The benefits and costs of writing a POSIX kernel in a high-level language

Cody Cutler, M. Frans Kaashoek, Robert T. Morris
MIT CSAIL

Abstract

This paper 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.

The paper 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 600 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.

1 Introduction

The default language for operating system kernels is C: Linux, macOS, and Windows all use C. C is popular for kernels because it can deliver high performance via flexible low-level access to memory and control over memory management (allocation and freeing). C, however, requires care and experience to use safely, and even then low-level bugs are common. For example, in 2017 at least 50 Linux kernel security vulnerabilities were reported that involved buffer overflow or use-after-free bugs in C code [34].

High-level languages (HLLs) provide type- and memory-safety and convenient abstractions such as threads. Many HLLs provide garbage collection to further reduce programmer burden and memory bugs. It is well-known that HLLs can be used in kernels: multiple kernels have been written in HLLs, often as platforms to explore innovative ideas (§2). On the other hand, leading OS designers have been skeptical that HLLs' memory management and abstractions are compatible with high-performance production kernels [51][47, p. 71].

While it would probably not make sense to re-write an existing C kernel in an HLL, it is worth considering what languages new kernel projects should use. Since kernels impose different constraints and requirements than typical applications, it makes sense to explore this question in the context of a kernel.

We built a new kernel, Biscuit, written in Go [15] for x86-64 hardware. Go is a type-safe language with garbage collection. Biscuit runs significant existing applications such as NGINX and Redis 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 device drivers in Go: one for AHCI SATA disk controllers and one for Intel 82599-based Ethernet controllers. Biscuit has nearly 28 thousand lines of Go, 1546 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.

In most ways the design of Biscuit is that of a traditional monolithic POSIX/Unix kernel, and Go was a comfortable language for that approach. In one respect the design of Biscuit is novel: its mechanism for coping with kernel heap exhaustion. 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 that much heap. Once a system call is allowed to continue, its allocations are guaranteed to succeed without blocking. 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.

We run several kernel-intensive applications on Biscuit and measure 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 a single NGINX client request was delayed (by many individual pauses) was a total of 600 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 modify 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 compare the performance of Biscuit and Linux on our kernel-intensive application benchmarks, finding that Linux is up to 10% faster than Biscuit. This result is not very illuminating about 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.

In summary, the main contributions of this paper are: (1) Biscuit, a kernel written in Go with good performance; (2) a novel scheme for coping with kernel heap exhaustion; (3) a discussion of qualitative ways in which use of an HLL in a kernel was and was not helpful; (4) measurements of the performance tax imposed by use of an HLL; and (5) a direct Go-vs-C performance comparison of equivalent code typical of that found in a kernel.

This paper does not draw any top-level conclusion about C versus an HLL as a kernel implementation language. Instead, it presents experience and measurements that may be helpful for others making this decision, who have specific goals and requirements with respect to programmability, safety and performance. Section 9 summarizes the key factors in this decision.

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 question of the impact of language choice on kernel performance, all else being equal, has not been explored.

Kernels in high-level languages. The Pilot [44] kernel and the Lisp machine [17] are early examples of use of a high-level language (Mesa [14] and Lisp, respectively) in an operating system. Mesa lacked garbage-collection, but it was a high-priority requirement for its successor language Cedar [48]. The Lisp machine had a real-time garbage collector [5].

A number of research kernels are written in high-level languages (e.g., Taos [49], Spin [7], Singularity [23], J-kernel [19], and KaffeOS [3, 4], House [18], the Mirage unikernel [29], and Tock [27]). 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. Singularity quantified the cost of hardware and software isolation [22], which is related to the use of a HLL, but didn't quantify the cost of safety features of a HLL language, as we do in §8.4.

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 Go, Java, C#, and Cyclone [25] (and, less recently, Cedar [48] and Modula-3 [37]). Other systems HLLs are less compatible with existing kernel designs. For example, Erlang [2] 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 [13]. Biscuit's goal is efficiency for kernels without modifying Go. Kernels have additional challenges such as dealing with user/kernel space, page tables, interrupts, and system calls.

