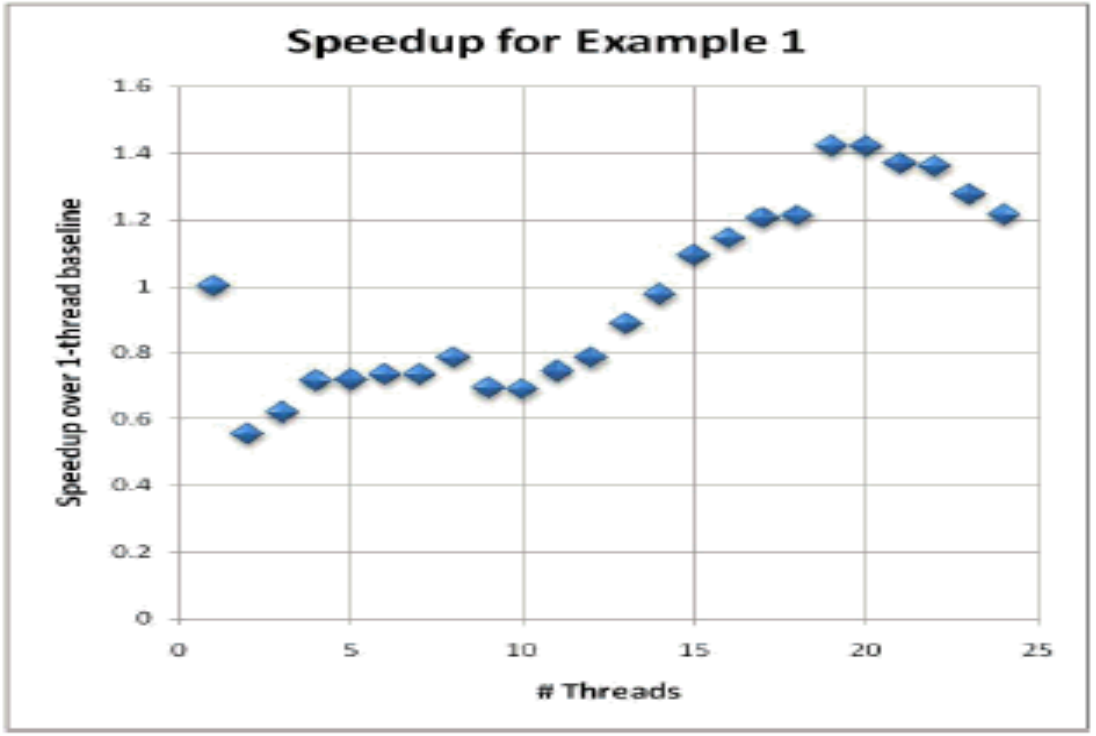
- Should be easy
  - Just make sure we do not have races

