Extracting parallelism in OS kernels using type-safe languages

Cody Cutler M. Frans Kaashoek Robert Morris Nickolai Zeldovich
MIT CSAIL
{ccutler, kaashoek, rtm, nickolai}@csail.mit.edu

Abstract

Operating system kernels are rife with potential concurrency, but exploiting this parallelism requires significant effort from programmers that write kernel code in C: the language provides little help for creating transient threads or packaging up arguments in closures, and fine-grained concurrency forces the programmer to carefully reason about what memory might be used by each thread and when it can be freed. This position paper argues that OS kernels should be written in a type-safe garbage-collected language, such as Go, which provides simple abstractions for threads and closures, as well as garbage-collection. We describe a number of opportunities for taking advantage of fine-grained parallelism in an OS kernel, illustrated by examples from our Go-based kernel, BISCUIT, challenges that must be addressed for this approach to be practical, and a preliminary performance evaluation.

1. Introduction

Operating system kernels are filled with performance opportunities that can be exploited by concurrency: background and speculative I/O, differing priorities for different incoming packet streams, and splitting up the work of expensive calls like `fork` and `exec` over multiple cores. A particularly streamlined style of programming for these situations creates transient threads as needed to perform specific tasks, allowing concurrency or parallelism to be expressed at fine granularity. However, this style is difficult in languages without garbage collection, because the threads complicate the programmer's reasoning about when data structures can be freed. This in turn has caused kernels written in C, such as Linux, to make limited use of transient threads, and to create many specialized mechanisms to manage concurrent activities.

Inspired by the example of the Firefly and Taos [13, 16], we believe that a kernel could expose and exploit much more concurrency if it were written in a language with support for threads, closures, and garbage collection. Such an arrangement would allow free use of transient threads to create concurrency, to improve the structure of concurrent kernel code, to unify existing ad-hoc techniques, and to allow concurrency where it is currently prohibitively complex to manage.

To explore the potential benefits of structuring kernels with free use of concurrency, we propose to build a traditional monolithic OS kernel in a garbage-collected language with thread support (Go [2] in our initial prototype). We believe that a monolithic kernel, rather than a microkernel, is the right approach for this project: we are looking for opportunities to exploit concurrency, and the more substantial services the kernel provides, the more likely we are to find concurrency.

We expect to address three challenges in this project, as follows. First, we need to find situations where threads and garbage collection can simplify implementations of existing parallel and concurrent techniques, or where free use of transient threads can enable improved software structure.

Second, there are likely to be performance problems when managing a large number of threads in the kernel. Although languages like Go have gotten better at handling many threads, performance problems can still arise [1]. In many situations a closure encapsulating the concurrent computation should be queued for a worker thread pool, or the number of transient threads should be chosen based on current core utilization.

Finally, we are depending on garbage collection, but garbage collection is often time-consuming. One significant concern is stop-the-world pauses during garbage collection, which may be problematic in an OS kernel that has to service periodic events like TCP packet acknowledgments or device interrupts. Some languages, such as Rust [3], have decided that garbage collection cannot be made efficient, and instead give the programmer explicit control over what memory is managed "by hand" and what memory is managed by different garbage collectors. Is such explicit control really necessary?

To address these questions, we have started to design and implement a type-safe monolithic kernel, BISCUIT, in Go.

In the rest of this paper, we lay out the opportunities for aggressive threading in OS kernels, and describe some of our initial experiences in exploiting fine-grained parallelism in BISCUIT.

2. Opportunity for parallelism

We believe that language support for fine-grained parallelism will lead to more natural designs for a number of OS kernel subsystems, as follows.

Existing uses in the Linux kernel. The Linux kernel developers already try to parallelize some aspects of kernel execution with a wide range of mechanisms, including work queues, tasklets, softirqs, bottom half handlers, timers, and so on. Each of these mechanisms are widely used; just as an example, Linux uses work queues for disk block I/O, cryptographic operations, ACPI callbacks, PCI devices, etc.

All of these are effectively threads but have different APIs that are tailored to specific use cases. Furthermore, using each of these mechanisms requires the programmer to package up their computation into a callback function with all state explicitly passed as arguments in a data structure that has to be manually allocated and freed.

We expect that a unified language mechanism will lead to much cleaner designs and more pervasive use of this programming pattern. Being able to use language-supported closures and threads will help the programmer to encapsulate the state for a computation and to create transient threads even for small tasks.

