CHANDLER CARRUTH
CPPCON2014

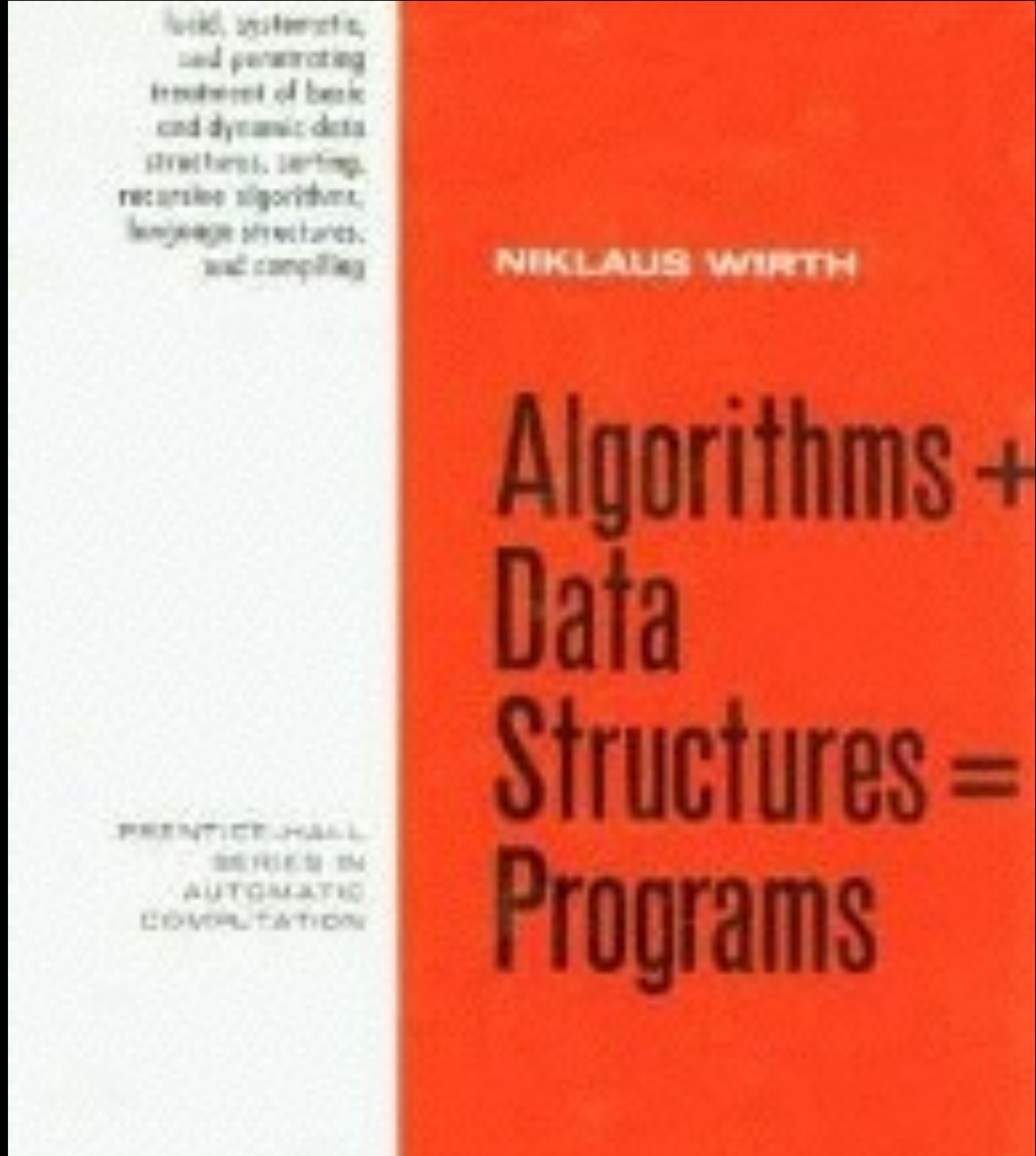
EFFICIENCY & PERFORMANCE ALGORITHMS & DATA STRUCTURES

SO WHAT?
WHO CARES?

