A Language Comparison Between Parallel Programming Features of Go and C

Skyler Hawthorne
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
skyler.hawthorne@sjsu.com

ABSTRACT

In 2009, Google released the first open-source version of their new compiled programming language Go. Go's new and unusual approach to facilitating parallel programming aims to make it much simpler to use than most other languages, namely C. This paper will examine and compare different aspects of each language's approach to, and ability to facilitate, parallel programming, including: ease of use, compile time, performance and efficiency, fine-grained control, and proneness to bugs such as deadlocks and race conditions.

1. INTRODUCTION

Google has created a new compiled programming language called Go. It was invented in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, and first released as open source in 2009 ([1] §1, [2] "What is the status of the project?"). Their main goal was to design a language that would make life easier for software engineers ([1] §1). Go aims to be as easy to use as a scripting language, but as fast as a compiled language ([2] "What is the purpose of the project?"). It features automatic memory management (garbage collection), type safety, a built-in suite of developer tools, and most importantly, its built-in concurrency primitives. Go takes a new and unusual approach to concurrency which makes it much simpler to use than C. The traditional approach to safe multithreading, as in C, is to use synchronization primitives, such as semaphores, locks, and condition variables. Go uses a different model, inspired by the formal language Communicating Sequential Processes (CSP);[1] the mantra of Go's memory model is: "*Do not communicate by sharing memory; instead, share memory by communicating.*"[3]

This paper will compare different aspects of C and Go's approach to, and ability to facilitate, parallel programming, including: ease of use, compile time, performance and efficiency, fine-grained control, and proneness to bugs such as deadlocks and race conditions. There are several C libraries which facilitate multithreaded programming, and since comparing C's approach with all libraries would be infeasible, this paper will only discuss the approach with the POSIX Threads (or `pthreads`) library, since it represents and showcases the most traditional and standard approach to managing multithreaded applications.

[1] The official web site distributes an ebook of a description of the language: http://www.usingcsp.com/

2. OVERVIEW OF GO

In order to fully understand Go's approach to concurrency, it is necessary to cover some of the fundamentals of the language. Go includes a few built-in primitives: slices, maps, and channels. The latter is specifically designed to facilitate concurrency through Go's shared memory model via goroutines.

2.1 Goroutines

The basic thread of execution in Go is called a *goroutine*, so named because the Go team felt the existing terms (threads, coroutines, processes, etc.) convey inaccurate connotations ([5] "Goroutines"). Goroutines have a simpler model. They are simply functions executing in the same address space. According to Golang.org's document "Effective Go,"

> [They are] lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.
>
> Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management ([5] "Goroutines").

The traditional approach to thread management can indeed be cumbersome. For example, in C with `pthreads`, to create a handful of threads which execute in parallel, the programmer has to do something like this:

Code Listing 2.1-1. Managing threads in C.

```
int i, *ret, *args;
pthread_t thd_handle[NUM_THREADS];

for (i = 0; i < NUM_THREADS; i++) {
    pthread_create(&thd_handle[i], NULL,
        SomeFunction, (void*)args);
}

for (i = 0; i < NUM_THREADS; i++) {
    pthread_join( thd_handle[i],
                  (void**)&ret );
    // do something with ret
}
```

The programmer must explicitly store all threads in a variable, and then wait for each individual thread to exit, doing something

with its return value after each successful termination. Also, because `pthreads` must take untyped (void) parameters, the data may have to be casted to and from its actual type multiple times, while making sure pointers are dereferenced appropriately—giving us such unwieldy code as `(void**)&ret`.

Go's approach is much simpler. To launch a goroutine, the programmer simply puts the keyword `go` in front of a function call. For example,

```
go list.Sort()
```

will launch a goroutine which sorts `list` concurrently, without waiting for it to finish. In this trivial example, though, we don't see how to handle return values or shared data synchronization. In order for goroutines to communicate, Go makes use of primitives called *channels*.

2.2 Channels

Go's approach to concurrency is designed to avoid explicit synchronization by the programmer, which can be difficult to manage. Traditionally, synchronization of data occurs through the use of primitives such as semaphores, locks, and condition variables. Typically, the programmer must protect shared data and critical code blocks from hazards by using something like a mutex variable with a condition variable.

