University of Glasgow

Review and Future Directions

Advanced Systems Programming (M)

Lecture 9

Review

- Systems Programming
- Memory Management
- Concurrency
- Security

Systems Programming

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

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.

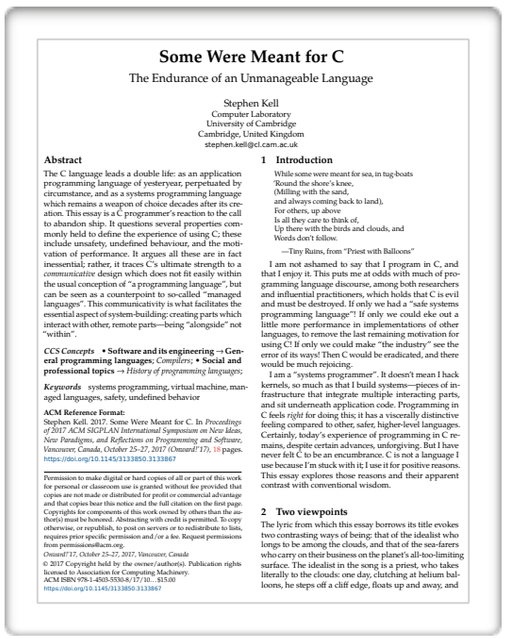
[2] By "transparent," I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

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

State of the Art: Unix and C

- 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 due to the C programming language
  - Simple, easy to understand, easy to port
- Explicit pointers, memory allocation, and control over data representation; uniform treatment of memory, devices and data structures
- Weak type system allows aliasing and sharing

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



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.

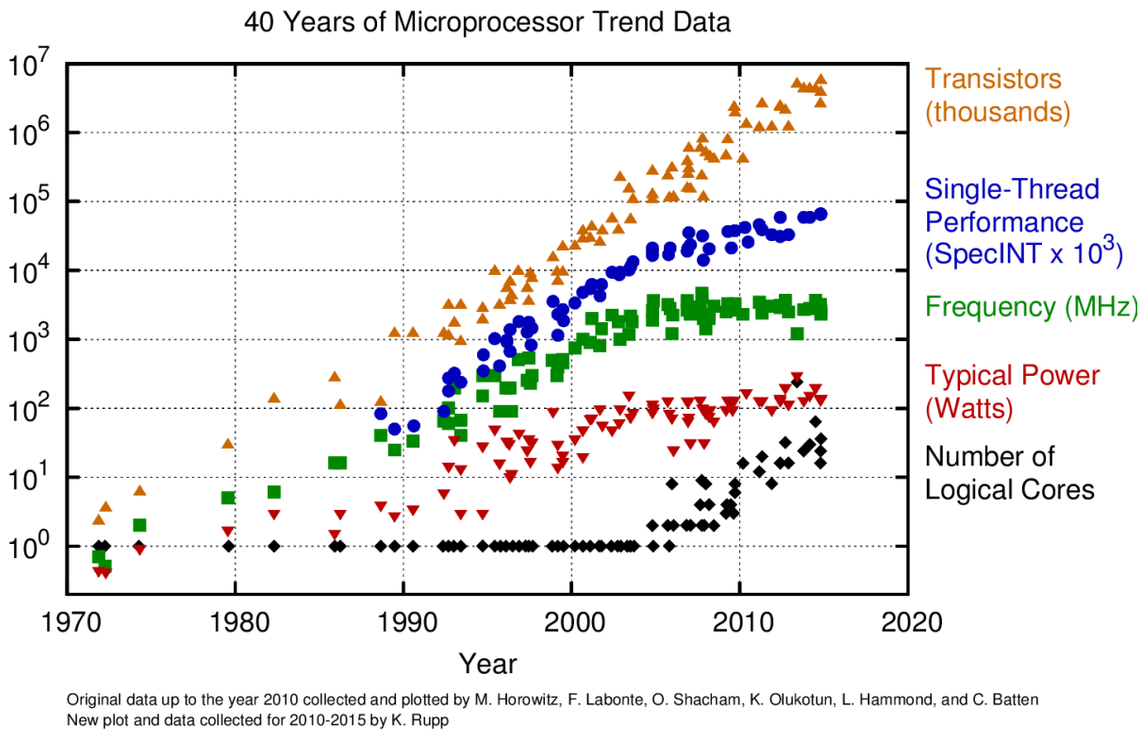
I am a "systems programmer". It doesn't mean I hack kernels, so much as that I build systems—pieces of infrastructure that integrate multiple interacting parts, and sit underneath application code. Programming in C feels *right* for doing this; it has a viscerally distinctive feeling compared to other, 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

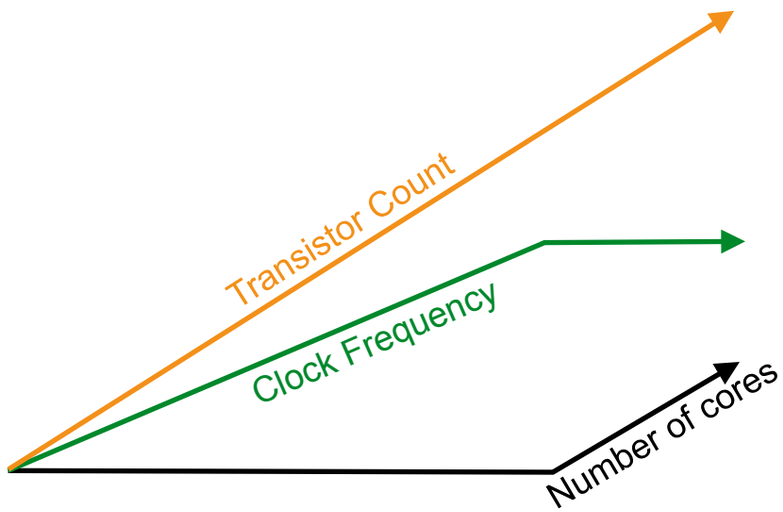
The lyric from which this essay borrows its title evokes two contrasting ways of being; that of the idealist who longs to be among the clouds, and that of the sea-farers who carry on their business on the planet's all-too-limiting surface. The idealist in the song is a priest, who takes literally to 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

No More Moore?



