The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language

by

Cody Cutler

B.S., University of Utah (2012)
M.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2019

Certified by . . . .
Robert T. Morris
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language

by

Cody Cutler

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This dissertation presents an evaluation of the use of a high-level language (HLL) with garbage collection to implement a monolithic POSIX-style kernel. The goal is to explore if it is reasonable to use an HLL instead of C for such kernels, by examining performance costs, implementation challenges, and programmability and safety benefits.

This dissertation contributes Biscuit, a kernel written in Go that implements enough of POSIX (virtual memory, `mmap`, TCP/IP sockets, a logging file system, `poll`, etc.) to execute significant applications. Biscuit makes liberal use of Go's HLL features (closures, channels, maps, interfaces, garbage collected heap allocation), which subjectively made programming easier. The most challenging puzzle was handling the possibility of running out of kernel heap memory; Biscuit benefited from the analyzability of Go source to address this challenge.

On a set of kernel-intensive benchmarks (including NGINX and Redis) the fraction of kernel CPU time Biscuit spends on HLL features (primarily garbage collection and thread stack expansion checks) ranges up to 13%. The longest single GC-related pause suffered by NGINX was 115 microseconds; the longest observed sum of GC delays to a complete NGINX client request was 582 microseconds. In experiments comparing nearly identical system call, page fault, and context switch code paths written in Go and C, the Go version was 5% to 15% slower.

Thesis Supervisor: Robert T. Morris
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Acknowledgments

I want to thank many people for contributing, in one way or another, to the completion of this dissertation. First and foremost, thank you Robert and Frans, for being my mentors and friends these past seven years, during which you taught me how to think, inspired me with your standards, and were constant catalysts for self-improvement.

Thank you Suzanne, my wife, partner, and best friend, for your inexhaustible support and dedication to us.

Thank you Eric Eide, Robert Ricci, Grant Ayers, and the Flux research group, for providing such a special environment and unique opportunity for me while I was an undergraduate. My life is greatly improved in many important ways thanks to my interaction with you during those years.

Thank you to my parents for starting me down this road by buying a copy of *Running MS-DOS QBasic* for a curious kid and, of course, for everything else.

Finally, I thank the following individuals:

Austin T. Clements • Adam Belay • Julian Straub • Malte Schwarzkopf • Eddie Kohler
Nickolai Zeldovich • Jonathan Perry • Eugene Wu • Albert Kim • Neha Narula
David Lazar • Srivatsa Bhat • Xi Wang • Yandong Mao • Taesoo Kim • Haogang Chen
Ramesh Chandra • Keith Winstein • Emily Stark • Rasha Eqbal • Tej Chajed
Jon Gjengset • Jonathan Behrens • Atalay Mert İleri • Akshay Narayan
Amy E. Ousterhout • Mike Hibler • Anton Burtsev • Ryan Jackson • Tarun Prabhu
Jonathan Duerig • Matt Strum • Gary Wong • David Johnson • Matthew Flatt
Joe Zachary • Bob Kessler • Erik Brunvand • Christopher and Kathryn Porter
Irwin Mark and Joan Klein Jacobs • Lisa Zaelit • Phillip and Tricia Collins
Adam Zabriskie • Andrew Brown • Jordan Crook • Chris Edwards • Linda Mellor
Ryan Thompson • Matt Shelley • Mike Holly • Patrick Hagen • Stephen Fiskell
James Thayne • Chad Ostler • Nina • Ralph Cutler • Marty Reimschissel

* * *

This dissertation extends work previously published in the following paper:

> Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, California, October 2018.

Contents

1 Introduction **13**

1.1 C, for better or worse . . . 13
1.2 An opportunity . . . 14
1.3 Biscuit . . . 15
1.4 Results . . . 16
1.5 Contributions . . . 17
1.6 Outline . . . 18

2 Related work **19**

2.1 Kernels in high-level languages . . . 19
2.2 High-level systems programming languages . . . 19
2.3 Memory allocation . . . 20
2.4 Kernel heap exhaustion . . . 21
2.5 Computing memory allocation bounds . . . 21

3 Overview **23**

3.1 Boot and Go runtime . . . 23
3.2 Processes and kernel goroutines . . . 23
3.3 Interrupts . . . 24
3.4 Multi-core and synchronization . . . 25
3.5 Virtual memory . . . 25
3.6 File system . . . 25
3.7 Network stack . . . 26
3.8 Limitations . . . 26

4 Garbage collection **27**

4.1 Go's collector . . . 27
4.2 Garbage collection cost model . . . 28
4.3 Biscuit's heap size . . . 29

5 Avoiding heap exhaustion **31**
5.1 Approach: reservations . . . 31
5.2 How Biscuit reserves . . . 32
5.3 Static analysis to find s . . . 33
5.3.1 Not just any s will do . . . 34
5.3.2 Reasoning about maximum simultaneously live data . . . 35
5.3.3 Approximating liveness via the call graph . . . 36
5.3.4 MAXLIVE operation . . . 38
5.3.5 MAXLIVE example . . . 38
5.3.6 MAXLIVE correctness . . . 40
5.3.7 MAXLIVE implementation . . . 41
5.3.8 Special allocations . . . 41
5.3.9 Handling loops . . . 41
5.3.10 Kernel threads . . . 42
5.3.11 Killer thread . . . 42
5.4 Limitations . . . 42
5.5 Heap exhaustion summary . . . 43

6 Implementation **45**

7 Evaluation **49**
7.1 HLL benefits . . . 49
7.1.1 Biscuit's use of HLL features . . . 49
7.1.2 Potential to reduce bugs . . . 51
7.2 HLL performance costs . . . 52
7.2.1 Experimental Setup . . . 52
7.2.2 HLL tax . . . 54
7.2.3 GC delays . . . 55
7.2.4 Sensitivity to heap size . . . 55
7.2.5 Go versus C . . . 56
7.2.6 Biscuit versus Linux . . . 57
7.2.7 Handling kernel heap exhaustion . . . 58
7.2.8 Lock-free lookups . . . 60
7.2.9 Scalability . . . 61

8 Discussion and future work **65**
8.1 HLL kernel challenges . . . 65
8.1.1 GC CPU overhead . . . 66
8.1.2 GC pauses . . . 67

8.2 HLL kernel benefits 67
8.2.1 Increased productivity 67
8.2.2 Memory-safety 67
8.2.3 Simpler concurrency 68
8.2.4 Simpler lock-free sharing 68
8.3 Future work 69

9 Conclusions **71**

Figures and tables

1-1 Biscuit's high-level design 16

3-1 Biscuit's overall structure 24

5-1 Pseudo code for heap reservations in Biscuit 33
5-2 Example call graph with allocations 35
5-3 Example call graph and state of MAXLIVE analysis 37
5-4 Pseudo code for MAXLIVE 39

6-1 Approximate lines of code in Biscuit 46
6-2 Biscuit's 58 system calls 46

7-1 Use of Go HLL features in Biscuit 50
7-2 Linux kernel CVEs which Biscuit's GC would prevent 51
7-3 Measured costs of HLL features in Biscuit 54
7-4 CMailbench throughput on Biscuit with different kernel heap sizes 55
7-5 Application throughput of Biscuit and Linux 58
7-6 The amount of live heap data during execution of an abusive program . . 59
7-7 CMailbench performance with and without directory cache read-locks . . 60
7-8 The performance of CMailbench with increasing core counts on Biscuit . . 61
7-9 The performance of CMailbench with increasing core counts on Linux . . 62
7-10 The throughput of Pstat with increasing core counts 63
7-11 The fraction of all CPU cycles spent on GC during the Pstat experiment . . 64

8-1 A simple case where threads share data 68

1 Introduction

This dissertation presents an evaluation of the use of a garbage-collected programming language to build a monolithic POSIX operating system kernel by examining performance costs, implementation challenges, and programmability and safety benefits.

The rest of this chapter explains the main reasons why widely-used kernels are written in C, explains why the question of whether to use a garbage-collected language instead of C for kernel implementation is worth investigating, and summarizes this dissertation's contributions.

1.1 C, for better or worse

Efficient use of hardware resources is one of the main goals of widely-used operating system kernels. Applications depend on the kernel to implement many fundamental operations; if the kernel's implementation of one of those operations is slow (for example, reading bytes from a file), all user programs using that operation will inherit that inefficiency, potentially reducing performance below acceptable bounds.

The goal of good kernel performance is one of the main reasons why widely-used operating system kernels are written in the C programming language. C gives the programmer a high degree of control over the generated program, providing opportunities to optimize for performance. For example, in C the programmer is responsible for memory allocation and deallocation. This allows the programmer to allocate an object on the stack instead of the heap in some situations, improving performance. Furthermore, C has few implicit, hidden costs: it is mostly obvious which instructions the compiler will emit to implement a C program. This close correspondence between source and executable can make finding performance problems more straightforward.

Another main goal of operating systems is to be robust. Bugs in the kernel are particularly damaging since a kernel bug may crash any number of user applications, corrupt user data, or provide a security hole for an unauthorized user to gain access to system resources.

Unfortunately it is challenging to build robust software in C, even for experts. Uninitialized variables and out-of-bounds and use-after-free accesses are some of the well-known sources of bugs that C programmers struggle with [15, 52, 65]. The result is system failures, security vulnerabilities, and programmer effort spent debugging and fixing software.

Memory-safety bugs (bugs where the kernel reads or writes the memory of an object other than the one intended by the programmer) in particular are a serious problem for C kernels and frequently result in security vulnerabilities. At least 40 Linux kernel security vulnerabilities were reported in 2017 that enable an attacker to execute malicious code in the kernel [47]; all 40 security vulnerabilities were caused by memory-safety bugs. Furthermore, seven of the 22 publicly-disclosed Linux kernel security vulnerabilities found by Google's Project Zero team from January 2018 to January 2019 are caused by memory-safety bugs. Kernel memory-safety bugs are enough of a problem that the Linux kernel includes a memory checker which detects some use-after-free and out-of-bounds accesses at runtime [60]. Nevertheless, Linux developers routinely discover and fix use-after-free bugs. For example, Linux has at least 36 commits from January to April of 2018 for the specific purpose of fixing use-after-free bugs.

1.2 An opportunity

Garbage-collected programming languages (hereafter referred to as "HLLs" or "high-level languages") automatically manage memory allocation and deallocation without any cooperation from the programmer. Together with runtime bounds checks, HLLs eliminate memory-safety bugs from software, preventing a class of common security vulnerabilities. None of the previously mentioned Linux kernel memory-safety bugs would have been able to execute malicious code had the kernel been written in a memory-safe language.

HLLs also have the potential to increase programmer productivity. Convenient language features like automatic allocation and deallocation, abstraction, type-safety, native complex data types, multi-value returns, closures, and language support for threads and synchronization, reduce the cognitive burden on programmers. Programmers are able to focus more on the important problems and less on minor details and debugging. Although it is difficult to meaningfully measure an HLL's effect on programmer productivity, this benefit is likely significant.

However, HLLs come with a price. Important mechanisms, like memory allocation, are hidden from the programmer in the runtime and the enforced abstraction and safety may forbid certain kinds of designs which are nevertheless safe and preferred by the programmer. Garbage collection (GC) can have a significant performance cost, reducing the amount of CPU cycles available to the kernel and user applications. This CPU overhead

is a result of the GC's periodic scanning of all allocated heap objects to identify heap memory that is safe to free. Other language features also cost CPU cycles, like bounds checks, nil pointer checks, and reflection. In addition to the CPU overhead, GC can cause seemingly random pauses during execution. An unluckily-timed GC pause during a latency-sensitive task, like redrawing a mouse pointer or processing an urgent client request, could degrade the quality of service. Furthermore, GC requires more RAM than does the manual allocation typically used in a kernel: in order to achieve competitive performance, the GC must have enough free heap RAM to keep the frequency of GCs low enough. All things considered, the net effect of an HLL on kernel performance is complex. If the performance cost is large enough, the downsides of using an HLL to build a kernel may outweigh the benefits.

Although there are programming languages which provide memory-safety without the use and thus without the downsides of garbage collection (e.g., Rust [49]), this dissertation focuses on the use of a language with garbage collection because of the additional convenience that it provides: garbage collection provides memory-safety without requiring any cooperation from the programmer. Furthermore, we wanted to explore whether garbage collection simplifies concurrent programming and the implementation of lock-free data structures, which are common in optimized operating system kernels [45].

The choice of whether to use an HLL to build an operating system kernel is therefore a trade-off with programmer productivity and system robustness on one hand and performance loss on the other. Since the potential benefits of using HLLs to build operating system kernels could be significant, it is important for kernel developers to understand this trade-off.

1.3 Biscuit

The goal of this dissertation is therefore to help kernel developers better understand the trade-off of using an HLL to build operating system kernels that can run existing applications with good performance. This dissertation does so by providing a detailed evaluation of the performance costs, implementation challenges, and programmability and safety benefits of the use of an HLL to build a monolithic POSIX operating system kernel. This work is, to our knowledge, the first comprehensive performance evaluation of an HLL kernel that is complete enough to run widely-used, kernel-intensive POSIX applications with good performance.

We built Biscuit, a new kernel in Go [27] for x86-64 hardware. Go is a type-safe language with garbage collection. Biscuit's design, shown in Figure 1-1, is similar to that of traditional C kernels to make comparison easier. Biscuit runs significant existing

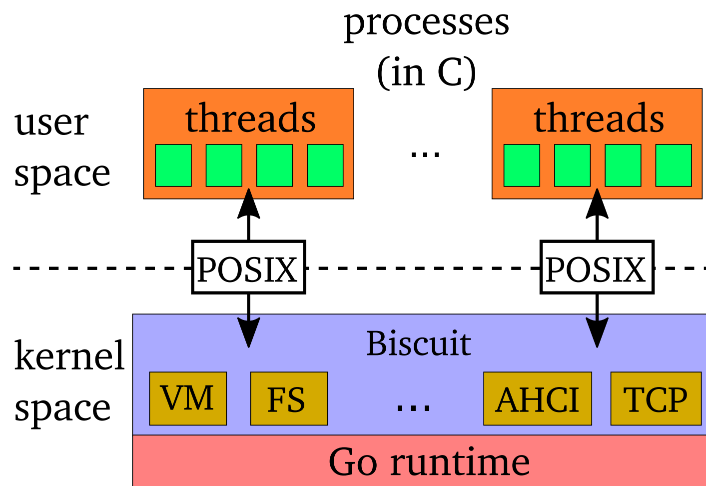

Figure 1-1: Biscuit's high-level design.

applications such as NGINX [36] and Redis [58] without source modification by exposing a POSIX-subset system call interface. Supported features include multi-core, kernel-supported user threads, futexes, IPC, mmap, copy-on-write fork, vnode and name caches, a logging file system, and TCP/IP sockets. Biscuit implements two significant DMA device drivers in Go: one for AHCI SATA disk controllers and one for Intel 82599-based Ethernet controllers. Biscuit has nearly 30 thousand lines of Go, about 1500 lines of assembler, and no C. We report lessons learned about use of Go in Biscuit, including ways in which the language helped development, and situations in which it was less helpful.