Packet processing. Many network workloads are easy targets for parallelism, such as IP packet forwarding, or TCP processing for different connections. Current Linux kernel implementations try to parallelize some packet processing with the help of multiple NIC hardware queues, but we believe language support for parallelism would lead to cleaner software structures.

The network stack currently performs processing at all layers for an incoming packet on the same core where the NIC initially delivered the packet. Splitting up this processing (e.g., performing cryptographic operations on a dedicated core, or re-distributing packets after extracting them from an IP tunnel) currently requires the developer to write explicit scheduling code in the network stack. Structuring the software around threads would simplify these kinds of scheduling decisions.

The current Linux network stack is also designed as a state machine, for maximum performance. However, a more natural structure might be to create a separate thread per TCP connection, with TCP connection state managed as variables on that thread's stack. This approach might lead to a simpler implementation that could better exploit multicore hardware.

System calls from a single process. As demonstrated by FlexSC [15], a single application can take advantage of other cores executing its system calls in parallel. In Linux, this required an explicit mechanism to send system calls to other cores, and process them there. With easy support for spawning a transient thread within a system call, implementing FlexSC-like optimizations would be simply a scheduling decision.

Parallelism within a system call. Today's operating systems typically do not create additional parallelism within a system call, under the assumption that it's the application developer's job to run a sufficient number of processes to keep all cores busy. However, this makes it difficult for applications with few threads to exploit a many-core processor.

On the other hand, there are plenty of opportunities for parallelism within a system call, especially for some of the complex system calls specified by POSIX. For example, the `fork` system call needs to copy the entire file descriptor table and the entire process address space. For a large process, each of these tasks could be significant. Given a convenient way to create transient threads, the kernel developer could create a separate thread to copy each virtual memory region, so that the work of copying a large address space can be parallelized over many cores. By creating parallelism within a system call, this reduces the burden on application programmers from creating enough parallelism to keep the OS busy on all cores.

There are many other examples in POSIX, such as the `exec` system call for a large binary, which involves loading many segments and pages from the ELF executable, and the `read` and `write` system calls for large buffers, which can farm out the work of copying a large buffer across many cores.

Speculative execution. Given language support for parallelism, it may be easy for kernel developers to implement speculative operations that execute in parallel with the "main" code, such as prefetching data, pre-populating page table entries, and so on.

If the language support were to include the ability to cancel threads or make their side-effects conditional on a later check, this would allow even more aggressive speculative execution, such as processing network inputs in parallel with checking signatures or checksums, or even executing code in parallel with flushing data to disk as in Speculator [12].

3. Exploiting parallelism

We present three examples which demonstrate some of the benefits of fine-grained parallelism with garbage collection in the BISCUIT kernel: IPSec packet transmission, the POSIX `read` system call, and simplified RCU protecting the routing table.

3.1 IPSec packet transmission

Figure 1 displays the IPSec transmission code. A thread is created for each packet-sized unit of the data to be sent.

```
func ipsec_send(so socket, p []byte) {
    done := make(chan int)
    ps := so.packetsize
    nthreads := len(p)/ps + 1

    for i := 0; i < nthreads; i++ {
        s := i*ps
        e := (i+1)*ps
        if e > len(p) {
            e = len(p)
        }
        go encrypt_send(so, p[s:e], done)
    }

    for i := 0; i < nthreads; i++ {
         <- done
    }
}

func encrypt_send(so socket, p []byte,
                  done chan int) {
    buf := encrypt(p)
    so_send(so, buf)
    done <- 1
}
```

Figure 1. Parallelism in the IPSec layer

These threads each encrypt their own allotment of the data and send it to the underlying network device. `ipsec_send` waits for all spawned threads to send their encrypted packets before returning.

This simple design is natural: both stages of packet processing are written in a separate function. This design is also powerful: if there are idle CPU cores, `ipsec_send` will take advantage of the available cores to parallelize the encryption of the data. Writing code that makes such use of transient threads is easy in a type-safe language with support for light-weight threads.

Consider `ipsec_send` if it were written in C. Since starting and synchronizing with transient threads is more cumbersome in C, it is more likely that the programmer would implement `ipsec_send` in a way that serializes the encryption and transmission of the packets unnecessarily.