A number of new languages have recently emerged for systems programming: D [11], Nim(rod) [42], Go [15], and Rust [36]. There are a number of kernels in Rust [12, 26, 27, 28, 39, 50], but none were written with the goal of comparing with C as an implementation language. Gopher OS is a Go kernel with a similar goal as Biscuit, but the project is at an early stage of development [1]. Other Go kernels exists but they don't target the questions that Biscuit answers. For example, Clive [6] is a unikernel and doesn't run on the bare metal. The Ethos OS uses C for the kernel and Go for user-space programs, with a design focused on security [41]. gVisor is a user-space kernel, written in Go, that implements a substantial portion of the Linux system API to sandbox containers [16].

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 [26].

Concurrent garbage collectors [5, 24, 30] reduce pause times by collecting while the application runs. Go 1.10 has such a collector [21], which Biscuit uses.

Several papers have studied manual memory allocation versus automatic garbage collection [20, 52], focusing on heap headroom memory's effect in reducing garbage collection costs in user-level programs. Headroom is also important for Biscuit's performance (§5 and §8.6).

Rafkind et al. added garbage collection to parts of Linux through automatic translation of C source [43]. 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 [8] 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.

Kernel heap exhaustion. All kernels 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 [9]. 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 [10]. 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 [9, 10]), 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.

3 Motivation

This section outlines our view of the main considerations in the choice between C and an HLL for the kernel.

3.1 Why C?

A major reason for C's popularity in kernels is that it supports low-level techniques that can help performance, particularly pointer arithmetic, easy escape from type enforcement, explicit memory allocation, and custom allocators [51][47, p. 71]. There are other reasons too (e.g. C can manipulate hardware registers and doesn't depend on a complex runtime), but performance seems most important.

3.2 Why an HLL?

The potential benefits of high-level languages are well understood. Automatic memory management reduces programmer effort and use-after-free bugs; type-safety detects bugs; runtime typing and method dispatch help with abstraction; and language support for threads and synchronization eases concurrent programming.

Certain kinds of bugs seem much less likely in an HLL than in C: buffer overruns, use-after-free bugs [40], and bugs caused by reliance on C's relaxed type enforcement. Even C code written with care by expert programmers has C-related bugs [40]. The CVE database for the Linux kernel [34] lists 40 execute-code vulnerabilities for 2017 which would be wholly or partially ameliorated by use of an HLL (see §8.2).

Use-after-free bugs are notoriously difficult to debug, yet occur often enough that the Linux kernel includes a memory checker that detects some use-after-free and buffer overrun bugs at runtime [46]. Nevertheless, Linux developers routinely discover and fix use-after-free bugs: Linux has at least 36 commits from January to April of 2018 for the specific purpose of fixing use-after-free bugs.

Another area of kernel programming that would benefit from HLLs is concurrency. Transient worker threads can be cumbersome in C because the code must decide when the last thread has stopped using any shared objects that need to be freed; this is easier in a garbage collected language.

However, use of a garbage-collected HLL is not free. The garbage collector and safety checks consume CPU time and can cause delays; the expense of high-level features may deter their use; the language's runtime layer hides important mechanisms such as memory allocation; and enforced abstraction and safety may reduce developers' implementation options.

4 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 facilitate comparison. 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

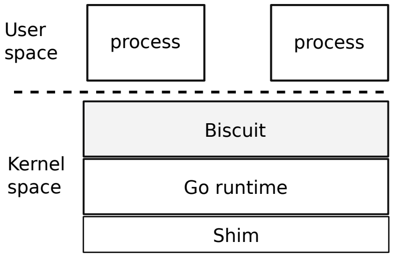

Figure 1: Biscuit's overall structure.

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.

Boot and Go Runtime. The boot block loads Biscuit, the Go runtime, and a "shim" layer (as shown in Figure 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, or in Go terminology, threads). 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.

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. "goroutine" is Go's name for a thread, and in this paper refers only to threads running inside the kernel.

Biscuit's 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.

Interrupts. A Biscuit device interrupt handler marks an associated device-driver goroutine as runnable and then returns, as previous kernels have done [35, 45]. 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.

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 execing in different processes. Biscuit uses read-lock-free lookups in some performance-critical code (see below).

