University of Glasgow

# Advanced Topics in Systems Programming

## Advanced Systems Programming (M)

## Lecture 1

# Introduction and Course Administration

# Contact Details and Website

- Lecturer and course coordinator:
  - Dr Colin Perkins / S101b Lilybank Gardens / colin.perkins@glasgow.ac.uk
  - No scheduled office hours – make appointments by email if necessary
- Lecture notes and other materials online:
  - https://csperkins.org/teaching/2018-2019/adv-systems-programming/ (or on the School's Moodle site)
- Paper handouts and lecture recordings will not be provided
  - The act of taking notes helps learning
  - The act of revising by reviewing your notes and other textbooks gives a useful additional perspective on the material, helps you understand the topic in depth
  - Lecture recordings discourage questions – it's already hard enough to ask a question in a large lecture theatre, but still harder when you know it's being recorded

# Rationale

- Technology shift: desktop PC → laptop, tablet, smartphone, cloud
  - Mobile, power-aware, concurrent, real-time, connected
  - But still programmed in C, running some variant of Unix – technology stack that's becoming increasing limiting
- This course will explore new techniques for safer, more effective, systems programming
  - Programming in an unmanaged environment, where data layout and performance matter
  - Operating systems kernels, device drivers, low-level networking code

# Aims and Objectives

- The course aims to explore the features of modern programming languages and operating systems that can ease the challenges of systems programming, considering type systems and run-time support.
- It will review the research literature on systems programming and operating system interfaces, discuss the limitations of deployed systems, and consider how systems programming might evolve to address the challenges of supporting modern computing systems.
- Particular emphasis will be placed on system correctness and secure programming, to ensure the resulting systems are safe to use in an adversarial environment.

# Intended Learning Outcomes (1/2)

- By the end of the course, students should be able to:
  - To discuss the advantages and disadvantages of C as a systems programming language, and to compare and contrast this with a modern systems programming language, for example Rust; to discuss the role of the type system, static analysis, and verification tools in systems programming, and show awareness of how to model system properties using the type system to avoid errors;
  - To discuss the challenges of secure low-level programming and write secure code in a modern systems programming language to perform systems programming tasks such as parsing hostile network input; show awareness of security problems in programs written in C;
  - To discuss the advantages and disadvantages of integrating automatic memory management with the operating system/runtime, to understand the operation of popular garbage collection algorithms and alternative techniques for memory management, and know when it might be appropriate to apply such techniques and managed run-times to real-time systems and/or operating systems;
  - …

# Intended Learning Outcomes (2/2)

- …
- To understand the impact of heterogeneous multicore systems on operating systems, compare and evaluate different programming models for concurrent systems, their implementation, and their impact on operating systems; and
- To construct and/or analyse simple programming to demonstrate understanding of novel techniques for memory management and/or concurrent programming, to understand the trade-offs and implementation decisions.

# Pre-requisites

- The following courses are pre-requisites:
  - Systems Programming (H)
  - Operating Systems (H)
  - Networked Systems (H)
  - Functional Programming (H)

- You are expected to be familiar with the C programming language, and to understand the basics of operating systems and networking
- A conceptual understanding of functional programming is assumed, but Haskell programming is not required

# Course Outline and Timetable

| Week | Lecture | Laboratory |
|---|---|---|
| 1 | #1: Introduction | #1: Introducing Rust (1/2) |
| 2 | #2: Types and Systems Programming | #2: Introducing Rust (2/2) |
| 3 | #3: Type-based Modelling and Design | #3: Types and Traits |
| 4 | #4: Resource Ownership and Memory Management | #4: Ownership, Pointers, and Memory |
| 5 | #5: Garbage Collection | |
| 6 | #6: Concurrency | #5: Concurrency |
| 7 | #7: Coroutines and Asynchronous Programming | #6: Coroutines and Asynchronous Code |
| 8 | #8: Security Considerations | |
| 9 | #9: Open Issues and Future Directions | |
| 10 | | |

# Laboratory Sessions

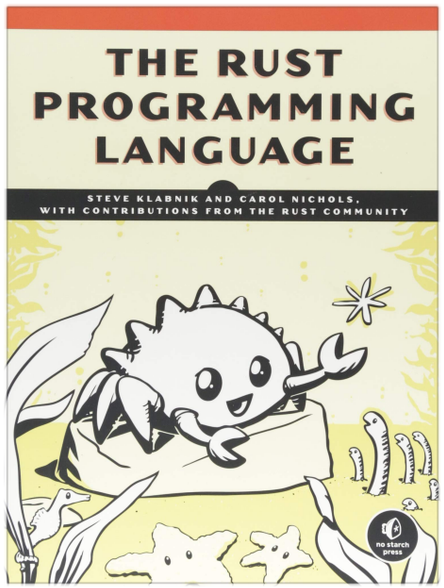
- Lab sessions are in Boyd Orr 1028, 13:00-14:00 on Mondays
- Self-study handouts will be available online before each lab – work through the handouts at own pace
- I'll be available in the labs to answer questions
- Lab handouts have been tested using Rust 1.31.1 on OS X 10.12.6, but are expected to run on the Linux machines in the lab

# Assessment

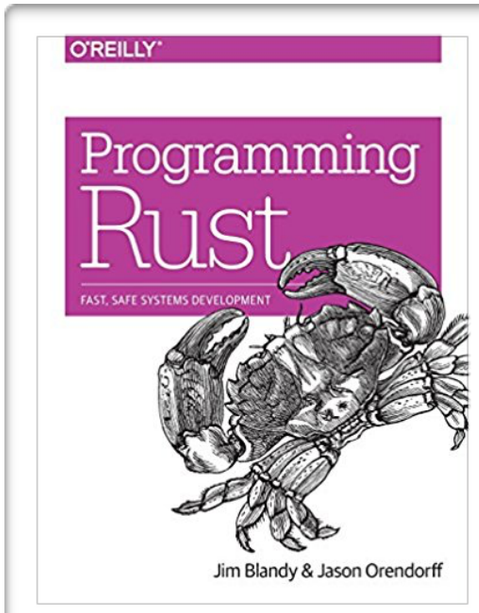
- This is a level M course, worth 10 credits
- Assessment is by examination (80%) and coursework (20%)
  - Sample exam paper, with worked answers, will be made available on the website/Moodle before the end of the semester
  - Material from the lectures, labs, and cited papers is examinable
    - Aim is to test your understanding of the material, not to test your memory of all the details; explain why – don't just recite what
- Coursework:

| Coursework | Date Set: | Date Due: | Weighting: | Topic: |
|---|---|---|---|---|
| Exercise 1 | Lecture 4 | Lecture 6 | 10% | Memory Management (essay) |
| Exercise 2 | Lecture 6 | Lecture 9 | 10% | Concurrent Programming (code) |

# Recommended Reading (1/2)




Steve Klabnik and Carol Nichols, "The Rust Programming Language", 2nd Edition, 2018, ISBN 978-1-59327-828-1 (Amazon, free online edition).

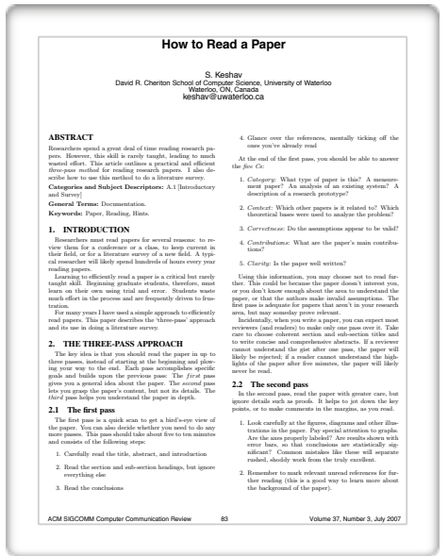



Jim Blandy and Jason Orendorff, "Programming Rust", O'Reilly, 2018, ISBN 978-1-491-92728-1 (Amazon).