A challenge we faced while building Biscuit was handling kernel heap exhaustion. Because Go, like many HLLs, implicitly allocates and does not expose failed allocations, Biscuit can't use the traditional method of manually checking and handling failed allocations. Instead, we use static analysis of the Biscuit source to determine how much heap memory each system call (and other kernel activity) might need, and each system call waits (if needed) when it starts until it can reserve the heap memory it needs. Once a system call is allowed to continue, all allocations made are guaranteed to succeed. This obviates the need for complex allocation failure recovery or deadlock-prone waiting for free memory in the allocator. The use of an HLL that is conducive to static analysis made this approach possible.

1.4 Results

We ran several kernel-intensive applications on Biscuit and measured the effects of Go's type safety and garbage collection on kernel performance. For our benchmarks, GC costs up to 3% of CPU. For NGINX, the longest single GC-related pause was 115

microseconds, and the longest total delay of a single NGINX client request (by many individual pauses) was 582 microseconds. Other identifiable HLL performance costs amount to about 10% of CPU.

To shed light on the specific question of C versus Go performance in the kernel, we modified Biscuit and a C kernel to have nearly identical source-level code paths for two benchmarks that stress system calls, page faults, and context switches. The C versions are about 5% and 15% faster than the Go versions.

Finally, we compared the performance of Biscuit and Linux on our kernel-intensive application benchmarks, and found that Linux is up to 10% faster than Biscuit. This result is not solely determined by choice of language, since performance is also affected by differences in the features, design and implementation of Biscuit and Linux. However, the results do provide an idea of whether the absolute performance of Biscuit is in the same league as that of a C kernel.

1.5 Contributions

The main contributions of this dissertation are:

- Biscuit, a kernel written in Go with good performance.
- A novel scheme for coping with kernel heap exhaustion which is deadlock-free and requires little recovery code, unlike the traditional method, at the cost of more frequently delayed system calls when free memory is low.
- A discussion of qualitative ways in which use of an HLL for kernel development was and was not helpful.
- Measurements of the performance tax imposed by use of an HLL when running widely-used, kernel-intensive applications.
- A direct Go-vs-C performance comparison of equivalent code typical of that found in a kernel.

The overall conclusion of this dissertation is that Go worked well for kernel development and has a reasonable performance cost for our applications. Nevertheless, the choice of whether to use Go or C to implement a kernel is a trade-off and the programmer should consider the requirements of the project. If a primary goal is avoiding common security pitfalls, then Go helps by avoiding some classes of security bugs (see section 7.1.2). If the goal is to experiment with OS ideas, then Go's HLL features may help rapid exploration of different designs (see section 7.1.1). If CPU performance is paramount, then C is the right answer, since it is faster (see sections 7.2.2 and 7.2.3). If efficient memory use is

vital, then C is also the right answer: Go's garbage collector needs at least twice as much RAM as there is live kernel heap data to run efficiently (see section 7.2.4). Finally, if performance is merely important, consider paying the CPU and memory overhead of GC for the safety and productivity benefits of Go.

1.6 Outline

The next chapter discusses how our work relates to prior research. Chapter 3 then describes the design and main features of Biscuit. Next, chapter 4 details Go's garbage collector and a cost model to explain its performance. Chapter 5 describes how Biscuit avoids kernel heap exhaustion and chapter 6 explains details of Biscuit's implementation. Chapter 7 presents measurements of some of the benefits and performance costs of the HLL. Lastly, chapter 8 discusses the results in more depth and chapter 9 concludes.

Biscuit's source code is publicly available at:

https://pdos.csail.mit.edu/projects/biscuit.html

2 Related work

Biscuit builds on multiple areas of previous work: high-level languages in operating systems, high-level systems programming languages, and memory allocation in the kernel. As far as we know, the performance impact of language choice on a kernel with a traditional architecture has not been explored.

2.1 Kernels in high-level languages

The Pilot [57] kernel and the Lisp machine [28] are early examples of use of a high-level language (Mesa [25] and Lisp, respectively) in an operating system. Mesa lacked garbage-collection, but it was a high-priority requirement for its successor language Cedar [62]. The Lisp machine had a real-time garbage collector [8].

There are many research kernels written in a high-level language (e.g., Taos [63], Spin [10], Singularity [34, 35], J-kernel [30], and KaffeOS [6, 7], House [29], the Mirage unikernel [42], and Tock [40]). The main thrust of these projects was to explore new ideas in operating system architecture, often enabled by the use of a type-safe high-level language. While performance was often a concern, usually the performance in question related to the new ideas, rather than to the choice of language. As a result, none of the prior projects comprehensively measure the performance costs of an HLL kernel when running widely-used, kernel-intensive POSIX applications as this dissertation does (see section 7.2.2).

2.2 High-level systems programming languages

A number of systems-oriented high-level programming languages with type safety and garbage collection seem suitable for kernels, including Ada [1], Go, Java, C#, and Cyclone [38] (and, less recently, Cedar [62] and Modula-3 [50]). Other systems HLLs are less compatible with existing kernel designs. For example, Erlang [5] is a “shared-nothing” language with immutable objects, which would likely result in a kernel design that is quite different from traditional C shared-memory kernels.

Frampton et al. introduce a framework for language extensions to support low-level programming features in Java, applying it to a GC toolkit [24]. Biscuit's goal is efficiency for kernels without modifying Go and must handle additional kernel challenges, such as dealing with user and kernel space privilege levels, page tables, interrupts, and system calls.

Several new languages have recently emerged for systems programming: D [20], Nim(rod) [55], Go [27], and Rust [49]. There are a number of kernels in Rust [22, 23, 39–41, 51], but none were written with the goal of comparing with C as an implementation language. Go kernels exists but they don't target the questions that Biscuit answers. For example, Clive [9] is a unikernel and doesn't run on the bare metal. Gopher OS doesn't support garbage collection or goroutines [3]. The Ethos OS uses C for the kernel and Go for user-space programs, with a design focused on security [53]. gVisor is a user-space kernel, written in Go, that implements a substantial portion of the Linux system API to sandbox containers [26].

2.3 Memory allocation

There is no consensus about whether a systems programming language should have automatic garbage-collection. For example, Rust is partially motivated by the idea that garbage collection cannot be made efficient; instead, the Rust compiler analyzes the program to partially automate freeing of memory. This approach can make sharing data among multiple threads or closures awkward [39].

Concurrent garbage collectors [8, 37, 43] reduce pause times by collecting while the application runs. Go 1.10 has such a collector [33], which Biscuit uses.

Several papers have studied manual memory allocation versus automatic garbage collection [31, 66], focusing on heap headroom memory's effect in reducing garbage collection costs in user-level programs. Headroom is also important for Biscuit's performance (sections 4.2 and 7.2.4).

Rafkind et al. added garbage collection to parts of Linux through automatic translation of C source [56]. The authors observe that the kernel environment made this task difficult and adapted a fraction of a uniprocessor Linux kernel to be compatible with garbage collection. Biscuit required a fresh start in a new language, but as a result required less programmer effort for GC compatibility and benefited from a concurrent and parallel collector.

Linux's slab allocators [13] are specifically tuned for use in the kernel; they segregate free objects by type to avoid re-initialization costs and fragmentation. A hypothesis in the design of Biscuit is that Go's single general-purpose allocator and garbage collector are suitable for a wide range of different kernel objects.

2.4 Kernel heap exhaustion

All kernels which dynamically allocate memory (i.e., all widely used kernels we know of) have to cope with the possibility of running out of memory for the kernel heap. Linux optimistically lets system calls proceed up until the point where an allocation fails. In some cases code waits and re-tries the allocation a few times, to give an "out-of-memory" killer thread time to find and destroy an abusive process to free memory. However, the allocating thread cannot generally wait indefinitely: it may hold locks, so there is a risk of deadlock if the victim of the killer thread is itself waiting for a lock [18]. As a result Linux system calls must contain code to recover from allocation failures, undoing any changes made so far, perhaps unwinding through many function calls. This undo code has a history of bugs [19]. Worse, the final result will be an error return from a system call. Once the heap is exhausted, any system call that allocates will likely fail; few programs continue to operate correctly in the face of unexpected errors from system calls, so the end effect may be application-level failure even if the kernel code handles heap exhaustion correctly.

Biscuit's reservation approach yields simpler code than Linux's. Biscuit kernel heap allocations do not fail (much as with Linux's contentious "too small to fail" rule [18, 19]), eliminating a whole class of complex error recovery code. Instead, each Biscuit system call reserves kernel heap memory when it starts (waiting if necessary), using a static analysis system to decide how much to reserve. Further, Biscuit applications don't see system call failures when the heap is exhausted; instead, they see delays.

2.5 Computing memory allocation bounds

Researchers have developed methods to calculate the amount of heap memory required by a particular program in many situations. Many approaches require a functional language or type-system or language support [2, 16, 32, 54, 64] and so it is not obvious how to use them in an existing language, like Go. The analysis most similar to this dissertation's is that of Braberman et al. [14]. Braberman's analysis, like this dissertation's, is an inter-procedural analysis on an annotated program, which Braberman et al. evaluated on several benchmarks that are hundreds of lines of code each. MAXLIVE demonstrates that it is possible to quickly calculate the amount of heap memory required for execution of system calls in an operating system kernel with 28 thousand lines of code.

3 Overview

Biscuit's main purpose is to help evaluate the practicality of writing a kernel in a high-level language. Its design is similar to common practice in monolithic UNIX-like kernels, to make comparison simpler. Biscuit runs on 64-bit x86 hardware and is written in Go. It uses a modified version of the Go 1.10 runtime implementation; the runtime is written in Go with some assembly. Biscuit adds more assembly to handle boot and entry/exit for system calls and interrupts. There is no C. This section briefly describes Biscuit's components, focusing on areas in which use of Go affected the design and implementation.

3.1 Boot and Go runtime

The boot block loads Biscuit, the Go runtime, and a "shim" layer (written in Go and shown in Figure 3-1). The Go runtime, which we use mostly unmodified, expects to be able to call an underlying kernel for certain services, particularly memory allocation and control of execution contexts (cores). The shim layer provides these functions, since there is no underlying kernel. Most of the shim layer's activity occurs during initialization, for example to pre-allocate memory for the Go kernel heap.

3.2 Processes and kernel goroutines

Biscuit provides user processes with a POSIX interface: `fork`, `exec`, and so on, including kernel-supported threads and futexes. A user process has one address space and one or more threads. Biscuit uses hardware page protection to isolate user processes. A user program can be written in any language; we have implemented them only in C and C++ (not Go). Biscuit maintains a kernel goroutine corresponding to each user thread; that goroutine executes system calls and handlers for page faults and exceptions for the user thread. A user thread starts a system call using the `sysenter` instruction and, after executing the system call, the kernel returns using the `sysexit` instruction. "goroutine" is Go's name for a thread, and in this dissertation refers only to threads running inside the kernel.

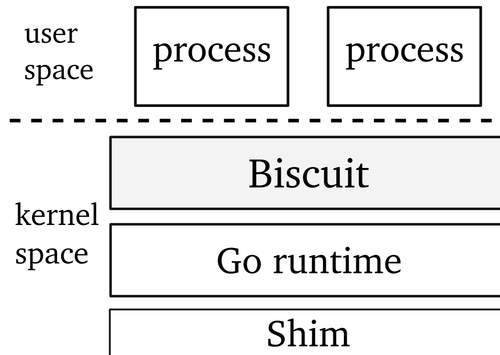

Figure 3-1: Biscuit's overall structure.

An unusual aspect of Biscuit is that it has two schedulers: one in the shim layer that schedules runtime threads to CPUs and one in the Go runtime that schedules kernel goroutines to runtime threads. In principle, the shim-layer scheduler is not necessary and could be eliminated, perhaps by instead having each CPU execute only one runtime thread. However, we specifically chose to use the shim-layer scheduler since it allows Biscuit to use the Go runtime with few modifications. Since the runtime may create helper threads (for instance, to manage timers), there may be more threads than there are CPUs. Without the shim-layer scheduler, the functionality of these helper threads would somehow have to be integrated into the main runtime threads. On the other hand, with the shim-layer scheduler, no changes are necessary to the main threads, helper threads, or any other threads that the Go developers later add.

Biscuit's Go runtime schedules the kernel goroutines of user processes, each executing its own user thread in user-mode when necessary. Biscuit uses timer interrupts to switch pre-emptively away from user threads. It relies on pre-emption checks generated by the Go compiler to switch among kernel goroutines.

3.3 Interrupts

A Biscuit device interrupt handler marks an associated device-driver goroutine as runnable and then returns, as previous kernels have done [48, 59]. Interrupt handlers cannot do much more without risk of deadlock, because the Go runtime does not turn off interrupts during sensitive operations such as goroutine context switch.

Handlers for system calls and faults from user space can execute any Go code. Biscuit executes this code in the context of the goroutine that is associated with the current user thread.

3.4 Multi-core and synchronization

Biscuit runs in parallel on multi-core hardware. It guards its data structures using Go's mutexes, and synchronizes using Go's channels and condition variables. The locking is fine-grained enough that system calls from threads on different cores can execute in parallel in many common situations, for example when operating on different files, pipes, sockets, or when forking or calling `exec` in different processes. Biscuit uses read-lock-free lookups in some performance-critical code (see below).

Some kernels use lock-free per-CPU variables to avoid synchronization, but lock-free per-CPU variables can be awkward to use in Biscuit. In C, the programmer synchronizes access to per-CPU variables by explicitly disabling interrupts during critical sections that access the per-CPU variables. This discipline prevents the scheduler from migrating the thread to a different CPU during the critical section, which is important since allowing such a migration could result in a data race where one CPU accesses a per-CPU variable belonging to a different CPU. However, safely using lock-free per-CPU variables in Go requires that the programmer both disable interrupts and avoid all Go code that may include a preemption check (like a function call or type assertion). Avoiding preemption checks in simple cases is straight-forward (by not calling any functions), but in code that does much more than read or write a per-CPU variable, avoiding the checks can be burdensome. For this reason, Biscuit does not use lock-free per-CPU variables, but instead protects per-CPU variables with locks and leaves interrupts enabled. As a result, critical sections can freely call code containing preemption checks, but they incur the overhead of locks.

3.5 Virtual memory

Biscuit uses page-table hardware to implement zero-fill-on-demand memory allocation, copy-on-write fork, shared anonymous and file mappings, and lazy mapping of files (e.g., for `exec`) in which the PTEs are populated only when the process page-faults, and `mmap`.

Biscuit records distinct, contiguous memory mappings compactly in a red-black tree, so in the common case large numbers of mapping objects aren't needed. Physical pages can have multiple references; Biscuit tracks these using reference counts.

3.6 File system

Biscuit implements a file system supporting the core POSIX file system calls. The file system has a file name lookup cache, a vnode cache, and a block cache. Biscuit guards each vnode with a mutex and resolves pathnames by first attempting each lookup in a

read-lock-free directory cache before falling back to locking each directory named in the path, one after the other. Biscuit runs each file system call as a transaction and has a journal to commit updates to disk atomically. The journal batches transactions through deferred group commit, and allows file content writes to bypass the journal. Biscuit has an AHCI disk driver that uses DMA, command coalescing, native command queuing, and MSI interrupts.