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



- Breakdown of Dennard scaling, impending end of Moore's law → concurrency
- Ubiquitous networks and cyber-physical systems pushing security to the fore
- Existing approaches to systems programming no longer sufficient

Do you agree with this claim?

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

Why is Strong Typing Desirable?

- “Well-typed programs don’t go wrong” – Robin Milner
  - The result is well-defined – although not necessarily correct
    - The type system ensures results are consistent with the rules of the language, but cannot check if you calculated the right result
    - A strongly-typed system will only ever perform operations on a type that are legal – there is no undefined behaviour
  - Types help model the problem, check for consistency, and eliminate common classes of bug

Segmentation fault (core dumped)

Segmentation faults should never happen:

- Compiler and runtime should strongly enforce type rules
- If program violates them, it should be terminated cleanly
- Security vulnerabilities – e.g., buffer overflow attacks – come from undefined behaviour after type violations

Challenges in Strongly-typed Systems Programming

- Four fallacies:
  - Factors of 1.5x to 2x in performance don't matter
  - Boxed representation can be optimised away
  - The optimiser can fix it
  - The legacy problem is insurmountable
- Four challenges:
  - Application constraint checking
  - Idiomatic manual storage management
  - Control over data representation
  - Managing shared state

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

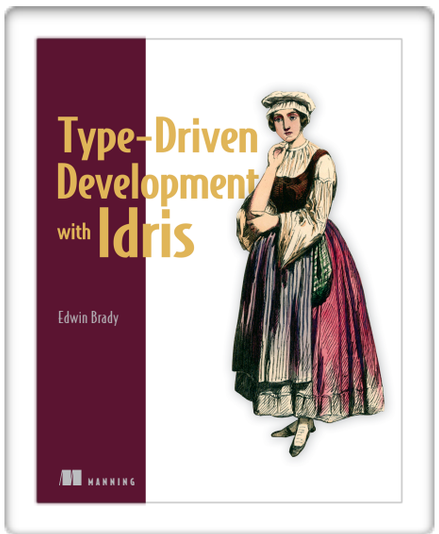
- Many good ideas in research languages and operating systems – only recently that these issues have been considered to make *practical* tools

The Rust Programming Language

- A practical, strongly typed, systems programming language
  - Enumerated types and pattern matching: **`Option`** and **`Result`**
  - Structure types and traits as alternative to object oriented design
  - Ownership, borrowing, and multiple reference types
  - Little is novel – adopts ideas from numerous other languages:
    - Syntax → C, Standard ML, and Pascal
    - Basic data types → C and C++
    - Enumerated types and pattern matching → Standard ML
    - Traits → Haskell type classes
    - The ownership and borrowing rules → Cyclone
- Strongly typed and permits no-cost abstractions

Type-driven Development

- **Define the types** first
  - Define concrete numeric types, identifiers
  - Define `enum` types to represent alternatives
  - Indicate optional values, results, error types
- Using the types as a guide, **write the functions**
  - Write the input and output types
  - Write the function, using the structure of the types as a guide
  - Make state machines explicit
  - Consider ownership of data
- **Refine** and edit types and functions as necessary
  - Use the compiler as a tool to help you debug your design
- Don't think of the types as checking the code, think of them as a plan, a model, for the solution – and as machine checkable documentation



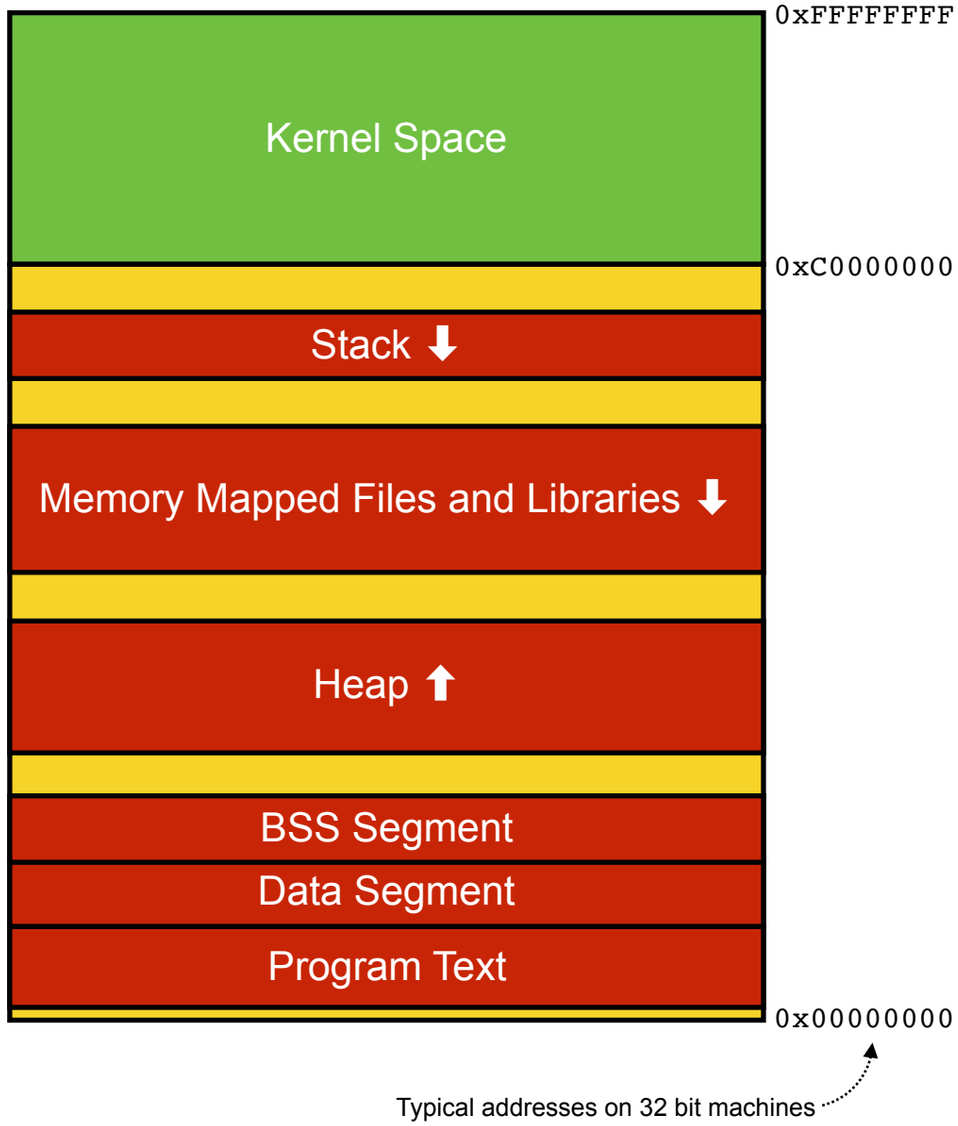
Type-drive development approach adapted from: E. Brady, "Type-Driven Development with Idris", Manning, March 2017.

