G.E.R.T: A Go-Based Toolkit For Embedded Applications

by

Yanni Coroneos

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2017

Certified by .....
Dr. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

G.E.R.T: A Go-Based Toolkit For Embedded Applications

by

Yanni Coroneos

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

0.1 abstract

Embedded systems are becoming increasingly complicated due to the emergence of SOCs (system-on-a-chip) with multiple cores, dizzying amounts of peripherals, and complicated virtual memory systems. Unfortunately, performant embedded systems for these SOCs are still largely written from scratch in C because heavy operating systems, such as Linux, can introduce more bugs and degrade real-time performance.

This thesis proposes a new system called G.E.R.T, the Golang Embedded Run-Time, for multicore ARM processors. GERT is a modified version of the Go runtime for bare-metal operation on multicore ARMv7a SOC's in order to evaluate the effectiveness of using a high-level, type-safe, and garbage collected language for embedded applications. G.E.R.T provides the multiprocessor support and basic memory abstractions of a typical embedded toolkit while also enabling the user to leverage the language features of Go in order to develop concurrent embedded programs that are easier to reason about than similar ones written in C.

Thesis Supervisor: Dr. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern embedded systems are composed of multicore SOCs that require careful thought about concurrency. C is the most commonly used language to program such low level systems because it is simple and expressive, but it is a double-edged blade. Low level code written in C can more easily and directly interface with hardware, but it can also be plagued with difficult bugs, especially concurrency-related bugs and memory bugs like use after free.

Concurrent, high-level user space programs that do not directly interface with hardware, on the other hand, are usually written in a high-level language, such as Rust or Go, which abstracts concurrency and provides memory safety. Rust, Go, and other High Level Languages (HLL's) will usually ensure that there is never an out of bounds error, use after free error, or unsafe cast. HLL's can also provide native concurrency support through primitives like channels. The downsides of HLLs is that those nice featues come at a cost of garbage collection and many runtime checks. Despite the shortcomings, HLLs are still preferred to C programs because fast, modern computers negate the performance cost. Now that embedded devices are also very fast, the possibility of using a HLL on a performant embedded system is undeniable.

Embedded programmers are often drawn towards using Linux for their work because then they can write their embedded program in user space with a high level language in order to avoid the difficulty and poor concurrency support of C. Popu-

lar platforms that use this paradigm include the Raspi and Beaglebone. However, embedded programs that run in user space suffer from significant event latency because external interrupts must shuffle their way through the kernel. Additionally, the kernel's scheduler will also constantly preempt the program. This performance loss is compounded by the performance penalties that HLL's also impose on programs. Usually, an experienced programmer does not even want or need the kernel, but they pay the price just to run the high-level embedded code.

Traditional operating systems impose expensive and redundant checks on high-level programs. For example, Go already ensures that different threads cannot access each other's memory and that null pointers will not be dereferenced. The OS is intended to work with buggy C programs though, so it must still switch page tables when switching processes and be prepared for page faults when bad accesses occur. This machinery is expensive and redundant for programs written in a HLL. Since embedded applications only run a single program, it is possible to completely do away with the OS underneath a high-level program and regain the lost performance without any degredation in safety.

There are ongoing efforts to bring high-level languages to desktop operating system kernels and also single-core microcontrollers, but there is no known system which provides a high-level language environment for multicore SOCs. Singularity <cite> and Biscuit <cite> are desktop operating system kernels, written in Sing# and Go, which focus on hosting user-space programs. Copper <cite> and MicroPython <cite> are small embedded toolkits, written in Rust and Python, which aim to provide a high-level programming environment for single-core microcontrollers. Multicore SOCs have, so far, been left out of the picture. This thesis presents a new embedded toolkit, the Golang Embedded RunTime (GERT), which is specifically intended for concurrent, bare-metal embedded applications.

1.1 Outline

Below is an outline of this thesis.