For example, with the `pthreads` library in C, to synchronize a multithreaded program that fits the producer–consumer model,[2] one first has to set up a synchronized data structure which encapsulates both the payload and its synchronization locks:

Code Listing 2.2-1. Mutex Encapsulation

```
typedef struct {
    pthread_mutex_t    mutex;
    pthread_cond_t     cv;
    int                data;
} flag;

flag ourFlag = {
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER,
    0
};
```

Then, any time the programmer wishes to perform operations on the data, and then have an independent thread wait for changes to that data before proceeding, it is necessary to explicitly lock the variable, perform the operations, and signal the condition to allow the waiting threads to execute.

Code Listing 2.2-2. Mutex/condition variable example 1.

```
int main() {
    ...
    // create worker threads
    ...
    pthread_mutex_lock(&ourFlag.mutex);
    ourFlag.data = someValue;
    pthread_cond_broadcast(&ourFlag.cv);
    pthread_mutex_unlock(&ourFlag.mutex);
    ...
}
```

[2] For a description of the producer–consumer problem, see the Wikipedia article.

Similarly, within the worker thread, one must wait on the mutex lock, then wait on the condition variable before doing something with the changed data.

Code Listing 2.2-3. Mutex/condition variable example 2.

```
void* workerThread(void* args) {
    ...
    pthread_mutex_lock(&ourFlag.mutex);
    pthread_cond_wait(&ourFlag.cv,
        &ourFlag.mutex);
    // do something with ourFlag.data
    pthread_mutex_unlock(&ourFlag.mutex);
    ...
}
```

In order to avoid data hazards and/or race conditions, access to the data must be explicitly managed by the programmer by bundling the synchronization primitives with the payload, and then acquiring and releasing locks and signaling to the other threads when they may continue. The use of *semaphores* can make this a bit simpler, but they are limited when dealing with multiple consumer threads, so condition variables are the best general solution ([6] p. 132). The explicit management required by this approach is exactly the kind of manual tracking that Go was designed to avoid.

Go takes a different approach to the problem of thread communication and synchronization through the use of built-in primitives called channels. A *channel* is defined by the Go Language Specification as a primitive which "provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type."[4] They work in a manner analogous to Unix pipes—data values are written to them (*sent*) and read from them (*received*) in a first-in, first-out (FIFO) manner. In fact, the document "Effective Go" says that they "can... be seen as a type-safe generalization of Unix pipes" ([5] "Share by communicating"). The typical flow of a multithreaded program in Go involves setting up communication channels, and then passing these channels to all goroutines which need to communicate. Worker goroutines send processed data to the channel, and goroutines which need to wait on work done by others will do so by receiving from this channel.

Code Listing 2.2-4. Channel example.

```
func WorkerGoroutine(data []int, c chan int) {
    for d := range data {
        result := SomeOperation(d)
        c <- result
    }
    close(c)
}

func main() {
    var data []int = // some data
    var c chan int = make(chan int, BUF_SIZE)

    go WorkerGoroutine(data, c)

    // blocks until there is data to receive
    // from c and stops when c has
    // been closed
    for result := range c {
        // do something with result
    }
```

```
}
```

As the reader can see, there is no need to set up any explicit data synchronization. Channels are inherently hazard safe, and blocking and signaling are done automatically.

The above example used *buffered* channels. Channels can either be buffered or not, depending on their use case. An unbuffered channel `c` would be declared like so (without a second parameter):

```
c := make(chan int)
```

Sending on an unbuffered channel will block until the receiver receives the value, whereas sending on a buffered channel will only block if the buffer is full, and receives will block only if the buffer is empty. One use of this is to limit throughput. The document "Effective Go" includes an example in which one can use channels like semaphores to limit the number of simultaneous goroutines that handle requests ([5] "Channels").