Virtual Memory. Biscuit uses page-table hardware to implement zero-fill-on-demand memory allocation, copy-on-write fork, 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 contiguous memory mappings compactly, 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.

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.

Network Stack. Biscuit implements a TCP/IP stack and a driver for Intel PCI-Express Ethernet NICs in Go. The driver uses DMA and MSI interrupts. The system-call API provides POSIX sockets.

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 pages. Biscuit lacks many security features like users, access control lists, or address space randomization.

5 Garbage collection

Biscuit's use of garbage collection is a clear threat to its performance. This section outlines the Go collector's design and describes how Biscuit configures the collector; §8 evaluates performance costs.

5.1 Go's collector

Go 1.10 has a concurrent parallel mark-and-sweep garbage collector [21]. 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 cores (a "stop-the-world" pause) twice during a collection: at the beginning to enable the write barrier on all cores 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 fraction of CPU time spent collecting is roughly proportional to the number of live objects, and inversely proportional to the interval between collections [20, 52]. This interval can be made large by devoting enough RAM to the heap that a substantial amount of space ("headroom") is freed by each collection.

The Go collector does most of its work during calls to the heap allocator, spreading out this work roughly evenly among calls. Thus goroutines see delays proportional to the amount that they allocate; §8.5 presents measurements of these delays for Biscuit.

5.2 Biscuit's heap size

At boot time, Biscuit allocates a fixed amount of RAM for its Go heap, defaulting to $1/32$nd of total RAM. Go's collector ordinarily expands the heap memory when live data exceeds half the existing heap memory; Biscuit disables this expansion. The next section (§6) explains how Biscuit copes with situations where the heap space is nearly filled with live data.

6 Avoiding heap exhaustion

Biscuit must address the possibility of live kernel data completely filling the RAM allocated for the heap ("heap exhaustion"). This is a difficult area that existing kernels struggle with (§2).

6.1 Approach: reservations

Biscuit is designed to tolerate heap exhaustion without kernel failure. In addition, 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, so deadlock is avoided.

The killer thread must distinguish between good and bad citizens, since killing a critical process (e.g., `init`) can make the system unusable. 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.

The mechanisms in this section do not apply to non-heap allocations. In particular, Biscuit allocates physical memory pages 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).

6.2 How Biscuit reserves

Biscuit dedicates a fixed amount of RAM M for the kernel heap. A system call only starts if it can reserve enough heap memory for the maximum amount of *simultaneously* live data that it uses, called s. A system call may allocate more than s from the heap, but the amount over s must be dead and can be freed by the collector. This means that, even in the extreme case in which all but s of the

```
reserve(s):
    g := last GC live bytes
    c := used bytes
    n := reserved bytes
    L := g + c + n
    M := heap RAM bytes
    if L + s < M:
        reserved bytes += s
    else:
        wake killer thread
        wait for OK from killer thread

release(s):
    a := bytes allocated by syscall
    if a < s:
        used bytes += a
    else:
        used bytes += s
    reserved bytes -= s
```

Figure 2: Pseudo code for heap reservations in Biscuit.

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 not known. Biscuit maintains a conservative over-estimate of live heap data using three counters: g, c, and n. g is the amount of live data marked by the previous garbage collection. c is the total amount of reservations made by system calls that have completed. n is the total outstanding reservations of system calls that are executing but not finished. Let L be the sum of g, c, and n.

Figure 2 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 $L+s < M$, the thread reserves by adding s to n, otherwise the thread wakes up the killer thread and sleeps. When finished, a system call calculates a, the total amount actually allocated, and uses a to (partially) release any over-reservation: if a <s, the system call adds a to c and subtracts s from n. Otherwise, $a \geq s$ and the system call adds s to c and subtracts s from n.

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

If heap memory is plentiful (live data $\ll M$), the amount of live+dead data in the heap usually grows faster than L, 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 $g+n$ 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.

6.3 Static analysis to find s

We have developed a tool, MAXLIVE, that analyzes the Biscuit source code and the Go packages Biscuit uses to find s for each system call. The core challenge is detecting statically when allocated memory can no longer be live, since many system calls allocate memory for transient uses. Other challenges include analyzing loops with non-constant bounds, 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 amenable to the reservation approach, for example to avoid recursion (we changed a few functions). Two modifications were required to standard Go packages that Biscuit uses (packages *time* and *fmt*).