“Software is getting slower more rapidly
than hardware becomes faster.”

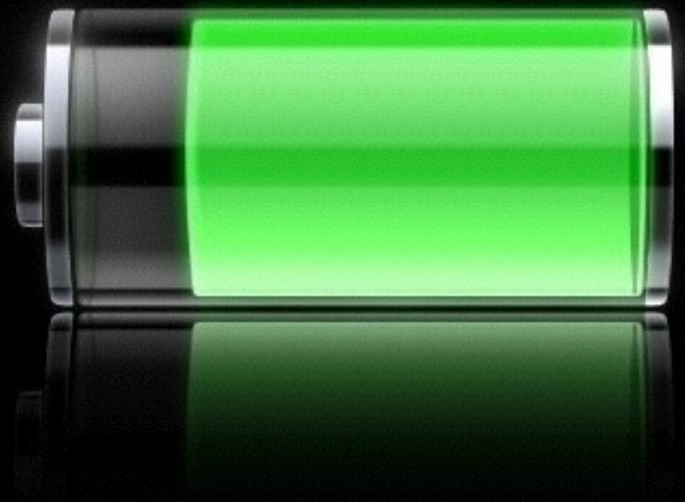
–NIKLAUS WIRTH, A PLEA FOR LEAN SOFTWARE

HE ALSO PREDICTED SOME ASPECTS OF THIS TALK...



EVEN MORE RELEVANT TODAY

SMALL, FAST — CHOOSE ONE



COMPUTE/WATT DOMINATES

RAW SPEED ISN’T EVERYTHING…

AFTER ALL, JAVA IS FASTER!


TROLL

Java is faster!

In terms of throughput in a long running system…

For specific applications…

On a well tuned JVM…

With near-optimal GC strategy selected…

SO WHY ARE WE USING C++?
JAVACON!!!

NOT SO FAST... (BA-DUM)

C++ DOESN'T GIVE YOU PERFORMANCE,
IT GIVES YOU CONTROL OVER PERFORMANCE

HIGH LEVEL THESIS:

- Efficiency through Algorithms
- Performance through Data Structures

EFFICIENCY:
HOW MUCH WORK IS REQUIRED BY A TASK

- Improving efficiency involves *doing less work*
- An efficient program is one which does the minimum (that we're aware of) amount of work to accomplish a given task

PERFORMANCE:
HOW QUICKLY A PROGRAM DOES ITS WORK

- Improving performance involves doing work faster
- But there is no such thing as “performant”, not even a word in the English language
  - There is essentially no point at which a program cannot do work any faster… until you hit Bremermann’s limit…