- Chapter 2 explains why Go was chosen for GERT and also the basic intuition behind the Go runtime implementation
- Chapter 3 presents the core work of this thesis, or how the runtime was modified to work on bare metal
- Chapter 4 shows how to use GERT, an essential aspect of any toolkit
- Chapter 5 evaluates GERT on embedded benchmarks and also two embedded case studies
- Chapter 6 concludes this thesis with a crystalization of the results and their implication

Chapter 2

Why Write System Code in Go?

At first glance, Go code looks a lot like C. There are no classes and the language has a strong type system. This already makes Go a good systems language, but Go's greatest feature is its built-in support for concurrency through goroutines and channels. Goroutines are lightweight threads that the Go runtime can schedule without help from the operating system. Channels are typed FIFO queues which can help to serialize asynchronous events, perhaps coming from several goroutines. With these features and familiarity, writing programs in Go comes naturally to a C programmer, but with the added bonus that a Go program is also memory safe because of runtime checks and a garbage collection system.

2.1 Go Runtime Organization

Go's implementation is reminiscent of a small OS. There are three basic abstractions in the runtime: G's, M's, and P's. A G, or goroutine, is an executable fragment of Go code, like a function or group of functions. In a concurrent program, each fragment of Go code will get its own G. M's are just OS threads; they can be executing Go code, blocked in a system call, or idle. Finally, a P represents a processor, or the resources necessary to allow execution. M's cannot execute code unless they are associated with a P. It is the Go scheduler's job to associate G's, M's, and P's with each other

so that code can run. Go's threads are lightweight and cooperatively scheduled[1] so that execution only transfers during blocking operations. The runtime also manages its own pool of memory and exports its own atomic primitives through the standard "sync" package. In fact, Go provides most of the common OS primitives natively in its standard libraries <https://golang.org/pkg/#stdlib>.

Go relies on the OS for priveleged operations. Since it runs in userspace, the Go runtime cannot interface with physical hardware so it uses the OS to get the time, allocate memory, receive signals.

2.2 Go Memory Safety

Like most high-level languages, Go does not allow the programmer to unsafely manipulate memory. This means that there is no way to unsafely cast from one data type to another, it is impossible to use a pointer after it is freed, and accessing an array out of bounds yields a runtime error. Go also does not allow pointer arithmetic. These properties are provided by bounds checking all array accesses, a strong type system, and the lack of a $free()$ function.

Instead of letting the programmer free memory, Go has a garbage collector which automatically frees memory with no incoming pointers. The garbage collector is a costly abstraction though because it must scan through all the pointers in the program. In the worst case scenario, the program must be checkpointed and stopped while the GC runs, but recent improvements to Go have enabled concurrent garbage collection. GC pauses are still noticeable though.

Compared to C, the properties of Go might seem like training wheels, but this is exactly the point. The majority of bugs in a C program can be traced to an out of bounds access (buffer overruns), use-after-free, or null pointer dereference. These issues are so commonplace that OS kernels attempt to preempt the problem by inserting guard pages all over the user program's address space. In a bare-metal

[1]Go code is not totally cooperative because goroutines can be pre-empted during a function call. Users can write critical sections to avoid this though

embedded system, though, there is no kernel which can help mitigate the effects of a memory safety bug. The outcome of triggering a memory safty bug in an embedded system is usually program curruption and total failure. Preventing such a scenario is one of the goals of GERT.

Platform	Index OOB	User-after-free	Null Dereference	Correctness	Race Conditions
Bare Metal C	✓	✓	✓	✓	✓
GERT	panic	✗	panic	✓	detectable

Figure 2-1: Bugs That GERT Programs Can Have

Compared to a bare-metal C program, a GERT program is memory safe. The table in 2-1 summarizes the bugs that GERT programs will not experience. Instead of crashing from a memory safety bug, a GERT program will panic. In Go, panics can be recovered so execution can be resumed. GERT does not categorically prevent any other bugs from occuring, such as correctness bugs or race conditions but Go does have a race detector which can help identify race conditions.