- The Rust Programming language (https://rust-lang.org/) will be used to illustrate principles of ownership, memory management, and type-driven programming – read one of the above books
- You are expected the learn the basics of programming in Rust

# Recommended Reading (2/2)

- Research papers will be cited to illustrate some concepts
  - Citations with URLs and/or DOIs are provided
  - Resolve DOIs via http://dx.doi.org/
  - All papers are accessible at no cost from the campus networks
- You are expected to read these research papers
  - Critical reading of a research paper requires practice
  - Read in a structured manner, not end-to-end, thinking about the material as you go
    - Focus on the concepts, not the details
  - Realise that research papers are written to explore new ideas
    - Some will be good ideas, some less so
    - Some will be interesting but impractical
    - What's impractical today might be important tomorrow – changes in technology and/or society can change what's feasible/desirable
  - Think and judge for yourself!



How to Read a Paper

S. Keshav
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, ON, Canada
keshav@uwaterloo.ca

**ABSTRACT**

Researchers spend a great deal of time reading research papers. However, this skill is rarely taught, leading to much wasted effort. This article outlines a practical and efficient *three-pass method* for reading research papers. I also describe how to use this method to do a literature *survey*.

**Categories and Subject Descriptors:** A.1 [Introductory and Survey]

**General Terms:** Documentation.

**Keywords:** Paper, Reading, Hints.

**1. INTRODUCTION**

Researchers must read papers for several reasons: to review them for a conference or a class, to keep current in their field, or for a literature *survey* of a new field. A typical researcher will likely spend hundreds of hours every year reading papers.

Learning to efficiently read a paper is a critical but rarely taught skill. Beginning graduate students, therefore, must learn on their own using trial and error. Students waste much effort in the process and are frequently driven to frustration.

For many years I have used a simple approach to efficiently read papers. This paper describes the 'three-pass' approach and its use in doing a literature survey.

**2. THE THREE-PASS APPROACH**

The key idea is that you should read the paper in up to three passes, instead of starting at the beginning and plowing your way to the end. Each pass accomplishes specific goals and builds upon the previous pass: The *first* pass gives you a general idea about the paper. The *second* pass lets you grasp the paper's content, but not its details. The *third* pass helps you understand the paper in depth.

**2.1 The first pass**

The first pass is a quick scan to get a bird's-eye view of the paper. You can also decide whether you need to do any more passes. This pass should take about five to ten minutes and consists of the following steps:

1. Carefully read the title, abstract, and introduction
2. Read the section and sub-section headings, but ignore everything else
3. Read the conclusions
4. Glance over the references, mentally ticking off the ones you've already read

At the end of the first pass, you should be able to answer the *five Cs*:

1. *Category*: What type of paper is this? A measurement paper? An analysis of an existing system? A description of a research prototype?
2. *Context*: Which other papers is it related to? Which theoretical bases were used to analyze the problem?
3. *Correctness*: Do the assumptions appear to be valid?
4. *Contributions*: What are the paper's main contributions?
5. *Clarity*: Is the paper well written?

Using this information, you may choose not to read further. This could be because the paper doesn't interest you, or you don't know enough about the area to understand the paper, or that the authors make invalid assumptions. The first pass is adequate for papers that aren't in your research area, but may someday prove relevant.

Incidentally, when you write a paper, you can expect most reviewers (and readers) to make only one pass over it. Take care to choose coherent section and sub-section titles and to write concise and comprehensive abstracts. If a reviewer cannot understand the gist after one pass, the paper will likely be rejected; if a reader cannot understand the highlights of the paper after five minutes, the paper will likely never be read.

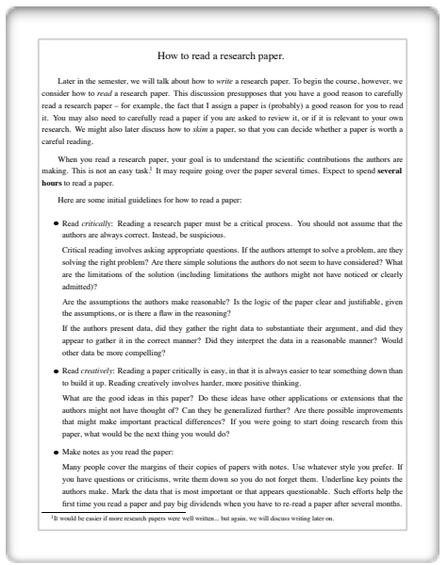
**2.2 The second pass**

In the second pass, read the paper with greater care, but ignore details such as proofs. It helps to jot down the key points, or to make comments in the margins, as you read.

1. Look carefully at the figures, diagrams and other illustrations in the paper. Pay special attention to graphs. Are the axes properly labeled? Are results shown with error bars, so that conclusions are statistically significant? Common mistakes like these will separate rushed, shoddy work from the truly excellent.
2. Remember to mark relevant unread references for further reading (this is a good way to learn more about the background of the paper).



S. Keshav, "How to Read a Paper", ACM Computer Communication Review, 37(3), July 2007
DOI: 10.1145/1273445.1273458



How to read a research paper.

Later in the semester, we will talk about how to *write* a research paper. To begin the course, however, we consider how to *read* a research paper. This discussion presupposes that you have a good reason to carefully read a research paper – for example, the fact that I assign a paper is (probably) a good reason for you to read it. You may also need to carefully read a paper if you are asked to review it, or if it is relevant to your own research. We might also later discuss how to *skim* a paper, so that you can decide whether a paper is worth a careful reading.

When you read a research paper, your goal is to understand the scientific contributions the authors are making. This is not an easy task.[1] It may require going over the paper several times. Expect to spend **several hours** to read a paper.

Here are some initial guidelines for how to read a paper:

- Read *critically*: Reading a research paper must be a critical process. You should not assume that the authors are always correct. Instead, be suspicious.

  Critical reading involves asking appropriate questions. If the authors attempt to solve a problem, are they solving the right problem? Are there simple solutions the authors do not seem to have considered? What are the limitations of the solution (including limitations the authors might not have noticed or clearly admitted)?

  Are the assumptions the authors make reasonable? Is the logic of the paper clear and justifiable, given the assumptions, or is there a flaw in the reasoning?

  If the authors present data, did they gather the right data to substantiate their argument, and did they appear to gather it in the correct manner? Did they interpret the data in a reasonable manner? Would other data be more compelling?

- Read *creatively*: Reading a paper critically is easy, in that it is always easier to tear something down than to build it up. Reading creatively involves harder, more positive thinking.

  What are the good ideas in this paper? Do these ideas have other applications or extensions that the authors might not have thought of? Can they be generalized further? Are there possible improvements that might make important practical differences? If you were going to start doing research from this paper, what would be the next thing you would do?

- Make notes as you read the paper:

  Many people cover the margins of their copies of papers with notes. Use whatever style you prefer. If you have questions or criticisms, write them down so you do not forget them. Underline key points the authors make. Mark the data that is most important or that appears questionable. Such efforts help the first time you read a paper and pay big dividends when you have to re-read a paper after several months.

[1] It would be easier if more research papers were well written... but again, we will discuss writing later on.

http://www.eecs.harvard.edu/~michaelm/postscripts/ReadPaper.pdf

# Systems Programming

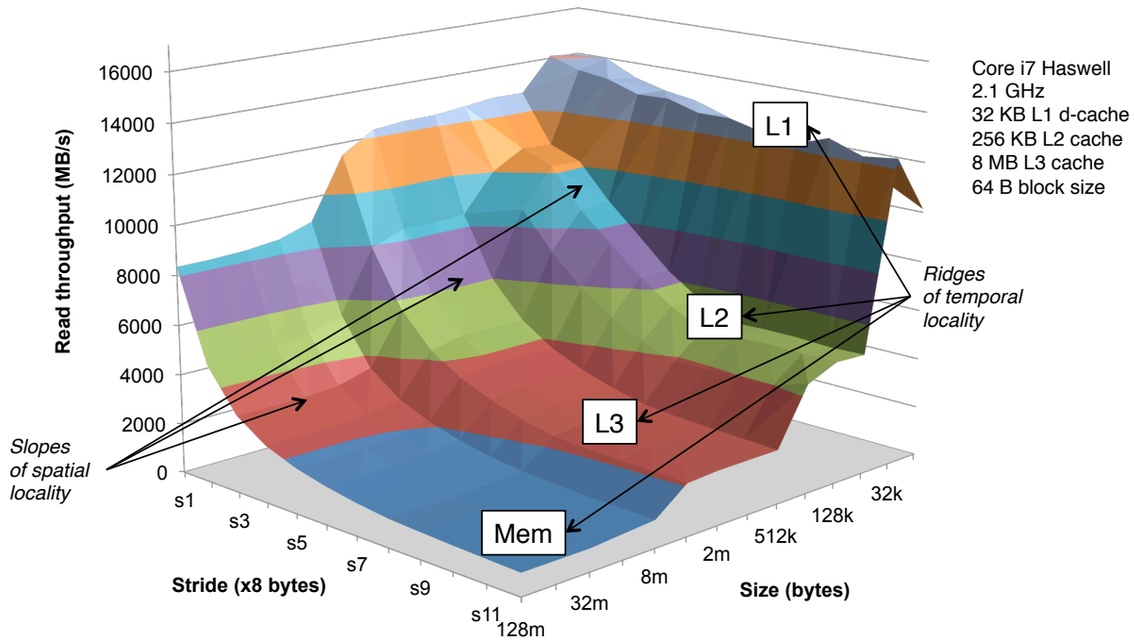
- What is systems programming?
- The state of the art
- Challenges and limitations