6.3.1 Basic MAXLIVE operation

MAXLIVE examines the call graph (using Go's *ssa* and *callgraph* packages) to detect all allocations a system call may perform. It uses escape and pointer analysis (Go's *pointer* package) to detect when an allocation does not "escape" above a certain point in the call graph, meaning that the allocation must be dead on return from that point.

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 object in the SSA 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.

6.3.2 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.

6.3.3 Kernel threads

A final area of special treatment applies to long-running kernel threads. An example is the filesystem 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 filesystem logging thread. Other kernel threads, like the ICMP packet processing thread, block and wait for heap reservations when needed and release them when idle.

6.3.4 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.

6.4 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 current implementation doesn't do this yet.

6.5 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.

7 Implementation

The Biscuit kernel is written almost entirely in Go: Figure 3 shows that it has 27,583 lines of Go, 1,546 lines of assembly, and no C.

Biscuit provides 58 system calls, listed in Figure 4. 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

Component	Lang	LOC
Biscuit kernel (mostly boot)	asm	546
Biscuit kernel	Go	
Core		1700
Device drivers		4088
File system		7338
Network		4519
Other		1105
Processes		935
Reservations		749
Syscalls		5292
Virtual memory		1857
Total		27583
MaxLive	Go	1299
Runtime modifications	asm	1,000
Runtime modifications	Go	3,200

Figure 3: 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.

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, 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.

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. 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.

8 Evaluation

This section analyzes the costs and benefits of writing a kernel in an HLL by exploring the following questions:

- 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 (§8.1).
- 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 (§8.2).
- How much performance does Biscuit pay for Go's HLL features? We measure the time Biscuit spends in garbage collection, bounds checking, etc., and the delays that GC introduces (§8.4,8.5,8.6).
- What is the performance impact of using Go instead of C? We compare nearly-identical pipe and page-fault handler implementations in Go and C (§8.7).
- Is Biscuit's performance in the same ballpark as Linux, a C kernel (§8.8)?
- Is Biscuit's reservation scheme effective at handling kernel heap exhaustion (§8.9)?
- Can Biscuit benefit from RCU-like lock-free lookups (§8.10)?

8.1 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 5 shows how often Biscuit uses Go's HLL features, and compares with two other major Go systems:

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

Figure 4: Biscuit's 58 system calls.

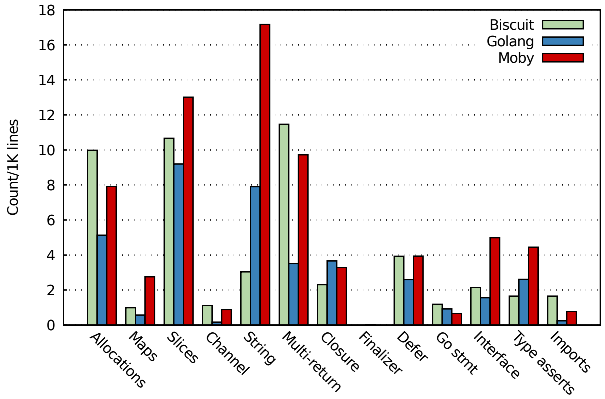

Figure 5: Uses of Go HLL features in the Git repositories for Biscuit, Golang (1,140,318 lines), and Moby (1,004,300 lines) per 1,000 lines. For data types (such as slices), the numbers indicate the number of declarations of a variable, argument, or structure field of that type.

the Golang 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 5 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. 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 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` packages 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.

8.2 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 [34]. There are 65 bugs

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

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

Figure 6: Linux kernel CVEs from 2017 that would *not* cause memory corruption, code execution, or information disclosure in Biscuit.

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 6), 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).

8.3 Experimental Setup

The performance experiments reported below 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 $\frac{O_{gc}-O_{nogc}}{O_{gc}}$, where O_{gc} is the time to execute a benchmark with garbage collection and O_{nogc} is the time without garbage collection. To measure O_{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 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 [38] (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 (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.