Code Listing 2.2-5. Channels as semaphores

```
var sem = make(chan int, MaxOutstanding)

func init() {
    for i := 0; i < MaxOutstanding; i++ {
        // initialize semaphore with one value
        sem <- 1
    }
}

func Serve(queue chan *Request) {
    // handle the requests as they come
    for req := range queue {
        // Wait for active queue to drain.
        <-sem
        go func(req *Request) {
            process(req)

            // Done; enable next request to run.
            sem <- 1
        }(req)
    }
}
```

The Go team does, however, acknowledge that the approach with channels is not a universal solution. The Go team says that "This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance" ([5] "Share by communicating"). Channels are ultimately a high level solution, and sometimes lower-level solutions are needed. For this purpose, the Go standard library does include packages with lower-level synchronization primitives, such as the `sync` package, which includes mutex types and condition types.[10] See the section FINE-GRAINED CONTROL for more detail.

2.3 Memory Model

From the previous examples, one can understand the difference in approach. With the traditional approach to synchronization in C, the memory model can be very cumbersome. Much of the hazard safety is left as the responsibility of the programmer, who must explicitly restrict access to shared data by communicating to other threads, via synchronization primitives, when the data is being used and when the data is free. Go makes this easier for the programmer with a simpler shared memory model. Data is never actively shared between separate threads of communication —it is just passed around in channels. Thus, data races cannot happen, by design, when the data are passed between channels. The document "Effective Go" explains that

> One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization ([5] "Share by communicating").

This is why the Go team has reduced the model to a slogan: "*Do not communicate by sharing memory; instead, share memory by communicating.*"[3]

3. PRONENESS TO DEADLOCKS AND RACE CONDITIONS

Although Go takes a very different approach to concurrency than C, it is ultimately a lower level language, and thus it can be easy to write code which does not behave as expected. While Go's concurrency primitives can make it easier to avoid data hazards, they are not a panacea—the programmer does still have to be mindful of race conditions.

3.1 A Simple Example

In C, without explicitly locking critical code, data races are very likely to happen. As a simple example, consider a multithreaded program which increments a shared integer variable a set number of times.[4]

Code Listing 3.1-1. Simple Race Condition Example in C

```
void* sum(void* ret) {
    for(int i = 0; i < 1000000; i++) {
        (*((int*)ret))++;
    }
}
```

If, say, 4 threads were running the `sum` function concurrently, then by the time all threads have finished, the value in the `ret` variable would not be 4 million. It would, in fact, be a different number every time the program was run. One could easily construct a Go program with analogous behavior.

Code Listing 3.1.2. Simple Race Condition Example in Go

```
func sum(ret *int, done chan int) {
    for i := 0; i < 1000000; i++ {
        *ret++
    }
    done <- 1
}
```

So the question that remains is: how easy is it to avoid race conditions like this?

---

[3] An excellent concrete example of this mantra can be seen in an interactive Code Walk on the Go web site, which was too lengthy to include in this paper. http://golang.org/doc/codewalk/sharemem/

[4] This is, of course, a simplified example. One could design a parallel summation using a more reasonable approach, such as data segmentation. There is no reason to use a single shared variable for a simple sum like this.

3.2 Avoiding Race Conditions

As we've seen, in C, to avoid race conditions, one could use a mutex variable to protect any shared data. To fix the example C code above, one could lock just before incrementing the shared variable, and then unlock just after.

Code Listing 3.2-1. Mutex fix

```
void* sum(void* ret) {
    for(int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);
        (*((int*)ret))++;
        pthread_mutex_unlock(&mutex);
    }
}
```

Similarly, for analogous Go code, one could use a buffered channel of size 1 to restrict access to the sum variable to one goroutine at a time.

Code Listing 3.2-2. Channel fix

```
func sum(sumChan chan int, done chan int) {
    for i := 0; i < 1000000; i++ {
        // sumChan has a buffer of size 1, so
        // receiving blocks if there is another
        // goroutine currently incrementing the
        // sum
        cur := <-sumChan
        cur++
        sumChan <- cur
    }
    done <- 1
}
```

In this simple case, the solutions are not terribly different. However, as we can gather from the previous Go code examples, channels can be very flexible in their usage. Where C needs distinct synchronization primitives, such as mutex variables, condition variables, and semaphores, to handle all of the different use cases, Go's channels can be used: as a hazard-safe conduit to share actual data; to signal when a goroutine is finished; and as a way to limit the throughput of the program. In other words, good solutions exist in both languages, but Go's channels provide a much more effortless way to design programs which are safe from race conditions.