# What is Systems Programming?

- Infrastructure components, operating systems, device drivers, network protocols, services
- Systems programs tend to be constrained by:
  - Memory management and data representation
  - I/O operations
  - Management of shared state
  - Performance

# Memory Management and Data Representation

- Predictability
  - Timing of allocations must be bounded for real-time applications
  - Bounds on memory usage
- Data locality
  - Cache line sharing impacts performance
- Data representation
  - Device drivers that must control via fixed layout control registers
  - Network protocol implementations must conform to specified packet layout
  - Ensuring data is aligned and packed into cache lines for high performance
- Systems programming languages offer control of memory management and data representation – others languages lack such controls
  - e.g., stack *vs.* heap allocation and explicit pointers in C, compared to Java where all objects are allocated on the heap and accessed via implicit references

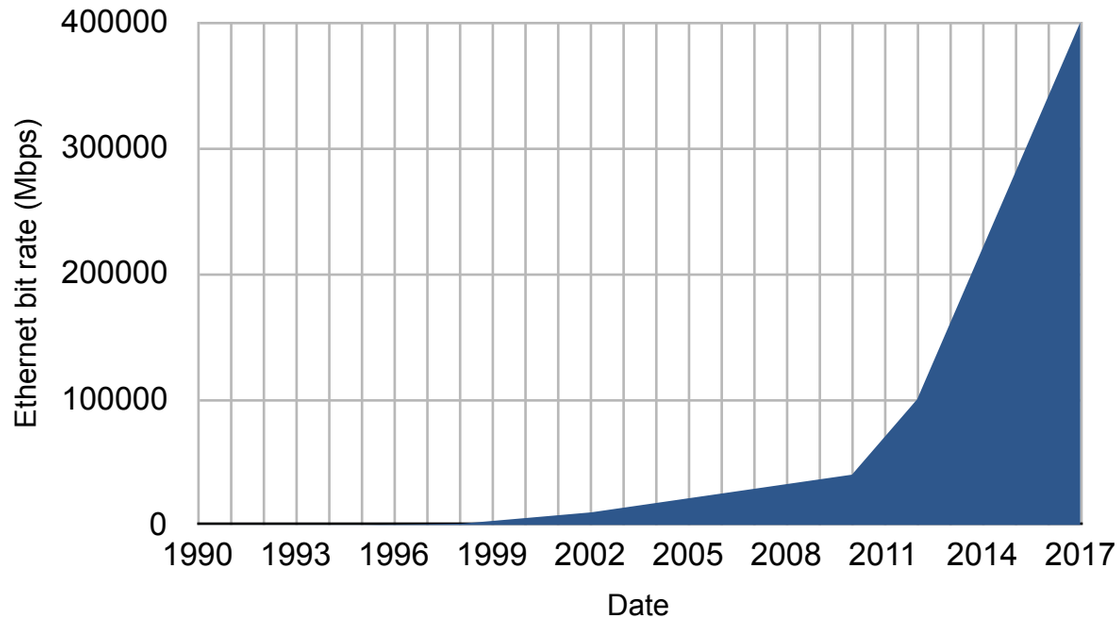



Smaller values of **stride** indicate data with better spatial locality; **size** is the total amount of data accessed

Source: Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Edition, Pearson, 2016, Fig. 6.41. http://csapp.cs.cmu.edu/3e/figures.html (Permission granted for lecture use with attribution)

# I/O Operations

- Network performance increasingly a bottleneck:
  - Chart shows Ethernet bit rate over time – wireless links follow a similar curve
  - Closely tracking exponential growth over time – unlike CPU speed, which stopped growing significantly mid-2000s
  - MTU remains constant but packet rate increases → fewer cycles to process each packet
- SSD performance on a similar trend for file system access




- I/O performance of systems software critical to overall system performance
  - Device drivers, network protocol stack, file system

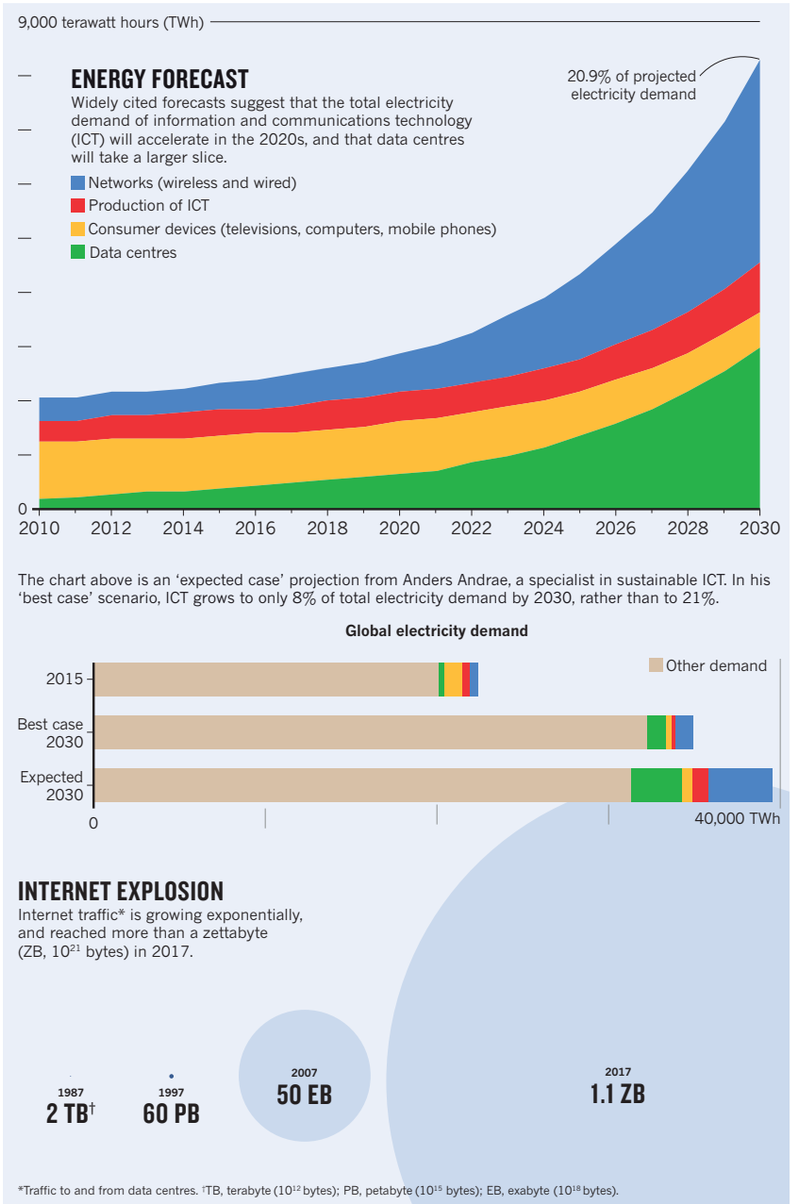
# Management of Shared State

- Systems programs responsible for coordinating shared mutable state:
  - State shared across layers/between kernel and applications
    - Data structures for zero-copy networking
    - Header processing and state for the TCP stack
  - State shared between threads
    - Internal state of the kernel
    - File systems
    - Network code
  - Highly performance critical