Solution Attempt #1

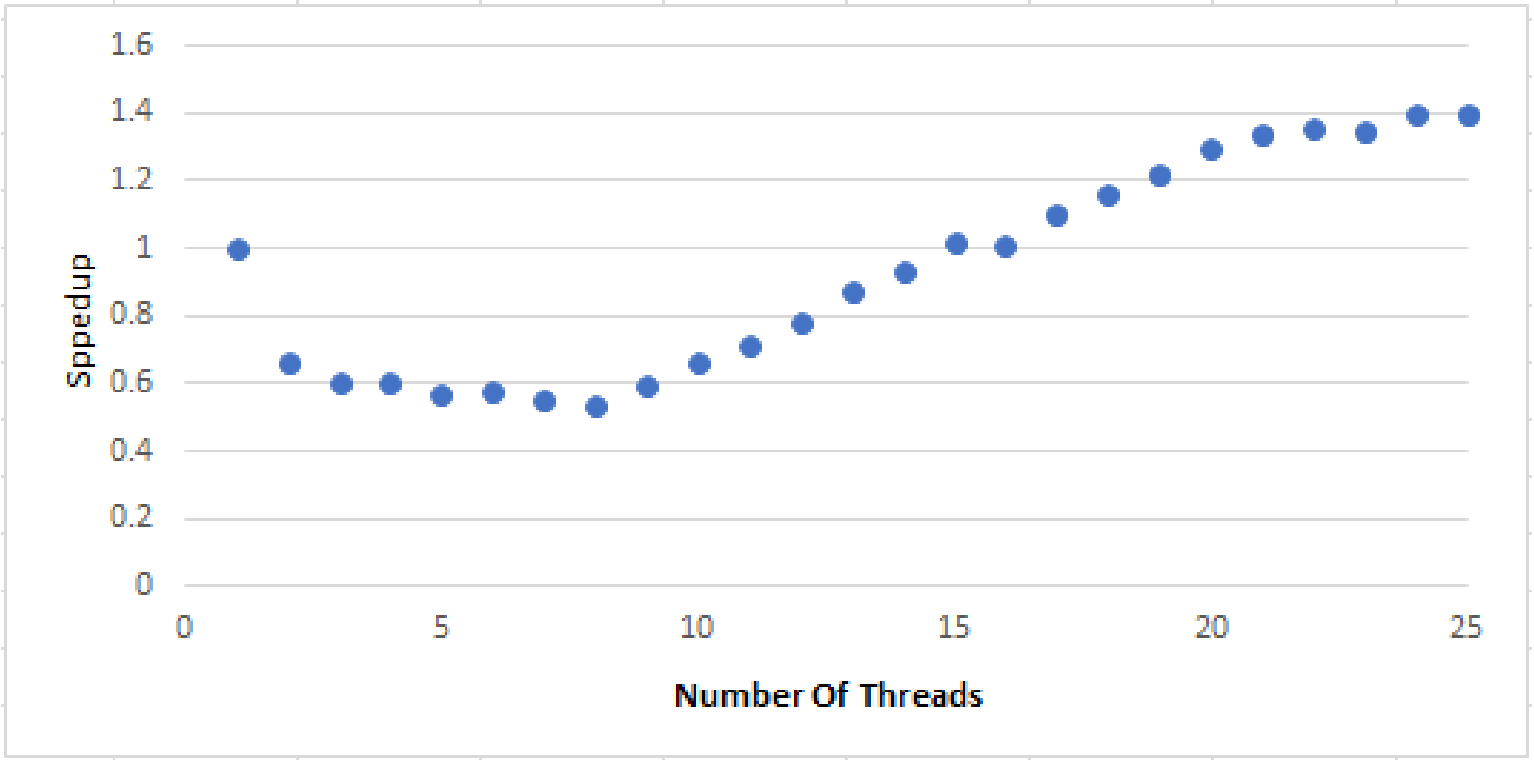
```
int counter[NumOfThreads];

for( int p = 0; p < NumOfThreads; ++p )
        pool.run( count(p) );

pool.join();
odds = 0;
for( int p = 0; p < NumOfThreads; ++p )
  odds += counter[p];
```