2.3 Tolerating the Garbage Collector

Go uses a garbage collector to clean up danging pointers and prevent use-after-free errors, but it can have serious performance implications for an embedded system. First of all, the garbage collector may allow an excessive amount of unused memory to build up before running a scan. This can cause SOCs with small amounts of RAM to run out of memory. GERT does not attempt to alleviate this problem in any way because the operating assumption is that powerful multicore SOCs will also have at least 1GB of RAM. A quick scan through DigiKey does confirm this assumption. The other problem with the garbage collector is that it has to "stop the world" sometimes in order to perform its function. This process involves checkpointing the executing program before pausing it and scanning all of its pointers.

GERT mitigates the effects of a GC collection in two ways: An intentional way and an unintentional way. In an embedded system, the danger of a GC collection

lies in missing external events. The way GERT intentionally mitigates the effects of the garbage collector is by allowing interrupts to be serviced even while the world is stopped. This is possible as long as the interrupt handler does not execute a blocking operation. Most embedded systems have bounded inputs and outputs so this allows for the same memory to be reused.

Chapter 3

The Go Runtime On Bare Metal

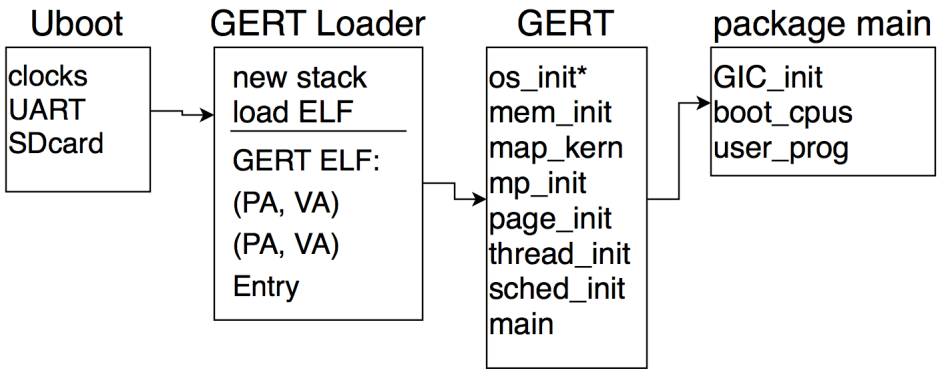

Figure 3-1: GERT Boot Process

Even though Go code is compiled, it relies on a runtime to coordinate certain actions with the OS and support its concurrency model. Timers, locks, and file descriptors are just a few of the OS abstractions that the runtime hinges on in order to function at all. This means that getting compiled Go code to run bare metal on an SOC requires more than just a boot loader, the Go runtime itself must be modified to work without any OS abstractions. This poses a bootstrapping problem because any modifications made to the Go runtime's initialization process must not inadvertently cause it to use an abstraction that does not yet exist. For example, creating a new object with *make*() during the boot process would be disasterous if the GC has not yet been initialized. Modifying the Go runtime to boot on bare metal is tricky process

because all additions should be made in a non-destructive way that still preserves all of Go's useful primitives, including the standard library.

In observation of these constraints, GERT boots via a 3-step process as shown in figure 3-1. In step 1, u-boot is used to set up device clocks and load step 2 off of an SD card. Step 2 prepares the Go stack with arguments and environment variables before jumping into GERT. Step 3 runs inside of GERT and it finishes the boot process by initializing virtual memory, threading, and interrupt handlers.

3.1 Step 1 Bring Up

The u-boot loader is used to initialize device clocks and chain load the GERT bootloader. Unlike desktop PCs, SOCs have no BIOS so it is entirely the programmer's job to initialize device clocks and power on essential peripherals, such as the memory controller. U-boot is a simple bootloader which abstracts this laborious process for all of the SOCs which it supports. GERT uses it to chain load its own, more specialized, bootloader.

3.2 Step 2 GERT Kernel Installation

The GERT loader is a C program which sets up the initial Go stack and decompresses the Go kernel ELF into RAM. GERT is compiled as a Linux Go program, so the Go runtime expects to find arguments and environment variables in its initial stack. This is a convenient channel for passing information, so the GERT loader gives the size of the GERT kernel as an input argument. GERT later uses this size to determine its location in memory and initialize virtual memory.