- Systems programming languages allow sharing data between layers and/or threads – other languages disallow/discourage such sharing

# Performance

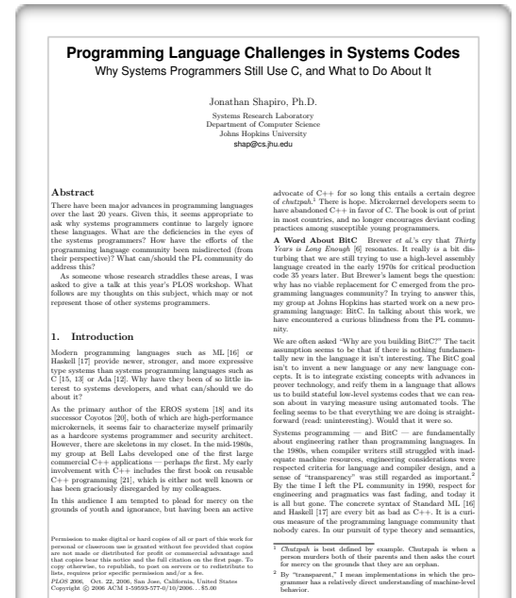
- Systems infrastructure performance fundamentally affects overall system and application performance
  - Mobile devices have limited battery life
  - Data centre efficiency and power consumption
- Systems components often the bottleneck in terms of overall performance and power efficiency
  - Simply because they're the basis on which the higher-level systems depend




Source: N. Jones, "The Information Factories", Nature, v.561, p.163–166, Sep. 2018.DOI:10.1038/d41586-018-06610-y

# Systems Programming

- Systems programming languages offer low-level control of data representation, memory management, I/O, and sharing.
- They are high-performance – concrete rather than high abstraction



**Programming Language Challenges in Systems Codes**
Why Systems Programmers Still Use C, and What to Do About It

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University
shap@cs.jhu.edu

**Abstract**

There have been major advances in programming languages over the last 20 years. Given this, it seems appropriate to ask why systems programmers continue to largely ignore these languages. What are the deficiencies in the eyes of the systems programmers? How have the efforts of the programming language community been misdirected (from their perspective)? What can/should the PL community do address this?

As someone whose research straddles these areas, I was asked to give a talk at this year's PLOS workshop. What follows are my thoughts on this subject, which may or not represent those of other systems programmers.

**1. Introduction**

Modern programming languages such as ML [16] or Haskell [17] provide newer, stronger, and more expressive type systems than systems programming languages such as C [15, 13] or Ada [12]. Why have they been of so little interest to systems developers, and what can/should we do about it?

As the primary author of the EROS system [18] and its successor Coyotos [20], both of which are high-performance microkernels, it seems fair to characterize myself primarily as a hardcore systems programmer and security architect. However, there are skeletons in my closet. In the mid-1980s, my group at Bell Labs developed one of the first large commercial C++ applications — perhaps the first. My early involvement with C++ includes the first book on reusable C++ programming [21], which is either not well known or has been graciously disregarded by my colleagues.

In this audience I am tempted to plead for mercy on the grounds of youth and ignorance, but having been an active advocate of C++ for so long this entails a certain degree of *chutzpah*.[1] There is hope. Microkernel developers seem to have abandoned C++ in favor of C. The book is out of print in most countries, and no longer encourages deviant coding practices among susceptible young programmers.

**A Word About BitC** Brewer *et al.*'s cry that *Thirty Years is Long Enough* [6] resonates. It really *is* a bit disturbing that we are still trying to use a high-level assembly language created in the early 1970s for critical production code 35 years later. But Brewer's lament begs the question: why has no viable replacement for C emerged from the programming languages community? In trying to answer this, my group at Johns Hopkins has started work on a new programming language: BitC. In talking about this work, we have encountered a curious blindness from the PL community.

We are often asked "Why are you building BitC?" The tacit assumption seems to be that if there is nothing fundamentally new in the language it isn't interesting. The BitC goal isn't to invent a new language or any new language concepts. It is to integrate existing concepts with advances in prover technology, and reify them in a language that allows us to build stateful low-level systems codes that we can reason about in varying measure using automated tools. The feeling seems to be that everything we are doing is straightforward (read: uninteresting). Would that it were so.

Systems programming — and BitC — are fundamentally about engineering rather than programming languages. In the 1980s, when compiler writers still struggled with inadequate machine resources, engineering considerations were respected criteria for language and compiler design, and a sense of "transparency" was still regarded as important.[2] By the time I left the PL community in 1990, respect for engineering and pragmatics was fast fading, and today it is all but gone. The concrete syntax of Standard ML [16] and Haskell [17] are every bit as bad as C++. It is a curious measure of the programming language community that nobody cares. In our pursuit of type theory and semantics,


*PLOS* 2006, Oct. 22, 2006, San Jose, California, United States


[1] *Chutzpah* is best defined by example. Chutzpah is when a person murders both of their parents and then asks the court for mercy on the grounds that they are an orphan.

[2] By "transparent," I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

J. Shapiro, "Programming language challenges in systems codes: why systems programmers still use C, and what to do about it", Workshop on Programming Languages and Operating Systems, San Jose, CA, October 2006. DOI:10.1145/1215995.1216004

# The State of the Art

- Most devices run some variant of Unix as their operating system, programmed in C
  - Original version of Unix written in assembly for PDP-7 minicomputer in 1969
  - Ported to the PDP-11/40 in early 1970s, re-writing into C at that language was developed
    - “The PDP-11/40 was designed to fit a broad range of applications, from small stand alone situations where the computer consists of only 8K of memory and a processor, to large multi-user, multi-task applications requiring up to 124K of addressable memory space. Among its major features are a fast central processor with a choice of floating point and sophisticated memory management, both of which are hardware options.” https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf
  - macOS, iOS, Linux and Android are moderns variants and reimplementations of Unix

https://dave.cheney.net/2017/12/04/what-have-we-learned-from-the-pdp-11
Image credit: Dennis Ritchie

- This has proven surprisingly resilient and portable – but is it still the right model?

# Unix and C: Strengths

- Unix gained popularity due to portability and ease of source code access, but also:
  - Small, relatively consistent set of API calls
  - Low-level control
  - Robust and high performance
  - Easy to understand and extend
- Portability was due to the C programming language
  - Simple, easy to understand, easy to port to new architectures
  - Explicit pointers, memory allocation, and control over data representation
  - Uniform treatment of memory, device registers, and data structures – easy to write device drivers, network protocols, and interface with external formats
  - Weak type system allows aliasing and sharing

```
struct {
    short errors     : 4;
    short busy       : 1;
    short unit_sel   : 3
    short done       : 1;
    short irq_enable : 1
    short reserved   : 3
    short dev_func   : 2;
    short dev_enable : 1;
} ctrl_reg;

int enable_irq(void)
{
    ctrl_reg *r = 0x80000024;
    ctrl_reg  tmp;

    tmp = *r;
    if (tmp.busy == 0) {
        tmp.irq_enable = 1;
        *r = tmp;
        return 1;
    }
    return 0;
}
```

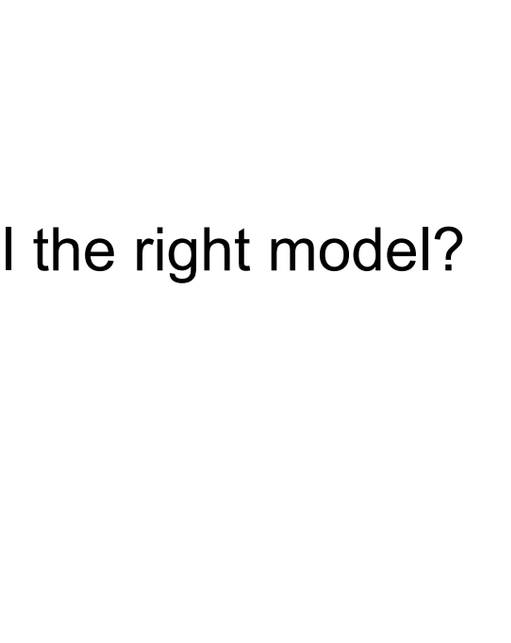
Example: hardware access in C

# Unix and C: Weaknesses

- Unix APIs reflect 1970s/1980s minicomputer architectures
  - Sockets and file system APIs significant performance bottlenecks
  - Security architecture insufficiently flexible
  - No portable APIs for mobility, power management, etc.
  - Assumes professional, interactive, systems administration
- C programming language
  - Limited concurrency support → memory model for pthreads poor supported
  - Undefined behaviour, buffer overflows → security risks
  - Weak type system → difficult to reason about correctness, effectively model problem domain