3.7 Network stack

Biscuit implements a TCP/IP stack, ARP, and a driver for Intel PCI-Express Ethernet NICs in Go. The driver uses DMA, MSI interrupts, TCP segmentation and checksum offloading, interrupt coalescing, and per-CPU transmit queues. The system-call API provides POSIX sockets.

3.8 Limitations

Although Biscuit can run many Linux C programs without source modification, it is a research prototype and lacks many features. Biscuit does not support scheduling priority because it relies on the Go runtime scheduler. Biscuit is optimized for a small number of cores, but not for large multicore machines or NUMA. Biscuit does not swap or page out to disk, and does not implement the reverse mapping that would be required to steal mapped physical pages. Biscuit lacks many security features like users, access control lists, or address space randomization.

4 Garbage collection

Biscuit's use of garbage collection is a clear threat to its performance. This section outlines the Go collector's design, describes a model of garbage collector performance, and explains how Biscuit configures the collector. Chapter 7 evaluates performance costs.

4.1 Go's collector

Go 1.10 has a concurrent parallel mark-and-sweep garbage collector [33]. The concurrent aspect is critical for Biscuit, since it minimizes the collector's "stop-the-world" pauses.

When the Go collector is idle, the runtime allocates from the free lists built by the last collection. When the free space falls below a threshold, the runtime enables concurrent collection. When collection is enabled, the work of following ("tracing") pointers to find and mark reachable ("live") objects is interleaved with execution: each allocator call does a small amount of tracing and marking. Writes to already-traced objects are detected with compiler-generated "write barriers" so that any newly installed pointers will be traced. Once all pointers have been traced, the collector turns off write barriers and resumes ordinary execution. The collector suspends ordinary execution on all CPUs (a "stop-the-world" pause) twice during a collection: at the beginning to enable the write barrier on all CPUs and at the end to check that all objects have been marked. These stop-the-world pauses typically last dozens of microseconds. The collector rebuilds the free lists from the unmarked parts of memory ("sweeps"), again interleaved with Biscuit execution, and then becomes idle when all free heap memory has been swept. The collector does not move objects, so it does not reduce fragmentation.

The Go collector does most of its work during calls to the heap allocator, spreading out this work roughly evenly among calls; each thread must complete an amount of GC work that is proportional to the amount that it allocates. Thus the frequency of GC delays during a goroutine's execution is proportional to the amount that it allocates; section 7.2.3 presents measurements of these delays for Biscuit.

Concurrent garbage collection typically begins collecting before the free lists are exhausted in order to satisfy concurrent allocations; if there is not enough free memory to satisfy concurrent allocations, the collector effectively becomes stop-the-world since any thread that allocates must block and wait until the collection completes.

4.2 Garbage collection cost model

This section describes a model for garbage collector performance. The observations of this section are not novel, but we nevertheless found the model helpful for two reasons. First, the model clearly explains why GC performance is mainly determined by a single ratio (h, described below) and not an absolute amount of memory. Second, the model makes obvious the two main opportunities available to the programmer for reducing the performance cost of GC.

The fraction of CPU time used by garbage collection is largely determined by the CPU time required for each collection and the interval between collections [31, 66]. The former is proportional to the number of live objects. The latter is a function of the amount of free heap space ("headroom") available after each collection and the rate at which the program (Biscuit in this case) allocates memory. The following analysis models these relationships.

First, define h to be the heap headroom ratio, the ratio of the memory used by live objects to the total heap memory. h turns out to largely determine the fraction of time spent in collection, independently of the absolute amount of live data. To simplify this presentation we'll define h in terms of numbers of objects, effectively assuming that all objects are the same size.

$$h = \frac{n_{live}}{\text{total heap size}} = \frac{n_{live}}{n_{live} + n_{free}}$$

The time consumed by a single collection is roughly the number of live objects times the time to read and write a cache line in RAM (to mark the object), plus the number of free objects in the heap times the time to add an object to a free list:

$$t_{gc} = n_{live} * k_m + n_{free} * k_s$$

The rate of collections per second is determined by the rate at which the program allocates per second and the amount of free space per collection. Note $n_{free} = (\frac{1}{h} - 1) * n_{live}$ by the definition of h.

$$r_{gc} = \frac{r_{alloc}}{n_{free}} = \frac{r_{alloc}}{(\frac{1}{h} - 1) * n_{live}}$$

The fraction of time spent collecting is the collection rate times the time per collection:

$$f_{gc} = r_{gc} * t_{tc}$$

Expanding both factors:

$$f_{gc} = \frac{r_{alloc}}{(\frac{1}{h} - 1) * n_{live}} * (n_{live} * k_m + n_{free} * k_s)$$

Re-arranging and canceling n_{live}:

$$f_{gc} = \frac{k_m * r_{alloc}}{\frac{1}{h} - 1} + k_s * r_{alloc} \tag{4.1}$$

k_m is much larger than k_s, because the random memory references during marking aren't very cacheable, while sweeping is sequential and thus benefits from the hardware prefetcher. Thus for larger values of h (i.e., the live data increases) the mark time dominates; for smaller values of h, the sweep time dominates.

The key implication of Equation 4.1 is that a kernel implementer can reduce the fraction of time spent in GC in two ways: by changing the kernel code to allocate less (reducing r_{alloc}), and by decreasing the ratio of live data size to total heap size (h). In particular, by dedicating enough memory to the kernel heap the overhead of collection can be kept low, even if there are millions of live objects. In practice, we expect Biscuit to be configured so that the heap RAM is at least twice the expected peak kernel heap live data size; see section 8.1.1 for further discussion.

4.3 Biscuit's heap size

At boot time, Biscuit allocates a fixed amount of RAM for its Go heap, defaulting to 1/32nd of total RAM. Go's collector ordinarily expands the heap memory when live data exceeds half the existing heap memory; Biscuit disables this expansion. Chapter 5 explains how Biscuit copes with situations where the heap space is nearly filled with live data.

5 Avoiding heap exhaustion

Biscuit must address the possibility of live kernel data completely filling the RAM allocated for the heap ("heap exhaustion"). For example, the kernel heap could become exhausted by a network server which rapidly processes requests where each request leaves metadata for a TCP socket that must remain live for some time (i.e., the socket is in the TIME-WAIT state).

Avoiding heap exhaustion is a difficult problem that existing kernels struggle with (see section 2.4). Widely-used C kernels avoid heap exhaustion by designing the heap memory allocator to return failure when memory is low and writing code to handle the potential failure of nearly all heap allocations. Writing code to handle nearly all allocation failures is challenging and error prone [18, 19].

The rest of this chapter describes how Biscuit handles heap exhaustion.

5.1 Approach: reservations

Biscuit is designed to tolerate heap exhaustion without kernel failure. In addition (and like widely-used C kernels), it can take corrective action when there are identifiable "bad citizen" processes that allocate excessive kernel resources implemented with heap objects, such as the structures describing open files and pipes. Biscuit tries to identify bad citizens and kill them, in order to free kernel heap space and allow good citizens to make progress.

Biscuit's approach to kernel heap exhaustion has three elements. First, it purges caches and soft state as the heap nears exhaustion. Second, code at the start of each system call waits until it can reserve enough heap space to complete the call; the reservation ensures that the heap allocations made in the call will succeed once the wait (if any) is over. Third, a kernel "killer" thread watches for processes that are consuming lots of kernel heap when the heap is near exhaustion, and kills them.

This approach has some good properties. Applications do not have to cope with system call failures due to kernel heap exhaustion. Kernel code does not see heap allocation

failure (with a few exceptions), and need not include logic to recover from such failures midway through a system call. System calls may have to wait for reservations, but they wait at their entry points without locks held, and thus cannot deadlock.

The killer thread must distinguish between good and bad citizens, since killing a critical process (e.g., `init`) can make the system unusable. An example of a situation with an obvious bad citizen is a buggy program that continuously creates non-contiguous anonymous memory mappings, resulting in many virtual memory objects in the kernel heap. If there is no obvious bad citizen, this approach may block and/or kill valuable processes. Lack of a way within POSIX for the kernel to gracefully revoke resources causes there to be no good solution in some out-of-memory situations; for instance, when nearly all memory is used by a critical database server.

The mechanisms in this section do not apply to non-heap allocations. In particular, Biscuit allocates physical memory pages (for user memory, file cache pages, or socket buffers) from a separate allocator, not from the Go heap; page allocations can fail, and kernel code must check for failure and recover (typically by returning an error to a system call).

5.2 How Biscuit reserves

Biscuit dedicates a fixed amount of RAM M for the kernel heap. A system call starts only if it can reserve enough heap memory for the maximum amount of *simultaneously* live data (MSLD) that it uses. s is the amount heap memory to reserve and is at least as large as the MSLD (precise calculation of the MSLD is difficult, see section 5.3.1). A system call may allocate more than s from the heap, but the amount over s must be dead (by the definition of MSLD and s) and can be freed by the collector. This means that, even in the extreme case in which all but s of the heap RAM is used by live data or is already reserved, the system call can execute (with collections as needed) to recover the call's own dead data in excess of s.

Ideally, a reservation should check that M minus the amount of live and reserved data in the heap is greater than or equal to s. However, except immediately after a collection, the amount of live heap data is unknown. Biscuit maintains a conservative over-estimate of live heap data using three counters: *lastgc*, *used*, and *current*. *lastgc* is the amount of live data marked by the previous garbage collection. *used* is the total amount of reservations made by system calls that have completed. *current* is the total outstanding reservations of system calls that are executing but not finished. Let L be the sum of *lastgc*, *used*, and *current*.

Figure 5-1 presents pseudo code for reserving and releasing the reservation of heap RAM in Biscuit. Before starting a system call, a thread checks that $L+s < M$. If $L+s < M$,

```
reserve(s):
    L := lastgc + used + current
    M := heap RAM bytes
    if L + s < M:
        current += s
    else:
        // wake and send s to killer thread
        // wait for OK from killer thread

release(s):
    if alloc < s:
        used += alloc
    else:
        used += s
    current -= s
```

Figure 5-1: Pseudo code for heap reservations in Biscuit.

the thread reserves by adding s to *current*, otherwise the thread wakes up the killer thread and sleeps. When finished, a system call calculates *alloc*, the total amount actually allocated, and uses *alloc* to (partially) release any over-reservation: if $alloc < s$, the system call adds *alloc* to *used* and subtracts s from *current*. Otherwise, $alloc \geq s$ and the system call adds s to *used* and subtracts s from *current*.

The reason for separate *used* and *current* is to carry over reservations of system calls that span a garbage collection; a collection sets *used* to zero but leaves *current* unchanged.

If heap memory is plentiful (live data $\ll M$), the amount of live+dead data in the heap usually grows faster than L (since most allocations become dead quickly), so collections are triggered by heap free list exhaustion rather than by $L + s \geq M$. Thus system calls do not wait for memory, and do not trigger the killer thread. As live heap data increases, and $lastgc + current$ gets close to M, $L + s$ may reach M before a collection would ordinarily be triggered. For this reason the killer thread performs a collection before deciding whether to kill processes.

5.3 Static analysis to find s

We have developed a tool, MaxLive, that inter-procedurally analyzes the Biscuit source code and the Go packages Biscuit uses to find s, a bound on the MSLD, for each system call. The core challenge is detecting statically when allocated memory must be dead and thus is unnecessary to include in s, resulting in a bound that is small enough to be useful (see section 5.3.1). Other challenges include analyzing loops whose maximum number of iterations (and thus amount of allocation) are not obvious and determining reservations for background kernel activities that are not associated with a specific system call.

We address these challenges by exploiting the characteristic event-handler-style structure of most kernel code, which does a modest amount of work and then returns (or goes idle); system call implementations, for example, work this way. Furthermore, we are willing to change the kernel code to make it easier for MAXLIVE to analyze, for example to avoid recursion or cycles in the call graph. Avoiding recursion and making the call graph acyclic was a minor inconvenience in Biscuit: it required us to change a few functions. Two modifications were also required to standard Go packages that Biscuit uses (packages *time* and *fmt*).

The discussion in the rest of this chapter refers to call graphs. The "call graph of function x" is the directed, acyclic graph where the nodes are the functions x and those that may be directly or transitively called by x. There is an edge from node a to node b if and only if function a contains a function call that may call function b.

This method requires that the programmer supply source code annotations for allocations whose size cannot be easily determined statically (see section 5.3.8) and for the maximum number of iterations of loops (see section 5.3.9).

5.3.1 Not just any s will do

We don't know of a way to quickly and precisely compute the MSLD of a system call. Instead of precisely computing the MSLD, MAXLIVE computes an upper bound on the MSLD, s. MAXLIVE takes annotated Go source code for a system call as input and outputs a number, which is a bound on the MSLD during execution of the given system call. MAXLIVE is conservative in that the computed s is at least as large as the MSLD during execution of the system call.

However, MAXLIVE cannot be too conservative. An s that is much larger than the MSLD could reduce the maximum number of in-progress system calls. In the worst case, at most M/s_{max} system calls can be in progress at any time, where s_{max} is the largest s of all system calls. Therefore MAXLIVE likely causes no significant problem if it overestimates s by a small factor, but overestimating by a factor of 100 could severely limit system concurrency.

An earlier version of MAXLIVE used the maximum total allocations (MTA, which is also a bound on the MSLD) instead of the method described in this section to find s for a system call. Calculating the MTA is significantly simpler, but unfortunately the MTA is far larger than the MSLD bound found by this method for many system calls since many allocations are short-lived and quickly become dead. Thus the MTA is not useful to use as s since it is hundreds of terabytes for some system calls.

As a demonstration of why the MTA can be much larger than the MSLD, consider the example call graph shown in Figure 5-2. The call graph in Figure 5-2 shows three functions, `foo`, `bar`, and `sue`, and all of them allocate heap memory. The MTA for the

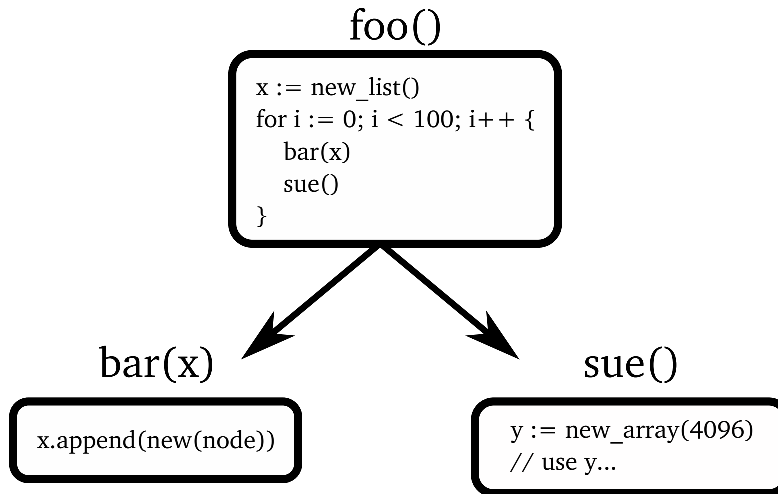

Figure 5-2: Example call graph with allocations.

call graph in Figure 5-2 is the sum of the sizes of the list head, 100 list nodes, and 100 of the 4096-byte arrays (due to the loop), even though at most one array can be live at any time. Thus the MTA includes the size of 99 superfluous arrays because it ignores deallocation, unlike the MSLD.