Is this a good approach to systems development?

Memory Management

Layout of a Processes in Memory

- Layout of process address space:
  - Kernel at top of address space
  - Program text, data, and global variables at bottom of virtual address space
  - Heap allocated upwards, above BSS
  - Stack grows downwards, below kernel
  - Memory mapped files and shared libraries between these



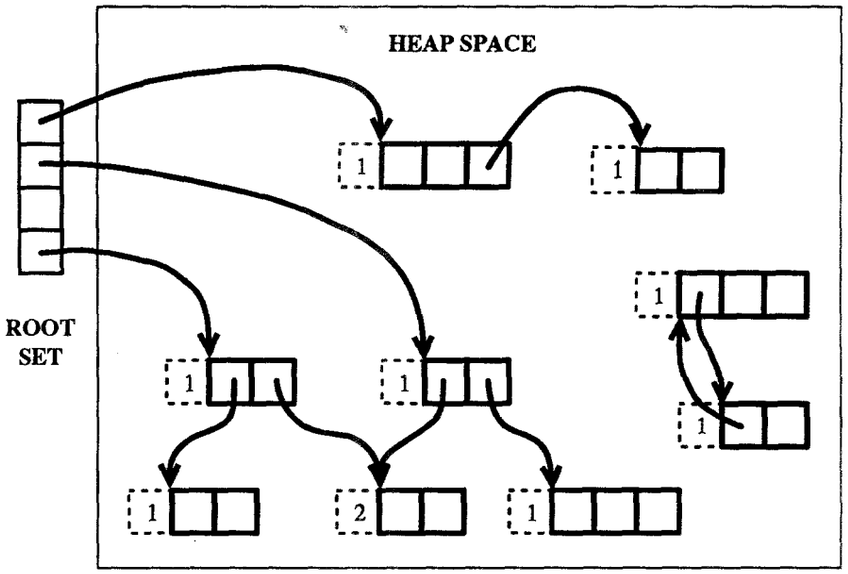
See also http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Automatic Memory Management

- Automatic memory management distrusted by systems programmers
  - Perceived high processor and memory overheads, unpredictable timing
  - But, memory management problems are common:
    - Unpredictable performance
      - Calls to **`malloc()`**/**`free()`** can vary in execution time by several orders of magnitude
    - Memory leaks
    - Memory corruption and buffer overflows
    - Use-after-free
    - Iterator invalidation
- New automatic memory management schemes solve many problems
  - Garbage collectors → lower overhead, more predictable
    - Also system performance improvements made overhead more acceptable
  - Region-based memory management → predictability, compile time guarantees

Reference Counting

- Simplest automatic heap management
- Each allocation also allocates space for an additional **reference count**
  - An extra **`int`** is allocated along with every object
  - Counts number of references to the object
    - Increased when new reference to the object is created
    - Decremented when a reference is removed
    - When reference count reaches zero, there are no references to the object, and it may be reclaimed
      - Reclaiming object removes references to other objects
      - May reduce their reference count to zero, so triggering further reclamation
- Incremental, predictable, and understandable
- Potentially high cost; memory leaks with cyclic data structures



Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI:10.1007/BFb0017182

Region-based Memory Management

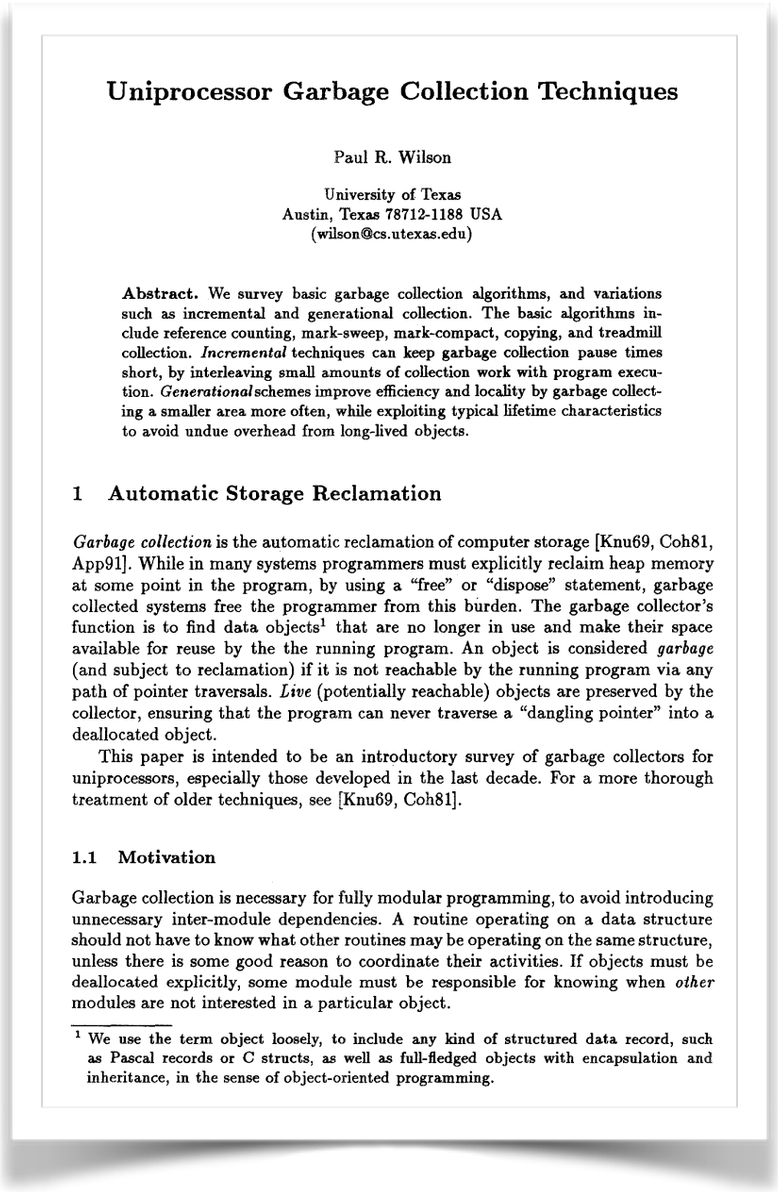
- Allocate objects with lifetimes corresponding to regions matching program scope
- Track object ownership, and changes of ownership:
  - What region owns each object at any time
  - Ownership of objects can move between regions
- Deallocate objects at the end of the lifetime of their owning region
  - Use scoping rules to ensure objects are not referenced after deallocation

- Efficient and predictable; correctness guarantees prevent common bugs
- Constrains the type of programs that can be written

- Forces programmer to consider resource ownership early in the design – but eliminates run-time misbehaviours and costs

Garbage Collection

- Garbage collection algorithms:
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational algorithms
- Incremental approaches
- Real-time collection



Uniprocessor Garbage Collection Techniques

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

P. R. Wilson, "Uniprocessor garbage collection techniques", Proceedings of the International Workshop on Memory Management, St. Malo, France, September 1992. DOI: 10.1007/BFb0017182

Memory Management Trade-offs

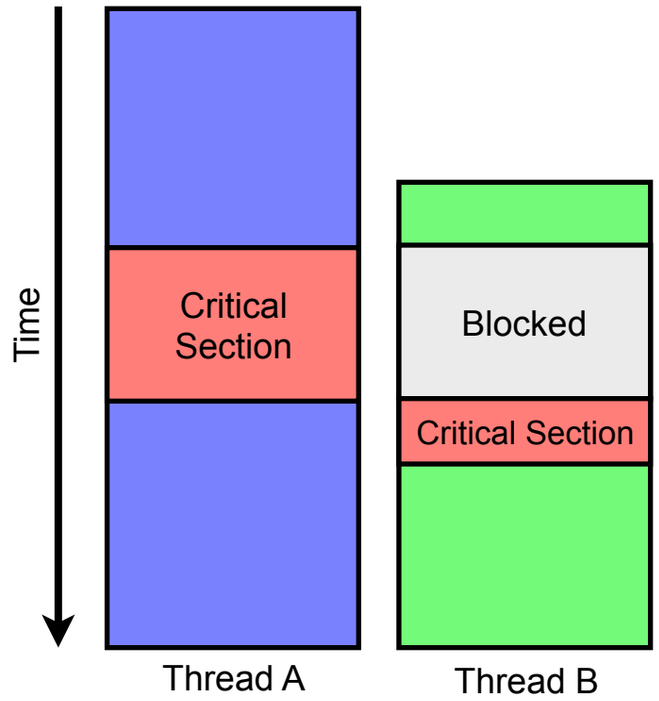
- Rust pushes memory management complexity onto the programmer
  - Predictable run-time performance, low run-time overheads
  - Uniform resource management framework, including memory
  - Limits the programs that may be expressed – matches common patterns in good C code
- Garbage collection imposes run-time costs and complexity, but simpler for the programmer

Does Rust make the right trade-off here?

Concurrency

Concurrency, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
- The processor/language memory models specify how concurrent access to shared memory works
  - e.g., synchronise by explicitly locking critical sections
    - **`synchronized`** methods and statements in Java
    - **`pthread_mutex_lock()`**/**`pthread_mutex_unlock()`**
  - Limited guarantees about unlocked concurrent access to shared memory

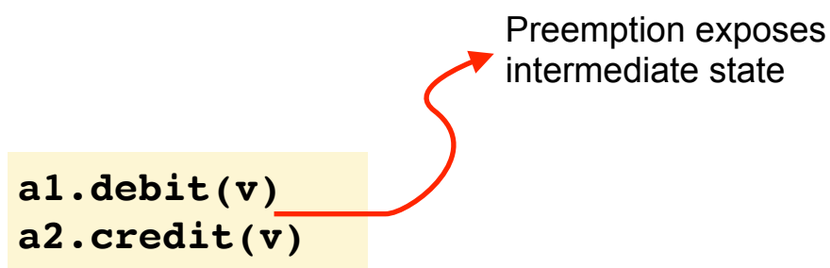


Limitations of Lock-based Concurrency

- Major problems with lock-based concurrency:
  - Difficult to define a memory model that enables good performance, while allowing programmers to reason about the code
  - Difficult to ensure correctness when composing code
    - Difficult to enforce correct locking
    - Difficult to guarantee freedom from deadlocks
    - Failures are silent – errors tend to manifest only under heavy load
    - Balancing performance and correctness difficult – easy to over- or under-lock systems

Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from **`a1`** and move it to **`a2`**, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to **`a1`** *and* **`a2`** while the transfer is in progress
  - The individual operations are correct, but the combined operation is not