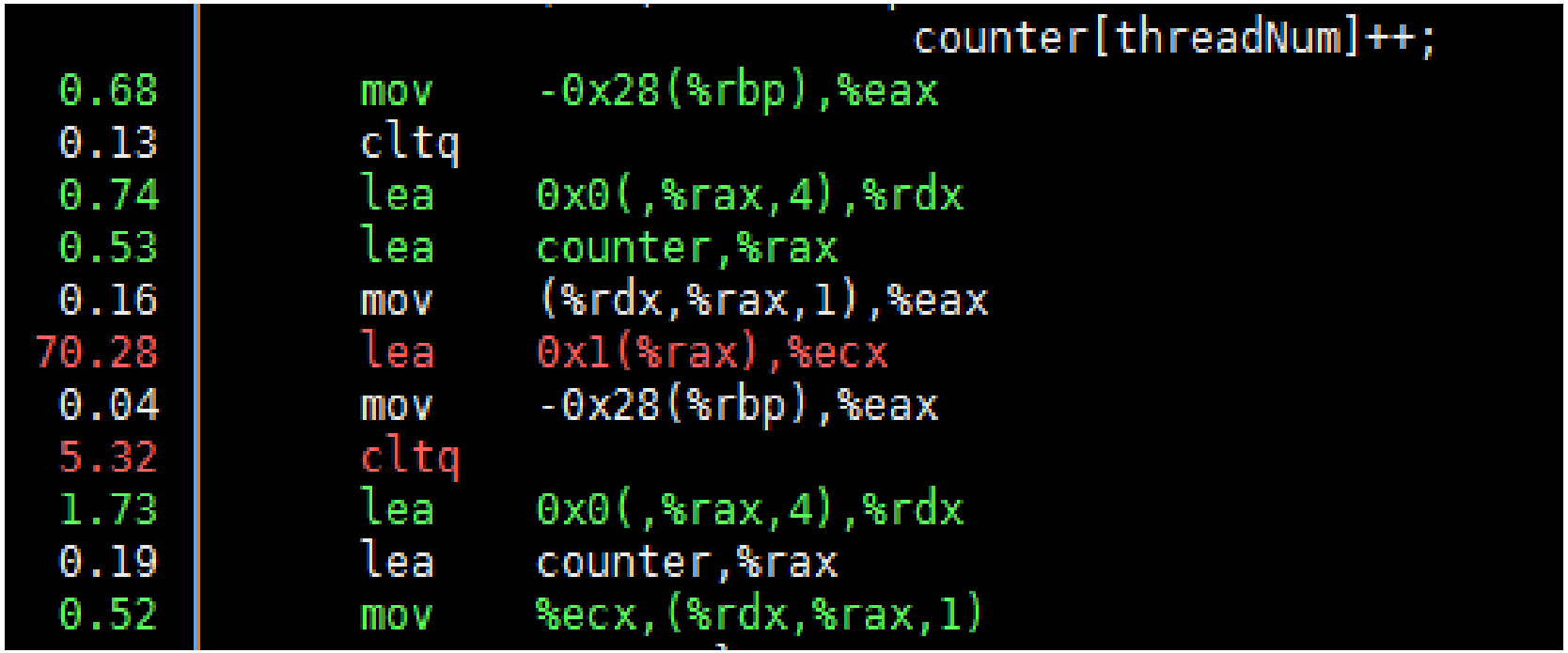
```
auto count = [] (int threadNum) {
   counter[threadNum] = 0;
   int chunkSize = DIM/(threadNum + 1);
   int myStart = threadNum * chunkSize;
   int myEnd = min( myStart+chunkSize, DIM );
   for( int row = myStart; row < myEnd; ++row )
     for( int col = 0; col < DIM; ++col )
       if( matrix[row*DIM + col] % 2 != 0 )
         ++counter[threadNum];
 } );
```

