An Attempt at Reducing Costs of Disk I/O in Go

Sean Wilson, Riccardo Mutschlechner

{swilson, riccardo}@cs.wisc.edu

July 9, 2017

Abstract

Concurrency is a hard-to-tame beast. Golang, or Go, is a programming language created by Google which offers easily navigable abstractions of concurrency through lightweight threads (Goroutines) to ease parallel programming difficulty. In order to support these Goroutines, Go uses a runtime to multiplex Goroutines onto OS threads. In order to facilitate disk I/O, Go keeps a pool of I/O threads around to improve CPU utilization while threads block during I/O. In our work we explore an alternative solution to this pool of threads. We propose an implementation where a single thread would receive, coalesce, and perform requests for disk I/O without the need to create a pool of threads specifically for I/O. This implementation would expose the same disk I/O interface to the user (Read, Write, Open, Close). In order to accomplish this we attempt to use Linux Asynchronous I/O (AIO) system calls to execute disk operations without blocking. While we are unable to see improved performance, we believe this is due to the limited support for AIO provided by the Linux kernel. In this paper, we present our motivation (section 1), provide background information necessary to understand our work (section 2), describe our implementation (section 3), and demonstrate or results (section 4). Finally, since we were unable to achieve our original goal due to limited kernel support, we discuss options for future work (section 5).

1 Motivation

Concurrency is difficult and tedious to program correctly. There are many existing solutions to ease programmer woes, however, many of these paradigms are not natively ingrained into their language. Each paradigm: `OMP`, `CILK`, `Pthreads`, and `C++11 Threads` are attached to an existing language, and each acts as a bandage. Programmers are not limited to a single implementation of threads in each of these contexts. This can lead to messy and buggy code as the number of programmers and size of a project increases due to conflicting programmer perspectives and composability issues. [1]

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! ");
   printf("It’s me, thread %ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: ");
     printf("creating thread %ld\n", t);
     rc = pthread_create(&threads[t],
                         NULL,
                         PrintHello,
                         (void *)t);
     if (rc) {
       printf("ERROR %d\n", rc);
       exit(-1);
       }
     }

   /* Let all pthreads finish. */
   pthread_exit(NULL);
}
```

(a) Pthreads Version of Hello World

```go
package main

import "fmt"

func PrintHello(tid int) {
    fmt.Printf("Hello World! ")
    fmt.Printf("It’s me, thread %d!\n", tid)
}