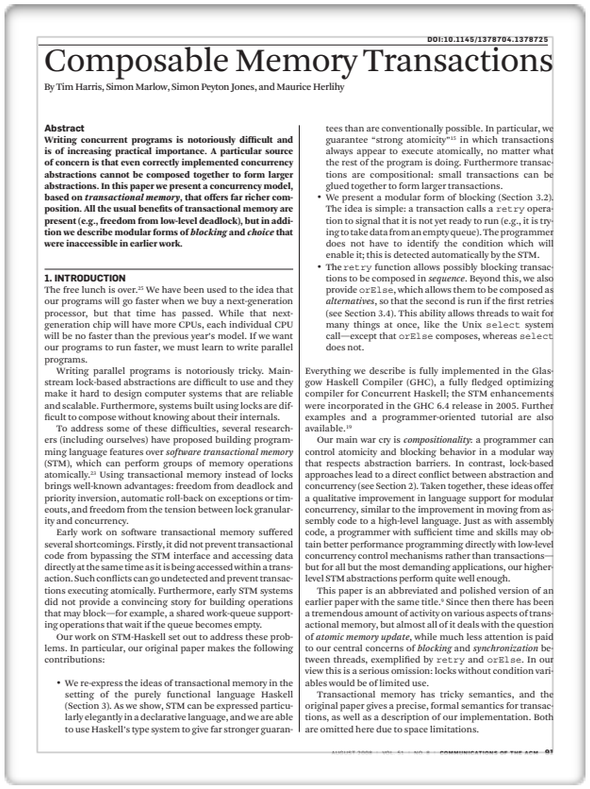
- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object

Alternative Concurrency Models

- Concurrency increasingly important
  - Multicore systems now ubiquitous
  - Asynchronous interactions between software and hardware devices
- Threads and synchronisation primitives problematic
- Are there alternatives that avoid these issues?
  - Transactions
  - Message passing

Transactions

- Are transactions a reasonable programming model?
- Is transactional memory a realistic technique?
  - Assumption: shared memory system, doesn't work with distributed and networked systems – is this true?
- Concurrent Haskell:
  - Monadic IO; do notation; IORefs; spawning threads
  - Type system separates state and stateless computation
  - The STM interface
- Do its requirements for a purely functional language, with controlled I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?



DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract

Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION

The free lunch is over.[22] We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.[23] Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee "strong atomicity"[11] in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a `retry` operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The `retry` function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide `orElse`, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call—except that `orElse` composes, whereas `select` does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.[19]

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.[3] Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by `retry` and `orElse`. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions", Communications of the ACM, 51(8), August 2008. DOI:10.1145/1378704.1378725.

http://www.cmi.ac.in/~madhavan/courses/pl2009/reading-material/harris-et-al-cacm-2008.pdf

Is this a realistic systems programming model?

Message Passing Systems

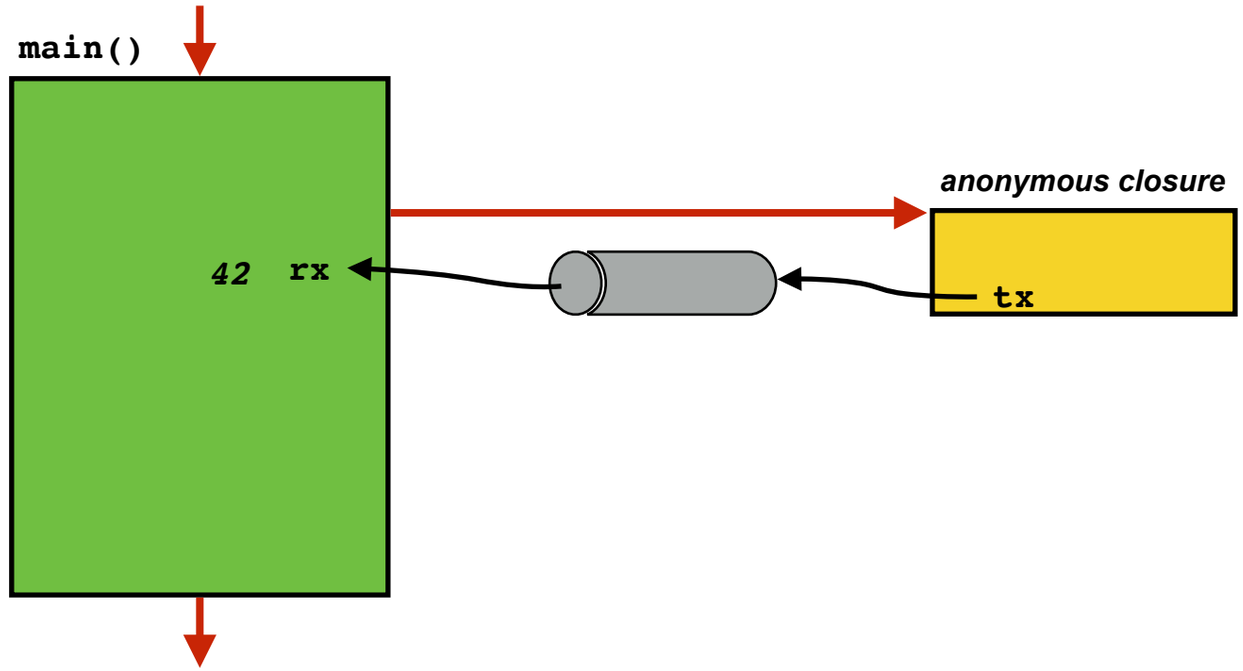
- System is a set of communicating processes – no shared mutable state
- Communication via exchange of messages
  - Immutable or linearly typed to ensure safety

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



Does this offer sufficient benefit over shared state concurrency?