The link address of the GERT binary must be also adjusted on a per-SOC basis in order for the Go runtime to avoid using inaccessable memory. By default, Go links and loads at 0x0. This is a disaster for most SOCs because they have reserved regions near that address. Fortunately, the Go compiler can emit code at a different link addresses so this is not a significant problem.

3.3 Step 3 Go Runtime Setup

The Go runtime forms the basis of GERT's functionality but it is not equipped to run on bare metal. The final steps of the boot process are accomplished in Go. GERT initializes the minimum set of OS abstractions which the Go runtime utilizes, before the runtime actually uses them. These abstractions are virtual memory, thread scheduling, interrupt handling, timers, and booting secondary cores.

3.3.1 Virtual Memory Setup

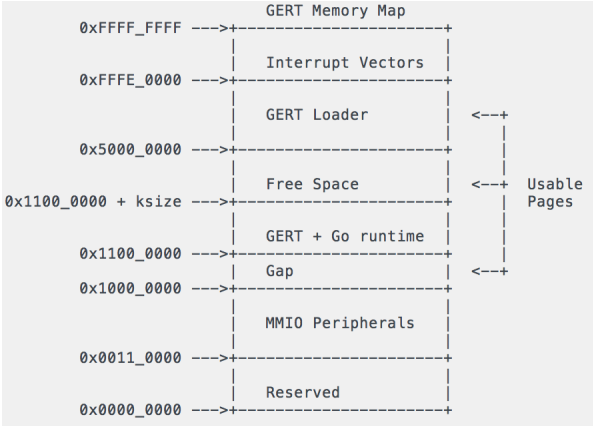

Figure 3-2: GERT Virtual Memory Layout

GERT needs to have virtual memory because the Go runtime will readily use addreses that are within reserved regions of physical memory. GERT still uses a single address space though because it has no concept of user mode or kernel mode, so there is no need to switch page tables often. Additionally, the virtual memory map changes infrequently because the Go runtime only maps memory in big chunks. For these reasons, only 1MB pages are used and they are organized in a single level table. The free pages are represented as a linked list so that map and unmap must only modify one link. The memory layout in 3-2 indicates the areas of memory that GERT can use as pages. Notice how the GERT loader is recycled, freeing the space it previously used.

3.3.2 Thread Scheduling and Trapframes

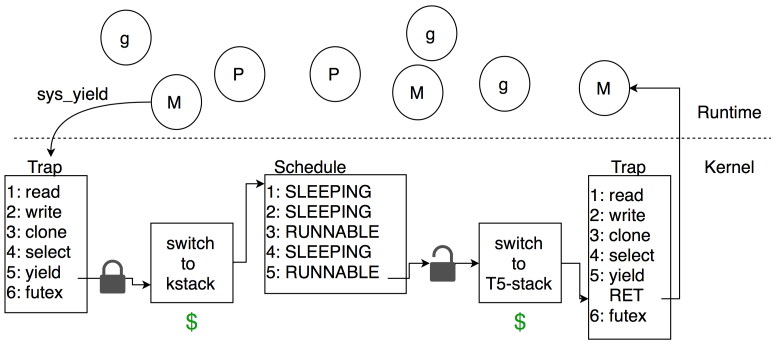

Figure 3-3: Handling Go Runtime Syscalls

GERT models the entire Go runtime as a black box whose only entry and exit points are through the syscalls it makes. This model is shown in 3-3. To be clear, there is no such thing as a syscall in GERT. Whenever the Go runtime makes a syscall, the processor mode is not changed and a new page table is not installed. Instead, every instance of the syscall instruction in the Go runtime has been replaced with a function call to *trap*. While execution is in *trap*, the Go runtime believes that the OS is servicing its syscall, but in actuality the M is still running Go code inside the GERT kernel. GERT maintains data structures that track the state of Go threads outside the runtime's knowledge. It is important that all state inside the GERT kernel is allocated outside the Go runtime's knowledge either with global variables or the kernel's static memory allocator. If this constraint is not observed, then it is possible for the garbage collector to potentially recycle memory from the kernel.