func main() {
    numThreads := 5

    for i := 0; i < numThreads; i++ {
        fmt.Printf("Creating thread %d\n", i)
        go PrintHello(i)
    }
}
```

(b) Go Version of Hello World

Figure 1: Threaded Hello World Example

In 2012 the Go programming language version 1.0 was released. Go addresses the problem seen in `OMP` and other such paradigms. Go has engrained parallelism deep into the syntax of the language. Spawning a thread of execution[1] is as

[1]not necessarily an OS thread

simple as writing `go functionName` In Figure 1, we compare the difference between C and Go implementations of a multithreaded ”hello world”. By line count alone, clearly the Go version wins.

$Pthreads$ and simple synchronization primitives are, in essence, the ”assembly language” of multi-threaded programming. Many paradigms have tried to abstract these primitives away with various frameworks and runtimes. However, none have established themselves as an authoritative approach as they have lacked true native language support. Go, with its easy-to-use, and most importantly, native, concurrency model offers a simple and built-in solution to parallel programming. Leveraging multicore systems through parallel programming is key to continuing to see Moores law trends in program speedup for years to come. We believe Go does this very well, and we are interested in helping push the boundaries on the efficiency of thread scheduling within Go itself. In general, we are very interested in Go as a stripped down systems language, while still enabling easy usage for higher-level programming such as building a REST/HTTP endpoint - something most would not dream of doing in C - as performant and clean as it is.

Since we have seen a large uptake in the usage of Go in production systems, and we believe that there is a need for a simple to use, performant, and native parallel programming language we started to explore Go more deeply.[2]

2 Background

Before we we introduce the problem we tried to solve, we should provide some background on the Go scheduler. A thread (specifically, a lightweight process) in Go is known as a Goroutine, or G. A kernel thread is known as an M. These $G's$ are scheduled onto $M's$, in what is known as a $G : M$ threading model, or more commonly known as the $M : N$ threading model, user space threads, or green threading model. [2]–[4] This threading model enables easy to use, lightweight threads in Go. In order to facilitate this threading model, there exists a runtime within the Go language which must schedule its own G green threads onto the M kernel threads. Green threads provide the benefit of reduced overheads present in kernel threads such as the cost of entering the kernel to perform context switches, wasted memory when reserving a whole page for stack, and green threads introduce faster alternatives to `futex_wait`.[3] On top of this multiplexed thread model, Go implements work stealing to improve thread utilization. [2], [3] If a thread, M_0, blocks in order to execute a $syscall$ another OS thread, M_1, (without any Goroutines in its work queue) will attempt to take Goroutines off of M_0's work queue, and M_1 will start to execute them.

The Go runtime has seen a few major overhauls since its 1.0 release . The original runtime lacked many performance optimizations which have since been added. One of the most extensive changes to the runtime since Go's conception

[2]See [D] for an up to date list of Go usage in production servers.

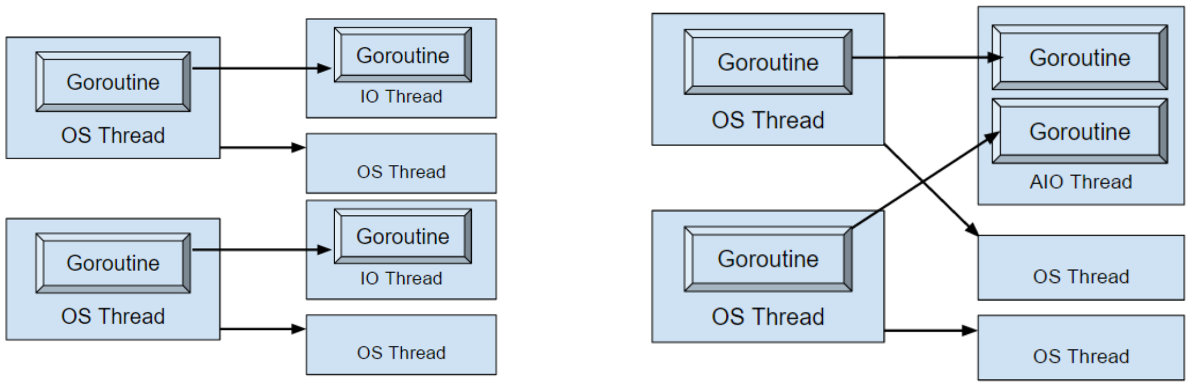
[3]Green threads can be woken by the user space runtime system rather than trapping into the kernel.