# Unix and C

- Unix has proven surprisingly resilient and portable – but is it still the right model?
  - No – numerous work-arounds for its limitations exist:
    - Kernel bypass networking
    - Increasingly baroque package management
    - Containers and sandboxing
- The C programming language is increasingly a liability
  - Too easy to introduce security vulnerabilities
  - Too easy to trip over undefined behaviour
  - Insufficient abstractions



**Some Were Meant for C**

The Endurance of an Unmanageable Language

Stephen Kell
Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
stephen.kell@cl.cam.ac.uk

**Abstract**

The C language leads a double life: as an application programming language of yesteryear, perpetuated by circumstance, and as a systems programming language which remains a weapon of choice decades after its creation. This essay is a C programmer's reaction to the call to abandon ship. It questions several properties commonly held to define the experience of using C; these include unsafety, undefined behaviour, and the motivation of performance. It argues all these are in fact inessential; rather, it traces C's ultimate strength to a *communicative* design which does not fit easily within the usual conception of "a programming language", but can be seen as a counterpoint to so-called "managed languages". This communicativity is what facilitates the essential aspect of system-building: creating parts which interact with other, remote parts—being "alongside" not "within".

***CCS Concepts*** • **Software and its engineering** → **General programming languages**; *Compilers*; • **Social and professional topics** → *History of programming languages*;

***Keywords*** systems programming, virtual machine, managed languages, safety, undefined behavior

**ACM Reference Format:**
Stephen Kell. 2017. Some Were Meant for C. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Vancouver, Canada, October 25-27, 2017 (Onward!'17),* 18 pages.
https://doi.org/10.1145/3133850.3133867


*Onward!'17, October 25-27, 2017, Vancouver, Canada*

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133867

**1 Introduction**

While some were meant for sea, in tug-boats
'Round the shore's knee,
(Milling with the sand,
and always coming back to land),
For others, up above
Is all they care to think of,
Up there with the birds and clouds, and
Words don't follow.

—Tiny Ruins, from "Priest with Balloons"

I am not ashamed to say that I program in C, and that I enjoy it. This puts me at odds with much of programming language discourse, among both researchers and influential practitioners, which holds that C is evil and must be destroyed. If only we had a "safe systems programming language"! If only we could eke out a little more performance in implementations of other languages, to remove the last remaining motivation for using C! If only we could make "the industry" see the error of its ways! Then C would be eradicated, and there would be much rejoicing.

I am a "systems programmer". It doesn't mean I hack kernels, so much as that I build systems—pieces of infrastructure that integrate multiple interacting parts, and sit underneath application code. Programming in C feels *right* for doing this; it has a viscerally distinctive feeling compared to other, safer, higher-level languages. Certainly, today's experience of programming in C remains, despite certain advances, unforgiving. But I have never felt C to be an encumbrance. C is not a language I use because I'm stuck with it; I use it for positive reasons. This essay explores those reasons and their apparent contrast with conventional wisdom.

**2 Two viewpoints**

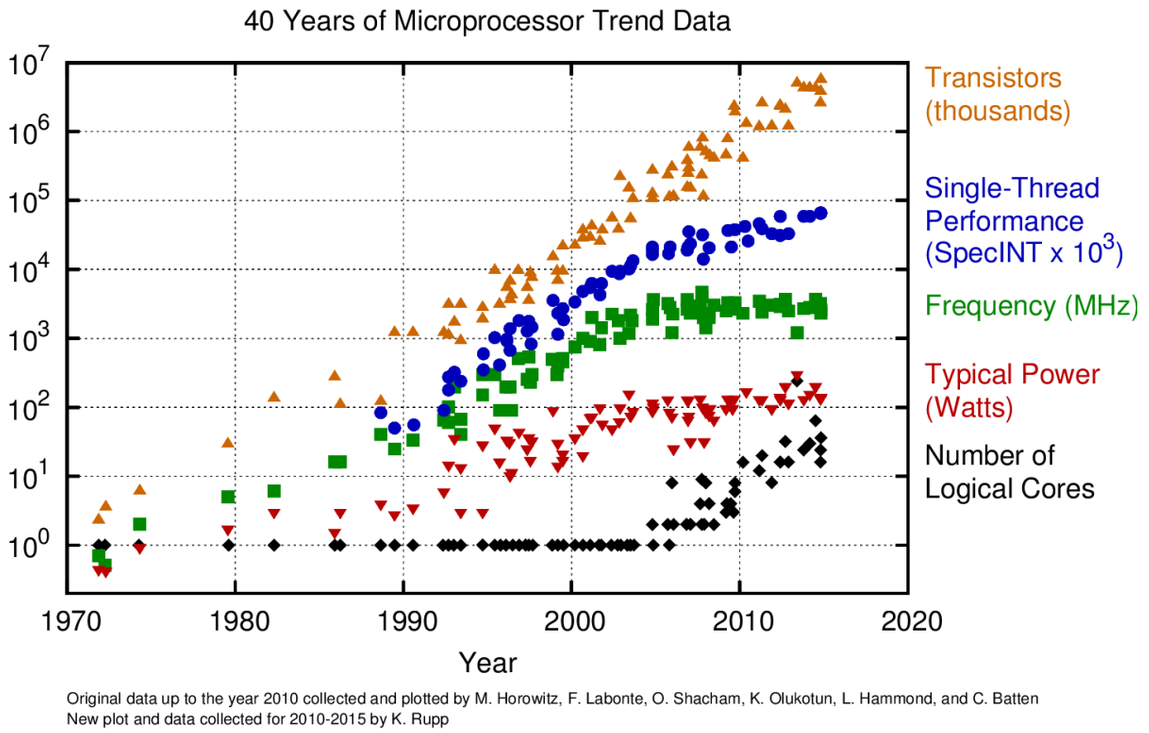
The lyric from which this essay borrows its title evokes two contrasting ways of being; that of the idealist who longs to be among the clouds, and that of the sea-farers who carry on their business on the planet's all-too-limiting surface. The idealist in the song is a priest, who takes literally to the clouds: one day, clutching at helium balloons, he steps off a cliff edge, floats up and away, and

S. Kell, "Some were meant for C: The endurance of an unmanageable language", International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Vancouver, BC, Canada, October 2017. ACM. DOI:10.1145/3133850.3133867

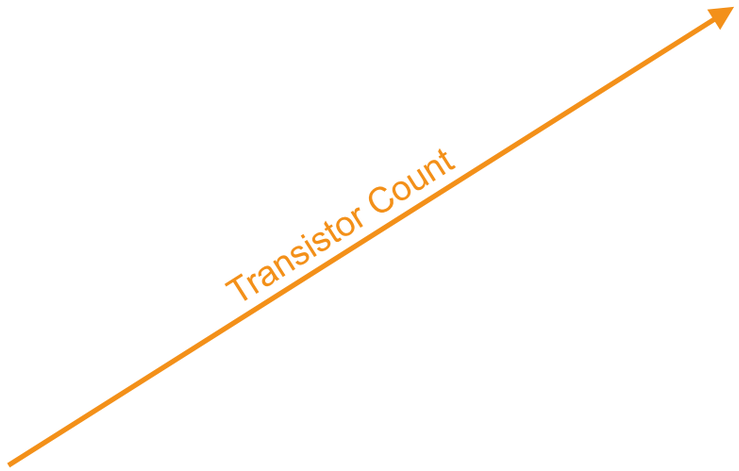
# Challenges and Limitations

- What changes in the environment are affecting systems programs?
  - The end of Moore's law
  - Increasing concurrency – imposed due to hardware changes
  - Increasing need for security – the Internet
  - Increasing mobility and connectivity