WAT

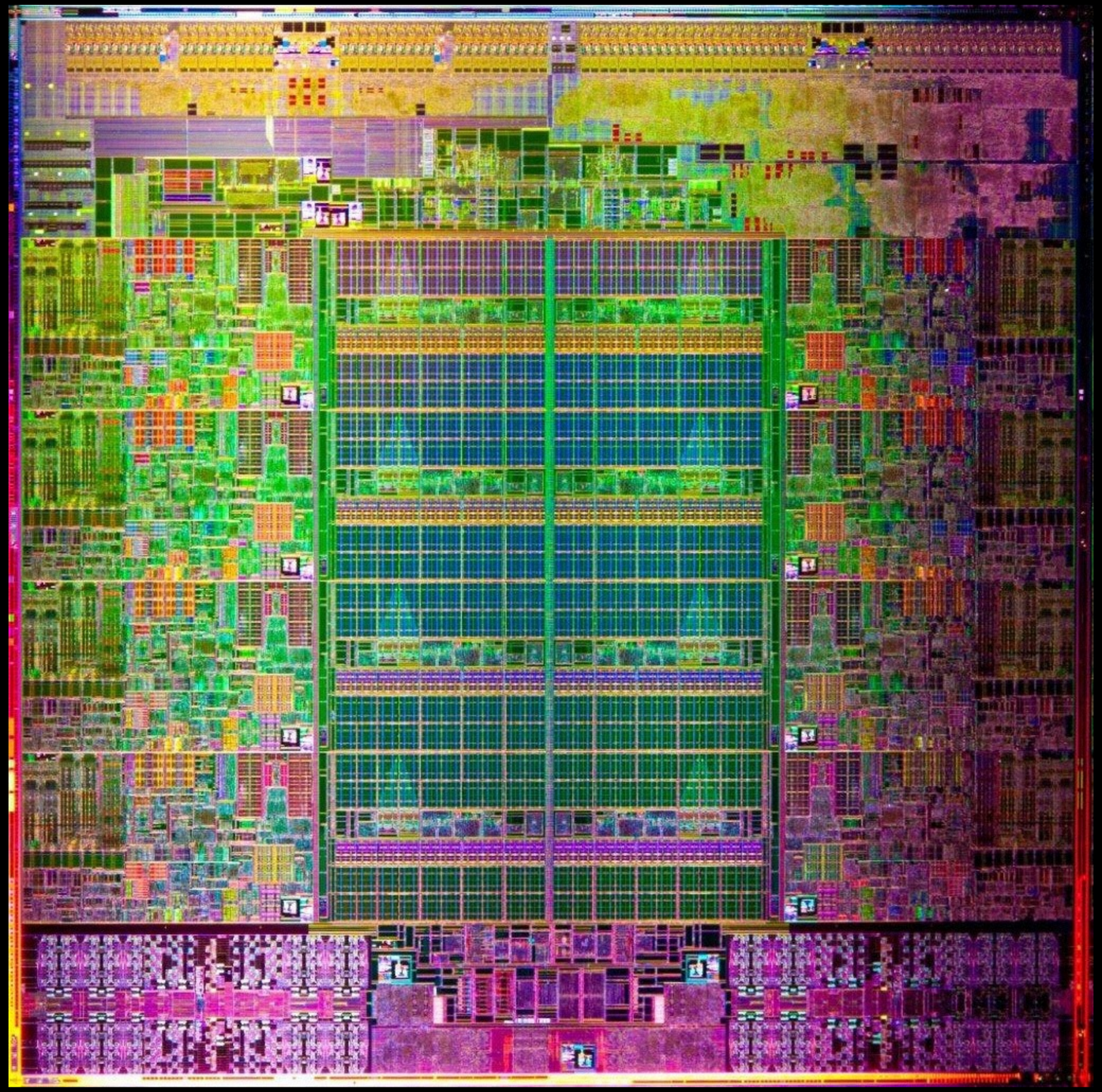
PERFORMANCE:
HOW QUICKLY A PROGRAM DOES ITS WORK

- Improving performance involves doing work faster
- But there is no such thing as “performant”, not even a word in the English language
  - There is essentially no point at which a program cannot do work any faster… until you hit Bremermann’s limit…

PERFORMANCE:
HOW QUICKLY A PROGRAM DOES ITS WORK

- What does it mean to improve the performance of software?
  - The software is going to run on a specific, real machine
  - There is some theoretical limit on how quickly it can do work

PERFORMANCE:
LIGHTING UP ALL OF THE TRANSISTORS



ALL COMES BACK TO WATTS

- Every circuit not used on a processor is wasting power
- Don’t reduce this to the absurd — it clearly doesn’t make sense to use more parts of the CPU without improving performance!

EFFICIENCY → ALGORITHMS

ALGORITHMS

- Complexity theory & analysis
- Common across higher level languages, etc.
- Very well understood by most (I hope?)
- Improving algorithmic efficiency requires finding a different way of solving the problem

SUB-STRING SEARCHING

- Initially, you might have a basic O(n^2) algorithm
- Next, we have Knuth-Morris-Pratt (a table to skip)
- Finally, we have Boyer-Moore (use the end of the needle)

DO LESS WORK BY NOT WASTING EFFORT

```
std::vector<X> f(int n) {
  std::vector<X> result;
  for (int i = 0; i < n; ++i)
    result.push_back(X(…));
  return result;
}
```