3.2.1 Detecting Race Conditions

No matter how skilled a programmer is, race conditions will inevitably arise. Therefore, tools to help the programmer detect race conditions are necessary. Various tools exist which analyze a C program's runtime to aid the detection of race conditions. A common tool is Valgrind, which includes the race detector Helgrind.[8] The Go tool chain comes packaged with a race detector, which is built on the ThreadSanitizer race detector for C/C++.[7] To run the race detector on a Go program, one can simply use the `-race` option when compiling (e.g. `go run -race mysrc.go`). Similarly, `gcc` comes bundled with ThreadSanitizer, so to run the race detector on a C program, one would simply compile with the `-fsanitize=thread` and `-fPIE` options and then link to `-fsanitize=thread -pie`.[9] Therefore, the options for helping detect race conditions are similar for both C and Go.

3.3 Deadlocks

Deadlocks are one of the trickiest bugs to remediate. Often, they are transient and very difficult to trace (especially in a large project). Deadlocks take care to avoid in both C and Go. Mechanisms and methodologies exist in both languages to aid in avoiding deadlocks.

3.3.1 Data Segmentation or Replication

If it is possible, replicating the shared resource for each thread will eliminate deadlocks, because no locking mechanism is needed for synchronization ([6] p. 178). For example, in C, if the shared resource is an array, then replication can be done with the `memcpy` function. Although, if the data set is not small, replicating can consume too much memory. Therefore, if the concurrent operations are independent from each other, it is better to segment the shared data, rather than replicate it.

In Go, segmentation can be done easily with its built-in `slice` type. Go's `slice` type is similar to Python's `slice` type. Slices are references to an underlying array, and they provide operators which make it easy to segment that array. For example, one can "reslice" a slice with the bracket operator. If `arr` is a slice variable, then `arr[:len(arr)/2]` will return a slice which can access the first half of `arr`'s elements.[5] Thus, if `sum` is a function which returns the sum of every element of an integer slice, then

Code Listing 3.3.1-1. Slice example

```
arr := []int{7, 2, 8, -9, 4, 0}
partSums := make(chan int)
go sum(a[:len(a)/2], partSums)
go sum(a[len(a)/2:], partSums)
x, y := <-partSums, <-partSums
fmt.Println(x+y)
```

will launch two goroutines which sum the first and second half of `arr`, respectively, and receive both partial sums from the `partSums` channel. Then, these two partial sums are added together to yield the correct sum of all the elements in `arr`—in this case, 12. Segmenting the data like this avoids the need for any communication between the two goroutines, as well as the need to replicate any data (since slices are simply references). A similar functionality could be achieved in C, although the segmentation would have to be done manually; the `sum` function, for instance, might need to take parameters for the beginning and ending indices of which elements in the shared array to sum. In both languages, if data segmentation is possible, this is a good way to avoid deadlocks. It obviates the need to use any kind of synchronization locks.

3.3.2 Try and Back Off

However, when locks are unavoidable, there exist methodologies in both C and Go to help avoid deadlocking. When using mutex variables to guard critical code sections, one approach is to instruct a thread to release any resources it currently holds if locking another resource results in failure ([6] p. 180). In C, this can be done with the `pthread` function `pthread_mutex_trylock`.[6]

---

[5] If the reader has never seen slices before, or if this is otherwise confusing, see "Slice expressions" in the Go Language Specification, or for a more detailed discussion, the Go Blog article "Go Slices: usage and internals".

[6] Although this approach may introduce a *live lock* if many threads continually conflict over the first resource and back off. In this case, a statement may be added to the end of the loop which waits a random amount of time ([6] p. 180).

Code Listing 3.3.2-1. "Try and Back Off" Example in C.

```
while (true) {
    pthread_mutex_lock(&mutex1);
    if(pthread_mutex_trylock(&mutex2) == 0) {
        break;
    }
    pthread_mutex_unlock(&mutex2);
}
```

In Go, however, if mutex variables absolutely must be used, this approach may not work, as mutex behavior is more strict than in C's `pthread` library. In Go, there is no analogous "trylock" function on its mutex type; furthermore, it is a runtime error if the `Unlock` function is called on a mutex which is already unlocked,[10] so greater care must be given to restrict the ordering of lock and unlock operations.

