CE 290 I Lecture Notes, C. Kirsch

Control and Information Management

Computer Science for Non-Computer Scientists

- Introduction
- Architecture
- Algorithms
- Computability & Complexity
- Languages
- Syntax & Semantics
- Compilers
- Data Structures
- Memory Management
- Concurrency
- Kernels & Virtual Machines
- Real Time & Model-driven Development

Software and Systems

- Target: learn how to construct complex software systems, in the cloud and on the ground
- Audience: everyone who does not care about computer science but needs it to solve a problem that involves computation (which is basically everyone)
- Approach: we explain all fundamental principles of computer science in one semester, accompanied by weekly programming assignments
- Prerequisite: basic programming experience

Myth Busting

- I prefer programming language X because it makes it more convenient to implement my app: wrong!
  - ⇝ the fundamental problem of programming is to establish functional correctness and adequate performance.
  - ⇝ different languages provide different kinds of automating the process of establishing correctness and performance and should therefore be chosen based on that insight.

configuration … program/model … library

- compiler
- code generator

} → tool

checks correctness and generates code

?

- binary
- operating system
- virtual machine

} → code

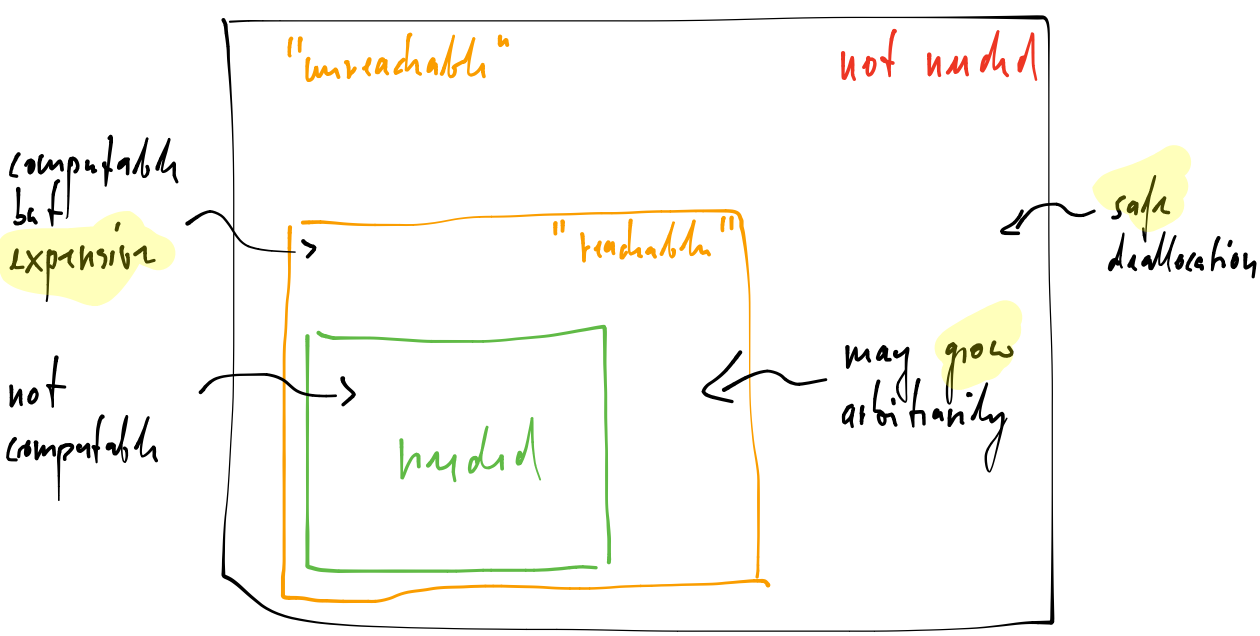
executes/simulates and checks correctness

Myth Busting

- I like garbage-collected languages because they free me from memory management: wrong!