8.4 HLL tax

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

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

The "Kernel time" column (fraction of time spent in the kernel, rather than in user space) shows 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 time 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.

"Safety 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 time 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 [8].

Figure 7 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, §8.6 shows that collection cost grows with the amount of live data, and it seems likely that prologue costs could be reduced.

8.5 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 hashtable.

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.

8.6 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 8 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 §8.4 / Figure 7 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.

8.7 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

	Tput	Kernel	Live	GCs	GC	Prologue	WB	Safety	Alloc
		time	data		cycles	cycles	cycles	cycles	cycles
CMailbench	15,862	92%	34 MB	42	3%	6%	0.9%	3%	8%
NGINX	88,592	80%	48 MB	32	2%	6%	0.7%	2%	9%
Redis	711,792	79%	18 MB	30	1%	4%	0.2%	2%	7%

Figure 7: Measured costs of HLL features in Biscuit for three kernel-intensive benchmarks. "Alloc cycles" are not an HLL-specific cost, since C code has significant allocation costs as well.

Live	Total	Headroom	Tput	GC%	GCs
(MB)	(MB)	ratio	(msg/s)		
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 8: 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.

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.

8.7.1 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.

	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 9: Application throughput of Biscuit and Linux. "Ratio" is the Linux to Biscuit throughput ratio.

8.7.2 Page-faults

The second Go-versus-C benchmark is a user-space program that repeatedly calls `mmap()` to map 4 MB 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 userspace 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 userspace. 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 §8.4 and §8.6.

8.8 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 of §8.4. The applications make the same system calls on Linux and on Biscuit. These results cannot be used to conclude

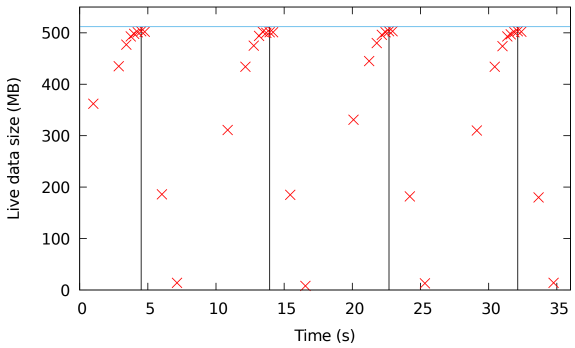

Figure 10: The amount of live data (in red) in the kernel heap during the first 35 seconds of the heap exhaustion experiment. The blue line indicates the RAM allocated to the kernel heap (512MB). The four vertical black lines indicate the points at which the killer thread killed the abusive child process.

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 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 9 presents the results: Linux achieves up to 10% better performance than Biscuit. The "HLL taxes" identified in §8.4 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).

8.9 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 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 10 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 500 MB, or 97% of the heap RAM. The main reason that live data does not reach 512 MB is that the reservation for the file system logger thread is 6 MB, 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.

8.10 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 [32, 33]. Linux's full set of RCU rules is complex; see "Review Checklist for RCU patches" [31].

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

Figure 11: The performance of CMailbench with two versions of Biscuit's directory cache, one read-lock-free and one using read locks.

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 11 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.

9 Discussion and future work

Should one write a kernel in Go or in C? We have no simple answer, 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 §8.2). If the goal is to experiment with OS ideas, then Go's HLL features may help rapid exploration of different designs (see §8.1). If CPU performance is paramount, then C is the right answer, since it is faster (§8.4, §8.5). If efficient memory use is vital, then C is also the right answer: Go's garbage collector needs a factor of 2 to 3 of heap headroom to run efficiently (see §8.6).

We have found Go effective and pleasant for kernel development. Biscuit's performance on OS-intensive applications is good (about 90% as fast as Linux). Achieving this performance usually involved implementing the right optimizations; Go versus C was rarely an issue.

An HLL other than Go might change these 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 that avoids garbage collection might provide higher performance as well as safety, though perhaps at some cost in programmability for threaded code.

There are some Biscuit-specific issues we would like to explore further. We would like Biscuit to expand and contract the RAM used for the heap dynamically. We would like to modify the Go runtime to allow Biscuit to control scheduling policies. We would like to scale Biscuit to larger numbers of cores. Finally, we would like to explore if Biscuit's heap reservation scheme could simplify the implementation of C kernels.