Coroutines and Asynchronous Code

- Provide language and run-time support for I/O multiplexing on a single thread, in a more natural style

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += input.read(&mut buf[cursor..])?;
    }
}
```

```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
    let mut cursor = 0;
    while cursor < buf.len() {
        cursor += await!(input.read(&mut buf[cursor..]))?;
    }
}
```

- Runtime schedules **`async`** functions on a thread pool, yielding to other code on **`await!()`** calls → low-overhead concurrent I/O

Do the benefits outweigh the cooperative multitasking concern?

Security

Memory Safety in Programming Languages

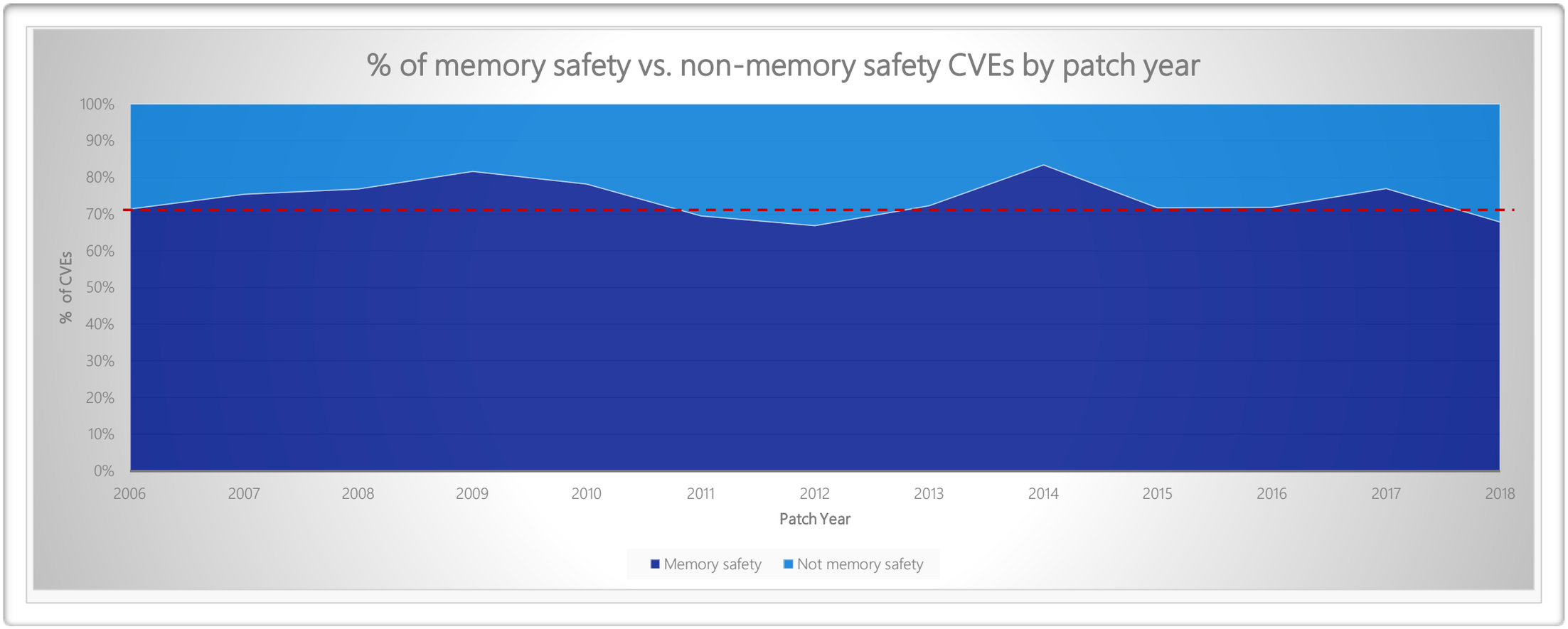
- A **memory safe** language is one that ensures that the only memory accessed is that owned by the program, and that such access is done in a manner consistent of the declared type of the data
  - The program can access memory through its global and local variables, or through explicit references to them
  - The program can access heap memory it allocated via an explicit reference to that memory
  - All accesses obey the type rules of the language
    - Array bounds are respected
    - Memory cannot be accessed after it's freed
    - References can only refer to objects of matching type
    - …

Memory Safe	Memory Unsafe
Java, Scala, C#, Rust, Go, Python, Ruby, Tcl, FORTRAN, COBOL, Modula-2, Occam, Erlang, Ada, Pascal, Haskell, …	C, C++, Objective-C

Impact of Memory Unsafe Languages

- Lack of memory safety breaks the machine abstraction
  - With luck, program crashes – segmentation violation
  - If unlucky, memory unsafe behaviour corrupts other data owned by program
    - Undefined behaviour occurs
    - Cannot predict without knowing precise layout of program in memory
    - Difficult to debug
    - **Potential security risk** – corrupt program state to force arbitrary code execution

Impact of Memory Unsafe Languages – Security (2/2)



Source: Matt Miller, Microsoft, presentation at BlueHat IL conference, February 2019
https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

- ~70% of Microsoft security updates fix bugs relating to unsafe memory usage
  - Interestingly, this has not significantly changed in >10 years
  - We're not getting better at writing secure code in memory unsafe languages

Mitigations for Memory Unsafe Languages (1/2)

- Use modern tooling for C and C++ development:
  - Compile C code with, at least, **`clang –W -Wall -Werror`**
  - Use **`clang`** static analysis tools during debugging:
- Use modern C++ language features:
  - I advise against C++ programming – language is too complex to understand
  - But if you have to use C++, modernise the code base, to use newer – safer – language features and idioms, where possible
- Consider rewriting the most critical sections of the code in a memory safe language:
  - Rust can call C functions directly, and can be compiled into a library that can be directly linked with C or C++
  - Objective-C and Swift can similarly be combined in the Apple ecosystem

Mitigations for Memory Unsafe Languages (2/2)

- Writing safe code in unsafe languages is difficult
  - Easy to rationalise each individual bug – “How could anyone write that?”

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

https://dwheeler.com/essays/apple-goto-fail.html

- But, people will continue to make mistakes
- Memory safe languages eliminate common classes of vulnerability; strong type systems help detect mistakes early

Parsing and LangSec

- Can we combine better parsing with modern, strongly typed, languages to improve network security?
- Is performance good enough?
- Can we improve the way we design protocols?

SECURITY

The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate Professor of computer science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural language processing research before coming to Dartmouth.