# The End of Moore's Law (1/2): Physical Limits

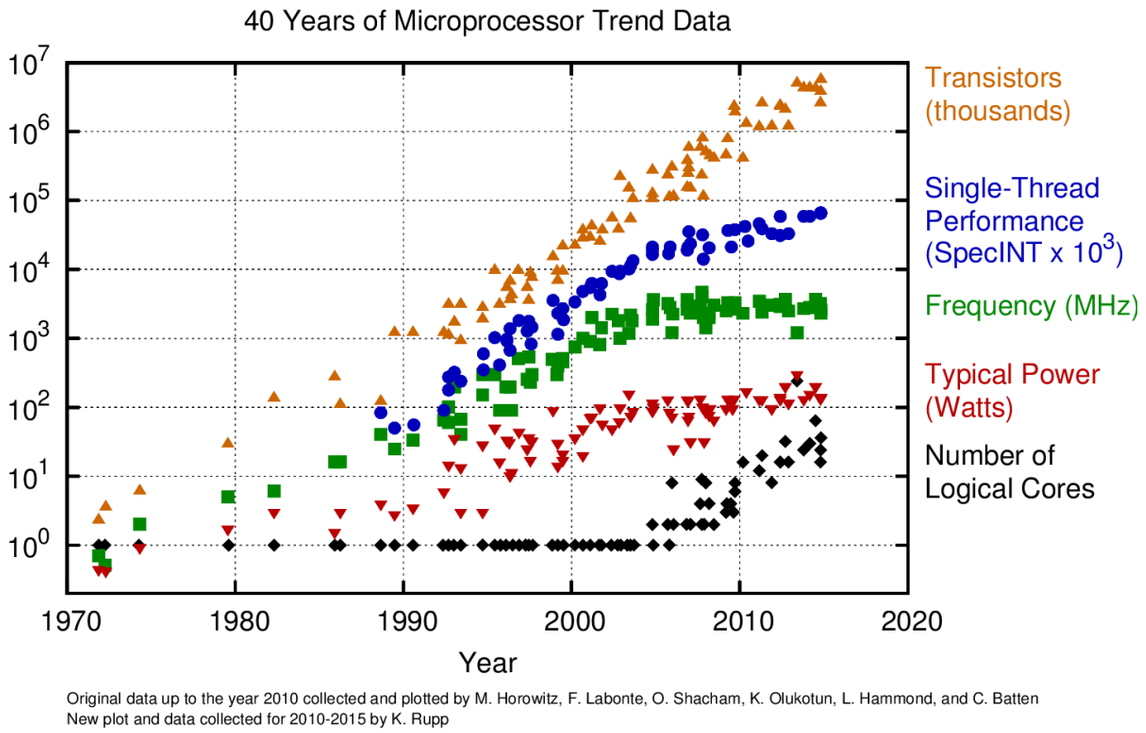



Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

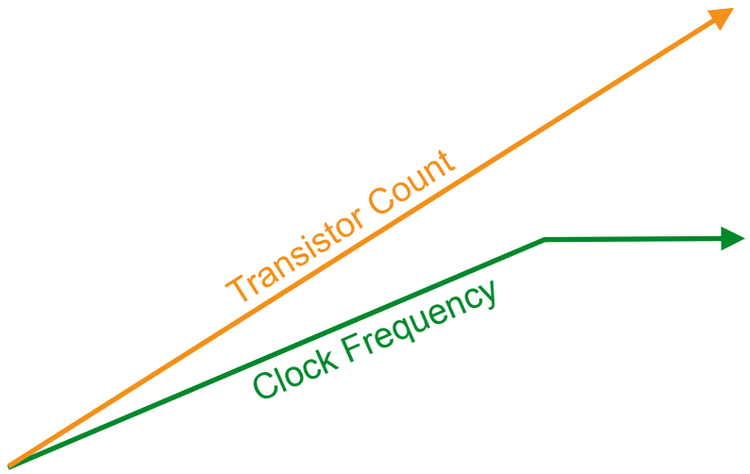



- Moore's law: advances in manufacturing double transistor count every two years
- But, rapidly approaching physical limits:
  - 10nm process → features ~40 atoms across
  - Transistors *will* stop shrinking soon

# The End of Moore's Law (2/2): Dennard Scaling



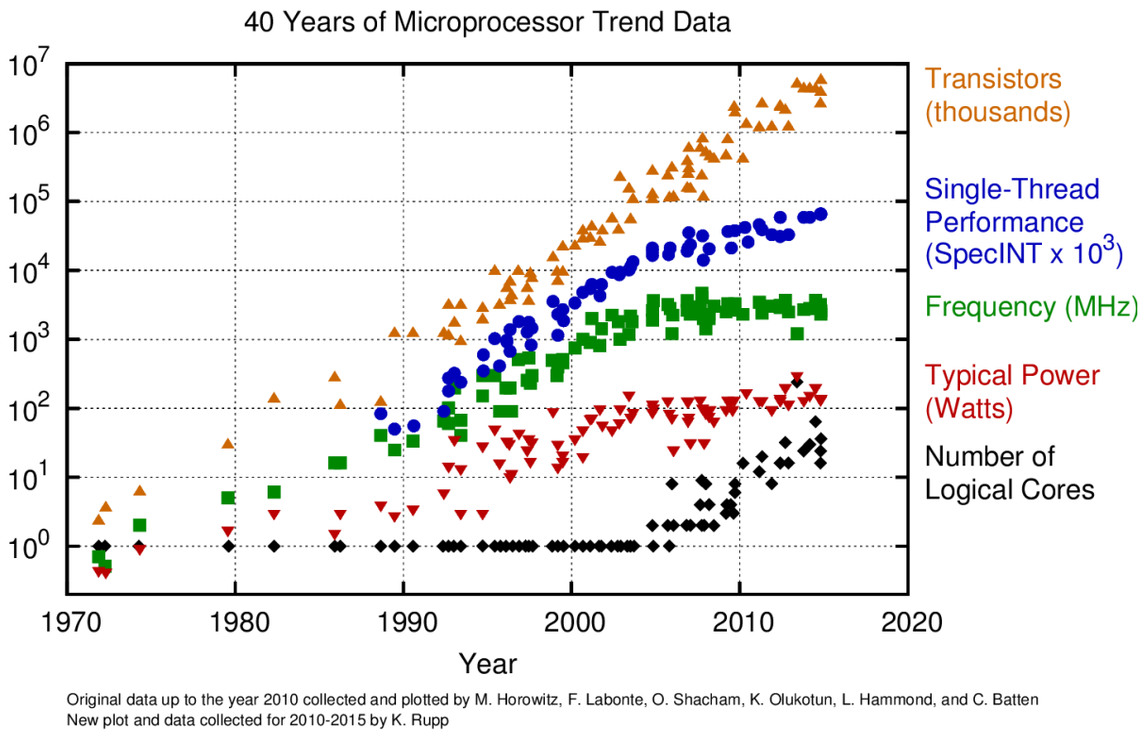

Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/



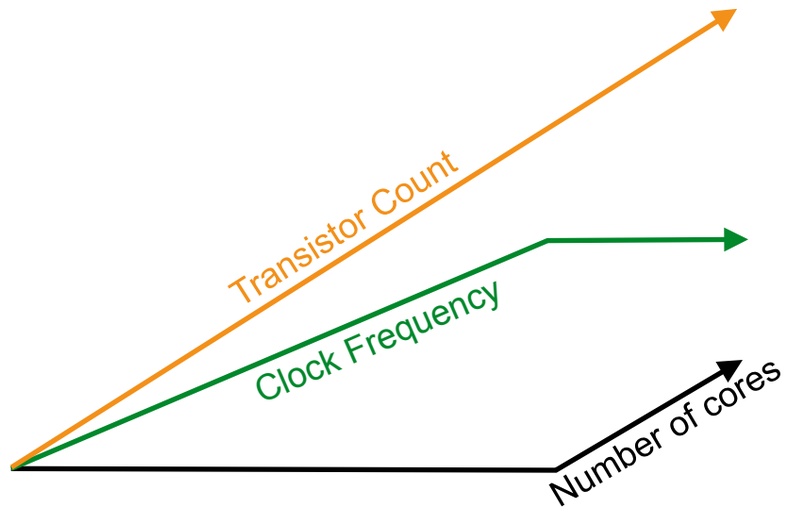

- Dennard scaling: smaller transistors reduce capacitance and voltage → frequency increase without corresponding increase in power consumption
- Scaling relation breaks down eventually, due to leakage → clock frequency increase stalls
- System performance constraints become more acute