5.3.2 Reasoning about maximum simultaneously live data

MAXLIVE therefore reasons about deallocation to find bounds smaller than the MTA and closer to the MSLD. Since static, precise detection of the moment a particular allocation becomes dead is difficult or impossible in general, our efforts to improve the bound focus on a particularly common case of an object becoming dead. Specifically, the case where some function in a subtree of the call graph allocates an object, no function in the subtree stores a reference to that object in a global variable or any object reachable from a global variable, and no function outside the subtree could possibly have a pointer to that object. Such an allocation must be dead once the function in the subtree closest to the root returns.

In other words, such objects are only reachable from stack frames and never from global variables (similar to escape analysis [12, 17]). Objects reachable only from stack frames must become dead when a particular function returns and thus don't count against *s* once that particular function returns. Therefore the high-level approach to compute the MSLD is to find the moment during execution of the system call where the peak amount of heap memory is reachable from all live stack frames.

The following two simplifications make it easier to reason about the MSLD of a function during execution. First, the rest of this chapter reasons about object liveness

by whether or not an object is reachable only from stack frames. In reality, an object reachable from a stack frame or a global variable is live. But, for simplicity, the rest of this chapter makes no further mention of global variables and treats any allocation that may be reachable from a global variable as live at all times. This method is conservative since such allocations always count against s, even though they may become dead during execution.

The second simplification is that objects only become dead once the function that destroyed the final pointer to them returns. A function could overwrite the only remaining pointer to a heap object with a different pointer or nil during execution, causing the object to become dead. But the rest of this chapter pretends that any objects that would normally become dead during the execution of a function don't become dead at that time, but instead only become dead once that function returns. This simplification makes reasoning about the MSLD slightly easier and doesn't affect correctness since it can only increase the amount of heap memory used and thus is strictly conservative.

5.3.3 Approximating liveness via the call graph

During execution of a function, only ancestors in the call graph of the currently-executing function can have live stack frames [1]. Thus only allocations that are reachable from the stack frames of the currently-executing and ancestors functions can be live. An executing function can store a pointer to an allocation in objects or data structures passed as function arguments, potentially making the allocation live from the stack frames of any number of ancestor functions. This allocation will become dead once some ancestor function returns, specifically that ancestor which is closest to the root and from whose stack frame the allocation is reachable.

Consider the following abstract function, P, which helps calculate the MSLD. Let $P(x)$ map the set of functions x to the set of all allocator calls whose allocations may be transitively reachable from any of the stack frames of the functions in x. (Note that an allocator call in $P(x)$ may be made by a function that is not in x.) For instance, $P(\{foo\})$ for the call graph shown in Figure 5-2 is the set containing the list node allocation in `bar` and the list head allocation in `foo` (the array allocation in `sue` is not reachable from `bar`). P can be implemented through pointer analysis [4, 61].

The set of live allocations at any point during execution of a system call are those made by $P(L)$ where L is the path (a set of nodes) from the root (the system call entry point) to the function executing at that point in the call graph. The reason is that only the ancestors of the currently-executing function can also have live stack frames at any

[1] Allocations reachable from a closure are also reachable from the stack frame of any function which has a pointer to the closure.

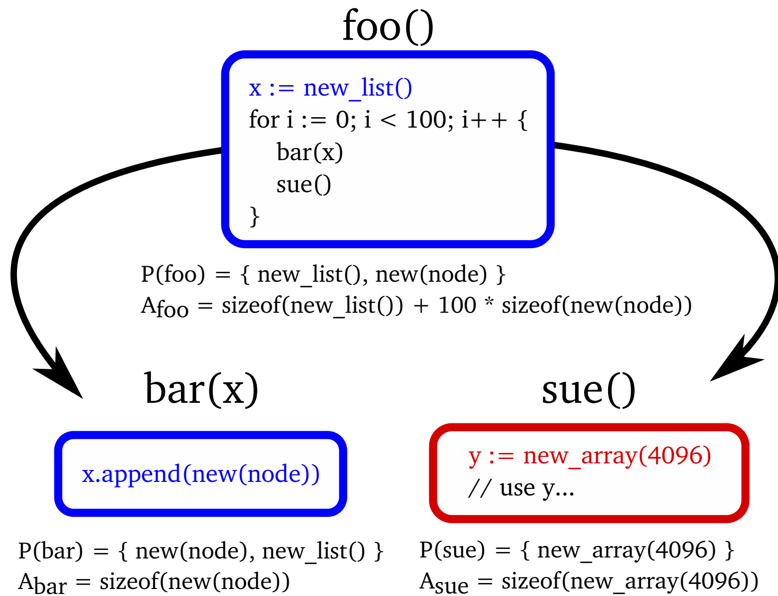

Figure 5-3: Example call graph and state of MAXLIVE analysis. The blue and red allocations are reachable from the stack frames of the blue and red functions, respectively.

point during the execution of the system call. Thus all allocations that are unreachable from all live stack frames must be dead.

Figure 5-3 shows an example call graph with allocations (the same call graph shown in Figure 5-2). Reachability of allocations is indicated by matching colors of the line of code containing the allocation and the border of the function's node in the call graph. Function `foo` allocates a new list, which is reachable from `foo`'s stack frame. `bar` allocates new list nodes and appends them to the same list, thus the allocation in `bar` is reachable from `foo`. `sue` allocates an array of 4096 bytes, but that allocation is not reachable from `foo` so it becomes dead once `sue` returns.

Consider the MSLD of `foo` in Figure 5-3. The maximum amount of heap memory used during the execution of `foo` occurs during the last iteration of `foo`'s `for` loop, after `sue` has allocated the array but not yet returned. At that point, the list head allocation, `bar`'s 100 linked list nodes, and `sue`'s 4096 byte array are all still live. Thus the MSLD is the number of bytes used by the linked list head, 100 linked list nodes, and one 4096-byte array.

One way to calculate a bound on the MSLD is to brute force search all possible executions of the system call. Unfortunately the runtime of a brute force search is roughly $O(2^n)$ in the total number of branches in all functions in the call graph, making such a search likely too slow to be useful. For this reason, MAXLIVE uses a different approach to keep the runtime short enough to be practical.

5.3.4 MaxLive operation

The purpose of MaxLive is to compute a low-enough bound on the MSLD of a system call in a reasonable amount of time, i.e. faster than a brute force search of all executions of the system call. MaxLive finds a bound on the MSLD much faster than a brute force search (MaxLive calculates s for all the system calls in Biscuit in about five minutes), but MaxLive's bound is typically larger than that found by a brute force search. Nevertheless, MaxLive's bounds are small enough to be useful in practice.

MaxLive computes the MSLD bound in the following way. First, for each function i in the call graph rooted at the system call entry point, MaxLive computes A_i, which is the maximum amount of allocations that can ever be reachable from i's stack frame. MaxLive computes A_i by searching all reachable functions in the call graph rooted at i in post order. Each visit finds the path through the function's basic blocks (including descendants, but ignoring back edges) that allocates the maximum number of bytes using only the allocator calls in $P(\{i\})$. Each function call counts as an allocation of size $\max_{x \in X}(A_x)$ where X is the set of all possible callees of the function call (e.g., an interface method call may have multiple potential callees). For each basic block that is in a loop, the maximum amount of allocation of that block is multiplied by the maximum amount of iterations of the containing (and possibly nested) loop. Post-order ensures that the maximum amount of allocations for a function has been computed before visiting any callers (the call graph is acyclic). Once the search terminates, A_i is the amount of allocation calculated by the visit to function i.

After calculating each A_i, MaxLive finds the moment during execution of the system call where the stack frames that potentially have the largest amount of simultaneously live data occur, i.e. the path L from root of the call graph (the system call entry point) to all leaves where $\sum_{i \in L} A_i$ is largest. The result for the largest path is a bound on the MSLD.

Pseudo code for MaxLive is shown in Figure 5-4. Each call to `max1` computes one of the A_i terms. `max1`(x) finds the path through function x and all descendants that allocates most using only allocations reachable from x. `maxpath` then finds the path L where the most simultaneous live data may occur during execution of the system call.

5.3.5 MaxLive example

Let's examine how MaxLive would compute an MSLD bound for `foo` in Figure 5-3. MaxLive would first calculate A_{foo} via `max1`. `max1`(*foo*) would find the list node allocations in `bar` since that allocation is in $P(\{foo\})$. Because the call to `bar` is in a loop with 100 iterations, MaxLive multiplies the total allocations made by `bar` by 100. MaxLive also observes that no allocations in `sue` are in $P(\{foo\})$, so none of `sue`'s allocations are

```
// takes a system call entry point and returns s, a bound on the MSLD of the
// system call
maxlive(root):
    maxes := empty map
    // compute A_i terms
    for each function f in call graph:
        maxes[f] = max1(f)
    // find path of live stack frames with most simultaneously live data
    return maxpath(root, maxes)
// takes a function, returns A_cur, the maximum amount of allocation that can be
// reachable from cur's stack frame
max1(cur):
    visited := empty map
    // "P" is the function described in section 5.3.3.
    // "live" is the set of all allocation calls whose allocations
    // may be reachable from globals or the stack frame of cur
    live := P(cur)
    search(cur, live, visited)
    return visited[cur]
// takes a function, a set of allocation calls, and a map of functions to
// maximum amount of allocation, returns nothing
search(f, live, visited):
    for c in f.callees:
        search(c, live, visited)
    visit(f, live, visited)
// same parameters as search, returns nothing
visit(f, live, visited):
    max_path := 0
    for each distinct path p in f:
        path_total := 0
        for each basic block b in p:
            block_total := 0
            for each instruction ins in b:
                if ins is allocation and in live:
                    block_total += alloc_size(ins)
                else if ins is function call:
                    allocs := [visited[c] for c in ins.callees]
                    block_total += max(allocs)
            if b is in loop:
                block_total *= loop product
            path_total += block_total
        if path_total > max_path:
            max_path = path_total
    visited[f] = max_path
// takes a function and a map of functions to maximum amount of allocation,
// returns the maximum amount of bytes allocated by any path from f to a leaf
// in the call graph
maxpath(f, maxes):
    callees := [maxpath(c, maxes) for c in f.callees]
    return maxes[f] + max(callees)
```

Figure 5-4: Pseudo code for MAXLIVE.

reachable from `foo`'s stack frame. Thus A_{foo} is the number of bytes used by the allocations of the list head and 100 list nodes, but not the array in `sue`.

MAXLIVE would then compute A_{bar} and A_{sue} which are the number of bytes used by a single list node and the array, respectively. `maxpath` would then compare $A_{foo} + A_{bar}$ and $A_{foo} + A_{sue}$ and find that the latter is larger and thus is a bound on the MSLD. This is correct: $A_{foo} + A_{sue}$ is the number of bytes used on the list head, 100 linked list nodes, and the 4096 byte array.

Although MAXLIVE's bound was minimal in this example, it is unlikely to be minimal in practice. The reason is that the paths that allocate the most for the A_i values are not necessarily the same path. For example, suppose `alpha` calls `bravo` and both `alpha` and `bravo` are in L. The execution of `alpha` which allocates the most memory reachable from `alpha`'s stack frame may not include a call to `bravo`. Thus $\sum_{i \in L} A_i$ may include multiple distinct paths, even though it may be that only some subset of those paths can occur at runtime. This further increases the bound beyond the actual MSLD, but reduces MAXLIVE's runtime to roughly $O(m^2)$ in the number of functions, improving over the brute force search's $O(2^n)$ in the total number of branches.

The bound computed by MAXLIVE could be reduced by a more sophisticated analysis. MAXLIVE assumes all paths through a function's basic blocks are possible, regardless of conditional control flow. Thus the execution through a function or the call graph found by MAXLIVE to exhibit the maximum amount of allocation may actually be impossible.

5.3.6 MAXLIVE correctness

The following is an argument for why the bound computed by MAXLIVE is at least as large as the actual MSLD. Let L_M be the call graph path of live stack frames during execution of the system call at the moment where MSLD bytes were in use. Let S_i be the number of bytes of allocations that are reachable from the stack frame of function i at that same moment.

Suppose that MAXLIVE is incorrect and produced a bound that is less than the MSLD. In other words,

$$\sum_{i \in L_M} A_i < \sum_{i \in L_M} S_i$$

and thus there is at least one b where $A_b < S_b$. So there is some path through the basic blocks of a function, reachable in the call graph rooted at b, that 1) differs from the path found by MAXLIVE and 2) allocates more than the path found by MAXLIVE. But this cannot be because MAXLIVE examined all paths from b and must have found this larger path.

5.3.7 MaxLive implementation

Our implementation of MaxLive analyzes the call graph and each function's basic blocks using Go's *callgraph* and *ssa* packages. MaxLive implements $P(x)$ using Go's *pointer* package, which provides pointer analysis [4, 61]; the *pointer* package takes as input a particular pointer in a stack frame or an object and outputs the set of all allocation calls which may have allocated the memory referenced by that pointer.

5.3.8 Special allocations

MaxLive handles a few kinds of allocation specially: *go*, *defer*, maps, and slices. *go* (which creates a goroutine) is treated as an escaping allocation of the maximum kernel stack size (the new goroutine itself must reserve memory when it starts, much as if it were itself a system call). *defer* is a non-escaping allocation, but is not represented by an allocator call instruction so MaxLive specifically considers it an allocation. Every insertion into a map or slice could double its allocated size; MaxLive generally doesn't know the old size, so it cannot predict how much memory would be allocated. To avoid this problem, we annotate the Biscuit source to declare the maximum size of slices and maps, which required 70 annotations.

5.3.9 Handling loops

For loops where MaxLive cannot determine a useful bound on the number of iterations, we supply a bound with an annotation; there were 78 such loops. Biscuit contains about 20 loops whose bounds cannot easily be expressed with an annotation, or for which the worst case is too large to be useful. Examples include retries to handle wakeup races in `poll`, iterating over a directory's data blocks during a path component lookup, and iterating over the pages of a user buffer in `write`.

We handle such loops with *deep reservations*. Each loop iteration tries to reserve enough heap for just the one iteration. If there is insufficient free heap, the loop aborts and waits for free memory at the beginning of the system call, retrying when memory is available. Two loops (in `exec` and `rename`) needed code to undo changes after an allocation failure; the others did not.