was designed by Dmitry Vyukov. [5], [6] Dmitry proposed the addition of a P struct which are meant to represent a core of the processor. By adding this additional abstraction, the Go runtime could be more aware of system resources. In order to execute Goroutines on its work queue, an M must hold aP. $P's$ are stolen from $M's$ by others $M's$ but, in order to reduce thrashing of program contexts, thus improving cache performance, $P's$ will not be stolen if there are idle $M's$ without $P's$. Lastly, more preference is given to reschedule $G's$ onto the same M. Increased locality between $M's$ and $G's$ results in improve affinity with physical processors, which benefits cache performance. This change to the Go runtime improved utilization of system resources and overall performance. However, the Go runtime is still very young and there are still plenty of opportunities for improvement.

Currently, Go's lightweight threads suffer from the same problem as many other languages which support green threads. [4], [7] When a thread executing on a Goroutine tries to write to the disk using the `write()` system call (or any other blocking syscall) the kernel thread will be descheduled by the OS. There is no way for the Linux kernel to know that the thread might still be utilized.[4] As seen in Figure 2a, in order to continue using all processor cores, the Go scheduler will keep a pool of extra threads specifically for I/O. Before entering the system call the Go runtime must first wake up one of its I/O threads and then execute the system call on said thread. This means more overhead is attached to executing system calls and more resources are wasted by keeping around thread contexts which are not always being used. This problem is exacerbated by the fact that Go will now create another M struct for each of these additional threads.

In order to reduce costs of executing disk I/O in Go, we explored an alternative to Go's I/O thread pool. Rather than allowing a thread to block in the kernel while it executes a disk read which we call Blocking I/O (BIO), we propose a scheduler (pictured in Figure 2b) which will receive requests from Goroutines to execute I/O. The requesting Goroutines will block until the scheduler has completed their request, but the M that Goroutine is blocking on will take another G from its work queue (or steal one) and continue execution. In order to perform disk operations and continue to receive requests, our scheduler will utilize Linux Asynchronous I/O (AIO) system calls to batch requests and allow the kernel to process them in the background.

[4]Much earlier work such as Scheduler Activations addresses this idea. [8]



(a) Go's I/O scheduler diagram

(b) Our I/O scheduler diagram

Figure 2: Scheduler implementations

Since AIO system calls are rarely used, and the reader may be unfamiliar with their usage, we would like to introduce semantics of their use by comparing them to BIO. When a programmer wishes to write to a disk on a traditional $N : N$ threading model, she will likely use the POSIX system call write. This will enter the OS kernel and lead to the kernel thread being descheduled, therefore halting program execution on said thread until the data has been written to the disk (or disk cache). To improve thread utilization a programmer might instead use the asynchronous Linux system call `io_submit`. This system call will still enter the kernel, however, instead of blocking execution of the thread by putting it to sleep while the write is executed, the kernel returns execution to the programmers thread and completes the write in the background. When the kernel is finished writing or fails to write, the thread will receive an interrupt from the kernel causing the users program to trap to a signal handler. In this signal handler the user can specify what must be done now that the write has completed. Using signals allows a programmer to utilize their threads resources to the fullest. An alternative to signal handling is to poll for completion. This approach allows for simpler reasoning about program execution but can lead to wasted cycles due to spinning.