Solution Attempt #1: Results



And in my run





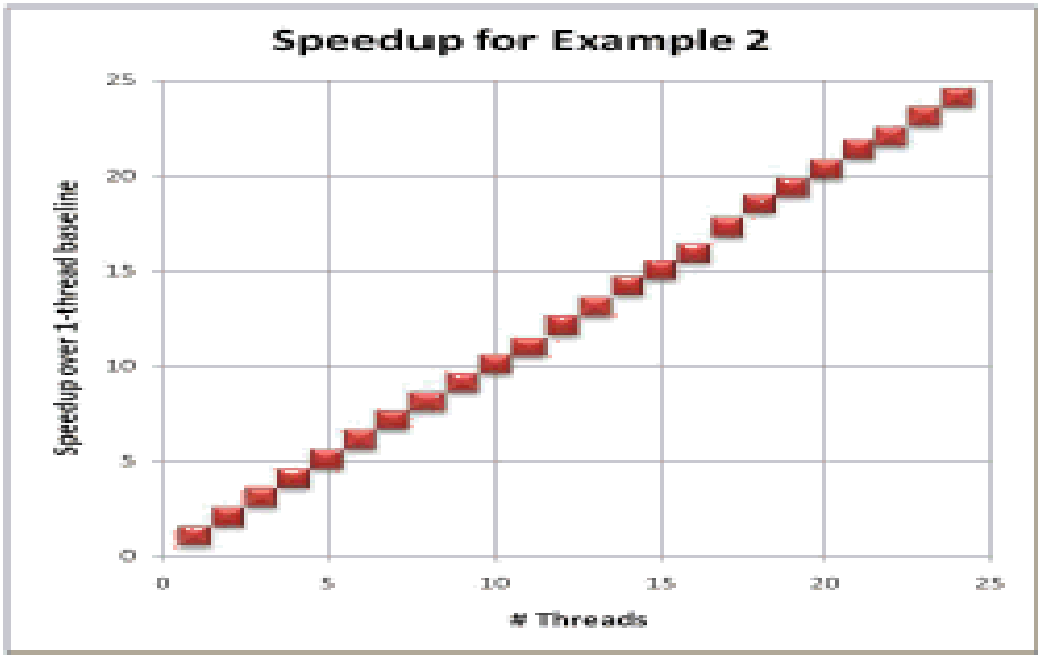
Solution Attempt #1

```
int counter[NumOfThreads];

for( int p = 0; p < NumOfThreads; ++p )
        pool.run( count(p) );

pool.join();
odds = 0;
for( int p = 0; p < NumOfThreads; ++p )
  odds += counter[p];
```

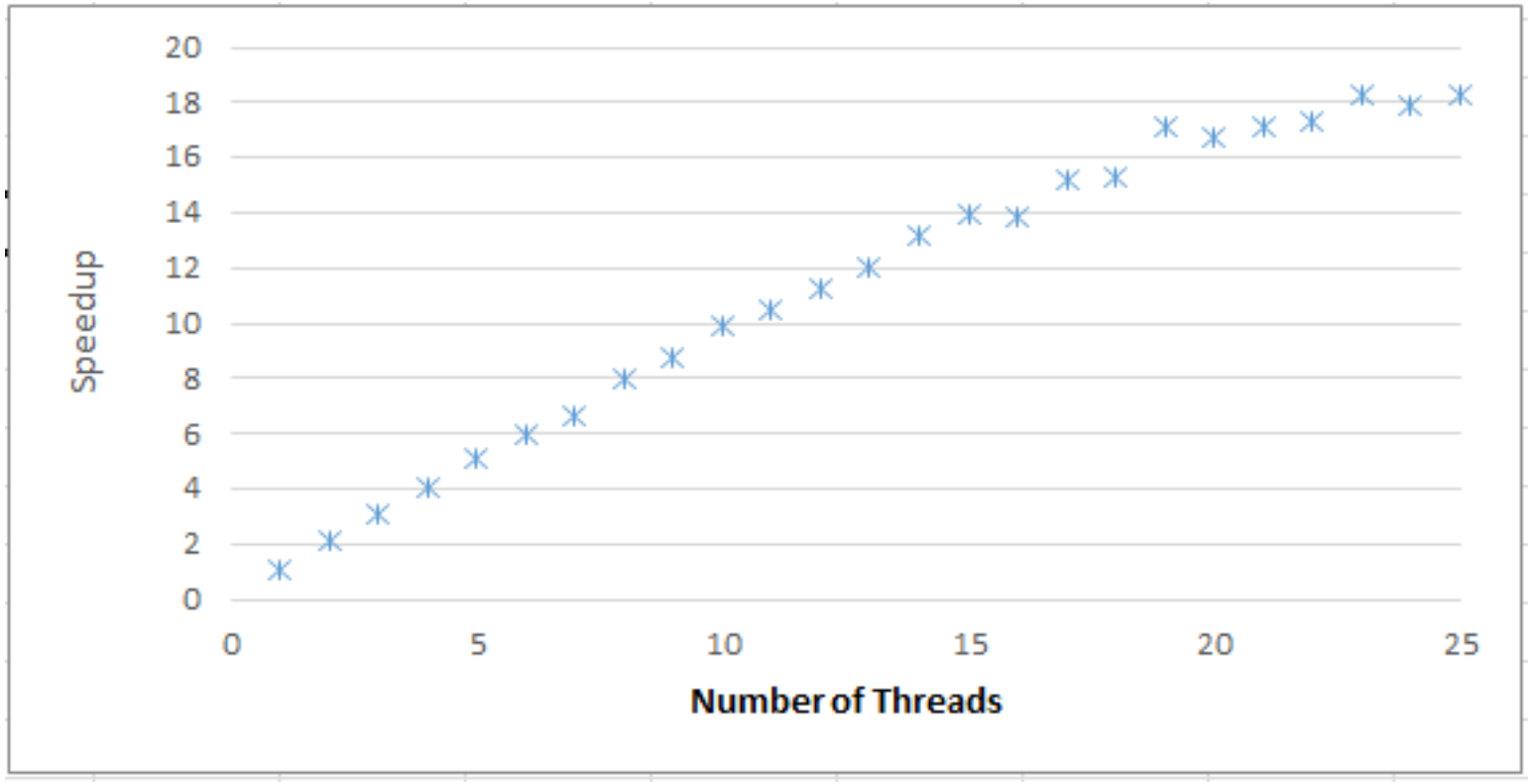
```
auto count =  [] (int threadNum) {
   counter[threadNum] = 0;
   int chunkSize = DIM/(threadNum + 1);
   int myStart = threadNum * chunkSize;
   int myEnd = min( myStart+chunkSize, DIM );
  int count = 0;
   for( int row = myStart; row < myEnd; ++row )
     for( int col = 0; col < DIM; ++col )
       if( matrix[row*DIM + col] % 2 != 0 )
         ++count; //++counter[threadNum];
     couter[threadNum] = count;
 } );
```