Three system calls have particularly challenging loops: `exit`, `fork`, and `exec`. These calls can close many file descriptors, either directly or on error paths, and each close may end up updating the file system (e.g., on last close of a deleted file). The file system writes allocate memory, and may create entries in file system caches. Thus, for example, an exiting process that has many file descriptors may need a large amount of heap memory for the one `exit` system call. However, in fact `exit`'s memory requirements are

much smaller than this: the cache entries will be deleted if heap memory is tight, so only enough memory is required to execute a single close. We bound the memory use of `close` by using MAXLIVE to find all allocations that may be live once `close` returns. We then manually ensure that all such allocations are either dead once `close` returns or are evictable cache entries. That way `exit`, `fork`, and `exec` only need to reserve enough kernel heap for one call to `close`. This results in heap bounds of less than 500kB for all system calls but `rename` and `fork` (1MB and 641kB, respectively). The `close` system call is the only one we manually analyze with the assistance of MAXLIVE.

5.3.10 Kernel threads

A final area of special treatment applies to long-running kernel threads. An example is the file system logging thread, which acts on behalf of many processes. Each long-running kernel thread has its own kernel heap reservation. Since `exit` must always be able to proceed when the killer thread kills a process, kernel threads upon which `exit` depends must never release their heap reservation. For example, `exit` may need to free the blocks of unlinked files when closing file descriptors and thus depends on the file system logging thread. Other kernel threads, like the ICMP packet processing thread, block and wait for heap reservations when needed and release them when idle.

5.3.11 Killer thread

The killer thread is woken up when a system call's reservation fails. The thread first starts a garbage collection and waits for it to complete. If the collection doesn't free enough memory, the killer thread asks each cache to free as many entries as possible, and collects again. If that doesn't yield enough free memory, the killer thread finds the process with the largest total number of mapped memory regions, file descriptors, and threads, in the assumption that it is a genuine bad citizen, kills it, and again collects. As soon as the killer thread sees that enough memory has been freed to satisfy the waiting reservation, it wakes up the waiting thread and goes back to sleep.

5.4 Limitations

Biscuit's approach for handling heap exhaustion requires that the garbage collector run successfully when there is little or no free memory available. However, Go's garbage collector may need to allocate memory during a collection in order to make progress, particularly for the work stack of outstanding pointers to scan. We haven't implemented it, but Biscuit could recover from this situation by detecting when the work stack is full and falling back to using the mark bitmap as the work stack, scanning for objects

which are marked but contain unmarked pointers. This strategy will allow the garbage collection to complete, but will likely be slow. We expect this situation to be rare since the work stack buffers can be preallocated for little cost: in our experiments, the garbage collector allocates at most 0.8% of the heap RAM for work stacks.

Because the Go collector doesn't move objects, it doesn't reduce fragmentation. Hence, there might be enough free memory but in fragments too small to satisfy a large allocation. To eliminate this risk, MAXLIVE should compute s for each size class of objects allocated during a system call. Our implementation doesn't do this.

5.5 Heap exhaustion summary

Biscuit borrows ideas for heap exhaustion from Linux: the killer thread, and the idea of waiting and retrying after the killer thread has produced free memory. Biscuit simplifies the situation by using reservation checks at the start of each system call, rather than Linux's failure checks at each allocation point; this means that Biscuit has less recovery code to back out of partial system calls, and can wait indefinitely for memory without fear of deadlock. Go's static analyzability helped automate Biscuit's simpler approach.

6 Implementation

The Biscuit kernel is written almost entirely in Go: Figure 6-1 shows that it has about 27,500 lines of Go, 1,500 lines of assembly, and no C.

Biscuit provides 58 system calls, listed in Figure 6-2. It has enough POSIX compatibility to run some existing server programs (for example, NGINX and Redis).

Biscuit includes device drivers for AHCI SATA disk controllers and for Intel 82599-based Ethernet controllers such as the X540 10-gigabit NIC. Both drivers use DMA. The drivers use Go's `unsafe.Pointer` to access device registers and in-memory structures (such as DMA descriptors) defined by device hardware, and Go's `atomic` package to control the order of these accesses. The code would be more concise if Go supported some kind of memory fence.

Biscuit contains 90 uses of Go's "unsafe" routines (excluding uses in the Go runtime). These unsafe accesses parse and format packets, convert between physical page numbers and pointers, read and write user memory, and access hardware registers.

We modified the Go runtime to record the number of bytes allocated by each goroutine (for heap reservations), to check for runnable device handler goroutines (woken up by interrupt handlers), and to increase the default stack size from 2kB to 8kB to avoid stack expansion for a few common system calls.

Biscuit lives with some properties of the Go runtime and compiler in order to avoid significantly modifying them. The runtime does not turn interrupts off when holding locks or when manipulating a goroutine's own private state. Therefore, in order to avoid deadlock, Biscuit interrupt handlers just set a flag indicating that a device handler goroutine should wake up. Biscuit's timer interrupt handler cannot directly force goroutine context switches because the runtime might itself be in the middle of a context switch. Instead, Biscuit relies on Go's pre-emption mechanism for kernel goroutines (the Go compiler inserts pre-emption checks in the generated code). Timer interrupts do force context switches when they arrive from user space.

Component	Lang	LOC
Biscuit kernel (mostly boot)	asm	500
Biscuit kernel	Go	
Core		1,700
Device drivers		4,100
File system		7,300
Network		4,500
Other		1,100
Processes		900
Reservations		700
System calls		5,300
Virtual memory		1,900
Total		27,500
MaxLive	Go	3,100
Runtime modifications	asm	1,000
Runtime modifications	Go	3,200

Figure 6-1: Approximate lines of code in Biscuit. Not shown are about 50,000 lines of Go runtime and 32,000 lines of standard Go packages that Biscuit uses.

accept	fcntl	getrlimit	listen	pipe2	recvfrom	socket	write
bind	fork	getrusage	lseek	poll	recvmsg	socketpair	writev
chdir	fstat	getsockopt	mkdir	pread	rename	stat	
close	ftruncate	gettid	mknod	prof	sendmsg	sync	
connect	futex	gettimeofday	mmap	pwrite	sendto	threxit	
dup2	getcwd	info	munmap	read	setrlimit	truncate	
execv	getpid	kill	nanosleep	readv	setsockopt	unlink	
exit	getppid	link	open	reboot	shutdown	wait4	

Figure 6-2: Biscuit's 58 system calls.

Goroutine scheduling decisions and the context switch implementation live in the runtime, not in Biscuit. One consequence is that Biscuit does not control scheduling policy; it inherits the runtime's policy. Another consequence is that per-process page tables are not switched when switching goroutines, so Biscuit system call code cannot safely dereference user addresses directly. Instead, Biscuit explicitly translates user virtual addresses to physical addresses, and also explicitly checks page permissions. Biscuit switches page tables if necessary before switching to user space.

We modified the runtime in three ways to reduce delays due to garbage collection. First, we disabled the dedicated garbage collector worker threads so that application threads don't compete with garbage collector threads for CPU cycles. Second, we made root marking provide allocation credit so that an unlucky allocating thread wouldn't mark many roots all at once (during GC, each thread completes GC work proportional to the number of bytes which it allocates). Third, we reduced the size of the pieces that large objects are broken into for marking from 128kB to 10kB.

Biscuit implements many standard kernel performance optimizations. For example, Biscuit maps the kernel text using large pages to reduce iTLB misses, uses per-CPU NIC transmit queues, and uses read-lock-free data structures in some performance critical code such as the directory cache and TCP polling. In general, we found that Go did not hinder optimizations.

7 Evaluation

This chapter analyzes the costs and benefits of writing a kernel in an HLL. Section 7.1 explores some of the benefits of using an HLL and section 7.2 measures the main performance costs of using an HLL.

7.1 HLL benefits

This section explores some of the ways that using an HLL was helpful for building an operating system kernel. The evaluation questions are:

- To what degree does Biscuit benefit from Go's high-level language features? To answer, we count and explain Biscuit's use of these features (section 7.1.1).
- Do C kernels have safety bugs that a high-level language might mitigate? We evaluate whether bugs reported in Linux kernel CVEs would likely apply to Biscuit (section 7.1.2).

7.1.1 Biscuit's use of HLL features

Our subjective feeling is that Go has helped us produce clear code and helped reduce programming difficulty, primarily by abstracting and automating low-level tasks.

Figure 7-1 shows how often Biscuit uses Go's HLL features, and compares with two other major Go systems: the Go repository (containing Go's compiler, runtime, and standard packages), and Moby[1], which contains Docker's container software and is the most starred Go repository on Github at the time of writing. Figure 7-1 shows the number of times each feature is used per 1,000 lines of code. Biscuit uses Go's HLL features about as much as other Go systems software.

To give a sense how these HLL features can benefit a kernel, the rest of this section provides examples of successful uses, as well as situations where we didn't use them.

[1] https://github.com/moby/moby

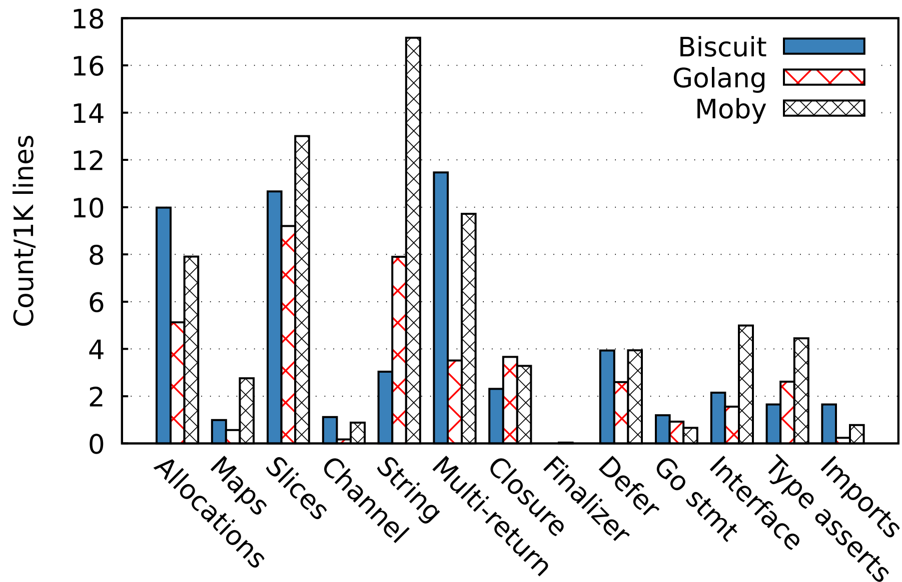

Figure 7-1: Uses of Go HLL features in the Git repositories for Biscuit, Go (1,140,318 lines), and Moby (1,004,300 lines) per 1,000 lines. For data types (such as slices), the numbers indicate the number of declarations of a variable, argument, or structure field of that type.

Biscuit relies on the Go allocator and garbage collector for nearly all kernel objects. Biscuit has 302 statements that allocate an object from the GC-managed heap. Some of the objects are compound (composed of multiple Go objects). For example, Biscuit's `Vmregion_t`, which describes a mapped region of virtual memory, has a red-black tree of `Vminfo_t`, which itself is compound (e.g., when it is backed by a file). The garbage collector eliminates the need for explicit code to free the parts of such compound data types.

Biscuit's only special-purpose allocator is its physical page allocator. It is used for process memory pages, file cache pages, socket and pipe buffers, and page table pages.

Biscuit uses many goroutines. For example, device drivers create long-running goroutines to handle events such as packet arrival. Biscuit avoids goroutine creation, however, in frequently executed code. The reason is that the garbage collector produces pauses proportional to the number of goroutines; these are insignificant for thousands of goroutines but a problem with hundreds of thousands.

The combination of threads and garbage collection is particularly pleasing, since it avoids forcing the programmer to worry about delaying frees for shared objects until the last sharing thread has finished. For example, Biscuit's `poll` system call installs a pointer

Type	CVE Identifier			
Use-after-free or double-free	2016-10290	2016-8480	2016-8436	2016-8391
	2016-10288	2016-8449	2016-8392	2016-6791
Out-of-bounds access	2017-1000251	2017-0612	2017-0453	2016-10289
	2017-6264	2017-0611	2017-0443	2016-10285
	2017-0622	2017-0608	2017-0442	2016-10283
	2017-0621	2017-0607	2017-0441	2016-8476
	2017-0620	2017-0521	2017-0440	2016-8421
	2017-0619	2017-0520	2017-0439	2016-8420
	2017-0614	2017-0465	2017-0438	2016-8419
	2017-0613	2017-0458	2017-0437	2016-6755

Figure 7-2: Linux kernel CVEs from 2017 that would *not* cause memory corruption, code execution, or information disclosure in Biscuit.

to a helper object in each file descriptor being polled. When input arrives on a descriptor, the goroutine delivering the input uses the helper object to wake up the polling thread. Garbage collection eliminates races between arriving input and freeing the helper object.

Some Biscuit objects, when the last reference to them disappears, need to take clean-up actions before their memory can be collected; for example, TCP connections must run the TCP shutdown protocol. Go's finalizers were not convenient in these situations because of the prohibition against cycles among objects with finalizers. Biscuit maintains reference counts in objects that require clean-up actions.

Biscuit uses many standard Go packages. For example, Biscuit imports *sync* in 28 files and *atomic* in 18 files. These packages provide mutexes, condition variables, and low-level atomic memory primitives. Biscuit's MaxLive tool depends on Go's code analysis packages (*ssa*, *callgraph*, and *pointer*).

Biscuit itself is split into 31 Go packages. Packages allowed some code to be developed and tested in user space. For example, we tested the file system package for races and crash-safety in user space. The package system also made it easy to use the file system code to create boot disks.

7.1.2 Potential to reduce bugs

An HLL might help avoid problems such as memory corruption from buffer overflows. To see how this applies to kernels, we looked at Linux execute-code bugs in the CVE database published in 2017 [47]. There are 65 bugs where the patch is publicly available. For 11 bugs of the 65, we aren't sure whether Go would have improved the outcome. 14 of the 65 are logic bugs that could arise as easily in Go as they do in C. Use of Go would have improved the outcome of the remaining 40 bugs (listed in Figure 7-2), based on manual inspection of the patch that fixed the bug. The impact of some of these 40 bugs

is severe: several allow remote code execution or information disclosure. Many of the bugs in the out-of-bounds category would have resulted in runtime errors in Go, and caused a panic. This is not ideal, but better than allowing a code execution or information disclosure exploit. The bugs in the use-after-free category would not have occurred in Go, because garbage collection would obviate them.

The Go runtime and packages that Biscuit relies on also have bugs. There are 14 CVEs in Go published from 2016 to 2018. Two of them allow code execution (all in `go get`) and two allow information gain (due to bugs in Go's *smtp* and *math/big* packages).

7.2 HLL performance costs

This section measures the main performance costs of the HLL when running complex, demanding applications. The evaluation questions are:

- How much performance does Biscuit pay for Go's HLL features? We measured the time Biscuit spends in garbage collection, bounds checking, etc., and the delays that GC introduces (sections 7.2.2 to 7.2.4).
- What is the performance impact of using Go instead of C? We compared nearly-identical pipe and page-fault handler implementations in Go and C (section 7.2.5).
- Is Biscuit's performance in the same ballpark as Linux, a C kernel (section 7.2.6)?
- Is Biscuit's reservation scheme effective at handling kernel heap exhaustion (section 7.2.7)?
- Can Biscuit benefit from RCU-like lock-free lookups (section 7.2.8)?
- Does Biscuit scale with a larger number of cores (section 7.2.9)?