Each thread in GERT has an id, status, and trapframe associated with it (3-4). The trap frame records the state of all the registers and the location of the stack at the time that the Go runtime made a syscall.

```
type thread_t struct {
    tf        trapframe
    state     uint32
    futaddr   uintptr
    sleeptil  timespec
    id        uint32
    lock      Spinlock_t
}
type trapframe struct {
    lr   uintptr
    sp   uintptr
    fp   uintptr
    r0   uint32
    r1   uint32
    r2   uint32
    r3   uint32
    r10  uint32
}
```

Figure 3-4: Thread state and Trapframes

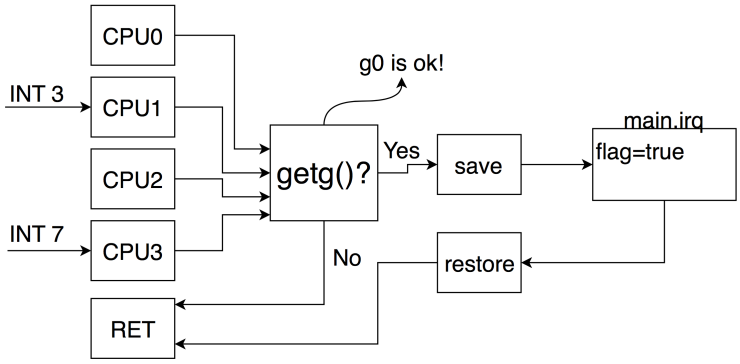

Figure 3-5: Handling Interrupts in GERT

3.3.3 Interrupts

3.3.4 Keeping Time

3.3.5 Booting Secondary CPUs

Chapter 4

How to Use GERT

Chapter 5

Evaluation

With GERT, the programmer should be able to implement concurrent embedded programs which are as performant as the equivalent C implementation, but without having to worry about concurency abstractions and memory safety bugs. In order to evaluate GERT, there are two questions that must be answered:

1. Does GERT retain good performance despite the costs of using a High Level Language?

2. Do the concurrency patterns of Go simplify the task of the programmer?

The first question is addressed by benchmarks below which measure GERT's speed in creating and responding to external events. Pin toggle (5.2) measures the maximum pin toggle frequency. Interrupt response time (5.3) measures the time it takes to respond to an external interrupt. Pulse counting (5.4) counts number of pulses at increasing frequencies. Concurrent response (**??**) measures how many external events GERT can concurrently respond to.

The second question is harder to answer because difficulty is a subjective measure. This thesis attemps to show that GERT presents a better framework for concurrency through two case studies: a robot sensor platform (5.7) ,which runs motors and reads sensors, and a galvo laser projector (5.8) which traces images onto a surface by rotating mirrors at high speeds. The case studies present a real-world experience for using GERT.

5.1 Experimental Setup

In all tests, GERT is run on an i.MX6Quad SOC and all measurements are taken with a Rigol DS1054Z oscilloscope. When GERT is compared to Linux, the SOC runs Debian 8 "Jessie" with hardfloat support. GERT is also occasionally compared to a Teensy 3.2 microcontroller running C. The Teensy 3.2 uses a Cortex M4, which is specifically intended for microcontroller applications, and has good real-time performance. Its event response times represent a good comparison point for GERT and Linux. The Teensy platform has poor concurrency support though because its Cortex M4 is a single core processor.

All of the tests make use of the GERT "embedded" package, which is an example driver library that was developed for the i.MX6Quad. The embedded package is not intrinsic to GERT's functionality, nor was it optimized for performance in any way. The embedded package just aims to provide a template for how drivers can be written in the Go language. It currently includes drivers for the UART, SPI, PWM, GIC, USDHC, GPIO, and GPT peripherals. The embedded package also includes an implementation of the FAT32 file system, which is currently unused.

5.2 Pin Toggle Frequency

This test measures the speed at which GERT can toggle a simple GPIO pin on the iMX6 Quad SOC. In ARM assembly, this can be implemented in 4 lines, but compilers and abstractions can increase the instruction count. Higher pin toggling frequency indicates less code in the critical path. GERT toggles the GPIO pin by directly interfacing with the GPIO peripheral on the iMX6, but userspace Linux code must use the sysfs driver. Results are shown in figure 5-1.

