High performance servers without the event loop
OSCON, Portland
21 July 2015
Dave Cheney
whoami(1)

- Sysadmin before Devops
- Transitioned to Go 5 years ago
- Work for Canonical contributing to the Go project
Who is this presentation for?

The horror!

As an admin in a past life the most stressful times in my career were defined by unsatisfying performance.

I got into Go because of its potential to build high performance servers.

As we're in a technical track, I'm not here to tell you that Go is fast, instead I going to explain why Go is fast.
An argument for an efficient programming language
Moore's Law states that the number of transistors per square inch doubles roughly every 18 months.

However, clock speeds topped out a decade ago with the Pentium 4 (Penryn) and have been slipping backwards since.
From space constrained to power constrained

Sun Enterprise e450—about the size of a bar fridge, about the same power consumption.
Where does this power consumption come from?

CMOS stands for Complementary Metal Oxide Semiconductor, the *complementary* part is the key.

When the circuit is on or off, no current flows directly from the source to the drain. However during transition there is a brief period where *both* transistors are conducting.

Power consumption, and thus heat dissipation, is directly proportional to number of transition per second—Clock speed.
CPU feature size reductions are primarily aimed at reducing power consumption.

Reducing power consumption doesn't just mean "green". The primary goal is to keep power consumption, and thus heat dissipation, below levels that will damage the CPU.

Conversely, performance improvements come mainly from microarchitecture tweaks and esoteric vector instructions which are not directly useful for the general computation.

Added up, each microarchitecture (5 year cycle) change yields at most 10% improvement per generation, and most recently 4-6%.

Moore's Law is still in effect, but for the people in this room, the free lunch is over.
What's your point?

So, why am I rambling on about hardware at a software conference?

The old adage that a slow language doesn't matter because hardware is getting faster, do not apply anymore.

If performance and scale is important to you, and arguably it is, as you're here in this session, then you'll agree with me that the days of throwing hardware at the problem are over - at least in the conventional sense.

You need a language which is efficient, because inefficient languages just do not justify themselves in production, at scale, on a capex basis.
An argument for a concurrent programming language
CPUs are not getting faster, but they are getting wider.

Image Credit: Intel / AnandTech (http://www.anandtech.com/show/5875/dual-core-i2-ivy-bridge-die-measured-121mm2)

So, CPUs are not getting faster, but they are getting wider.

- Hyper threading
- More cores, dual core on mobile parts, quad core on desktop parts, even more cores on server parts.
Go on the server

A common refrain when talking about Go is its a language that works well on the server; static binaries, powerful concurrency, and high performance.

This talk focuses on the last two items, how the language and the runtime transparently let Go programmers write highly scalable network servers without having to worry about thread management or blocking I/O.
Processes, threads and Goroutines
Your grandfather's server

Image Credit: Tom Scott (https://www.flickr.com/photos/tascott/3357632896/in/album-7215761525758987/) (CC BY-NC-SA 2.0)

The first web server, circa March 1989 (http://info.cern.ch/Proposal.html).

NCSA was the web server, which grew into apache.
Processes

In the beginning computers ran one process at a time. Then in the 60’s the idea of multiprocessing, or time sharing became popular.

By the 70's this idea was well established for network servers, ftp(1), telnet(1), rlogin(1).

This was the world of Burners-Lee's NCSA Mosaic server running on a 25Mhz Next Cube, every active HTTP session was handled by its own process.

In a time-sharing system the operating systems maintains the illusion of concurrency by rapidly switching the attention of the CPU between active processes by recording the state of the current process, then restoring the state of another.

This is called context switching.
There are three main costs of a context switch.

- The kernel needs to store the contents of all the CPU registers for that process, then restore the values for another process.
- The kernel needs to flush the CPU’s virtual memory to physical mappings (TLB).
- Overhead of the operating system context switch, and the overhead of the scheduler function to choose the next process to occupy the CPU.
Threads

This lead to the development of threads, which are conceptually the same as processes, but share the same memory space.

As threads share address space, they are lighter than processes so are faster to create and faster to switch between.

Threads still have an expensive context switch cost, a lot of state must be retained.

Goroutines take the idea of threads a step further.
Goroutines

Goroutines are cooperatively scheduled, rather than relying on the kernel to manage their time sharing.

The compiler knows the registers which are in use and saves them automatically.

The switch between goroutines only happens at well defined points, when an explicit call is made to the Go runtime scheduler.

- Channel send and receive operations, if those operations would block.
- The Go statement, although there is no guarantee that new goroutine will be scheduled immediately.
- Blocking syscalls like file and network operations.
- After being stopped for a garbage collection cycle.

In other words, places where the goroutine cannot continue until it has more data, or more space to put data.
Goroutines

Many goroutines are multiplexed onto a single operating system thread.

- Super cheap to create.
- Super cheap to switch between as it all happens in user space.
- Tens of thousands of goroutines in a single process are the norm, hundreds of thousands not unexpected.

This results in relatively few operating system threads per Go process, with the Go runtime taking care of assigning a runnable Goroutine to a free operating system thread.
func grep(r io.Reader, needle string) {
    br := bufio.NewReader(r)
    lines := make(chan string, 20)

    go func() {
        defer close(lines)
        line, err := br.ReadString('
')
        if err != nil {
            return
        }
        lines <- line
    }()

    for line := range lines {
        if strings.Contains(line, needle) {
            fmt.Println(line)
        }
    }
}
Go uses a M:N scheduler in user space.