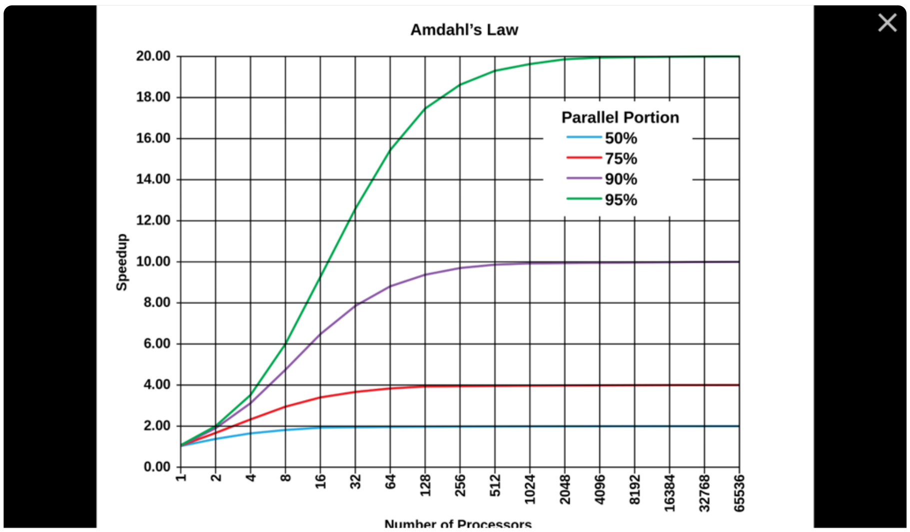
  ⇝ garbage collectors provide safe deallocation of unneeded memory but the programmer still needs to say what is unneeded, otherwise the system will run out of memory

  ⇝ programmers need to determine when memory is not needed anymore (which is a difficult, highly app-dependent property)



Myth Busting

- let's buy a 100-core machine, create 100 threads of our app, and then be 100 times faster: wrong!



The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used.

Details

speedup → $S(N) = \dfrac{1}{(1-P) + \frac{P}{N}}$ ← parallel portion

\# processors

i.e. $S(N) = N$ if $P = 1$ but $S(\infty) = \dfrac{1}{1-P}$

- sequentiality is not a property of a program but a property of program execution!

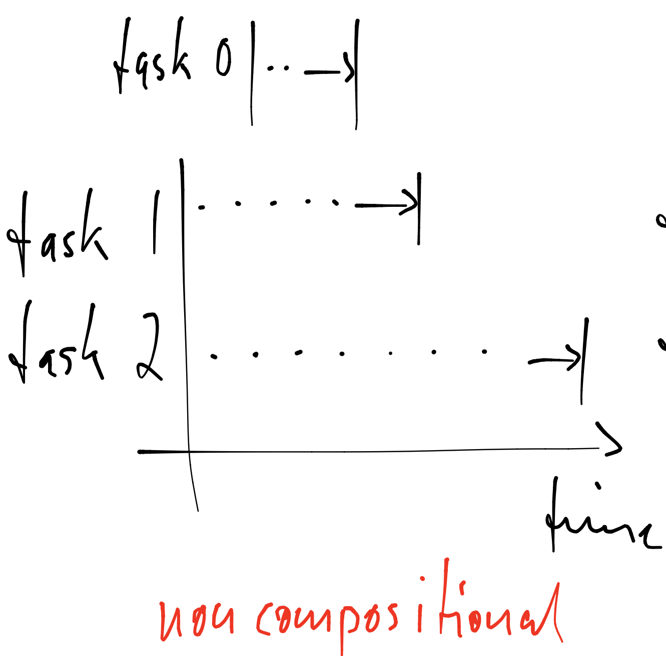
  - processor architecture
  - memory hierarchy
  - communication infrastructure