7.2.1 Experimental Setup

With the exception of the scalability experiments, all performance experiments were run on a four-core 2.8 GHz Xeon X3460 with hyper-threading disabled and 16 GB of memory. Biscuit uses Go version 1.10. Except where noted, the benchmarks use an in-memory file system, rather than a disk, in order to stress the CPU efficiency of the kernel. The in-memory file system is the same as the disk file system, except that it doesn't append disk blocks to the in-memory log or call the disk driver. The disk file system uses a Samsung 850 SSD.

The network server benchmarks have a dedicated ten-gigabit Ethernet switch between a client and a server machine, with no other traffic. The machines use Intel X540 ten-gigabit network interfaces. The network interfaces use an interrupt coalescing period of 128 μs. The client runs Linux.

Except when noted, Biscuit allocates 512MB of RAM to the kernel heap. The reported fraction of CPU time spent in the garbage collector is calculated as $\frac{T_{gc}-T_{nogc}}{T_{gc}}$, where T_{gc} is the time to execute a benchmark with garbage collection and T_{nogc} is the time without garbage collection. To measure T_{nogc}, we reserve enough RAM for the kernel heap that the kernel doesn't run out of free memory and thus never collects. This method does not remove the cost to check, for each write, whether write barriers are enabled.

We report the average of three runs for all figures except maximums. Except when noted, each run lasts for one minute, and variation in repeated runs for all measurements is less than 3%.

Many of the performance experiments use three applications, all of which are kernel-intensive:

CMailbench CMailbench is a mail-server-like benchmark which stresses the virtual memory system via `fork` and `exec`. The benchmark runs four server processes and four associated clients, all on the same machine. For each message delivery, the client forks and execs a helper; the helper sends a 1660-byte message to its server over a UNIX-domain socket; the server forks and execs a delivery agent; the delivery agent writes the message to a new file in a separate directory for each server. Each message delivery involves two calls to each of `fork`, `exec`, and `rename` as well as one or two calls to `read`, `write`, `open`, `close`, `fstat`, `unlink`, and `stat`.

NGINX NGINX [36] (version 1.11.5) is a high-performance web server. The server is configured with four processes, all of which listen on the same socket for TCP connections from clients. The server processes use `poll` to wait for input on multiple connections. NGINX's request log is disabled. A separate client machine keeps 64 requests in flight; each request involves a fresh TCP connection to the server. For each incoming connection, a server process parses the request, opens and reads a 612-byte file, sends the 612 bytes plus headers to the client, and closes the connection. All requests fetch the same file.

Redis Redis [58] (version 3.0.5) is an in-memory key/value database. We modified it to use `poll` instead of `select` (since Biscuit doesn't support `select`). The benchmark runs four single-threaded Redis server processes. A client machine generates load over the network using two instances of Redis's "redis-benchmark" per Redis server process, each of which opens 100 connections to the Redis process and keeps a single GET outstanding on each connection. Each GET requests one of 10,000 keys at random. The values are two bytes.

	Tput	Kernel time	Live data	GCs	CPU cycles				
					GC	*Prologue*	*WB*	*Safe*	*Alloc*
CMailbench	15,862	92%	34 MB	42	3%	6%	<1%	3%	8%
NGINX	88,592	80%	48 MB	32	2%	6%	<1%	2%	9%
Redis	711,792	79%	18 MB	30	1%	4%	<1%	2%	7%

Figure 7-3: Measured costs of HLL features in Biscuit for three kernel-intensive benchmarks. *GC, Prologue, WB, Safe,* and *Alloc* show the proportion of total CPU cycles used on garbage collection, function prologues, write barriers, safety checks, and allocation, respectively. *Alloc* cycles are not an HLL-specific cost, since C code has significant allocation costs as well.

7.2.2 HLL tax

This section investigates the performance costs of Go's HLL features for the three applications. Figure 7-3 shows the results.

The "Tput" column shows throughput in application requests per second.

The "Kernel time" column shows the fraction of time spent in the kernel (rather than in user space) and that the results are dominated by kernel activity. All of the benchmarks keep all four cores 100% busy.

The applications cause Biscuit to average between 18 and 48 MB of live data in the kernel heap. They allocate transient objects fast enough to trigger dozens of collections during each benchmark run ("GCs"). These collections use between 1% and 3% of the total CPU time.

"Prologue" cycles are the fraction of total CPU cycles used by compiler-generated code at the start of each function that checks whether the stack must be expanded, and whether the garbage collector needs a stop-the-world pause. "WB" cycles reflect compiler-generated write barriers that take special action when an object is modified during a concurrent garbage collection.

"Safe" cycles reports the cost of runtime checks for nil pointers, array and slice bounds, divide by zero, and incorrect dynamic casts. These checks occur throughout the compiler output; we wrote a tool that finds them in the Biscuit binary and cross-references them with CPU time profiles.

"Alloc" cycles measures the CPU cycles spent in the Go allocator, examining free lists to satisfy allocation requests (but not including concurrent collection work). Allocation is not an HLL-specific task, but it is one that some C kernels streamline with custom allocators [13].

Figure 7-3 shows that the function prologues are the most expensive HLL feature. Garbage collection costs are noticeable but not the largest of the costs. On the other hand, section 7.2.4 shows that collection cost grows with the amount of live data, and it seems likely that prologue costs could be reduced.

Live (MB)	Total (MB)	Headroom ratio	Tput (msg/s)	GC%	GCs
640	960	0.66	10,448	34%	43
640	1280	0.50	12,848	19%	25
640	1920	0.33	14,430	9%	13
1280	2560	0.50	13,041	18%	12

Figure 7-4: CMailbench throughput on Biscuit with different kernel heap sizes. The columns indicate live heap memory; RAM allocated to the heap; the ratio of live heap memory to heap RAM; the benchmark's throughput on Biscuit; the fraction of CPU cycles (over all four cores) spent garbage collecting; and the number of collections.

7.2.3 GC delays

We measured the delays caused by garbage collection (including interleaved concurrent work) during the execution of NGINX, aggregated by allocator call, system call, and NGINX request.

0.7% of heap allocator calls are delayed by collection work. Of the delayed allocator calls, the average delay is 0.9 microseconds, and the worst case is 115 microseconds, due to marking a large portion of the TCP connection hash table.

2% of system calls are delayed by collection work; of the delayed system calls, the average delay is 1.5 microseconds, and the worst case is 574 microseconds, incurred by a *poll* system call that involved 25 allocator calls that performed collection work.

22% of NGINX web requests are delayed by collection work. Of the delayed requests, the average total collection delay is 1.8 microseconds (out of an average request processing time of 45 microseconds). Less than 0.3% of requests spend more than 100 microseconds garbage collecting. The worst case is 582 microseconds, which includes the worst-case system call described above.

7.2.4 Sensitivity to heap size

A potential problem with garbage collection is that it consumes a fraction of CPU time proportional to the "headroom ratio" between the amount of live data and the amount of RAM allocated to the heap. This section explores the effect of headroom on collection cost.

This experiment uses the CMailbench benchmark. We artificially increased the live data by inserting two or four million vnodes (640 or 1280 MB of live data) into Biscuit's vnode cache. We varied the amount of RAM allocated to the kernel heap.

Figure 7-4 shows the results. The two most significant columns are "Headroom ratio" and "GC%;" together they show roughly the expected relationship. For example,

comparing the second and last table rows shows that increasing both live data and total heap RAM, so that the ratio remains the same, does not change the fraction of CPU time spent collecting; the reason is that the increased absolute amount of headroom decreases collection frequency, but that is offset by the fact that doubling the live data doubles the cost of each individual collection.

In summary, while the benchmarks in section 7.2.2 and Figure 7-3 incur modest collection costs, a kernel heap with millions of live objects but limited heap RAM might spend a significant fraction of its time collecting. We expect that decisions about how much RAM to buy for busy machines would include a small multiple (2 or 3) of the expected peak kernel heap live data size; see section 8.1.1 for further discussion.

7.2.5 Go versus C

This section compares the performance of code paths in C and Go that are nearly identical except for language. The goal is to focus on the impact of language choice on performance for kernel code. The benchmarks involve a small amount of code because of the need to ensure that the C and Go versions are very similar.

The code paths are embedded in Biscuit (for Go) and Linux (for C). We modified both to ensure that the kernel code paths exercised by the benchmarks are nearly identical. We disabled Linux's kernel page-table isolation, retpoline, address space randomization, transparent hugepages, hardened usercopy, cgroup, fair group, and bandwidth scheduling, scheduling statistics, ftrace, kprobes, and paravirtualization to make its code paths similar to Biscuit. We also disabled Linux's FS notifications, *atime* and *mtime* updates to pipes, and replaced Linux's scheduler and page allocator with simple versions, like Biscuit's. The benchmarks allocate no heap memory in steady-state, so Biscuit's garbage collector is not invoked.

Ping-pong

The first benchmark is "ping-pong" over a pair of pipes between two user processes. Each process takes turns performing five-byte reads and writes to the other process. Both processes are pinned to the same CPU in order to require the kernel to context switch between them. The benchmark exercises core kernel tasks: system calls, sleep/wakeup, and context switch.

We manually verified the similarity of the steady-state kernel code paths (1,200 lines for Go, 1,786 lines for C, including many comments and macros which compile to nothing). The CPU-time profiles for the two showed that time was spent in near-identical ways. The ten most expensive instructions match: saving and restoring SSE registers on context switch, entering and exiting the kernel, *wrmsr* to restore the thread-local-storage register, the copy to/from user memory, atomic instructions for locks, and *swapgs*.

The results are 465,811 round-trips/second for Go and 536,193/second for C; thus C is 15% faster than Go on this benchmark. The benchmark spends 91% and 93% of its time in the kernel (as opposed to user space) for Go and C, respectively. A round trip takes 5,259 instructions for Go and 4,540 for C. Most of the difference is due to HLL features: 250, 200, 144, and 112 instructions per round-trip for stack expansion prologues, write barrier, bounds, and nil pointer/type checks, respectively.

Page-faults

The second Go-versus-C benchmark is a user-space program that repeatedly calls `mmap()` to map 4MB of zero-fill-on-demand 4096-byte pages, writes a byte on each page, and then unmaps the memory. Both kernels initially map the pages lazily, so that each write generates a page fault, in which the kernel allocates a physical page, zeroes it, adds it to the process page table, and returns. We ran the benchmark on a single CPU on Biscuit and Linux and recorded the average number of page-faults per second.

We manually verified the similarity of the steady-state kernel code: there are about 480 and 650 lines of code for Biscuit and Linux, respectively. The benchmark spends nearly the same amount of time in the kernel on both kernels (85% on Biscuit and 84% on Linux). We verified with CPU-time profiles that the top five most expensive instructions match: entering the kernel on the page-fault, zeroing the newly allocated page, the user-space store after handling the fault, saving registers, and atomics for locks.

The results are 731 nanoseconds per page-fault for Go and 695 nanoseconds for C; C is 5% faster on this benchmark. The two implementations spend much of their time in three ways: entering the kernel's page-fault handler, zeroing the newly allocated page, and returning to user space. These operations use 21%, 22%, and 15% of CPU cycles for Biscuit and 21%, 20%, and 16% of CPU cycles for Linux, respectively.

These results give a feel for performance differences due just to choice of language. They don't involve garbage collection; for that, see sections 7.2.2 and 7.2.4.

7.2.6 Biscuit versus Linux

To get a sense of whether Biscuit's performance is in the same ballpark as a high-performance C kernel, we report the performance of Linux on the three applications described in section 7.2.1. The applications make the same system calls on Linux and on Biscuit. These results cannot be used to conclude much about performance differences due to Biscuit's use of Go, since Linux includes many features that Biscuit omits, and Linux may sacrifice some performance on these benchmarks in return for better performance in other situations (e.g., large core counts or NUMA).

We use Debian 9.4 with Linux kernel 4.9.82. We increased Linux's performance by disabling some costly features: kernel page-table isolation, retpoline, address space

	Biscuit	**Linux**	**Ratio**
CMailbench (mem)	15,862	17,034	1.07
CMailbench (SSD)	254	252	0.99
NGINX	88,592	94,492	1.07
Redis	711,792	775,317	1.09

Figure 7-5: Application throughput of Biscuit and Linux. "Ratio" is the Linux to Biscuit throughput ratio.

randomization, transparent hugepages, TCP selective ACKs, and SYN cookies. We replaced *glibc* with *musl* (nearly doubling the performance of CMailbench on Linux) and pinned the application threads to CPUs when it improves the benchmark's performance. We ran CMailbench in two configurations: one using an in-memory file system and the other using an SSD file system (*tmpfs* and *ext-4* on Linux, respectively). The benchmarks use 100% of all cores on both Biscuit and Linux, except for CMailbench (SSD), which is bottlenecked by the SSD. The proportion of time each benchmark spends in the kernel on Linux is nearly the same as on Biscuit (differing by at most two percentage points).

Figure 7-5 presents the results: Linux achieves up to 10% better performance than Biscuit. The "HLL taxes" identified in section 7.2.2 contribute to the results, but the difference in performance is most likely due to the fact that the two kernels have different designs and amounts of functionality. It took effort to make Biscuit achieve this level of performance. Most of the work was in understanding why Linux was more efficient than Biscuit, and then implementing similar optimizations in Biscuit. These optimizations had little to do with the choice of language, but were for the most part standard kernel optimizations (e.g., avoiding lock contention, avoiding TLB flushes, using better data structures, adding caches).

7.2.7 Handling kernel heap exhaustion

This experiment demonstrates two things. First, that the system calls of a good citizen process do not fail when executing concurrently with an application that tries to exhaust the kernel heap. Second, that Biscuit's heap RAM reservations aren't too conservative: that the reservations allow most of the heap RAM to be used before forcing system calls to wait.