3.2 Parallel read system call

Figure 2 shows example code for a parallel `read` system call specialized for the case in which the data is already in the buffer cache. The arguments to `sys_read` are a file descriptor and a slice[1] into which the data should be copied. A thread is created for each block to be copied. Each thread locates the buffer caching the desired block and copies it.

[1] a "slice" is a Go type that can be thought of as an array with corresponding length

```
func sys_read(fd int, p []byte) int {
    done := make(chan int)
    // update fd metadata...
    cnt := fd_size(fd)
    if len(p) < cnt {
        cnt := len(p)
    }
    nthreads := cnt / BLKSIZE
    for i := 0; i < nthreads; i++ {
        s := i*BLKSIZE
        e := (i+1)*BLKSIZE
        go parcopy(fd, s, p[s:e], done)
    }
    left := cnt % BLKSIZE
    if left != 0 {
        t := cnt - left
        from := buffer_get(fd, t)
        for i, c := range from {
            p[t + i] = c
        }
    }
    for i := 0; i < nthreads; i++ {
        <- done
    }
    return cnt
}

func parcopy(fd int, offset int, to []byte,
             d chan int) {
    from := buffer_get(fd, offset)
    for i, c := range from {
        to[i] = c
    }
    done <- 1
}
```

Figure 2. Example parallel `read` system call

The final partial block is copied without creating a thread. Finally, the main thread waits for the worker threads to complete.

One desirable feature of this implementation of `read` is that if the size of the data to read is larger than a few blocks and CPU cores are available, the data will be copied to the destination buffer in parallel. A large, parallel copy will take advantage of the RAM bandwidth of modern machines which is unlikely to be saturated by a single CPU core [9].

An implementation of parallel `read` in a language without good support for transient threads would be considerably more awkward.

3.3 Read-Copy Update

Read-copy update (RCU) [11] is a technique for read-lock-free parallel data access. In languages without garbage

```
var route_table *Table

func route_get(dst addr) *route {
    return route_table.lookup(dst)
}

func route_insert(r *route, dst addr) {
    rtlock.Lock()
    defer rtlock.Unlock()

    newrt := copy_table()
    newrt.insert(dst) = r

    // make new table available
    route_table = newrt
}
```

Figure 3. A read-lock-free routing table lookup, along with routing table update, illustrating an RCU-like technique that is simplified by having garbage collection.

collection, RCU requires a lock-free mechanism for reliably determining whether any reader may hold a reference to an old version of the data structure in order to safely free the old data structure.

Figure 3 shows example code demonstrating how RCU can be simplified with garbage collection. Readers of a routing table perform the route lookup directly using the `route_table` pointer. To update the routing table, writer threads must serialize access with a lock. Once the lock has been acquired, the routing table is copied and the new route is inserted into the new copy. Finally, the `route_table` pointer is updated to reference the new routing table.

This example RCU code is simple because the garbage collector takes care of reclaiming the old copy of the routing table after updates instead of the programmer. In a C implementation, the programmer must at some point free the stale version of the routing table once it is known that no readers will attempt to access a stale copy.

4. Evaluation

In this section we describe our preliminary performance evaluation of BISCUIT– we describe several experiments using microbenchmarks. The goal is to answer the following questions:

- is BISCUIT's performance competitive with a kernel written in C?
- are there performance penalties for using many threads?
- how much can garbage collection degrade performance?
- what performance benefits can we achieve by exploiting fine-grained parallelism?

In the first experiment, we compare against xv6 [14], a kernel written in C. xv6 and BISCUIT are practically identical in terms of features. All experiments, using BISCUIT or xv6, are run on the same machine with 8 logical CPUs (4 cores) and 12 GB of memory. The block cache capacity for BISCUIT and xv6 is set to 1024 to minimize the number of disk seeks required, making the microbenchmarks CPU bound so that we can compare performance of both kernels by comparing elapsed execution time of the microbenchmarks.

4.1 Smallfile microbenchmark

This experiment helps answer our first two evaluation questsions ("is BISCUIT's performance competitive with a kernel written in C?" and "are there performance penalties for using many threads?").

For this experiment we run a program (called "smallfile") on both BISCUIT and xv6 with a single CPU and record the wall-clock time of the program's execution. Smallfile performs many file system related system calls (using the POSIX interface i.e. *open, write, read*, etc) in order to test the performance of BISCUIT and xv6. Precisely, smallfile first iteratively creates, writes one byte to, and closes 100 separate files in a few different directories ("create" phase). Smallfile then opens, reads, and closes all 100 files ("read" phase). Finally, smallfile unlinks all the files ("unlink" phase).