```
// Open file for reading
int file = open("aio-example.c", O_RDONLY, 0);

// Request io_context handle from the kernel
io_context_t ctx;
memset(&ctx, 0, sizeof(ctx));
io_setup(1, &ctx);

// Fill in struct info for a pread request
struct iocb iocb_request;
io_prep_pread(&iocb_request, file, &buffer, SIZE_TO_READ, 0);

// Submit our AIO request

io_submit(ctx, 1, &iocb_request);

// Ask for one and only one completed AIO event
struct io_event e;
struct timespec timeout;

io_getevents(ctx, 1, 1, &e, &timeout);

// Cleanup
io_destroy(ctx);
close(file);
```

Figure 3: Example of a Single AIO request

Linux features its own non-standard API for disk AIO. In order to perform AIO the following steps must be taken: First, we call `io_setup()` which is passed the max number of concurrent requests we wish to handle. `io_setup()` returns an `io_context` which we will use to request future AIO operations (similar to a file handle). Second we prepare AIO specific struct that we will need to keep track of in order to check on our return values later on. Next, we call `io_prep_pread()` or `io_prep_pwrite()`, given the context object from above, an `IOCB` struct (which will get populated by this call), as well as a file descriptor, and input or output buffer respectively. After we do this, we must submit the IO request, using `io_submit()`, which takes the context as well as the `IOCB` from above with the actual syscall metadata. Finally, we must poll using our IO context above, specifically with the `io_getevents()` syscall. This will populate a `ioevent` struct with the relevant data: return values, errors, and data. This is not incredibly complex, however it is not as trivial as simply calling `open()`, followed by `read()` or `write()`. There is also a much more state to keep track of, rather than a simple file descriptor, we need to keep track of which AIO request buffer is tied to which `io_context` Furthermore, with asynchronous code it is difficult to reason about program correctness and order, and it usually means more code and longer debug cycles.

3 Implementation

Our first step after collecting background information, was to make the AIO system calls, structures, and flags accessible in Go. This took us quite some time and a lot of digging.[5] We originally planned to use Go's interface for communicating with C, `cgo`, which would have allowed us to import C header's for the AIO system calls directly. However, calling into C code from Go adds various issues such as allocating a whole C thread stack, complicating garbage collection, and moving the executing C code to its own thread. [9] With all these complications, using `cgo` did not seem to be a good option.To avoid the overheads induced with `cgo` usage, we decided to add AIO system calls directly into the Go compiler's `syscall` package. [A]

With the system calls now exposed through our modded compiler, we started to design our library. We decided to implement our scheduler as a simple library rather than placing it in the Go runtime itself for two reasons. First, there is no difference in the produced executable between a library within the Go compiler, and one imported from another source. Second, time constraints meant it was more important to get a working library, than to modify the compiler further. We know how important it is to a stable API, so we decided to keep the same file I/O API exposed through the BIO scheduler. Calls such as `Write()` and `Read()` in our AIO scheduler keep the same blocking semantics of BIO (they wait until completion or error to return), but `Write()` and `Read()` now send respective requests to the thread our AIO scheduler is running on through a Gochannel.[6]

One of our biggest challenges with the project was simply trying to track down a coherent explanation for single-file AIO. Many online resources for "Linux AIO" actually refer to POSIX AIO.[7] Since POSIX AIO uses BIO and a pool of I/O threads (the exact problem we tried to address in Go), it did not suit our needs. Before we started work on the request handler we finished minimal working examples first in C and then in GO.