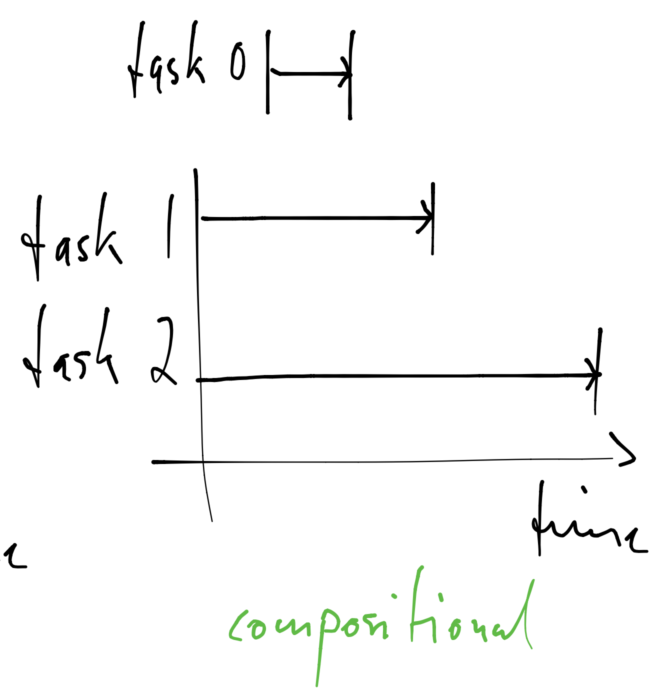
Myth Busting

- I need some real-time performance but my system is too slow, let's just get a faster machine and be done with it: wrong!

⇝ program execution time is a global, highly non-linear property that depends on all aspects of a system (hardware and all software)

⇝ real-time programming needs predictability, not speed!





Bad Habits

- Let's use this fancy library without understanding any of its implementation details: bad!
- I have been using all these compilers, who cares what they do and how: bad!
  - operating systems
  - machines
- I like virtualization, no idea how it works, but who cares, let's use it anyway: bad!

....

Problem Definition:

Software systems typically involve nontrivial functional and nonfunctional (performance) characteristics implemented by a possibly large number of interacting software and hardware components

The challenge is to fully understand that interaction and not just how to develop code (this is not a software engineering class)

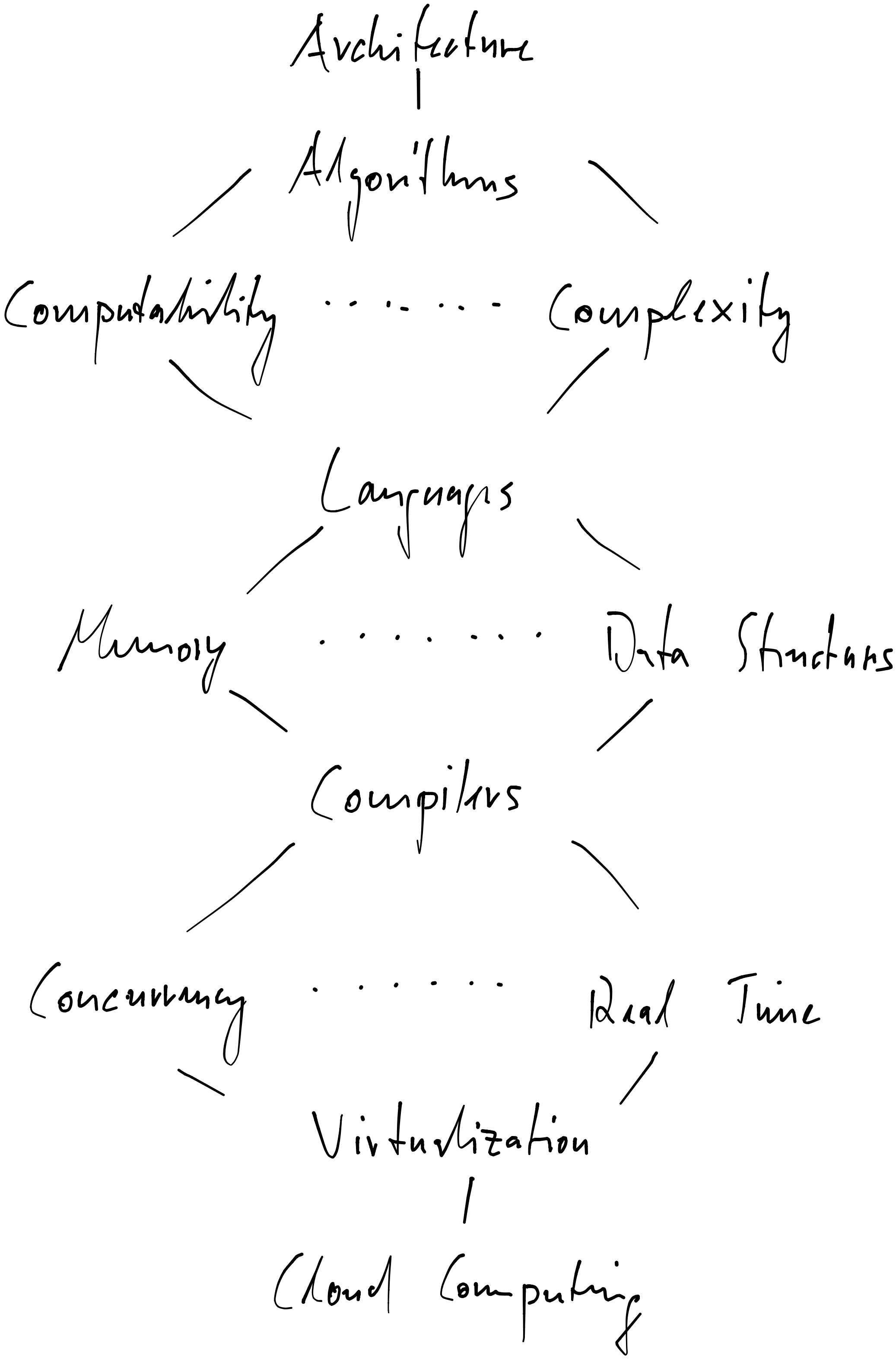
Software systems development is nevertheless mostly done by non-computer scientists with domain-specific backgrounds (our target)