10 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 paper 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 paper compares the performance of equivalent kernel code paths written in C and Go, finding that the C version is about 15% faster.

We hope that this paper helps readers to make a decision about whether to write a new kernel in C or in an HLL.

Acknowledgements

We thank Nickolai Zeldovich, PDOS, Austin Clements, the anonymous reviewers, and our shepherd, Liuba Shrira, for their feedback. This research was supported by NSF award CSR-1617487.

[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.

References

[1] A. Anagnostopoulos. gopher-os, 2018. `https://github.com/achilleasa/gopher-os`.

[2] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, Sept. 2010.

[3] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *In Proceedings of the 2000 Usenix Annual Technical Conference*, pages 197–210. USENIX Association, 2000.

[5] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.

[6] F. J. Ballesteros. The Clive operating system, 2014. `http://lsub.org/ls/clive.html`.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. 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, CO, Dec. 1995.

[8] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer Conference*, 1994.

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

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

[11] D Language Foundation. D programming language, 2017. `https://dlang.org/`.

[12] D. Evans. cs4414: Operating Systems, 2014. `http://www.rust-class.org/`.

[13] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[14] C. M. Geschke, J. H. Morris, Jr., and E. H. Satterthwaite. Early experience with Mesa. *SIGOPS Oper. Syst. Rev.*, Apr. 1977.

[15] Google. The Go Programming Language, 2017. `https://golang.org/`.

[16] Google. gvisor, 2018. `https://github.com/google/gvisor`.

[17] R. D. Greenblatt, T. F. Knight, J. T. Holloway, and D. A. Moon. A LISP machine. In *Proceedings of the Fifth Workshop on Computer Architecture for Non-numeric Processing*, CAW '80, pages 137–138, New York, NY, USA, 1980. ACM.

[18] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[19] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, 1998.

[20] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM OOPSLA*, 2005.

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

[22] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 37–49, Stevenson, WA, Oct. 2007.

[23] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, Oct. 2005.

[24] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: A Wait-free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, Beijing, China, 2012. ACM.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect

of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[26] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26, New York, NY, USA, 2015. ACM.

[27] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM.

[28] A. Light. Reenix: implementing a Unix-like operating system in Rust, Apr. 2015.

[29] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. 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, TX, Mar. 2013.

[30] B. McCloskey, D. F. Bacon, P. Cheng, and D. Grove. Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors. Technical report, IBM, 2008.

[31] P. McKenney. Review list for RCU patches. `https://www.kernel.org/doc/Documentation/RCU/checklist.txt`.

[32] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU usage in the Linux kernel: One decade later. 2012.

[33] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

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

[35] J. Mogul. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX 1996 Annual Technical Conference*, January 1996.

[36] Mozilla research. The Rust Programming Language, 2017. `https://doc.rust-lang.org/book/`.

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

[38] NGINX. Nginx, 2018. `https://www.nginx.com/`.

[39] P. Oppermann. Writing an OS in Rust, 2017. `http://os.phil-opp.com/`.

[40] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[41] W. M. Petullo, W. Fei, J. A. Solworth, and P. Gavlin. Ethos' deeply integrated distributed types. In *IEEE Security and Privacy Workshop on LangSec*, May 2014.

[42] D. Picheta. Nim in action, 2017. `http://nim-lang.org/`.

[43] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise Garbage Collection for C. In *Proceedings of the 9th International Symposium on Memory Management*, ISMM '09, Dublin, Ireland, June 2009. ACM.

[44] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, 1979. ACM.

[45] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 83–90, New York, NY, USA, 1989. ACM.

[46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, 2012. USENIX.

[47] A. S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, 2008.

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

[49] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Apr. 1987.

[50] Ticki. Redox - Your Next(Gen) OS, 2017. `https://doc.redox-os.org/book/`.

[51] L. Torvalds. `http://harmful.cat-v.org/software/c++/linus`, Jan 2004.

[52] T. Yang, M. Hertz, E. Berger, S. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *ACM ISMM*, 2004.