The experiment involves two programs. An abusive program repeatedly forks a child and waits for it. The child creates many non-contiguous memory mappings, which cause the kernel to allocate many heap objects describing the mappings. These objects eventually cause the kernel heap to approach fullness, at which point the out-of-memory killer kills the child. Meanwhile, a well-behaved program behaves like a UNIX mail

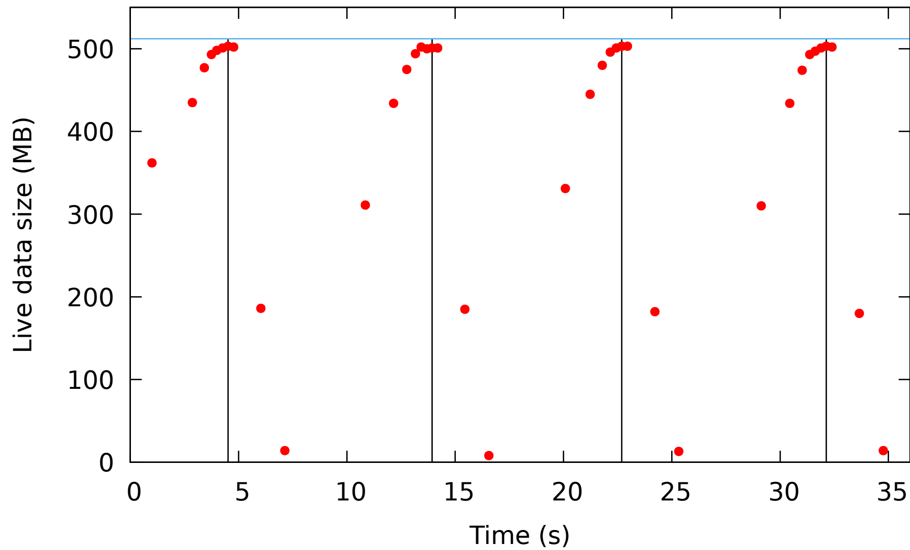

Figure 7-6: The amount of live data (in red) in the kernel heap during the first 35 seconds of the heap exhaustion experiment. The blue line indicates the RAM allocated to the kernel heap (512MB). The four vertical black lines indicate the points at which the killer thread killed the abusive child process.

daemon: it repeatedly delivers dozens of messages and then sleeps for a few seconds. This process complains and exits if any of its system calls returns an unexpected error. The kernel has 512MB of RAM allocated to its heap. The programs run for 25 minutes, and we record the amount of live data in the kernel heap at the end of every garbage collection.

Figure 7-6 shows the first 35 seconds of the experiment. Each red cross indicates the amount of live kernel heap data after a GC. The blue line at the top corresponds to 512MB. The four vertical lines show the times at which the out-of-memory killer killed the abusive program's child process.

Biscuit allows the live data in its heap to grow to about 500MB, or 97% of the heap RAM. The main reason that live data does not reach 512MB is that the reservation for the file system logger thread is 6MB, more than the thread actually uses. When the child is killed, it takes a couple seconds to release the kernel heap objects describing its many virtual memory mappings. The system calls of the good citizen process wait for reservations hundreds of thousands of times, but none return an error.

Directory cache	Tput
Lock-free lookups	15,862 msg/s
Read-locked lookups	14,259 msg/s

Figure 7-7: The performance of CMailbench with two versions of Biscuit's directory cache, one read-lock-free and one using read locks.

7.2.8 Lock-free lookups

This section explores whether read-lock-free data structures in Go increase parallel performance.

C kernels often use read-lock-free data structures to increase performance when multiple cores read the data. The goal is to allow reads without locking or dirtying cache lines, both of which are expensive when there is contention. However, safely deleting objects from a data structure with lock-free readers requires the deleter to defer freeing memory that a thread might still be reading. Linux uses read-copy update (RCU) to delay such frees, typically until all cores have performed a thread context switch; coupled with a rule that readers not hold references across context switch, this ensures safety [45, 46]. Linux's full set of RCU rules is complex; see "Review Checklist for RCU patches" [44].

Garbage collection automates the freeing decision, simplifying use of read-lock-free data structures and increasing the set of situations in which they can safely be used (e.g., across context switches). However, HLLs and garbage collection add their own overheads, so it is worth exploring whether read-lock-free data structures nevertheless increase performance.

In order to explore this question, we wrote two variants of a directory cache for Biscuit, one that is read-lock-free and one with read-locks. Both versions use an array of buckets as a hash table, each bucket containing a singly-linked list of elements. Insert and delete lock the relevant bucket, create new versions of list elements to be inserted or updated, and modify next pointers to refer to the new elements. The read-lock-free version of lookup simply traverses the linked list.[2] The read-locked version first read-locks the bucket (forbidding writers but allowing other readers) and then traverses the list. We use CMailbench for the benchmark since it stresses creation and deletion of entries in the directory cache. The file system is in-memory, so there is no disk I/O.

Figure 7-7 shows the throughput of CMailbench using the read-lock-free directory cache and the read-locked directory cache. The read-lock-free version provides an 11% throughput increase: use of Go does not eliminate the performance advantage of read-lock-free data in this example.

[2] We used Go's atomic package to prevent re-ordering of memory reads and writes; it is not clear that this approach is portable.

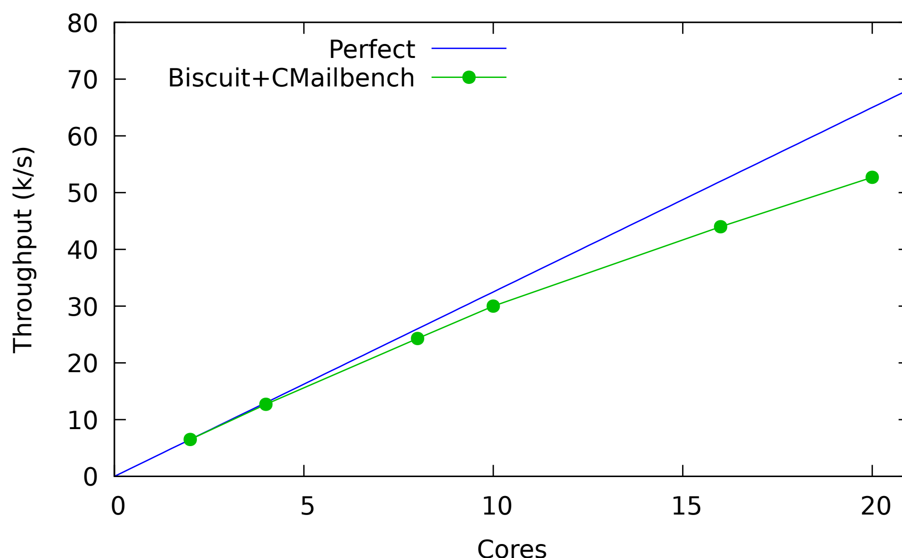

Figure 7-8: The performance of CMailbench with increasing core counts on Biscuit. The throughput units are thousands of messages per second.

7.2.9 Scalability

This section explores the scalability of the HLL through two experiments. The performance experiments in this section were run on a machine with two 2.4 GHz Xeon E5-2640 packages (two sockets with ten cores each) with hyper-threading disabled and 64 GB of memory. In these experiments, the cores used in each run of the benchmark were evenly spread between both NUMA nodes in order to keep the worst-case cost of memory access independent of the number of cores.

CMailbench scalability

The purpose of this experiment is to determine whether the HLL becomes a scalability bottleneck when running a complex and demanding application, CMailbench, with up to 20 cores. Each CMailbench server/client pair has its own copy of the binaries so that each server/client pair can `exec` them without contending on a file's reference count with other cores. Each run of CMailbench ran long enough to GC dozens of times, used either 2, 4, 8, 10, 16, or 20 cores, and we recorded throughput.

Figure 7-8 shows the results. The x-axis is the number of cores available to Biscuit and the y-axis shows the throughput of CMailbench in thousands of messages per second. The blue line shows the throughput of a perfectly scalable kernel and the green line shows the throughput of CMailbench. Figure 7-8 shows that CMailbench throughput scales with cores, achieving 92% and 81% of perfect scalability with 10 and 20 cores, respectively.

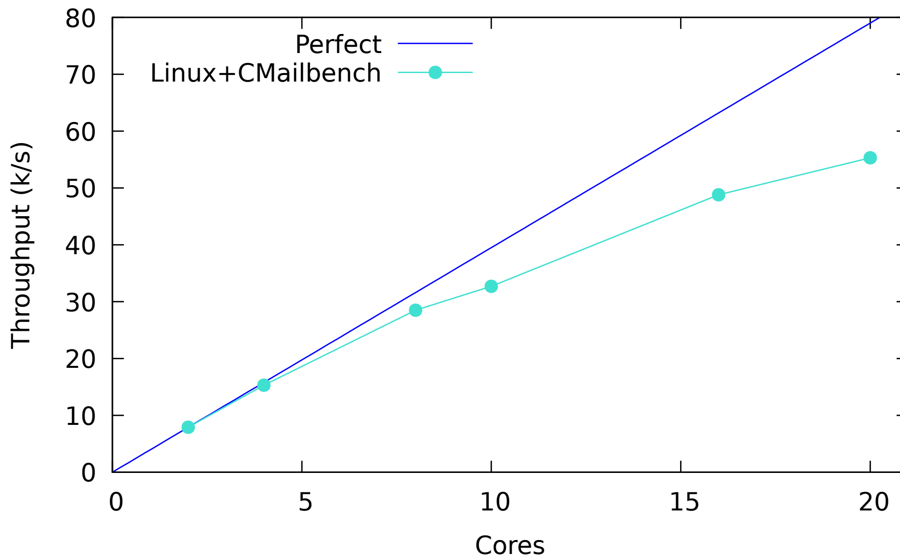

Figure 7-9: The performance of CMailbench with increasing core counts on Linux. The throughput units are thousands of messages per second.

The main reason why per-core throughput of 20 cores is less than with 10 cores is that CMailbench is not a perfectly-parallel benchmark: CMailbench causes all cores to contend on the locks protecting a few directories. The performance cost of this contention is small when using 10 cores, but becomes significant when using 16 or 20 cores.

We had to fix four serious scalability bottlenecks in Biscuit to achieve this performance. First, we modified Biscuit to, instead of broadcasting TLB shootdowns to all CPUs, send TLB shootdowns to only those CPUs which are using the updated page tables at the moment of invalidation. CMailbench causes Biscuit to frequently send TLB shootdowns because of the frequent calls to `fork`, which require modifying the permissions of user pages to convert them to or from copy-on-write pages. Second, we modified the page allocator to use per-CPU free lists before checking the global list under a lock. Third, we changed the data structure used for the process table from a native `map` protected by a lock to a hash table with per-bucket locks so that inserts and deletes on different PIDs can likely be done in parallel. Finally, we modified Biscuit to acquire the global `rename` lock (whose purpose is to prevent "orphaned loop" directories) only when moving directories, similar to ScaleFS [11]. Since CMailbench calls `rename` only on files, this modification prevents all cores from contending on this lock. Altogether, these modifications increased CMailbench throughput when using 20 CPUs by about a factor of 6.

As additional evidence that the HLL is not the main scalability bottleneck of CMailbench on Biscuit, Figure 7-9 shows the results of the same experiment when run on Linux (and the same machine). Figure 7-9 shows that CMailbench scales similarly whether

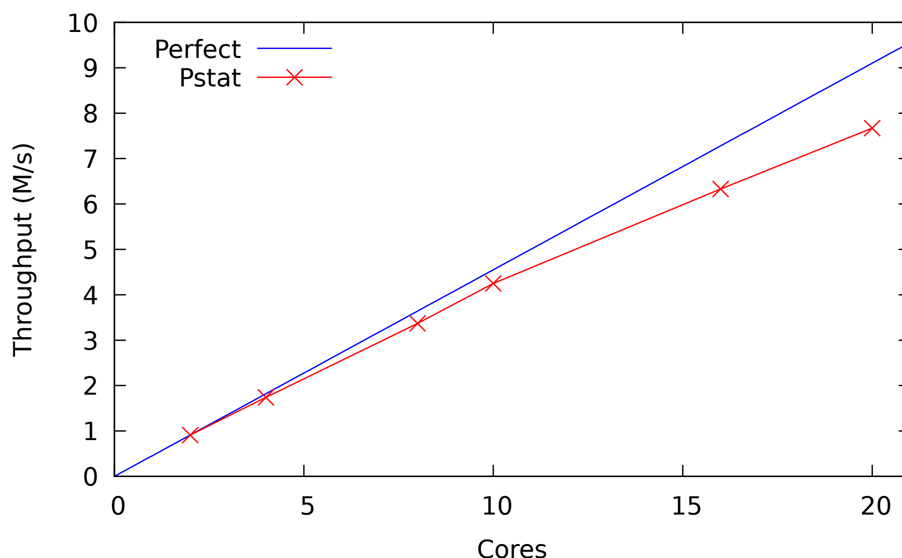

Figure 7-10: The throughput of Pstat with increasing core counts. The throughput units are millions of `stat` system calls per second.

running on Biscuit or Linux, though Linux does achieve higher throughput. Thus the main bottleneck when scaling CMailbench up to 20 cores was not due to the HLL.

GC scalability

The purpose of the next experiment is to determine whether the GC scales up to 20 cores for a GC-intensive application. The benchmark is Pstat, which scales perfectly to 20 cores when the GC is disabled. Pstat creates one process per core which calls `stat` on its own per-core file repeatedly in a loop. We artificially increased the live kernel heap data by inserting two million vnodes (resulting in 600MB of live data) into Biscuit's vnode cache and we set the amount of RAM allocated to the kernel heap to 900MB (i.e., so the ratio of live data to kernel heap RAM was was 0.66) in order to stress the garbage collector. Each run of Pstat uses either 2, 4, 8, 10, 16, or 20 cores and runs long enough so that dozens of GCs occur. With 20 CPUs, the total allocation rate is about 650MB per second.

Figure 7-10 shows the results. The x-axis is the number of cores available to Biscuit and the y-axis shows the throughput of Pstat in millions of `stat` system calls per second. The blue line shows the throughput of a perfectly scalable kernel and the red line shows Pstat throughput. Figure 7-10 shows that Pstat achieves 84% of perfect scalability with 20 cores.

2	4	8	10	16	20
24%	30%	31%	32%	35%	36%

Figure 7-11: The fraction of all CPU cycles spent on GC during execution of Pstat with different numbers or cores.

Figure 7-11 shows the fraction of total CPU cycles spent on GC during each run of Pstat with different numbers of cores. Figure 7-11 shows that the fraction of CPU cycles spent on GC increases with the number of cores, using between 24% and 36% with two and 20 cores, respectively.

The main reason why the proportion of CPU cycles spent on GC increases with more cores is that the GC progressively overloads the memory subsystem: with 20 cores, the GC spends an additional 8% of total CPU cycles stalled on RAM accesses than it does with two cores. As the number of cores increases, more cores execute the GC in parallel and the frequency of RAM accesses increases. Eventually, the frequency of RAM accesses becomes higher than what the memory subsystem can service. As a result, RAM accesses are stalled proportionally to the number of outstanding RAM accesses, which is proportional to the number of cores. Optimizing the GC for NUMA so that each core scans mostly NUMA-local live data may reduce the rate at which the CPU cycles used on the GC increases.

This experiment demonstrates that Go's GC does not scale perfectly, but nevertheless scales reasonably well, achieving 84% of perfect scalability with 20 cores.

8 Discussion and future work

Should one write a kernel in Go or in C? We have no simple answer since the choice is a trade-off, but we can make a few observations. For existing large kernels in C, the programming cost of conversion to Go would likely outweigh the benefits, particularly considering investment in expertise, ecosystem, and development process. The question makes more sense for new kernels and similar projects such as VMMs.

If a primary goal is avoiding common security pitfalls, then Go helps by avoiding some classes of security bugs (see section 7.1.2). If the goal is to experiment with OS ideas, then Go's HLL features may help rapid exploration of different designs (see section 7.1.1). If CPU performance is paramount, then C is the right answer, since it is faster (see sections 7.2.2 and 7.2.3). If efficient memory use is vital, then C is also the right answer: Go's garbage collector needs at least twice as much RAM as there is live kernel heap data to run efficiently (see section 7.2.4). Finally, if performance is merely important, consider paying the CPU and memory overhead of GC for the safety and productivity of the HLL.

