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

CODY CUTLER, M. FRANS KAASHOEK, AND ROBERT MORRIS

Cody Cutler is a PhD candidate in computer science at MIT. Cody loves baffling bugs, performance optimization, and building systems. ccutler@csail.mit.edu

Frans Kaashoek is the Charles Piper Professor in MIT's EECS department and a member of CSAIL, where he co-leads the Parallel and Distributed Operating Systems Group (http://pdos.csail.mit.edu/). Frans is a member of the National Academy of Engineering and the American Academy of Arts and Sciences, and is the recipient of the ACM SIGOPS Mark Weiser award and the 2010 ACM Prize in Computing. He was a co-founder of Sightpath, Inc. and Mazu Networks, Inc. His current research focuses on multicore operating systems and certification of system software. kaashoek@mit.edu

Robert Morris is a Professor of Computer Science at MIT. rtm@csail.mit.edu

Biscuit is a POSIX-subset operating system kernel for x86_64 CPUs, which we wrote from scratch over the last four years. Biscuit is a bit more than a research toy. It can run Nginx and Redis with good performance and has some important operating system features, like multicore support, kernel-supported threads, a journaled file system, virtual memory, a TCP/IP stack, and device drivers for AHCI SATA disks and Intel 10 Gb network cards. Building Biscuit was a lot of fun and a lot of work.

Unlike most kernels, Biscuit is written in Go instead of C. C is the usual programming language choice for kernels because it can deliver high performance via flexible low-level access to memory and control over memory management (allocation and freeing). But C 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 [7].

High-level languages (HLLs) have the potential to eliminate or reduce the impact of some common classes of bugs, particularly those having to do with memory and type safety. HLLs can also reduce programmer effort, thanks to automatic memory management, type safety, support for abstraction, and support for threads and synchronization.

However, OS designers have been skeptical about whether HLLs' memory management and abstraction are compatible with high-performance production kernels [9, 10]. Garbage collection (GC), runtime safety checks, and abstraction all cost CPU cycles, and many suspect that the benefits may not be worth the performance cost. For example, Rust [8] is partially motivated by the idea that GC cannot be made efficient; instead, the Rust compiler analyzes the program to partially automate freeing of memory.

Whether or not to use HLLs for kernels, then, requires an investigation of their performance in that context. There has been little research exploring this question, so we set out to shed a bit more light on it.

Our first step was to build a new POSIX-subset kernel, called Biscuit, in Go. Biscuit can run many programs that also run on Linux (after recompilation), so we were able to compare total application+kernel performance for Biscuit versus Linux. We did this for the Nginx and Redis servers, both of which make intensive use of the kernel. We found that throughput on Biscuit was within 10% of throughput on Linux, though this comparison should be taken with a grain of salt: although we examined both kernels' code and numerous CPU profiles to verify that they executed the applications' system calls in nearly the same way, we cannot completely rule out the possibility that Linux's performance was understated due to having many more features than Biscuit. Nevertheless, we suspect the performance difference between the two is approximately correct. To better focus on the HLL's impact on performance, we then measured the CPU overhead of Go's HLL features while running our applications on Biscuit. The CPU overhead of HLL features was at most 15%, with GC accounting for up to 3%. We presented these results in detail at the OSDI 2018 conference [11].

Paying a performance cost of 15% for the benefits of an HLL seems worthwhile in non-performance-critical situations. Similar tradeoffs regularly occur in existing kernels; for example, the Linux kernels included in Ubuntu and Debian have several compile-time features for security and debugging enabled. These features (hardened user copy, scheduling stats, and ftrace) reduce performance (by up to 25% in one microbenchmark), but most people probably don't disable them. Go has a performance cost, but it improves both security and programmability.

Readers may wonder why we used Go instead of Rust, given that Rust has no GC and thus wouldn't pay GC's performance price. We specifically wanted a language with GC in order to explore whether GC simplifies concurrent code.

In the remainder of the article, we will discuss a few challenges faced by HLL kernels, some benefits of HLL kernels, and reflect on our experience building Biscuit.

HLL Kernel Challenges

This section discusses some common concerns about HLLs and GC in kernels, and outlines what we learned about them while building Biscuit.

A kernel in Go cannot recover from low-memory situations since Go does not expose allocation failure. Linux and the BSDs handle kernel heap RAM exhaustion ("out of memory," or OOM) by returning NULL from the allocator; the calling kernel code must detect and handle the failure. Biscuit can't do this because Go implicitly allocates and does not have a way to indicate allocation failure.

Biscuit therefore uses a different approach: each kernel operation (system call, interrupt, etc.) reserves the maximum amount of heap RAM that the operation could possibly allocate before executing the operation. If the reservation isn't immediately available, the code waits until it is, after waking a separate thread that attempts to free heap memory by evicting from caches and perhaps by killing memory-hogging processes. The reservation guarantees that all allocations made by the operation cannot fail and thus no code is needed to detect and handle their failure. Additionally, since Biscuit waits for memory before executing the operation and thus while holding no locks, this approach cannot deadlock, a problem that Linux has struggled with [2, 3]. The challenging part is deciding how much memory each operation should reserve.