3.3.3 Go's `select` *Statement*

However, in a typical Go program where channels suffice for synchronization, a different approach may be used to avoid deadlocking on channel blocks. In Go, the `select` statement is analogous to a `switch` statement for channels. Essentially, the cases will try to execute a given operation on a channel, and if that operation *would* block, then it tries a different case.

Code Listing 3.3.3-1. `select` Statement Example

```
select {
case a := <-ch1:
    // use a
case b := <-ch2:
    // use b
default:
    // receiving from both a and b would block
}
```

Here, if the program can receive from the channel `ch1`, then it assigns the received value to the variable `a`, and `a` can be used within the code below the case statement. Similarly, if the program can receive from `ch2`, it assigns the received value to the variable `b`, which can be used in the code following the case statement. If both `ch1` and `ch2` are ready, the program picks one case at random. If neither are ready, the code in the default case is run. Therefore, a tactic analogous to the "try and back off" method can be used.[7]

Code Listing 3.3.3-2. "Try and Back Off" Example in Go

```
OuterLoop:
for {
    a := <-ch1
    select {
    case b := <-ch2:
        break OuterLoop
    default:
        ch1 <- a
        // wait
    }
}
// use a and b
```

[7] Go's `break` statement stops the execution of only the innermost `for`, `select`, or `switch` statement, so a label must be used to specify it is the `for` loop we want to stop. For more detail, see the "Labeled statements" and "Break statements" sections of the Go Programming Language Specification.[4]

Given how flexible channels are in general, there are, of course, many more ways one can use the `select` statement to avoid deadlocks. For example, in his article "Go Concurrency Patterns: Timing out, moving on," Andrew Gerrand describes how one could implement a simple timeout operation with channels.[11]

Code Listing 3.3.3-3. Timing out with Channels

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1 * time.Second)
    timeout <- true
}()

select {
case <-ch:
    // a read from ch has occurred
case <-timeout:
    // the read from ch has timed out
}
```

In the above code, a "timeout" channel is created, and an anonymous function is launched in a goroutine which sends a value to the `timeout` channel after one second. In the `select` statement that follows, the main goroutine is blocked until one of the `select` cases can happen. If it takes too long to receive from `ch` (i.e., if the operation takes too long), then the `select` statement receives from the `timeout` channel and the program continues.

4. FINE-GRAINED CONTROL

As we have seen, Go is intended to be somewhere in between the level of a scripting language and that of a lower-level language, like C. It is not surprising, then, to learn that C provides a finer level of control over the behavior of the program's threads than Go does. However, Go does include some reflective functions which provide some finer-grained control over the execution of the program, if so needed. Ultimately, though, the memory models of Go and C are anything but isomorphic, so the comparison between each language's runtime may not be apt.

4.1 Thread Attributes in C

In C with `pthreads`, there are a few standard thread attributes which can be set when threads are created. These attributes include: its detached or joinable state; scheduling inheritance, policies, parameters, and contention scopes, and stack size.[12] As an example, if the programmer knows that a thread will never need to join with another, they can set the thread's attributes such that it will always be detached, and thus, not joinable with any thread, for the duration of its execution.

Code Listing 4.1-1. Thread Attributes in C

```
pthread_t thread;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_DETACHED);

pthread_create(&thread,&attr,SomeFunction,args);
```

Similarly, one can set the scheduling policy of a thread with attributes. There are three scheduling types in `pthreads`: `SCHED_FIFO` (first-in, first-out), `SCHED_RR` (round robin), or `SCHED_OTHER`,[13] which can be set in a similar manner.

Code Listing 4.1-2. Changing the Scheduling Policy