Power consumption $\propto C \cdot F \cdot V^2$
- C = capacitance (transistor size)
- F = frequency (clock rate)
- V = voltage

# Increasing Concurrency

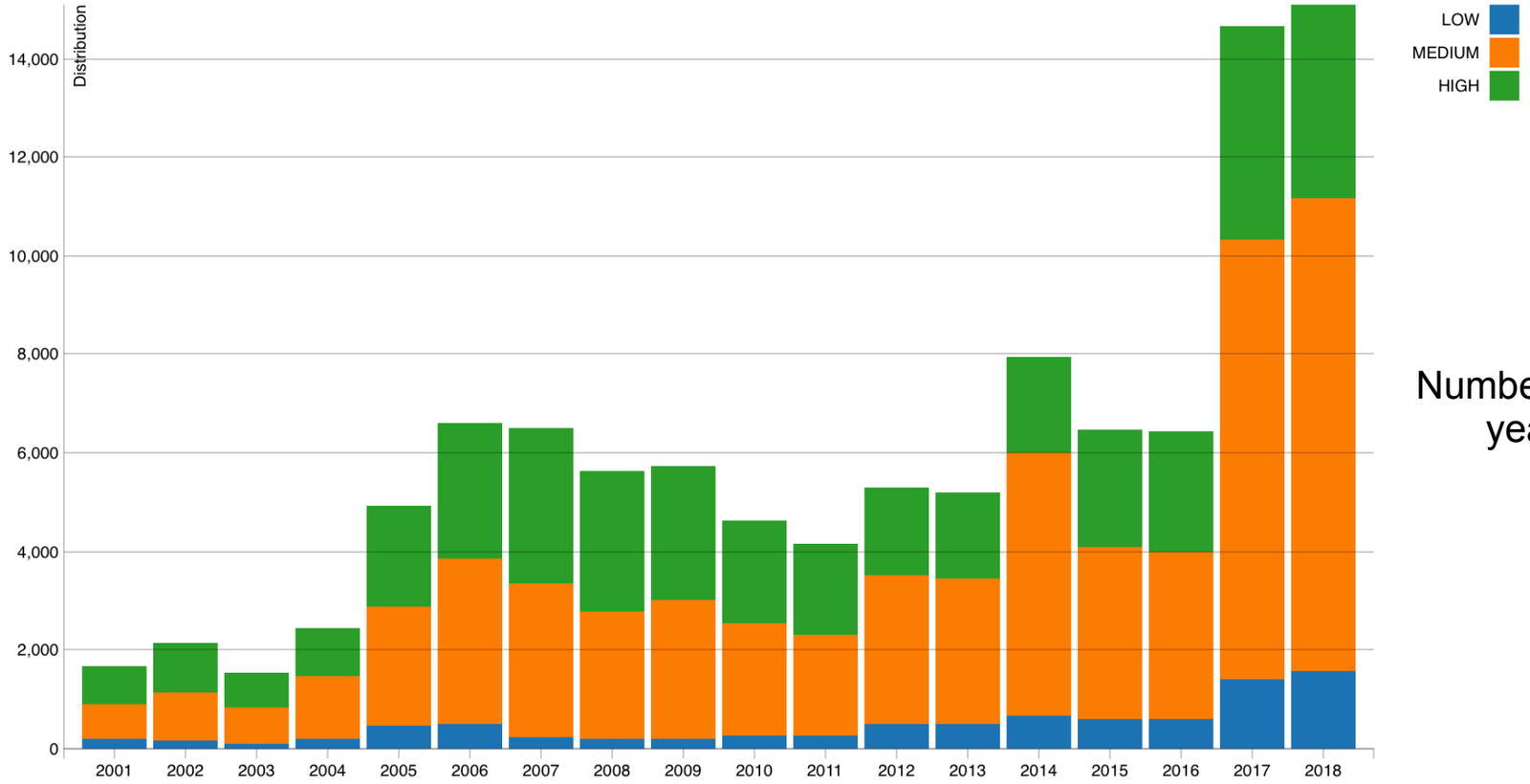



Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/




- Breakdown of Dennard scaling limits clock frequency, but Moore's law gives more transistors → used to increase number of cores
- Concurrency related problems become more severe:
  - Ensuring correctness, avoiding race conditions and deadlocks
  - Ensuring good performance

# Increasing Need for Security




Number of security vulnerabilities reported per year – US National Vulnerability Database

Source: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

- Weaponisation of the Internet
- The combination of C and Unix has not proven easy to secure

# Increasing Mobility and Connectivity

- Computing increasing implies mobile devices
  - Always on – but constrained by limitations of battery power
  - Always connected – by increasingly heterogenous networks
  - Personal – but vulnerable
- Do we have the APIs, tools, and programming models to make effective use of such devices?

# Next Steps in Systems Programming

# Next Steps in Systems Programming

- Need a programming language and environment that:
  - Improves memory management and safety – while maintaining control over allocation and data representation
  - Improves security – eliminates common classes of vulnerability
  - Improves support for concurrency – eliminates race conditions
  - Improves correctness – eliminates common classes of bug

- Advances in programming language design are starting to provide the necessary tools – and beginning to be applied to systems languages
  - Modern type systems
  - Functional programming techniques

# What is a Modern Type System?

- A modern type system is expressive enough to:
  - Provide useful guarantees about program behaviour
    - Prevent buffer overflows, use-after-free bugs, race conditions, iterator invalidation, …
  - Provide a model of the problem that prevents inconsistencies in the solution, while avoiding run-time overheads
    - No cost abstractions – compile-time checking that has no run-time cost
    - Describe constraints on program behaviour in the types – the compiler as a debugger

```
ACCEPT(2)                 BSD System Calls Manual                 ACCEPT(2)

NAME
     accept -- accept a connection on a socket

SYNOPSIS
     #include <sys/socket.h>

     int
     accept(int socket, struct sockaddr *restrict address,
         socklen_t *restrict address_len);

DESCRIPTION
     The argument socket is a socket that has been created with socket(2),
     bound to an address with bind(2), and is listening for connections after
     a listen(2).  accept() extracts the first connection request on the queue
     of pending connections, creates a new socket with the same properties of
     socket, and allocates a new file descriptor for the socket.  If no pend-
```

Common bug in networking code: call `read()` on listening socket, not socket returned from `accept()` that represents the connection

Both file descriptors are represented as type `int` – compiler can't check for such misuse

If *listening socket* and *connected socket* were separate types, and the `read()` call took a connected socket parameter, this bug would be detected at compile-time

Trivial example of important principle – try to describe behaviour in types so compiler can detect logic errors

# What is Functional Programming?

- A programming style that highlights:
  - Pure, referentially transparent, functions
  - No side effects
  - No shared mutable state
  - Control over I/O

  with language support for functions as first class types
- Pure functional languages constrain programs
  - Haskell is a testbed for exploring pure functional programming – what are its benefits and costs?
    - Principled, but perhaps impractical for large-scale systems programs
    - Not intended as an industrial-strength platform for deployment
    - The pure functional style is unsuited to some programs, but natural for others
- But concepts are widely applicable
  - Pure functional code is easy to test and debug – no hidden state
  - Pure functional code is thread safe – no side effects or mutable state
  - Eliminating shared mutable state and controlling I/O avoids race conditions
  - Use functional programming ideas where they make sense – prevent certain classes of bugs

# Improving Memory Management and Safety

- C has manual memory management
  - Pointers, `malloc()` and `free()`
  - Arrays represented as pointers to their first element, and don't store length
  - Access outside allocated memory "undefined" but no checks to prevent
  - Good reasons for this at the time:
    - Machines were slow and had limited memory
    - Bounds checks and garbage collection too expensive
    - Not all of these are still valid
- These design decisions lead bugs:
  - Use after free, memory leaks, buffer overflows
  - (C++ also suffers from) iterator invalidation
- Modern type systems can eliminate these *classes* of bug
  - Enforce bounds checks
  - Enforce ownership of data – code that tries to use data after it's been freed won't compile; similar for iterator invalidation

```
int foo[5];

// The following are equivalent:
foo[3] = 42;
*(foo+3) = 42;
*(3+foo) = 42;
3[foo] = 42;
```