Fortunately, Go was helpful in overcoming this challenge: it turns out that it is easy to statically analyze Go code. We used publicly available static analysis packages to write a tool that inspects Biscuit's source and performs an analysis similar to escape analysis. The tool does most of the work of choosing reservation sizes, with reasonably tight bounds, but some manual effort is still required.

GC will use too much total CPU. The GC must follow the pointers in all live heap objects, which typically requires a RAM fetch per object. If there are millions of objects, the total time required can be on the order of hundreds of milliseconds. However, there are a couple of reasons why the CPU cycles used by the GC in practice is likely to be acceptably low.

Kernel heaps are typically small. Kernel heap objects are usually small metadata 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 noncontiguous 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 and 16 GB of total resident memory. The total kernel heap RAM (the sum of allocated and free kernel heap RAM) was less than 2 GB on each machine. On the OpenBSD desktop machine on which the first author edited this article, the total resident user memory is 1.4 GB, but the total kernel heap RAM is less than 170 MB.

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. Therefore, 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.

GC pause times will be too long. 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.

Go uses a technique called concurrent collection to reduce collection pauses. The main idea is to split the GC work into small units and interleave them with ordinary execution. The result is that individual pauses caused by GC will last only for the duration of a unit of work. There are still two potential problems. One is that smaller units of GC work are less efficient than larger ones. The other problem is that spreading collection work out over time increases the time that *write barriers* must be active. Write barriers are code the compiler inserts before each write that perform bookkeeping if a heap object is written during a collection. Concurrent collection therefore trades decreased pause times for decreased efficiency.

We measured the pauses caused by Biscuit's GC while running a kernel-intensive server, Nginx. The maximum single pause incurred by kernel GC was 115 microseconds. A given client request, however, may be delayed by multiple individual pauses. So we also measured the total accumulated pauses during each Nginx client request and found that the maximum was 582 microseconds. Such pauses are rare: less than 0.3% of Nginx 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 [4].

The Go compiler will generate slower code than C compilers. Readily available C compilers have been optimized for decades. Go's compiler is comparatively young and must generate additional instructions for safety checks (bounds checks, nil-pointer checks, etc.) and write barriers.

We compared the performance of generated code from Go and GCC by modifying Biscuit and Linux to have near-identical code paths for two kernel-intensive microbenchmarks, pipe ping-pong, and zero-fill-on-demand page faults. We found that the Go versions were 15% and 5% slower than the C versions, respectively. The main reason pipe ping-pong is slower in Go is that it executes more instructions for safety checks and write barriers. The performance of the page fault handler in Go is closer to that of C because the generated instructions are less important: the main bottlenecks are the fundamental CPU operations of entering/exiting the kernel and copying the zero page.

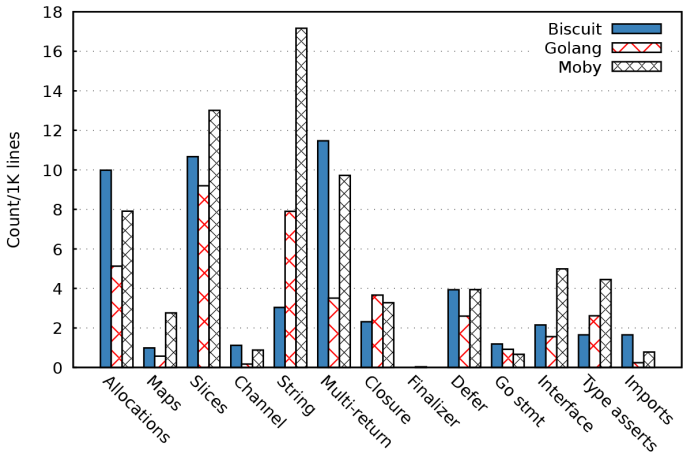

Figure 1: Uses of Go HLL features in the Git repositories for Biscuit, Golang, and Moby per 1,000 lines of code

Thus, for these two examples of typical kernel code, Go produced 5% to 15% slower executable code than C. For many situations, this is probably an acceptable price for the increased safety and programmability of Go.

HLL Kernel Benefits

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 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, Moby (https://github.com/moby/moby) and Golang (containing Go's compiler, runtime, and standard packages). Each bar in Figure 1 shows the number of uses of a particular feature per thousands of lines of code in the indicated project. Biscuit's use of most of the HLL features is in line with the other projects.

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. Had this buggy code been written in Biscuit, the GC and runtime safety checks would have prevented malicious code execution in all 40 cases.