Could this make a difference??

- Yes…



Again, in my run



Not only arrays

- Heap allocation, globals, statics
  - Even from different translation units

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

Applications

Check Point
SOFTWARE TECHNOLOGIES LTD

Cache Oblivious Algorithms

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

Check Point
SOFTWARE TECHNOLOGIES LTD

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 time
- Ignore preparation time

How should we store the sequence??

Check Point
SOFTWARE TECHNOLOGIES LTD

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

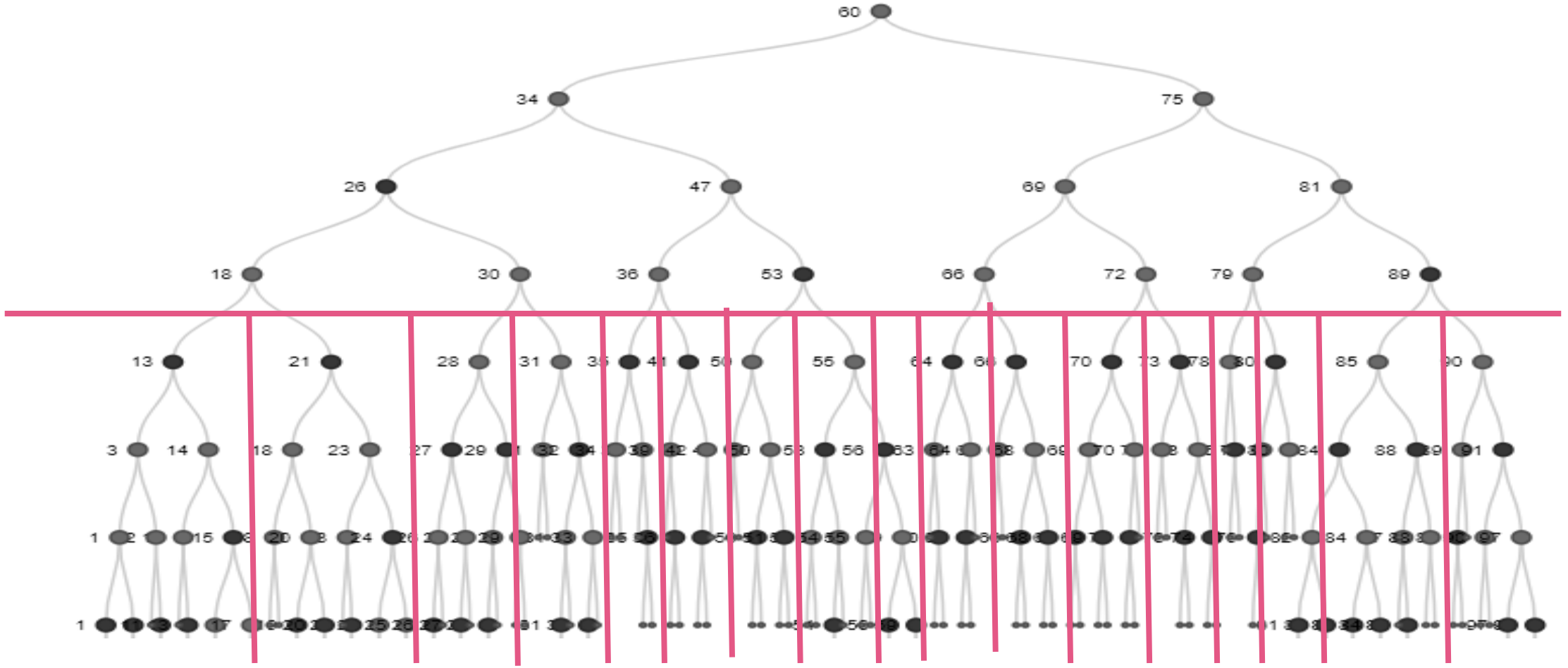
- Set B to our cache-line size
- We will require $\log_B(N)$ steps
- Each node will be loaded in one cache line
- O(log (N) / log(B)) memory accesses