With all the groundwork complete, we started construction of our lightweight I/O scheduler library [B]. The request handler ran on a single Goroutine that was constantly locked to its own thread, rather than some set number of I/O Goroutines waiting around, as shown in Figure 2a. Our implementation, as a diagram, is shown in Figure 2b. Once the scheduler receives a new operation, it queues it up for asynchronous processing by the OS. When the scheduler is not currently receiving any new operations (which it prioritizes), it checks to see if the previously submitted AIO operations have completed, and if so, marks them as such.

Once a basic implementation of our scheduler was complete we made various optimizations to obtain acceptable performance with AIO. The first optimization is a context manager which would manage `io_context`'s - we very

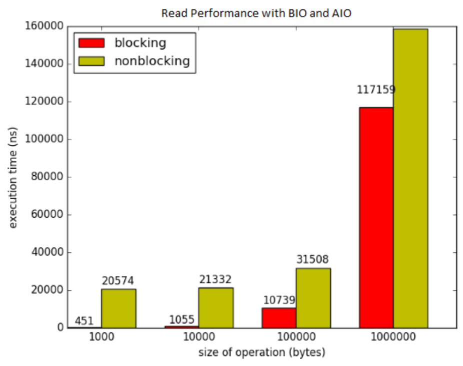
[5]Strangely enough, not many people add new system calls to Go everyday.
[6]Go channels are Go's implementation of CSP channels see [F] for an example.
[7]The top two responses for "linux aio" on Google link to POSIX examples [H], [G]

quickly learned that generating one context per AIO operation dominated the runtime, and this optimization helped significantly. Our other main optimization was the ability to coalesce and batch AIO operations into single AIO system calls. Rather than simply submitting a single request and then waiting for that request to complete before servicing any additional operations, we now were able handle multiple in flight operations at the same time.

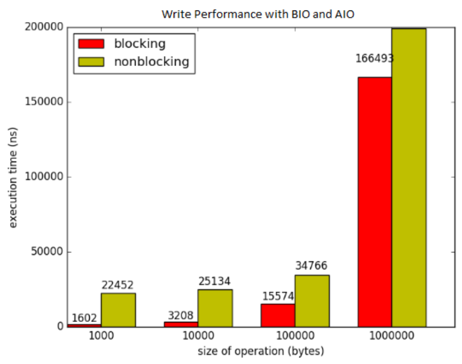
In order to test our scheduler and facilitate collection of performance results we created a testbench application [B] which swapped between the blocking and nonblocking I/O schedulers. The testbench would spawn up to a given number Goroutines that would then call into the scheduler under test with a read or write request. In order to actually track performance we create an tracing library which would log timing information in memory until test completion, then once testing completed, the trace would output to a file for processing. With this tracing information we were able to recognize areas for optimization. Our traces also helped us realize one of our key results: AIO is actually implemented to be blocking due to missing functionality in Linux.

4 Results

Using our tracing library (described in Section 3) we observed the time taken to return success from reads and writes using BIO. We compare these times to those times observed using our AIO scheduler in Figure 4. These results were taken from runs of our testbench where two Goroutines at a time were either executing BIO system calls using Go's old implementation (blocking) or requesting our AIO scheduler to perform the IO request for it (nonblocking). We decided to use two threads since this allowed us to observe performance gains obtained through batched requests, but limits the variance which might occur from context switches between multiple threads.



(a) Reads, Execution Times



(b) Writes, Execution Times

Figure 4: Execution Times for AIO v. BIO

The first thing we noticed was that our nonblocking approach performs significantly worse than the traditional blocking approach when executing smaller reads and writes. Our original explanation for this was that our implementation was adding too many overheads to the AIO system calls, and this was leading to performance issues that could be recovered with an improved implementation. As mentioned in the discussion of our Implementation, (Section 3) we tried to address some of these overheads by batching requests and reusing `io_contexts`. However, after these improvements were made, we could no longer able think of any changes which might lead to better performance. We also recognized that viewing return times of reads and writes is not the best indicator of performance. To get a sense of thread utilization, we need to see how much time is spent spinning and/or blocking in system calls. Thus, we decided to trace our system more deeply.

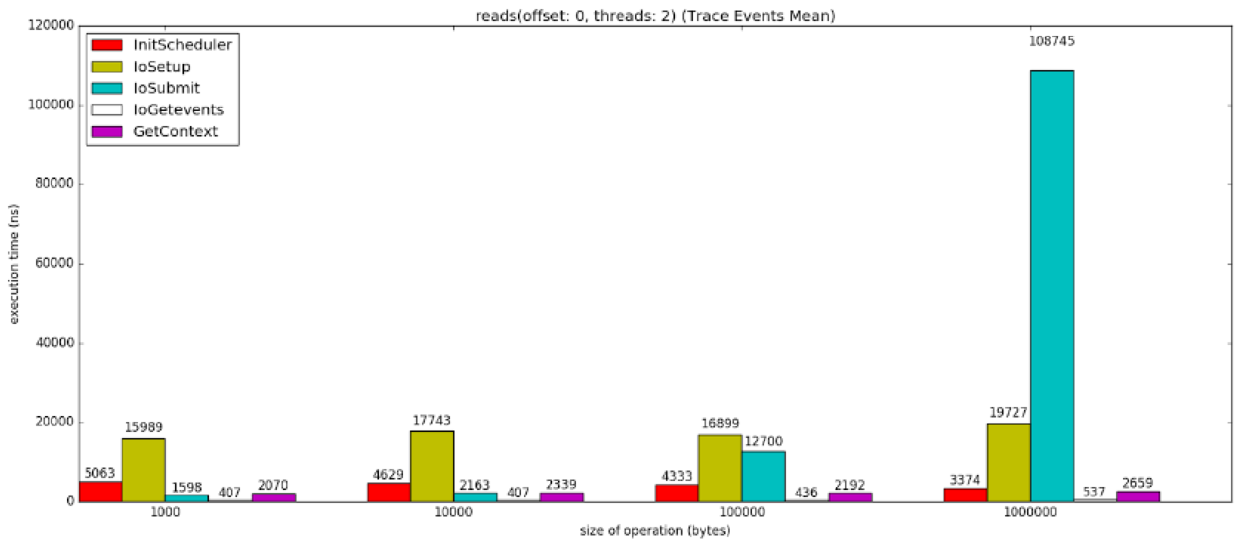


Figure 5: Reads, Detailed Trace