Solution:

We develop the background necessary for comprehensively understanding system behavior:

- Architecture and Algorithms
- Computability and Complexity
- Languages and Compilers
- Memory and Data Structures
- Concurrency and Real Time
- Virtualization and Cloud Computing

The key is to find the correct <u>abstractions</u> for <u>modeling</u> the relevant aspects of a solution to a given problem in system design.


Architecture
Algorithms
Computability
Complexity
Languages
Memory
Data Structures
Compilers
Concurrency
Real Time
Virtualization
Cloud Computing

Cloud
Virtualization
Networks
Servers

Functional

APIs
Libraries
Algorithms
Data Structures
Complexity

Hardware

Processors
Memory
Storage
I/O
Parallelism

Nonfunctional

Time
Space
Energy
Reliability
Robustness

Languages
Compilers
Operating Systems
Silicon

Syllabus

area	content	motivation
architecture	DLX machine · registers · memory · I/O	foundation
algorithms	DLX programs · arithmetic · memory · branching · subroutines	completeness
complexity	DLX execution · size of input · # instructions · asymptotic behavior	performance
languages	C subset · arithmetic · assignment · while loops · functions	correctness
semantics	C compiler · EBNF · recursive-descent parsing · straight code generation	automation

complexity	C Execution • C code vs. DLX instructions	performance
architecture	DLX modes • index vs. address • time vs. space • fragmentation	performance

data structures	composite types
	• arrays
	• records
	  • lists
	  • trees
	  • hashtables
		abstraction

memory	memory management • heap vs. stack • dynamic memory allocation • garbage collection	spatial isolation
concurrency	models & implementations • threads vs. processes • virtual memory • shared memory vs. message passing • synchronization	abstraction performance

communication	models & implementations · programmed vs. interrupt-driven I/O · I/O instructions vs. memory-mapped I/O · asynchronous I/O	interaction
concurrency	scheduling · throughput, latency · fairness · algorithms (tests) · real time	temporal isolation
model-driven development	DSLs · ZET, BET, LET · SR, LET programming	abstraction
cloud computing	virtualization · virtual machines (system vs. process) · migration · load balancing	scalability