sergey@cs.dartmouth.edu

Meredith L. Patterson is the founder of Upstanding Hackers. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium.
mlp@upstandinghackers.com

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society and maintains the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total.
ashubina@cs.dartmouth.edu

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory's classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers' design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

Parsers, Parsers Everywhere

To quote Koprowski and Binsztok [1]:

> Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers [2] (and even the hardware: turning PHY layer symbol streams into frames is parsing, too!). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.

S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. ;login:, 40(4):4–10, August 2015. http://langsec.org/papers/the-bugs-we-have-to-kill.pdf

Causes of Security Vulnerabilities

- Security vulnerabilities generally caused by persuading a program to do something that the programmer did not expect
  - Write past the end of a buffer
  - Treat user input as executable
  - Confuse a permission check
  - Violate an assumption in the code
- Strong typing makes assumptions explicit
  - Use explicit types rather than generic types
  - Define safe conversion functions

No Silver Bullet

Memory safety and strong typing won’t eliminate security vulnerabilities

But, used carefully, they eliminate certain classes of vulnerability, and make others less likely by making hidden assumptions – and their consequences – visible

Do you agree with this characterisation?

Future Directions

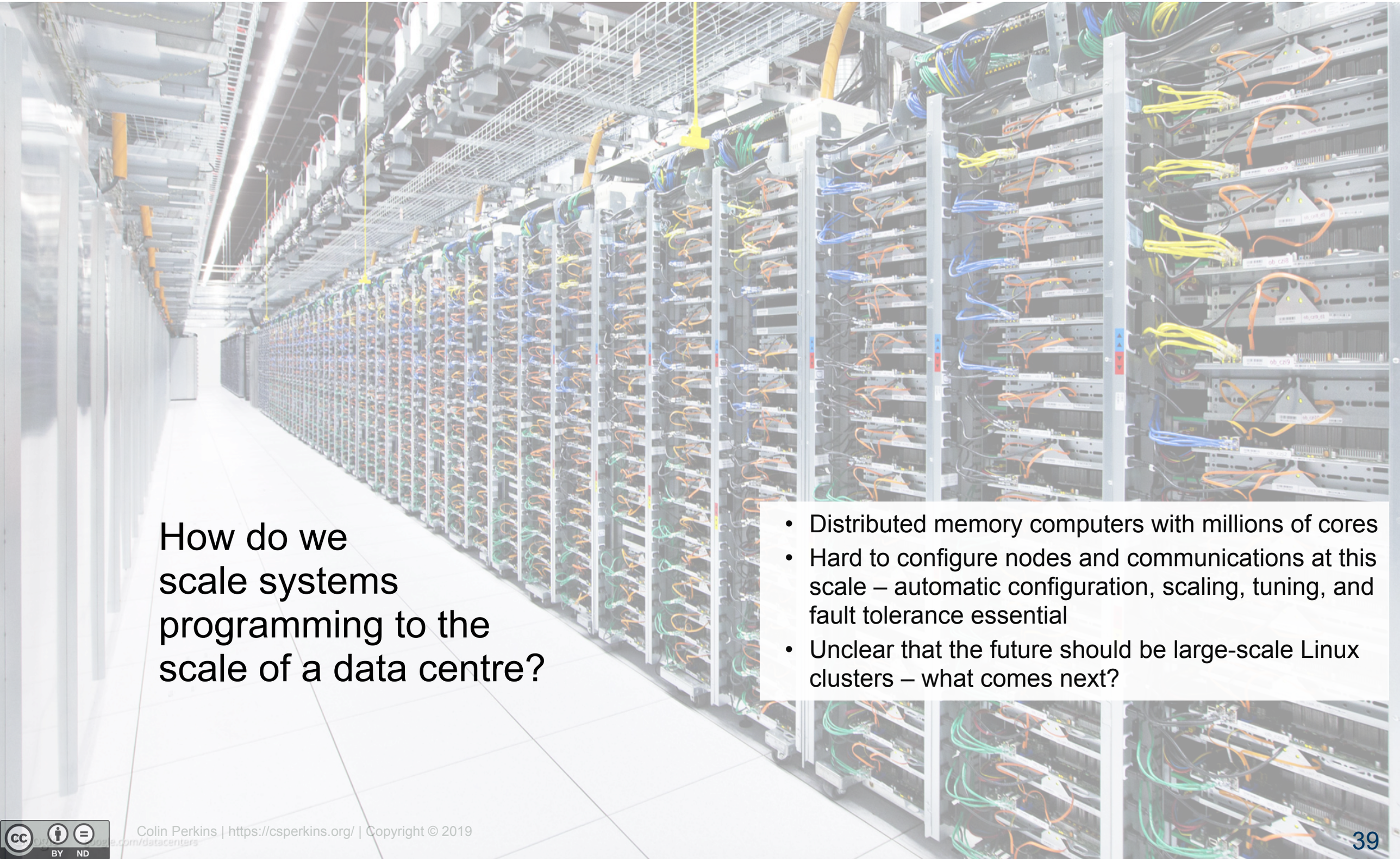
- Scaling systems programming

Concurrency

- Still no good solutions for concurrent programming
  - Transactional memory doesn't sit well with impure imperative languages
  - Message passing has issues with back pressure, race conditions, deadlocks, large-scale orchestration
  - Asynchronous code is a work-around for heavyweight threads

- None scale to GPU programming → implicit parallelism?

Data Centre Scale Computing

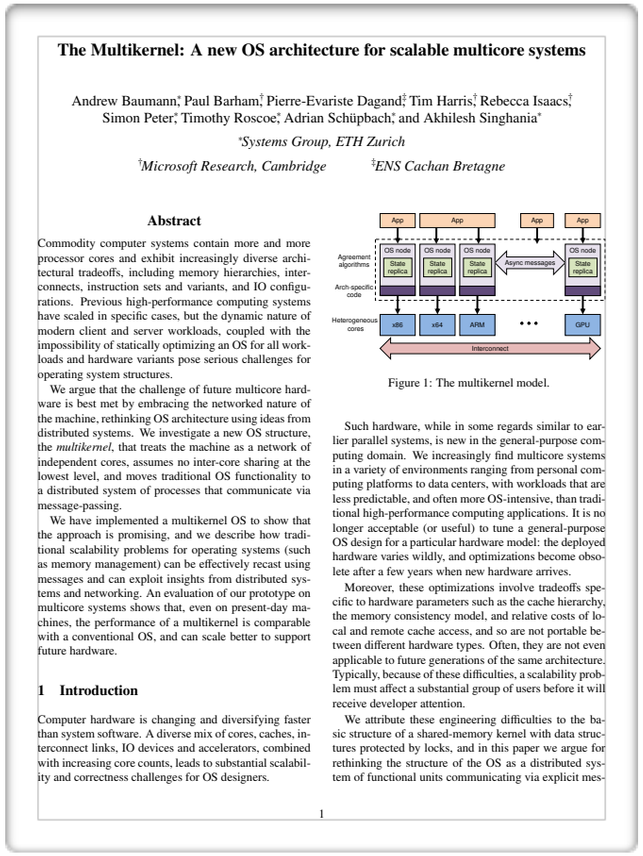


How do we scale systems programming to the scale of a data centre?

- Distributed memory computers with millions of cores
- Hard to configure nodes and communications at this scale – automatic configuration, scaling, tuning, and fault tolerance essential
- Unclear that the future should be large-scale Linux clusters – what comes next?

Distributed Operating Systems

- Barrelfish – a message passing kernel for multicore systems
  - Each core runs a separate operating systems instance
  - **All** communication between cores is via message passing; no “main” CPU to act as central orchestrator
- A research prototype to test an idea, not a product
  - Where is the boundary for a Barrelfish-like system?
  - Distinction between a distributed multi-kernel and a distributed system of networked computers? Should there be such a distinction?
  - How does it relate to concurrent programming, data centres, distributed edge computing, etc?



The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann*, Paul Barham†, Pierre-Evariste Dagand‡, Tim Harris†, Rebecca Isaacs†, Simon Peter*, Timothy Roscoe*, Adrian Schüpbach*, and Akhilesh Singhania*

*Systems Group, ETH Zurich

†Microsoft Research, Cambridge ‡ENS Cachan Bretagne

Abstract

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

Figure 1: The multikernel model.

1 Introduction

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit mes-

1

A. Baumann *et al.*, “The Multikernel: A new OS architecture for scalable multicore systems”, Proc. ACM Symposium on Operating Systems Principles, 2009. DOI:10.1145/1629575.1629579

Future Systems Programming Languages

- Increasingly seeing research ideas incorporated into mainstream programming languages