```
pthread_attr_init(&attr);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
```

This example would set the execution policy of any threads created with the `attr` variable to run in a FIFO order. Similar mechanisms exist for all the other attributes. Some local implementations of `pthreads` include a `pthread_setaffinity_np` function, which allows the programmer to specify which processor/core they would like the thread to run on.[12]

4.2 Fine-grained Control in Go

As this author has previously discussed, the memory model in Go is much different. Go does not run traditional threads—it runs goroutines. This memory model is designed to be much more lightweight and easier to use than the traditional approach; thus, many of the fine-grained details are abstracted away from the programmer. However, because these details are not always avoidable, Go does provide some packages which allow access to lower-level synchronization primitives and runtime scheduling.

For example, the `sync` package provides more traditional synchronization primitives, such as mutex and condition variables, as well as the `WaitGroup` type, which allows the programmer to explicitly wait on a set of goroutines to finish. The subpackage `sync/atomic` provides even lower-level functions which assist in implementation of synchronization algorithms by providing various operations which are guaranteed to be atomic.[14] The `runtime` package provides functions which interact with Go's runtime directly. Some example functions include: `runtime.Gosched()`, which yields the current goroutine, allowing others to run; `runtime.LockOSThread()`, which locks a goroutine to whichever OS thread it happens to be running on; and various reflective functions which report runtime information, such as `runtime.Stack()`, which prints stack trace information.[15]

Go does not provide mechanisms to control thread attributes as seen in section 4.1, however, because goroutines operate on a higher level than threads. As discussed in section 2.1, goroutines are managed dynamically and multiplexed onto multiple OS threads, so providing lower-level control mechanisms than those shown above is not really feasible. Most of the time, this works out in the programmer's favor, as they do not have to manage as much manually, but these control mechanisms can be missed if the application calls for them.

5. PERFORMANCE

For the most part, the comparisons here are not directly related to each language's ability to facilitate parallel programs, but performance is nonetheless an important factor when deciding which language to implement an application with.

5.1 Compilation Time

One of the major reasons why Google decided to write a new compiled language was because the compile times of their large C++ code bases grew to be a significant engineering problem ([1] §7). Rob Pike discusses this problem in the document "Google at Go" ([1] §5). C and C++ use preprocessor directives to manage dependencies, which does not scale well for very large projects. Typically, a C/C++ header file will use `#ifndef` guards to protect against errors when multiple source files `#include` the same library. If a dependency is included multiple times, after the first compilation, the contents are disregarded, but the compiler still opens and scans the whole file. As an example, Pike noted that for one large project at Google in 2007, a code base of 4.2 MB expanded to over 8 GB when delivered to the compiler because of all the repeated dependencies ([1] §5). For this reason, it was impractical to build large projects on a single machine, so Google had to design a large distributed compilation system. With an early version of this distributed system, the binary took 45 minutes to compile. At the time of this writing, Pike's document says the same 2007 binary takes 27 minutes to compile today.

Pike says that "the origin myth for Go states that it was during one of those 45 minute builds that Go was conceived" ([1] §6). The Go compiler was based on an approach that the Plan 9 team designed to remedy the repeated dependency problems described above.[8] Essentially, how it works is it builds a dependency graph from the source files themselves by reading all of the import statements from every file, and then compiling the lowest-level dependencies first by compiling every source file in topologically sorted order. Note that this implies that there are no cycles in the dependency graph—it was decided to make circular dependencies illegal ([1] §7). When the compiler progresses in its traversal of the sorted dependency graph, if the file being built includes a file which has already been compiled, it only reads that dependency's object file, not its source code. Pike notes that this design "has the important effect that when the compiler executes an import clause, *it opens exactly one file*, the object file identified by the string in the import clause... in effect, the compiler writes the header file when the Go source file is compiled" ([1] §7).[9]