DO LESS WORK BY NOT WASTING EFFORT

```
std::vector<X> f(int n) {
  std::vector<X> result;
  result.reserve(n);
  for (int i = 0; i < n; ++i)
    result.push_back(X(…));
  return result;
}
```

DO LESS WORK BY NOT WASTING EFFORT

```
X *getX(std::string key,
        std::unordered_map<std::string,
                           std::unique_ptr<X>> &cache) {
  if (cache[key])
    return cache[key].get();

  cache[key] = std::make_unique<X>(…);
  return cache[key].get();
}
```

DO LESS WORK BY NOT WASTING EFFORT

```
X *getX(std::string key,
        std::unordered_map<std::string,
                           std::unique_ptr<X>> &cache) {
  std::unique_ptr<X> &entry = cache[key];
  if (entry)
    return entry.get();

  entry = std::make_unique<X>(…);
  return entry.get();
}
```

ALWAYS DO LESS WORK

DESIGN APIS TO HELP

PERFORMANCE → DATA STRUCTURES

DISCONTIGUOUS DATA STRUCTURES ARE THE ROOT OF ALL (PERFORMANCE) EVIL

JUST SAY NO TO LINKED LISTS

MODERN CPUS ARE TOO FAST

- 1,000,000,000 cycles per second
- 12+ cores per socket
- 3+ execution ports per core
- 36,000,000,000 instructions per second
- All of the time spent waiting for data (ok, 50%)

WE NEED FASTER MEMORY

CPUS HAVE A HIERARCHICAL CACHE SYSTEM

One cycle on a 3 GHz processor	1	ns			
L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2, 200x L1
Compress 1K bytes with Snappy	3,000	ns			
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms	
Read 4K randomly from SSD*	150,000	ns	0.15	ms	
Read 1 MB sequentially from memory	250,000	ns	0.25	ms	
Round trip within same datacenter	500,000	ns	0.5	ms	
Read 1 MB sequentially from SSD*	1,000,000	ns	1	ms	4X memory
Disk seek	10,000,000	ns	10	ms	20x datacenter RT
Read 1 MB sequentially from disk	20,000,000	ns	20	ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms	

STD::LIST

- Doubly-linked list
- Each node separately allocated
- All traversal operations chase pointers to totally new memory
- In most cases, every step is a cache miss
- Only use this when you rarely traverse the list, but very frequently update the list

JUST USE STD::VECTOR

STACKS? QUEUES? MAPS?

- Just use std::vector. Really.
- It is already a perfectly good stack.
- If the queue can have a total upper bound and/or is short-lived, consider using a vector with an index into the front
  - Grow the vector forever, chase the tail with the index

USING STD::MAP IS AN EXERCISE IN SLOWING DOWN CODE

STD::MAP

- It’s just a linked list, oriented as a binary tree
- All the downsides of linked lists
- Insertion and removal are **also** partial traversals
- Even with hinting, every rebalancing is a traversal
- But you can use std::unordered_map, right?

NO.

STD::UNORDERED_MAP

- Essentially required to be implemented with buckets of key-value pairs for each hash entry.
- These buckets are… you guessed it… linked lists.
- You essentially always have some pointer chasing.

A GOOD HASH TABLE DESIGN

- No buckets! Use open addressing into a table of the key-value pairs.
- Table stored as contiguous range of memory
- Use local probing on collisions to find an open slot in the same cache line (usually)
- Keep both key and values **small**

EFFICIENCY → ALGORITHMS

PERFORMANCE → DATA STRUCTURES

EXCEPT THAT THIS IS ALL A LIE

DATA STRUCTURES AND ALGORITHMS

- They’re tightly coupled, see Wirth’s book
  - You have to keep both factors in mind to balance between them
- Algorithms can also influence the data access pattern regardless of the data structure used
- Worse is better: bubble sort and cuckoo hashing

CONCLUSION

- Both efficiency and performance matter, today more than ever
- C++ helps you control them
- Attend to your algorithms!
- Develop pervasive habits of using APIs in a way that avoids wasted work
- Use contiguous, dense, cache-oriented data structures
- Have fun writing crazy fast C++ code


radio flyer

THANK YOU!

QUESTIONS?