# Improving Security

**Vulnerabilities By Type**

| Year | # of Vulnerabilities | DoS | Code Execution | Overflow | Memory Corruption | Sql Injection | XSS | Directory Traversal | Http Response Splitting | Bypass something | Gain Information | Gain Privileges | CSRF | File Inclusion | # of exploits |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1999 | 894 | 177 | 112 | 172 | | | 2 | 7 | | 25 | 16 | 103 | | | 2 |
| 2000 | 1020 | 257 | 208 | 206 | | 2 | 4 | 20 | | 48 | 19 | 139 | | | |
| 2001 | 1677 | 403 | 403 | 297 | | 7 | 34 | 123 | | 83 | 36 | 220 | | 2 | 2 |
| 2002 | 2156 | 498 | 553 | 435 | 2 | 41 | 200 | 103 | | 127 | 74 | 199 | 2 | 14 | 1 |
| 2003 | 1527 | 381 | 477 | 371 | 2 | 49 | 129 | | | | | | | | |
| 2004 | 2451 | 580 | 614 | 410 | 3 | 148 | 291 | | | | | | | | |
| 2005 | 4935 | 838 | 1627 | 657 | 21 | 604 | 786 | | | | | | | | |
| 2006 | 6610 | 893 | 2719 | 663 | 91 | 967 | 1302 | | | | | | | | |
| 2007 | 6520 | 1101 | 2601 | 954 | 95 | 706 | 884 | | | | | | | | |
| 2008 | 5632 | 894 | 2310 | 699 | 128 | 1101 | 807 | | | | | | | | |
| 2009 | 5736 | 1035 | 2185 | 700 | 188 | 963 | 851 | | | | | | | | |
| 2010 | 4652 | 1102 | 1714 | 680 | 342 | 520 | 605 | 275 | 8 | 234 | 282 | 238 | 86 | 73 | 1493 |
| 2011 | 4155 | 1221 | 1334 | 770 | 351 | 294 | 467 | 108 | 7 | 197 | 409 | 206 | 58 | 17 | 557 |
| 2012 | 5297 | 1425 | 1459 | 843 | 423 | 243 | 758 | 122 | 13 | 343 | 389 | 250 | 166 | 14 | 624 |
| 2013 | 5191 | 1455 | 1186 | 859 | 366 | 156 | 650 | 110 | 7 | 352 | 511 | 274 | 123 | 1 | 205 |
| 2014 | 7946 | 1598 | 1574 | 850 | 420 | 305 | 1105 | 204 | 12 | 457 | 2104 | 239 | 264 | 2 | 401 |
| 2015 | 6484 | 1791 | 1826 | 1079 | 749 | 218 | 778 | 150 | 12 | 577 | 748 | 367 | 248 | 5 | 127 |
| 2016 | 6447 | 2028 | 1494 | 1325 | 717 | 94 | 497 | 99 | 15 | 444 | 843 | 600 | 87 | 7 | 1 |
| 2017 | 14714 | 3154 | 3004 | 2805 | 745 | 503 | 1516 | 274 | 11 | 629 | 1706 | 459 | 327 | 18 | 6 |
| 2018 | 16555 | 1852 | 3035 | 2451 | 400 | 516 | 2001 | 509 | 11 | 709 | 1374 | 247 | 461 | 31 | 4 |
| 2019 | 4 | 2 | | | | 1 | | | | | | | | | |
| Total | 110603 | 22685 | 30435 | 17226 | 5043 | 7438 | 13667 | 3823 | 162 | 5880 | 10104 | 4877 | 2123 | 2195 | 4333 |
| % Of All | | 20.5 | 27.5 | 15.6 | 4.6 | 6.7 | 12.4 | 3.5 | 0.1 | 5.3 | 9.1 | 4.4 | 1.9 | 2.0 | |

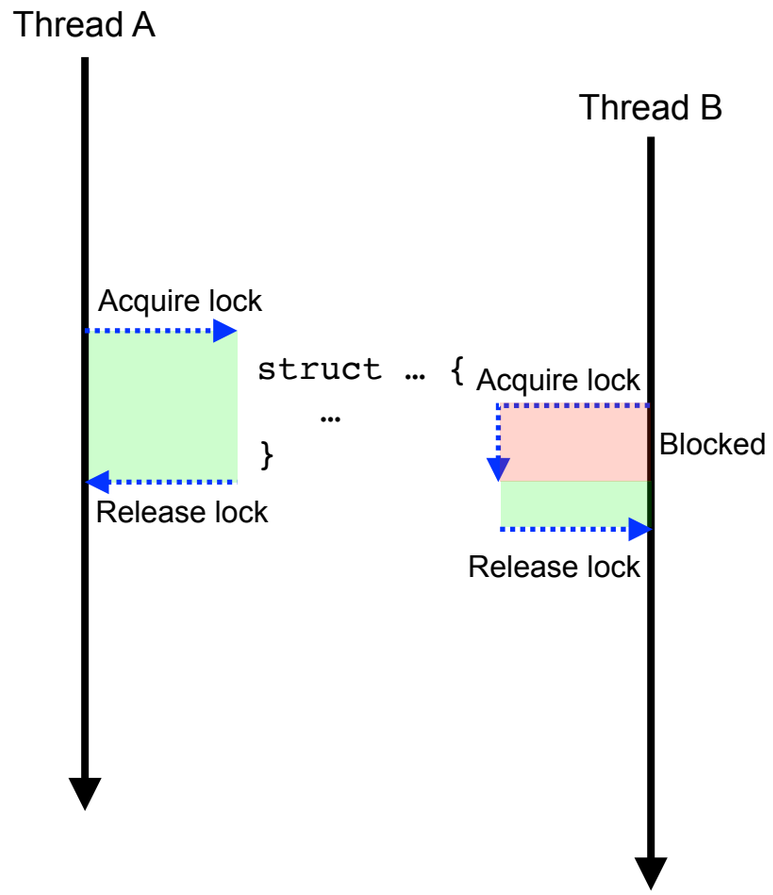
Source: https://www.cvedetails.com/vulnerabilities-by-types.php

Half of all reported security vulnerabilities are memory safety violations that should be caught by a modern type system –
buffer overflows, use-after-free, memory corruption, treating data as executable code

Use of type based modelling of the problem domain can help address others – by more rigorous checking of assumptions

# Improving Support for Concurrency

- Software increasingly concurrent to make effective use of multicore hardware
- Common abstraction: threads, locks, shared mutable state
- Prone to race conditions:
  - Too many or too few locks held
  - Locks held at the wrong time
  - Locks don't compose




- Program in a function style: avoid races by avoiding shared mutable state
- Avoid races by enforcing single ownership of data – message passing, rather than sharing

# Improving Correctness

- Modern systems programming languages can eliminate certain classes of bug that are common in C
  - Use-after-free, memory leaks, buffer overflows, iterator invalidation
  - Data races in multi-threaded code
  - Don’t fix the bug – eliminate the class of bugs
- Modern type systems allow for better modelling of the problem domain, and so more checking of code for consistency
  - Define types representing the problem domain, rather than using generic types – e.g., if you pass around `PersonId` and `VehicleId` rather than `int`, the compiler will warn if you pass the wrong type of identifier to a function
  - Represent program states in the types – e.g., `ListeningSocket` *vs.* `ConnectedSocket`
  - Modern languages allow you to define types and abstractions easily and without run-time cost – type-first design allows code that is correct by construction
- Use the compiler to debug your design

# Next Steps in Systems Programming

- People can't manage the complexity – need better tooling to help
  - C and Unix solve many systems programming problems
  - Control over data representation, memory management, sharing of state
  - Emerging languages and systems give the same degree of control – with added safety
- Strongly typed languages
  - Types help model the problem domain, structure code
  - Types and associated tooling help detect logic errors early – correct by construction
- This course will explore these ideas using the Rust programming language

# Summary

- What is systems programming?
- The state of the art
- Challenges and limitations
- Next steps in systems programming