An HLL other than Go might change the considerations. A language without a compiler as good as Go's, or whose design was more removed from the underlying machine, might perform less well. On the other hand, a language such as Rust [49] that avoids garbage collection might provide higher performance as well as safety, though perhaps at some cost in programmability for threaded code.

The rest of this chapter explains what we think are the main challenges and main benefits of building practical kernels in an HLL and a few possibilities for future work.

8.1 HLL kernel challenges

This section explains what we expect to be the main challenges with building practical kernels in an HLL and why.

8.1.1 GC CPU overhead

Perhaps the main downside of Biscuit is that system performance depends on an unusual and poorly-understood side-effect of application behavior: the size of live kernel heap data. The CPU overhead of GC will be significant if a single user application causes much of the kernel heap to be live most of the time. However, there are a couple reasons why the proportion of CPU cycles used by the GC is likely to be acceptably low in practice.

Kernel heaps are typically small. Kernel heap objects are usually small meta-data describing resources like files, sockets, virtual memory mappings, routing table entries, etc. The kernel heap does not contain large data items, such as user memory pages or file-cache pages. Few programs cause the kernel to accumulate millions of files, sockets, or non-contiguous virtual memory mappings. Thus the kernel heap typically uses a relatively small fraction of RAM even if user applications use many gigabytes of user memory.

To understand kernel heap sizes, we inspected four of MIT's big time-sharing machines. All four run Ubuntu Linux, had at least 79 users logged in, and had at least 800 processes with between 9 to 16 GB of total resident memory. The total kernel heap RAM (the sum of allocated and free kernel heap RAM) was less than 2GB on each machine. On the OpenBSD desktop machine on which the author wrote this dissertation, the total resident user memory is 1.8GB, but the total kernel heap RAM is less than 170MB.

One potential source of large kernel heaps is the vnode cache. Careful eviction of the vnodes may keep the number of kernel heap objects low without hurting application performance, depending on the access pattern.

If a large kernel heap is necessary, one can provision extra RAM to reduce the fraction of CPU time spent in GC. The collector only has to run when the kernel heap has no free space. Thus the amount of free heap RAM (and allocation rate) determines the frequency of GCs: doubling the amount of free heap RAM halves the frequency of GCs. So long as a machine has enough extra RAM that can be donated to the kernel heap, the GCs can be made rare enough that total CPU cycles used by GC will be low.

We suspect that dedicating extra memory to kernel heaps will often be an acceptable cost: many applications probably wouldn't be affected if the RAM available to them or the buffer cache was decreased by a few hundred megabytes.

Finally, it may be possible to further reduce the CPU overhead even when there is little free heap RAM by modifying Go's GC to be generational. Generational collection is effective at reducing GC overhead for most programs and we suspect Biscuit would benefit from it similarly.

8.1.2 GC pauses

Even if the interval between collections can be made long, the collector must eventually execute. If the collector causes kernel execution to pause for substantial periods, it could delay latency-sensitive tasks such as redrawing a moved mouse pointer or processing an urgent client request.

The largest sum of GC delays while processing a request in the NGINX experiment was 582 microseconds (see section 7.2.3). Such pauses are rare: less than 0.3% of requests spent more than 100 microseconds executing GC work.

Some applications can't tolerate even rare pauses of hundreds of microseconds, but we suspect that many can. For example, servers in one Google service had a 99th-percentile latency of 10 milliseconds [21].

8.2 HLL kernel benefits

This section summarizes some of the main benefits of using an HLL to build an operating system kernel.

8.2.1 Increased productivity

One of the main benefits of writing Biscuit in Go is the increased productivity over C. Unfortunately, we don't know a direct way of measuring productivity. Nevertheless, we believe Go significantly reduced the effort required to build Biscuit. Some of our favorite language features are GC'ed allocation, slices, `defer`, multi-value returns, closures, strings, and maps. Individually, none of these features are transformative, but together they result in significantly simpler code.

HLL features can increase productivity, but initially we weren't sure whether a kernel would be able to make good use of them. We compared the rate of use of several HLL features in Biscuit to two other large Go projects and found that Biscuit's use of most of the HLL features is in line with the other projects (see section 7.1.1).

8.2.2 Memory-safety

Manual memory management in C is error-prone and the consequences of bugs can be severe: 40 out of the 65 publicly-available, execute-code CVEs found in Linux during 2017 were due to manual memory management bugs and all of them allow an attacker to execute malicious code in the kernel (see section 7.1.2). Had this buggy code been written in Biscuit, the GC and runtime safety checks would have prevented malicious code execution in all 40 cases.

```go
func serve() {
	buf := new(request_t)
	read_next_request(buf)
	go func() {
		// log_request() occasionally blocks on IO
		log_request(buf)
	}()
	process_request(buf)
}
```

Figure 8-1: A simple case where threads share data.

8.2.3 Simpler concurrency

Garbage collection makes threaded sharing of transient heap objects particularly convenient. For example, consider the request processing code in Figure 8-1. A network server calls the `serve` function to receive and process the next request. The code calls `log_request` in a separate thread in order to prevent file writes from delaying the processing of the request. Each thread accesses `buf` while logging or processing. The GC automatically ensures that `buf` will be freed only after both threads have finished using it.

In contrast, this style of threaded programming can be awkward in C, because of the need for code that decides when the last thread has finished using the object. Consider writing Figure 8-1 in C. The C programmer would allocate `buf` via `malloc`. Neither thread could simply free `buf` before returning since the other thread may still be accessing `buf`. The programmer must delay the call to `free` until both threads have finished accessing `buf`. One solution would be to embed a reference count in `buf`, manipulated with atomic instructions. This is eminently possible in C, but requires more programmer thought than in Go, and thus more chance of error.

8.2.4 Simpler lock-free sharing

GC is convenient in the example described in the previous section, but GC is more than convenient when threads share data without locks (which is common in optimized kernels [45]) because the resulting code is significantly simpler than in C. In C, each thread must increase and decrease the corresponding reference count before and after accessing an object. Forgetting to increase or decrease a reference count will result in corrupted or leaked memory. Since threads may concurrently modify the same reference counter, all modifications must be atomic with respect to other counter accesses. Furthermore, the reference counters themselves cannot be stored in the same memory of the object which they protect, since then a thread may modify freed memory. Thus the programmer needs to find the counter belonging to each object.

The atomic operations to maintain reference counts can reduce performance. This is the main reason why Linux uses RCU [45, 46] to safely free memory that is shared among threads. RCU requires significantly fewer atomic operations and thus achieves good performance, but it is not simple to use: code which accesses memory managed by RCU must follow a list of rules [44] and be surrounded by a special prologue and epilogue. All such code cannot sleep, schedule, or block in any way, in addition to a few other rules.

For example, Linux uses RCU to safely free the objects storing per-thread metadata (`task_struct`). RCU enables readers to lookup the object corresponding to a particular PID without acquiring locks or accessing a reference counter, which increases performance. However, Linux still uses a reference counter in addition to RCU to safely free these objects. The reason is that there are two operations that may cause the object to become free (the deletion of the object from the PID map during `exit` and the last context switch away from the thread) which may occur in either order. But the object must only be freed after both operations have completed, therefore both operations decrement a reference counter to determine when the object can be safely freed. As a result, all code that accesses or wakes up a runnable thread must be careful to lookup and increase the reference counter of the per-thread object in the same RCU critical section (and eventually decrement the reference counter) to ensure the scheduler doesn't unsafely free the object out from under them.

GC makes these programming difficulties disappear. Biscuit code can freely share all heap objects among threads without worrying about when to free the objects. The reduction of programmer effort is especially evident in the case of read-lock-free data structures, which Biscuit uses in its directory cache, routing table, and network interface table. The result is high performance with less programmer effort, particularly in the directory cache.

8.3 Future work

The following are some potential areas of future work with Biscuit:

- Modify Biscuit to expand and contract the RAM used for the heap dynamically. Biscuit will have to page out user memory and cooperate with the kernel heap reservation mechanism.
- Modify the Go runtime to allow Biscuit to control scheduling policies.
- Scale Biscuit to larger numbers of cores, which likely requires that Biscuit be made NUMA aware.
- Investigate whether Biscuit's GC pauses cause problems when running demanding, latency-sensitive applications, like video games.
- Explore whether Biscuit's heap reservation scheme could simplify the implementation of C kernels.

9 Conclusions

Our subjective experience using Go to implement the Biscuit kernel has been positive. Go's high-level language features are helpful in the context of a kernel. Examination of historical Linux kernel bugs due to C suggests that a type- and memory-safe language such as Go might avoid real-world bugs, or handle them more cleanly than C. The ability to statically analyze Go helped us implement defenses against kernel heap exhaustion, a traditionally difficult task.

The dissertation presents measurements of some of the performance costs of Biscuit's use of Go's HLL features, on a set of kernel-intensive benchmarks. The fraction of CPU time consumed by garbage collection and safety checks is less than 15%. The dissertation compares the performance of equivalent kernel code paths written in C and Go, finding that the C version is about 15% faster.

We hope this dissertation helps readers to choose between C and an HLL when building operating system kernels.

9 Bibliography

[1] AdaCore. Homepage – Adacore. https://www.adacore.com/.

[2] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. Live heap space analysis for languages with garbage collection. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM)*, pages 129–138, Dublin, Ireland, June 2009.

[3] Achilleas Anagnostopoulos. gopher-os, 2019. https://github.com/achilleasa/gopher-os.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[5] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010.

[6] Godmar Back and Wilson C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Languages and Systems*, 27(4):583–630, July 2005.

[7] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (ATC)*, pages 197–210, San Diego, California, June 2000.

[8] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[9] Francisco J. Ballesteros. The Clive operating system, 2014. http://lsub.org/ls/clive.html.

[10] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, Colorado, December 1995.

[11] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–86, Shanghai, China, October 2017.

[12] Bruno Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.

[13] Jeff Bonwick. The Slab Allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, pages 87–89, Berkeley, California, 1994.

[14] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *Proceedings of the 7th International Symposium on Memory Management (ISMM)*, pages 141–150, Tucson, Arizona, June 2008.

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSYS)*, Shanghai, China, July 2011.

[16] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for OO programs. In *Proceedings of the 12th Annual International Static Analysis Symposium (SAS)*, pages 70–86, London, United Kingdom, September 2005.

[17] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, November 2003.

[18] Jonathan Corbet. The too small to fail memory-allocation rule. from https://lwn.net/Articles/627419/, Dec 2014.

[19] Jonathan Corbet. Revisiting too small to fail. from https://lwn.net/Articles/723317/, May 2017.

[20] D Language Foundation. D programming language, 2019. https://dlang.org/.

[21] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[22] Redox developers. Overview – the Redox operating system, 2019. https://doc.redox-os.org/book/.

[23] David Evans. cs4414: Operating Systems, 2014. http://www.rust-class.org/.

[24] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 81–90, Washington, DC, 2009.

[25] Charles M. Geschke, James H. Morris, Jr., and Edwin H. Satterthwaite. Early experience with Mesa. *ACM SIGOPS Operating Systems Review*, April 1977.

[26] Google. gVisor, 2019. https://github.com/google/gvisor.

[27] Google. The Go Programming Language, 2019. https://golang.org/.

[28] Richard D. Greenblatt, Thomas F. Knight, John T. Holloway, and David A. Moon. A LISP machine. In *Proceedings of the Fifth Workshop on Computer Architecture for Non-numeric Processing (CAW)*, pages 137–138, Pacific Grove, California, 1980.

[29] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 116–128, Tallinn, Estonia, September 2005.

[30] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, pages 259–270, New Orleans, Louisiana, June 1998.

[31] Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–326, San Diego, California, October 2005.

[32] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 359–373, Paris, France, January 2017.

[33] Richard Hudson. Go GC: Prioritizing low latency and simplicity. from https://blog.golang.org/go15gc, Aug 2015.

[34] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 37–49, Stevenson, Washington, October 2007.

[35] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, Washington, October 2005.

[36] NGINX inc. NGINX: High performance load balancer, web server, & reverse proxy, 2019. https://www.nginx.com/.

[37] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. The Collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM)*, pages 85–96, Beijing, China, June 2012.

[38] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference (ATC)*, pages 275–288, Berkeley, California, June 2002.

[39] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 21–26, Monterey, California, 2015.

[40] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 234–251, Shanghai, China, October 2017.

[41] Alex Light. Reenix: implementing a Unix-like operating system in Rust, April 2015. https://cs.brown.edu/research/pubs/theses/ugrad/2015/light.alex.pdf.

[42] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, Houston, Texas, March 2013.

[43] Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Technical report, IBM, 2008.

[44] Paul McKenney. Review list for RCU patches. https://www.kernel.org/doc/Documentation/RCU/checklist.txt.

[45] Paul E. McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the Linux kernel: One decade later. Technical report, 2012.

[46] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, Nevada, October 1998.

[47] MITRE Corporation. CVE Linux Kernel Vulnerability Statistics, 2018. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[48] Jeffrey Mogul. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[49] Mozilla research. The Rust Programming Language, 2019. https://doc.rust-lang.org/book/.

[50] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[51] Philipp Oppermann. Writing an OS in Rust (second edition), 2019. http://os.phil-opp.com/.

[52] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, California, March 2011.

[53] W. Michael Petullo, Wenyuan Fei, Jon A. Solworth, and Patrick Gavlin. Ethos' deeply integrated distributed types. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 167–180, San Jose, California, May 2014.

[54] Tuan-Hung Pham, Anh-Hoang Truong, Ninh-Thuan Truong, and Wei-Ngan Chin. A fast algorithm to compute heap memory bounds of Java card applets. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 259–267, Cape Town, South Africa, 2008.

[55] Dominik Picheta. Nim Programming Language, 2019. https://nim-lang.org/.

[56] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM)*, pages 39–48, Dublin, Ireland, June 2009.

[57] David Redell, Yogen Dalal, Thomas Horsley, Hugh Lauer, William Lynch, Paul McJones, Hal Murray, and Stephen Purcell. Pilot: An operating system for a personal computer. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 81–92, Pacific Grove, California, 1979.

[58] Redis Labs. Redis, 2019. http://redis.io/.

[59] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 83–90, Litchfield Park, Arizona, 1989.

[60] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 28–38, Boston, Massachusetts, June 2012.

[61] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, St. Petersburg Beach, Florida, January 1996.

[62] Warren Teitelman. The Cedar programming environment: A midterm report and examination. Technical Report CSL-83-11, Xerox PARC, 1984.

[63] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–172, Palo Alto, California, April 1987.

[64] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized live heap bound analysis. In *Proceedings of the 4th International Conference of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 70–85, London, United Kingdom, January 2003.

[65] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, Farmington, Pennsylvania, November 2013.

[66] T. Yang, M. Hertz, E. Berger, S. Kaplan, and J. Eliot B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management (ISMM)*, pages 61–72, Vancouver, BC, Canada, June 2004.