In addition to this, the language itself was designed with efficient compile time in mind. Its compact grammar and regular syntax makes it much easier to parse.[16] For instance, requiring opening braces to be kept on the same line as its function declaration allowed semicolon insertion to happen without the need for lookahead ([2] "Why are there braces but no semicolons?"). Go does not have inheritance; instead, it relies on a simple interface system, which greatly simplifies the underlying type system, and the structure of Go programs ([2] "Why is there no type inheritance?").

When testing the results of this design on a large binary, Google recorded a speedup of about 40x ([1] §7). During one of Google's Tech Talks in 2009, Rob Pike gave a demonstration of how fast the Go compiler was. On Pike's single-core laptop, building the entire Go source tree, which he estimated at about 120,000–130,000 lines and included "regular expressions, flags, concurrency primitives, the runtime, [and] the garbage collector..." took about 8 seconds.[17] Today, the standard library comes in at over 400,000 lines of code, and on this author's

[8] Plan 9 is an experimental operating system designed by Bell Labs as the potential successor to Unix, used primarily for research purposes at the Computing Sciences Research Center. See http://en.wikipedia.org/wiki/Plan_9_from_Bell_Labs.

[9] Pike also notes: "It's worth mentioning that this general approach to dependency management is not original; the ideas go back to the 1970s and flow through languages like Modula-2 and Ada. In the C family Java has elements of this approach" ([1] §7).

laptop with a 4-core Intel i3-3110M CPU, compile time averages around 8 seconds on a cold cache. It is clear that for large projects, the compile time of Go wins over C/C++.

5.2 Performance

Unsurprisingly, typical performance of a Go program is much faster than most scripting languages and slightly slower than most other lower-level languages, like C. In Debian's Computer Language Benchmarks Game, in their "Ten Tiny Examples," Go rates at 8th on a scale comparing (program time / fastest program time), coming in at the same order of magnitude of relative performance of Java, Scala, Haskell, and OCaml.[18] These were slightly slower than C++, Ada, C, and Fortran, but much faster than PHP, Perl, Python, Ruby, and Hack. It is worth mentioning, however, that Debian does disclaim that language comparison benchmarks are not representative of real-world performance; they warn not to jump to conclusions based on these tests because "programming languages are compared against each other as though their designers intended them to be used for the exact same purpose - that just isn't so."[19]

In 2011, Robert Hundt, one of Google's own software engineers, published a paper entitled "Loop Recognition in C++/Java/Go/Scala" which compared performances of those languages with a particular loop recognition algorithm.[20] Hundt found that the Go implementation described in the paper fell behind all other languages considered by a significant margin. However, Russ Cox, from the Go team, wrote an article that examined the paper, and found that the reason for the slow performance was mostly because of the naïveté of the implementation.[21] Cox took the paper as an opportunity to showcase the Go performance profiler. After some analysis and a few minor tweaks to the code published in the paper, the program improved from 25.2 seconds and 1302MB of memory to 2.29 seconds and 351MB of memory—a speedup of 11x. Cox also noticed some inefficiencies in the C++ implementation and fixed those as well so that a new comparison would be fair. It was found that Go performed slightly slower than C++—as expected. Cox concluded that "benchmarks are only as good as the programs they measure" and that Go can be competitive with C++ (and by extension, all other compiled languages, depending on the application) when care is taken to avoid generating excess garbage.[21]

These are, of course, only two examples of performance measurement, but in general, it is expected that Go, in its current state, will follow this pattern—being much faster than scripting languages, but slightly behind most other compiled languages. This is not surprising, given that Go is such a new language. Being a modern compiled language, it will be much faster than any interpreted language, but the compiler has not had the decades of optimizations that, say, the C compiler has had. In addition, as discussed in section 5.1, because one of the primary goals of the language was to compile fast, it does not necessarily spend as much time passing the code through optimizers. According to a question on the Go FAQ, "one of Go's design goals is to approach the performance of C for comparable programs" while being much easier to use ([2] "Why does Go perform badly on benchmark X?"). The same question concludes that, while there is certainly room for improvement in the compiler, standard library implementations, and the garbage collector, Go can be competitive for many applications.