Smallfile stresses the file system and exposes the overhead of system call handling (receiving a CPU trap, handling it, and returning to the user program). Because BISCUIT's file system creates a thread for each i-node, BISCUIT will create on the order of 100s of threads: one for each distinct file and one for each directory containing each file. Thus this microbenchmark also stresses BISCUIT's handling of many threads.

Figure 4 shows the results. The unit in the "cycles" columns is millions of cycles. While the "read" and "unlink" phases have similar times, BISCUIT's "create" phase is almost 20% faster than xv6's. The reason BISCUIT's "create" phase is faster is that, for each file, BISCUIT allocates a file, file descriptor, and inode object on the heap while xv6 allocates said objects by iteratively searching a static array for each object type. Despite BISCUIT allocating these objects in the heap, garbage collection is not costly during the smallfile microbenchmark: BISCUIT garbage collects 100s of times but the total wall-clock time of garbage collection pauses amount to less than 0.1% of total execution time.

This experiment shows two things: first, that for some workloads, BISCUIT has comparable performance to a kernel written in C. Second, that BISCUIT can manage 100s of threads without suffering significant performance degradation.

Smallfile phase	xv6's cycles	BISCUIT's cycles
Create	2,066	1,742
Read	36	37
Unlink	1,014	956
Total	3,117	2,735

Figure 4. Smallfile microbenchmark results. The units are millions of cycles.

Number of kernel objects	Millions of cycles/fork
.1 million	56.5
1 million	67.7
10 million	80.2
50 million	87.9

Figure 5. BMGC microbenchmark results. The units are millions of cycles.

4.2 BMGC microbenchmark

The purpose of this experiment is to demonstrate the impact on performance BISCUIT's garbage collection can have in a worst-case scenario (our third evaluation question). We run a program, BMGC, several times using a single CPU on BISCUIT. BMGC does the following: first, BMGC uses a special system call which causes the kernel to create and populate a hash table where the number of objects are specified by the special system call's argument. BMGC then forks 500 times (the child process immediately exits). Finally, BMGC outputs only the elapsed wall-clock time taken to fork 500 times (i.e. the elapsed time of the special system call is not included). We vary the number of objects added to the kernel hashtable each run.

Although BISCUIT uses copy-on-write fork, the *fork* syscall requires a significant amount of allocation since page tables must be allocated for the child process. Furthermore, both the parent and child processes will allocate a few more pages to replace the copy-on-write pages as they fault on them. Since BMGC causes many large allocations, BISCUIT will be forced to garbage collect often. However, total heap size remains stable since each child process frees its page tables by exiting immediately. Thus allocations are satisfied by reclaiming free memory, not by using new pages. We vary the number of objects added to the hash table by the special system call to increase garbage collector work – the garbage collector must trace every live object once per garbage collection.

Figure 5 shows the results. The time to fork increases as the number of live kernel objects increases. A fork takes 55% longer when hash table contains 50 million objects than when the hash table has only 100 thousand objects. Informal experiments indicate that OpenBSD on a commodity laptop allocates approximately 375 thousand objects from pools in steady state. Thus 50 million objects is a reasonable worst-case amount of live kernel objects.

4.3 Parallel fork microbenchmark

To answer our final evaluation question ("what performance benefits can we achieve by exploiting fine-grained parallelism?"), we implemented a parallel version of the *fork* system call. The parallel version of fork creates a thread for each page in the page map. Each thread then copies the page table entries to the new page and insert a reference to the new page into the new page tables. We run this microbenchmark with eight CPUs to observe the performance increase due to parallelism.

We test the parallel fork with a program that fills it address space with 4GB of memory and then forks 500 times and outputs the elapsed wall-clock time necessary to complete all 500 forks. The program allocates 4GB of memory into its address space in order to ensure that the fork system call takes up the vast majority of the CPU time executing the program. In the non-parallel version of fork, each fork takes 525 million cycles while each parallel fork takes only 70 million cycles to complete, yielding a speedup of 7×. Thus fine-grained parallelism can achieve significant speedups.