Because our tracing library logs information in memory without performing IO until all traces have been gathered, we were able to leverage the library for very fine grained tracing. In order to better understand where we might find additional opportunities for optimization we traced and monitored the execution times of all the system calls used when performing disk AIO. In Figure 5 we again have chosen to run our test suite using two Goroutines, and again, we vary the size of requests from 1,000 to 1,000,000 bytes. We see that the time it takes to run IoSubmit grows proportional to the size of the requested reads, whereas none of the other system call execution times change. In the context of asynchronous IO, it is perplexing that any system call should longer as the size of the operation is increased. We would hope that AIO would block the same amount of time for small requests as large, since the majority of the work should be done in the background (i.e. asynchronously). But, from our results, this clearly was not the case.

We discovered that AIO support in the Linux kernel is, and always has been extremely limited since it was introduced in the 2.5 Kernel. [10], [11] Linux AIO support only exists for files opened with the `O_DIRECT` flag. No vanilla Linux kernel to date (up to 4.9) has support for performing AIO through Linux's Virtual File System (VFS). [12] Instead, when `IoSubmit` is called on a file which must Go through VFS, the system call blocks until the operation is completed. This means what should actually be an asynchronous system call is now turned into an even slower blocking system call since there is an even longer code path taken. On top of this longer system call, we still are maintaining much more state, and we need to call `IoGetevents` to discover our ”asynchronous” system call has been completed.

Opening files with `O_DIRECT` means that access to files now goes directly to disk. This implies that subsequent disk operations on the same block will perform poorly since we will no longer be using kernel disk caching mechanisms. Even worse, reads and writes must be aligned to disk block boundaries and a multiple of disk block size. What this means for our work is that even if we were to ignore the performance implications of opening with `O_DIRECT`, the user would would not get the same reads and writes committed to their files. If a user were to try and write a single character to a file opened with `O_DIRECT`, once closed the file would contain that single character with 511 null bytes following.[8] Based on our original goal of modifying IO in Go in a user transparent manner, direct disk access is not an option, neither performance nor read/write semantics match Go’s current IO API. Go users expect the traditional, cached (highly performant), byte granular operations provided by the current API.

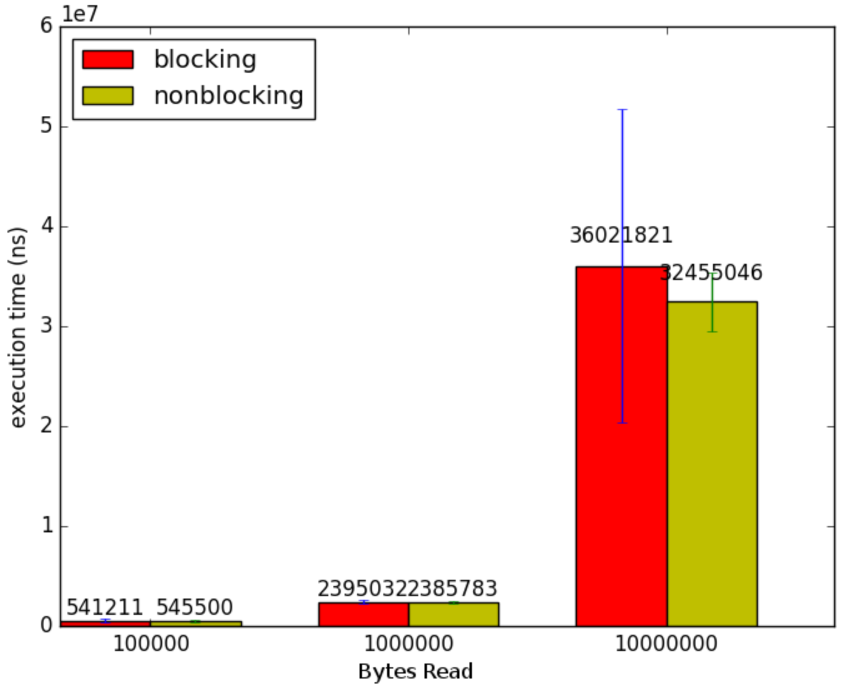


Figure 6: Reads, Preliminary O_DIRECT Execution Times

Due to time constraints we were unable to fully implement support for `O_DIRECT` file access within our test bench (since it complicates usage so much). However, we were able to gather results for single threaded access to a file with

[8]assuming a disk block size of 512 bytes

various read sizes. In Figure 6 we show early results comparing our nonblocking scheduler to Go's blocking approach for variable read request sizes using direct access.[9] These preliminary results are encouraging

To summarize our results, we were unable to improve the performance of scheduling disk IO requests in Go. We believe this is not due to the costs introduced by our AIO scheduler but is due to non-existent kernel support for AIO VFS operations. The restriction to direct disk access means we are unable to modify Go in a transparent manner, direct access would introduce massive performance decreases on re-access of data, and users would not be able to modify files a byte granularity.

5 Future Work

While our scheduler is not an improvement for the average user, we see that there are possible uses to explore in future work. With the current state of AIO in Linux, we see the possibility of our scheduler acting as a library for database disk IO. The `O_DIRECT` flag itself is used most often in databases since they manage their own cache and try to assert as much control on the system as possible. [13] We believe our scheduler could see success as an IO wrapper for applications such as databases, where the programmer understands the performance implications of using `O_DIRECT`.