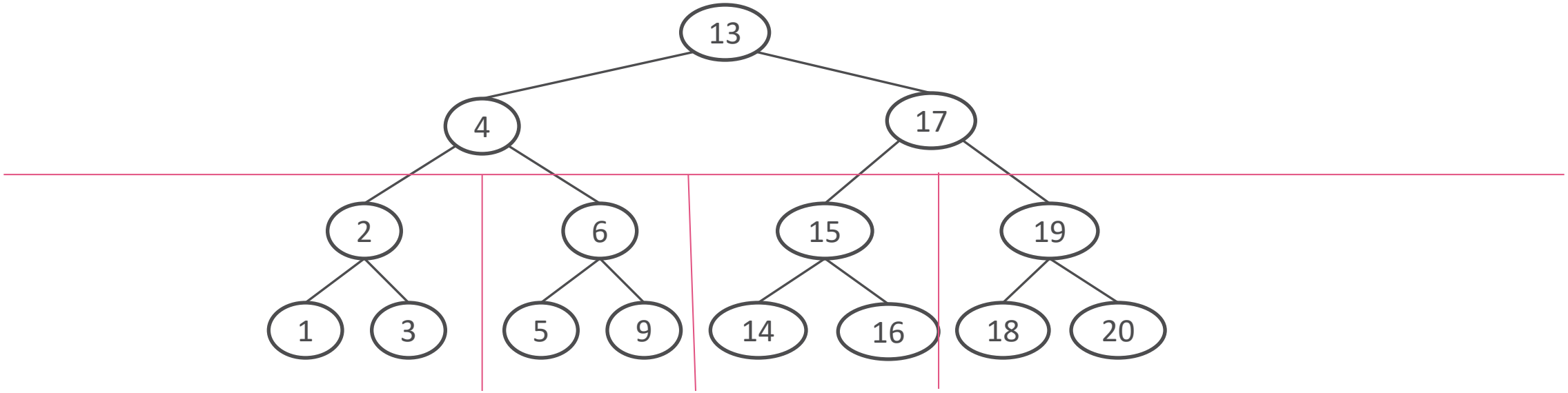
- But….
  - We need to know B

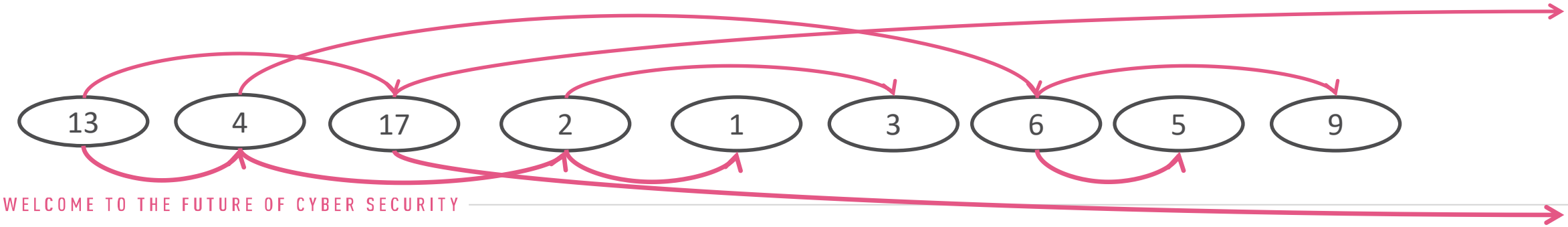
Check Point
SOFTWARE TECHNOLOGIES LTD

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







Van Emde Boas, intuitive analysis

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

Check Point
SOFTWARE TECHNOLOGIES LTD

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

_pos
_speed
_model
_name
_foo

Can the compiler help?

- 50 out of 450 cycles are real work

Compiler's domain is the 50 cycles

Not very much…

Check Point
SOFTWARE TECHNOLOGIES LTD

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

_pos
_speed
_foo
_model
_name

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 on them

Can we do better?

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

Regroup the data

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

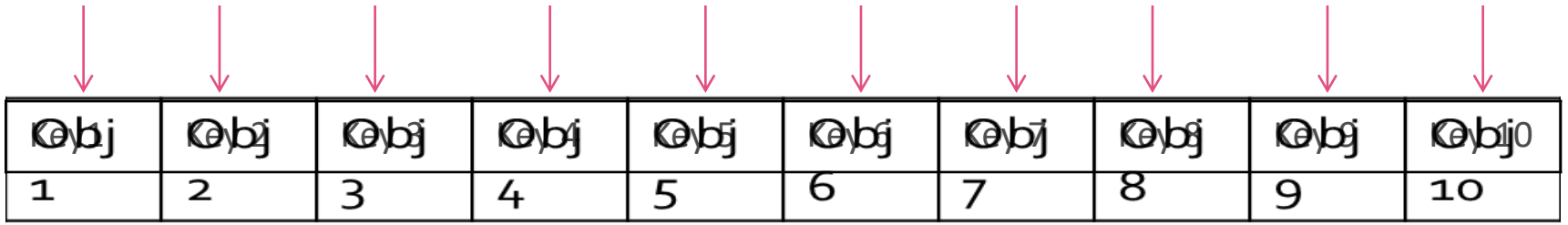
```
class ObjectPosition{
  float _pos[2];
  float _speed;
  int _foo;
};
```

Check Point
SOFTWARE TECHNOLOGIES LTD

The new cost

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

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, apple rectangle….

Polymorphism, Resolution

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

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

If time permits:

A touch on atomics

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

Check Point
SOFTWARE TECHNOLOGIES LTD

Attempt #1: Adding temporary

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

Check Point
SOFTWARE TECHNOLOGIES LTD

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;
}
```

Check Point
SOFTWARE TECHNOLOGIES LTD

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 ;
}
```

Check Point
SOFTWARE TECHNOLOGIES LTD

This works!

- But uses sequential consistency
- Can be expensive

- Can we do better?

Check Point
SOFTWARE TECHNOLOGIES LTD

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);
```

Check Point
SOFTWARE TECHNOLOGIES LTD

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;
}
```