5. Discussion

While a coding style that makes use of many transient threads may simplify some kernel subsystems, heavy use of threads is likely to incur performance costs. The main cost is the CPU time spent on thread creation, scheduling, and context switching. It seems likely that under high load thread creation will need to be adaptively turned into queuing of closures for thread pools. A closure-based solution is much easier to write in a type-safe language with garbage collection than in C where the programmer has to manage callbacks by "hand".

Garbage collection (GC) is particularly desirable when using closures and transient worker threads, which may hold references to data that should be freed *after* the last relevant thread exits.

The price of this convenience is GC pauses. Some kernel work is latency sensitive and a GC pause may prevent the kernel from meeting deadlines. For example, a network card interrupt signalling a full receive buffer may not be serviced quickly enough if a GC occurs during or near the time that the interrupt is raised and thus packets may be dropped, reducing the performance of the receiving application.

In order to reduce GC pauses in BISCUIT, it may be possible to modify Go's garbage collector to better suit the kernel environment. For example, instead of tracing the table of open network connections (which may contain millions of entries) during every GC, the garbage collector could cache the identities of pages which contain at least one object describing an open connection during some previous GC and only trace network connection objects that have been recently added. Tracing the entire open network connections table would then become a last resort when system memory

is critically low. Thus most GCs would be likely to skip the tracing of the network connection table and therefore have reduced tracing time.

6. Related work

The usefulness of threads for structuring and parallelism has long been appreciated, and many useful design patterns are known [6, 17].

The Firefly [16] project explored use of a high-level garbage-collected language (Modula-2+) for the operating system on a multi-processor. The kernel, the file system, the window system, and many other services used threads extensively both for structuring and for parallelism [13]. This idea is worth revisiting now because multi-core hardware is more common and because the performance characteristics of hardware have changed significantly (CPUs are now orders of magnitude faster than memory, in contrast to Firefly's hardware).

Another line of investigation has been the use of high-level languages to improve security or secure extensibility in operating system kernels; SPIN [5] and Singularity [7] pursue these ideas. However, neither SPIN nor Singularity focused on the opportunities for fine-grained parallelism.

Developments in garbage collection for interactive and real-time systems [4, 8, 10] reduce collection pause times by performing most of the collection while the application runs. These collectors have short pause times, and might be appropriate for use in a kernel.

7. Conclusion

Building a monolithic kernel in a type-safe, garbage collected language using fine-grained parallelism seems likely to yield new insights into structuring kernels for concurrency. We have discussed several possibilities for how kernels may benefit from fine-grained parallelism and presented three examples from the BISCUIT kernel that demonstrate some of these benefits: design simplification, exploiting idle CPUs, and efficient use of RAM bandwidth. We also presented a preliminary performance evaluation showing that a kernel written in a high level language can achieve performance comparable to kernel written in C for some workloads. Garbage collection is critical to this programming style since it enables easy data sharing between transient worker threads.

References

[1] runtime: limit number of operating system threads. from `https://github.com/golang/go/issues/4056`, .

[2] The go programming language. from `https://golang.org/`, .

[3] The rust programming language. from `https://www.rust-lang.org/`.

[4] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978. ISSN 0001-0782.

[5] 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*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[6] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 94–105, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8. doi: 10.1145/168619.168627. URL `http://doi.acm.org/10.1145/168619.168627`.

[7] 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.

[8] 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.

[9] Y. Mao, C. Cutler, and R. Morris. Optimizing RAM-latency dominated applications. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, Singapore, July 2013.

[10] 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.

[11] 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.

[12] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, Oct. 2005.

[13] S. Owicki. *Experience with the Firefly Multiprocessor Workstation*. Digital. Systems Research Center, 1989.

[14] R. M. Russ Cox, Frans Kaashoek. Xv6, a simple Unix-like teaching operating system. Technical report. URL `http://pdos.csail.mit.edu/6.828/2012/xv6.html`.

[15] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, Oct. 2010.

[16] C. P. Thacker and L. C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 164–172, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-8186-0805-6. doi: 10.1145/36206.36199. URL `http://dx.doi.org/10.1145/36206.36199`.

[17] M. Vandevoorde and E. Roberts. Workcrews: An abstraction for controlling parallelism. *International*

Journal of Parallel Programming, 17(4):347–366, 1988. ISSN 0885-7458. doi: 10.1007/BF01407910. URL http://dx.doi.org/10.1007/BF01407910.