If you lived through green threads in Java or user space threads on Linux, then you may be feeling uncomfortable at this point. Let me assure you that in practice this user space scheduler works well. This is because it is integrated with the runtime.

A small number of operating system threads service runnable goroutines

- Go versions 1.4 and lower, defaults to 1
- Go versions 1.5 and above, defaults to the number of CPUs visible to the operating system.

Compare this to threaded applications, where a thread can be preempted at any time, at any instruction. In Go, the compiler handles this as a natural by product of the function call preamble.

From the point of view of the language scheduling looks like a function call, and has the same function call semantics. The thread of execution calls into the scheduler with a specific goroutine stack, but may return with a different goroutine stack.
Stack management
This is a diagram of the memory layout of a process. The key thing we are interested is the location of the heap and the stack.
Stacks and guard pages

Because the heap and stack overwriting each other would be catastrophic, the operating system arranges an area of unwritable memory between the stack and the heap.
Threads share the same address space, so for each thread, it must have its own stack, and its own guard page.
Goroutine stack management

The early process model, allowed the programmer to view the heap and the stack as effectively infinite. The downside was a complicated and expensive subprocess model.

Threads improved the situation a bit, but require the programmer to guess the most appropriate stack size; too small, your program will abort, too large, you run out of virtual address space.

We've seen that the Go runtime schedules a large number of goroutines onto a small number of threads, but what about the stack requirements of those goroutines?
Goroutine stack growth

Each goroutine starts with an small stack, allocated from the heap. The size has fluctuated over time, but in Go 1.5 each goroutine starts with a 2k allocation.

Instead of using guard pages, the Go compiler inserts a check as part of every function call to test if there is sufficient stack for the function to run.

If there is insufficient space, the runtime will allocate a large stack segment on the heap, copy the contents of the current stack to the new segment, free the old segment, and the function call restarted.
Because of this check, a goroutines initial stack can be made much smaller, which in turn permits Go programmers to treat goroutines as cheap resources.
Integrated network poller
Dan Kegel's c10k problem

www.kegel.com/c10k.html circa 2002

Question: How to hold up 10,000 concurrent TCP sessions on commodity hardware of the day.

Conventional wisdom suggested that high performance servers require native threads, or more recently, event loops.

Threads carry a high overhead in terms of scheduling cost and memory footprint.

Event loops ameliorate those costs, but introduce their own requirements for a complex callback driven style.

Go provides the best of both worlds.
Go's answer to c10k

In Go, syscalls are *usually* blocking operations; `read(2)`, `write(2)`.

Scheduler would let the goroutines' backing thread block, then find or spawn another to continue to service other goroutines.

This works well for file IO as a small number of blocking threads can exhaust your local IO bandwidth.

For network sockets, almost all of your goroutines are going to be blocked waiting for network IO.

In a naive implementation this would mean as many threads as goroutines blocked waiting on network IO.
func (fd *netFD) Read(p []byte) (n int, err error) {
    // preamble
    for {
        n, err = syscall.Read(fd.sysfd, p)
        if err != nil {
            n = 0
            if err == syscall.EAGAIN {
                if err = fd.pd.WaitRead(); err == nil {
                    continue
                }
            }
        }
        err = fd.eofError(n, err)
        break
    }
    // epilog
    return
}
Integrated poller

In older versions of Go, this was handled by a single goroutine which used select(2) / kqueue(2) / epoll(2), IOCP (windows).

In current versions of Go this has been integrated into the runtime.

The runtime knows which goroutine is waiting for the socket to become ready and can put it back on the same CPU.

Async file IO slightly harder problem due to spotty operating system support. Maybe for Go 1.6.
Goroutines, stack management, and an integrated network poller

Goroutines provide a powerful abstraction that free the programmer from worrying about thread pools or event loops.

The stack of a goroutine is as big as it needs to be without being concerned about sizing thread stacks or thread pools.

The integrated network poller lets Go programmers avoid convoluted callback styles while still leveraging the most efficient IO completion logic available with the operating system.

The runtime make sure that there will be just enough threads to service all your goroutines and keep your cores active.

All of these features are transparent to the programmer.
But wait, there's more

As this is the performance track, I've mostly constrained my remarks to this area, but there is a lot more to the Go success story than just being fast, and these are arguably the more notable features of the language.

- Super fast compilation.
- Excellent tooling; go build, go test, go fmt, go lint, go vet, godoc, -race detector.
- Excellent cross platform support, cross compilation as well without tears
- Single static binary enables a ridiculously simple deployment story, scp(1) and you're done -- no runtime interpreter required.
Success stories

- Vitess (mysql proxy that sits between every db call behind Youtube)
- dl.google.com
- New York Times
- Hailo
- Facebook/Parse
- Docker
- CoreOS; fleet, etcd, rkt,
- Kubernetes
- Baidu
- Qihoo 360
- ... and many more (https://github.com/golang/go/wiki/GoUsers)
Would you like to know more?

Go take the tour

tour.golang.org (http://tour.golang.org)

Go join a meetup

go-meetups.appspot.com (https://go-meetups.appspot.com)

Go watch some more talks

gophervids.appspot.com (http://gophervids.appspot.com)
Thank you

Dave Cheney
dave@cheney.net (mailto:dave@cheney.net)
http://dave.cheney.net/ (http://dave.cheney.net/)
@davecheney (http://twitter.com/davecheney)