6. CONCLUSION

Go is an exciting new language which introduces many interesting features which make many aspects of programming much easier, particularly in implementing parallel programs. Go makes use of a simple memory model with goroutines to automatically manage multiple threads of execution, which can be much easier than the manual approach that is required in C. Through the use of channels, Go makes inter-process communication much simpler than traditional mechanisms. Both Go and C are equally prone to deadlocks and race conditions, and both come with comparable tools and techniques to help debug these issues. Since Go is designed to be simpler, it is not possible to reach quite the level of control over threads that C can achieve, so if such control is necessary, C may be better suited. For very large projects, Go compiles faster than C and C++ by an order of magnitude, while actual program performance will typically fall slightly behind C/C++. The goal of Go was to approach the performance of C and to make software engineering much simpler; given the analysis in this paper, it appears the Go team has achieved that goal remarkably well.

There are many other interesting features and idioms in Go which are outside the scope of this paper. To learn more, the official web site provides excellent and thorough documentation of the language, and the Go Blog has many in-depth articles that cover: the inner-workings of Go, idiomatic practices, and how to use the included tool suite. Many large companies have started adopting Go, including: BBC Worldwide, Bitbucket, GitHub, Canonical, Heroku, SoundCloud, and Docker, among others.[22]

7. REFERENCES

[1] Pike, Rob. (n.d.). Go at Google: Language Design in the Service of Software Engineering. *Golang.org*. Retrieved from http://talks.golang.org/2012/splash.article

[2] Golang.org. Frequently Asked Questions (FAQ). (n.d.). Retrieved 7 March 2014 from http://golang.org/doc/faq

[3] Gerrand, Andrew. (2010). Share Memory By Communicating. *The Go Blog*. Retrieved 8 March 2014 from http://blog.golang.org/share-memory-by-communicating

[4] Golang.org. The Go Programming Language Specification. (2013). Retrieved 8 March 2014 from http://golang.org/ref/spec

[5] Golang.org. Effective Go. (n.d.) Retrieved 8 March 2014 from http://golang.org/doc/effective_go.html

[6] Akhter, Shameem, & Roberts, Jason. (2006). *Multi-Core Programming: Increasing Performance through Software Multi-threading*. United States: Intel Press.

[7] Vyukov, Dmitry, & Gerrand, Andrew (2013). Introducing the Go Race Detector. *The Go Blog*. Retrieved on 15 March 2014 from http://blog.golang.org/race-detector.

[8] Valgrind.org. Helgrind: a thread error detector. (2013). Retrieved 15 March 2014 from http://valgrind.org/docs/manual/hg-manual.html.

[9] Code.google.com. Cpp Manual. (2014). Retrieved 15 March 2014 from http://code.google.com/p/thread-sanitizer/wiki/CppManual

[10] Golang.org. Package sync. (n.d.). Retrieved 16 March 2014 from http://golang.org/pkg/sync/

[11] Gerrand, Andrew. (2010). Go Concurrency Patterns: Timing out, moving on. *The Go Blog*. Retrieved on 16 March 2014 from http://blog.golang.org/go-concurrency-patterns-timing-out-and

[12] Barney, Blaise. (n.d.) POSIX Threads Programming. Retrieved 23 March 2014 from https://computing.llnl.gov/tutorials/pthreads

[13] Marshall, Dave. (1 May 1999). Further Threads Programming: Thread Attributes (POSIX). Retrieved on 23 March 2014 from http://www.cs.cf.ac.uk/Dave/C/node30.html

[14] Golang.org. Package atomic. (n.d.). Retrieved 23 March 2014 from http://golang.org/pkg/sync/atomic/

[15] Golang.org. Package runtime. (n.d.). Retrieved 23 March 2014 from http://golang.org/pkg/runtime/

[16] Pike, Rob. Go's Declaration Syntax. (2010). Retrieved 24 March 2014 from http://blog.golang.org/gos-declaration-syntax

[17] Pike, Rob. "The Go Programming Language." Talk given at Google's Tech Talks. (2009). Starts at 8:53. Retrieved from http://www.youtube.com/watch?v=rKnDgT73v8s#t=8m53

[18] Debian's Computer Language Benchmarks Game on a x64 Ubuntu : Intel® Q6600® quad-core. Retrieved 26 March 2014 from http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.php?gpp=on&java=on&go=on&ghc=on&sbcl=on&csharp=on&hipe=on&python3=on&yarv=on&calc=chart

[19] Debian's Computer Language Benchmarks Game. "And please don't jump to conclusions!" Retrieved on 26 March 2014 from http://benchmarksgame.alioth.debian.org/dont-jump-to-conclusions.php

[20] Hundt, Robert. "Loop Recognition in C++/Java/Go/Scala." (2011). Retrieved from http://research.google.com/pubs/pub37122.html

[21] Cox, Russ. "Profiling Go Programs." (2011). Updated May 2013 by Shenghou Ma. *The Go Blog*. Retrieved 26 March 2014 from http://blog.golang.org/profiling-go-programs

[22] go-wiki. "Currently Using Go." Retrieved on 26 March 2014 from https://code.google.com/p/go-wiki/wiki/GoUsers