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

Listing 1: A simple case where threads share data

Simpler concurrency. Garbage collection makes threaded sharing of transient heap objects particularly convenient. For example, consider the request processing code in Listing 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 Listing 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.

Simpler lock-free sharing. GC is convenient in the above example, but GC is more than convenient when threads share data without locks (which is common in optimized kernels [5]) 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 as the object that 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 [5, 6] to safely free memory 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 (see https://www.kernel.org/doc/Documentation/RCU/checklist.txt) and be surrounded by a special prologue and epilogue. All such code cannot sleep, schedule, or block in any way, in addition to following a few other rules.

GC makes these programming difficulties disappear. Biscuit code can share 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.

Experience and Reflections

Biscuit was a really exciting project because we had no idea what to expect of Go. Would Go make optimizing low-level code difficult or impossible? Can interrupt handlers tolerate GC pauses? Is a language runtime with its own state and invariants compatible with the degree of concurrency kernels have to handle? When we started, we expected to spend at most a couple of months on the project and quickly find an indisputable, concrete reason why a fast kernel could not be built in Go. We did not expect to end up with a kernel that runs Nginx and Redis on 10 Gb NICs with performance similar to Linux.

The focus of the project wasn't always performance. At the beginning, we hoped that Go's good support for threads and interthread communication and synchronization would allow simpler or more powerful designs for kernel code. For example, we hoped that a kernel in Go could make free use of transient worker threads to parallelize operations on multicore hardware. Unfortunately, we found few such situations. As a result, we switched goals away from exploring new kernel architectures and towards evaluating the effect of language choice and GC on performance. Thus the design of Biscuit started to become more and more traditional and similar to Linux in order to isolate performance differences due to the language as opposed to differing architectures.

Building an operating system is a huge amount of work, and it took months before Biscuit could run even the most trivial of programs. Biscuit currently has 58 system calls, and nearly all of them are required to run Nginx, Redis, and CMailbench.

As much work as it took to allow Biscuit to run complex programs, the optimization effort to run the programs well was far greater. We knew that Linux delivered good performance when we started, but we were stunned at how much effort it took to build a kernel whose performance was even within a factor of two of Linux's. Getting decent performance required implementing some interesting optimizations: mapping kernel text with

large pages to reduce iTLB misses, implementing TCP timers via streamlined timer-wheels, building a directory cache with store-free lookups that is correct with racing eviction, etc. But most were less interesting details: reducing lock contention by dedicating a NIC TX queue to each CPU instead of sharing one queue among all CPUs, avoiding unnecessary allocations or function calls, carefully batching TCP ACKs, sometimes using a linked list instead of an array, etc. Despite the effort, optimizing Biscuit's performance was the most fun part of the project and that's mainly because it honed our performance debugging skills. If we had to do it over again, we would write the code to profile via the CPU performance-monitoring counters as early as possible; those profiles were by far the most helpful tool for debugging performance problems.

We are grateful for QEMU [1], which has been a critical tool for building and testing Biscuit. We were amazed at how little work it took to get Biscuit to successfully boot on real hardware despite running it exclusively on QEMU up to that point. Real hardware did expose a few bugs in Biscuit (E820 memory map parsing, PCI interrupt routing, and the BIOS's INT 13h implementation apparently doesn't restore the interrupt flag), but it was generally painless, and that speaks to the quality of QEMU's emulation.

Our overall experience has been that building a kernel in Go was similar to building one in C: good kernel performance is more about implementing the right optimizations and less about the choice of programming language. Go didn't prevent us from implementing important kernel optimizations, which suggests that Go is a good choice for kernel programming.

Conclusion

Our 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.

We measured 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%. We compared the performance of equivalent kernel code paths written in C and Go, finding that the C version is about 15% faster.

The paper and Biscuit's code are available at https://pdos.csail.mit.edu/projects/biscuit.html.

References

[1] QEMU, the FAST! processor emulator, 2018: https://www.qemu.org.

[2] J. Corbet, "The Too Small to Fail Memory-Allocation Rule," LWN.net, December 2014: https://lwn.net/Articles/627419/.

[3] J. Corbet, "Revisiting Too Small to Fail," LWN.net, May 2017: https://lwn.net/Articles/723317/.

[4] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, February 2013, pp. 74–80.

[5] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole, "RCU Usage in the Linux Kernel: One Decade Later," 2012.

[6] P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

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

[8] S. Klabnik and C. Nichols, *The Rust Programming Language* (No Starch, 2018): https://doc.rust-lang.org/book/.

[9] A. S. Tanenbaum, *Modern Operating Systems* (Pearson Prentice Hall, 2008), p. 71.

[10] L. Torvalds, On C++, January 2004: http://harmful.cat-v.org/software/c++/linus.

[11] C. Cutler, M. F. Kaashoek, R. 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 '18),* pp. 89–105: https://www.usenix.org/system/files/osdi18-cutler.pdf.