In addition to a database library, we also see the possibility of layering a disk caching library on top of our scheduler in order to continue using the existing AIO in the Linux kernel while avoiding the uncached performance hits. However, there are quite a few issues with this approach. First, each Go program will now be managing disk caches so they each will be wasting their own program memory space. Second ever time a program starts, it will take time for the disk cache to repopulate. General purpose disk caching is better executed globally in the kernel, so adding caching on top of our scheduler is also not a general purpose solution.

Since user space solutions do not appear to be general purpose we turn to adding kernel support. We have uncovered multiple kernel patches which add support for VFS AIO to Linux. [11], [14], [C] We believe that while these patches may be useful for future testing of our implementation, they are not worth further exploration. Linus Torvalds has responded negatively to multiple patch requests to AIO, and if we were to go further down this route, our solution would still not be widely available to users. Expecting users of Go to patch their kernels is not practical, so exploring these existing patches would be imprudent. Rather than patching what Linus believes to be a broken system (the AIO system calls), he prefers to create a generalized asynchronous system call interface. [14] Users of this interface might then fill out structs which request multiple different system calls to be executed asynchronously. Since this is the most likely channel for bringing AIO support into the mainline Linux kernel, we believe this is the best way to proceed in future endeavors.

[9]Read requests in Figure 6 are slightly larger than their axis labels in order to align them to disk sectors

6 References

[1] Edward A. Lee. “The Problem with Threads”. In: *IEEE Computer Society* 18.5 (May 2006), pp. 33–42.

[2] Daniel Morsin. *The Go scheduler.* 2013. URL: `https://morsmachine.dk/go-scheduler` (visited on 2016-12-19).

[3] Neil Deshpander, Erica Sponsler, and Nathaniel Weiss. *Analysis of the Go Runtime Scheduler.* URL: `http://www.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf` (visited on 2016-12-19).

[4] The Rust Authors. *Green Crate Documentation.* 2016. URL: `https://doc.rust-lang.org/0.11.0/green/` (visited on 2016-12-19).

[5] Dmitry Vyukov and Martin Konecek. *Goroutines vs OS threads - Golang Forums.* Nov. 8, 2013. URL: `https://groups.google.com/forum/#!topic/golang-nuts/j51G7ieoKh4/` (visited on 2016-12-19).

[6] Dmitry Vyukov. *Scalable Go Scheduler Design Doc.* May 2, 2012. URL: `https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit` (visited on 2016-12-19).

[7] C. Wikstrom, J. Armstrong, R. Virding, et al. “Concurrent Programming in Erlang”. In: (1996).

[8] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, et al. “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”. In: *ACM Transactions on Computer Systems* 10.1 (Feb. 1992), pp. 53–79.

[9] Dave Cheney. *cgo is not Go.* Jan. 18, 2016. URL: `https://dave.cheney.net/2016/01/18/cgo-is-not-go` (visited on 2016-12-19).

[10] Suparana Bhattacharya, Steven Pratt, Badari Pulavarty, et al. “Asynchronous I/O Support in Linux 2.5”. In: *Ottawa Linux Symposium* (2003), pp. 351–366.

[11] Suparna Bhattacharya. *Linux Asynchronous I/O Design: Evolution & Challenges.* 2008. URL: `http://ftp.naist.jp/pub/linux/kernel/people/suparna/aio-linux.pdf` (visited on 2016-12-19).

[12] Jonathan Corbet. *Fixing asynchronous I/O, again.* Jan. 13, 2016. URL: `https://lwn.net/Articles/671649/` (visited on 2016-12-19).

[13] Linux Torvalds. *O_DIRECT performance impact on 2.4.18.* From the fa.linux.kernel mailing list. May 2002. URL: `http://yarchive.net/comp/linux/o_direct.html`.

[14] Linux Torvalds and Benjamin LaHaise. *aio: add support for async openat().* From the Linux kernel mailing list. Jan. 2016. URL: `https://lwn.net/Articles/671657/`.

7 Appendix

[A] Modified Go Compiler: http://github.com/spwilson2/cs758-project-compiler

[B] Go AIO Scheduler and Testbench: https://github.com/spwilson2/cs758-project

[C] AIO Submitted Patch git://git.kvack.org/ bcrl/aio-next

[D] Go in Production: https://github.com/golang/go/wiki/GoUsers

[E] AIO Example: https://github.com/spwilson2/cs758-project/blob/master/docs/Presentations/aio-example.c

[F] Gochannels: https://gobyexample.com/channels

[G] AIO Manpage: http://man7.org/linux/man-pages/man7/aio.7.html

[H] Stackoverflow Incorrectly Answered: http://stackoverflow.com/questions/6637191/linux-aio-not-posix-examples