  - Rust, Swift – abstraction, resource management, strong and expressive type systems, low-level control
  - Erlang, Go – concurrency, fault tolerance, communication

- Choose the language based on problem domain, available tooling, and expertise
  - Good solutions for programming single systems
  - We still have a lot to learn about scaling to large clusters

Scaling Systems Programming

- Unix and C made sense when programming simple single-core systems
  - And form the basis of most existing systems – **but what comes next?**
- Modern massively concurrent and distributed applications need new approaches:
  - Low-level performance still crucial, but also need type systems and abstractions to hide the complexity – *no-one* can hold all the details in their head
  - Tooling is essential – to check invariants, document assumptions, ensure consistency
  - Deployment, configuration, and management must become increasingly autonomic – DevOps, infrastructure as code, self-managing

What are we missing?

Assessment

Assessment

- This is a level M course, worth 10 credits
- Coursework (20%)
  - Essay (10%) and programming exercise (10%)
  - Exercise 1 marks available from teaching office
  - Exercise 2 marks will be returned by 22 March 2019
- Examination (80%):
  - 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
  - Two hours duration; answer three out of four questions
  - Sample paper will be provided by 22 March 2019

The End

;login: logout

The Night Watch

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

As a highly trained academic researcher, I spend a lot of time trying to advance the frontiers of human knowledge. However, as someone who was born in the South, I secretly believe that true progress is a fantasy, and that I need to prepare for the end times, and for the chickens coming home to roost, and fast zombies, and slow zombies, and the polite zombies who say "sir" and "ma'am" but then try to eat your brain to acquire your skills. When the revolution comes, I need to be prepared; thus, in the quiet moments, when I'm not producing incredible scientific breakthroughs, I think about what I'll do when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY. The main thing that I ponder is who will be in my gang, because the likelihood of post-apocalyptic survival is directly related to the size and quality of your rag-tag group of associates. There are some obvious people who I'll need to recruit: a locksmith (to open doors); a demolitions expert (for when the locksmith has run out of ideas); and a person who can procure, train, and then throw snakes at my enemies (because, in a world without hope, snake throwing is a reasonable way to resolve disputes). All of these people will play a role in my ultimate success as a dystopian warlord philosopher. However, the most important person in my gang will be a systems programmer. A person who can debug a device driver or a distributed system is a person who can be trusted in a Hobbesian nightmare of breathtaking scope; a systems programmer has seen the terrors of the world and understood the intrinsic horror of existence. The systems programmer has written drivers for buggy devices whose firmware was implemented by a drunken child or a sober goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish country, where the problem was transmitted in the front left hoof of a mule named Deliverance. The systems programmer has read the kernel source, to better understand the deep ways of the universe, and the systems programmer has seen the comment in the scheduler that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLed, and the systems programmer has submitted a kernel patch to restore balance to The Force and fix the priority inversion that was causing MySQL to hang. A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law.

J. Mickens. The night watch. ;login: logout, pages 5–8, November 2013. https://www.usenix.org/system/files/1311_05-08_mickens.pdf