The results of the pin toggle initially show that GERT underperforms compared to user-space Linux C. The reason became clear after tracing GERT's execution in QEMU: the slowdown is caused by Go's interfaces in the embedded package. The GPIO driver in the embedded package uses Go interfaces to abstract all of the different

Platform	Avg GPIO Toggle Rate
ASM	1.65MHz
GERT Static	568KHz
Linux C	263KHz
GERT	154KHz
Linux Go	127KHz

Figure 5-1: GPIO Toggle Rates of Different Platforms

pins. In order to toggle a single pin with an interface requires 47 instructions: 2 function calls, 19 loads, and 11 stores. In order to increase the toggle speed, a new static GPIO driver was developed for GERT. The new driver is just a thin layer over the memory-mapped registers. With static device ID's, the toggled pin can be inferred at compile time instead of run time. The performance of the static driver is also shown in the GERT static row of figure 5-1. With a static driver, GERT is able to toggle a pin faster than user-space C code, but it is still very slow compared to assembly.

But what if a Linux kernel module toggled the pin instead of userland code? Pin toggle from inside a kernel module would certainly be faster than userland C and it would also likely be faster than GERT. Operating from within kernel space is very dangerous though because it lacks the protections that GERT and userspace have.

5.3 Interrupt Response Time

This test measures the time it takes GERT to respond to an external event with another external event. Specifically, it is the time it takes to produce a rising edge on a GPIO pin in response to a falling edge on a different GPIO pin. Faster response times are important for real time control systems, such as ABS brakes in a car. GERT and the Teensy detect the event with hardware interrupts but Linux polls the input pin in a tight loop because the userspace sysfs driver does not expose interrupt attachment points. Results are shown below in figure 5-2.

The event response times follow the increasing abstraction cost for each system.

Platform	Event Reponse Time
Teensy 3.2	1μs
GERT	6.3μs
Linux C	10μs
Linux Go	30μs

Figure 5-2: Event Response Times of Different Platforms

The Teensy is very fast because its interrupt controller is vectored and interrupts do not cause a stack switch. This means that the Teensy can flip a pin within a few cycles of receiving the interrupt. GERT is slower because the iMX6 does not have a vectored interrupt controller, the interrupt stack must be switched, and the interrupt handler is written in Go. When GERT gets an interrupt, it must save its current state, decide which interrupt it received, and execute its Go handler. Despite this complexity, the iMX6 can execute more instructions in less time because of its very high clock rate (792MHz vs 96MHz) so it can keep up with the Teensy.

The Linux configuration is slower because external interrupts cannot directly trigger a response from userspace. In Linux, the GPIO pins are represented by file descriptors so IO is performed by reading/writing from the appropriate file. In response to an external interrupt, the Linux kernel sets a flag on the file descriptor which means that there is data to read. The userspace program does not actually see the data until it is scheduled again.

As before, a Linux kernel module would likely attain the same performance as GERT (or even more), but would lose the benefits of a HLL.

5.4 Pulse Counting

This test measures GERT's ability to count incoming pulses. Missed pulses indicate an excessively long interrupt handler that is still executing when the next pulse arrives. None of the platforms were configured for re-entrant interrupt handlers so they can all potentially miss pulses. The pulses are provided by a Xilinx Artix 7 FPGA and they are variable in frequency and count. The Linux benchmarks are run with polling again

because the userspace sysfs driver does not support interrupt attachment. Results are shown in figure 5-3.

Platform	Pulse Count	Max Pulse Rate	Missed Pulses
Teensy 3.2	10	2.50MHz	2
GERT	10	161KHz	1
Linux C	10	161KHz	4
Linux Go	10	50KHz	1

Figure 5-3: Pulse Counts of Different Platforms

GERT and userspace Linux C achieve similar performance on this test. Linux just missed a few pulses.

The Teensy registers more pulses than any other platform because of its compact and efficient architecture. A dissassembly of the Cortex M4 pulse count binary reveals a fully vectorized interrupt whose routine only contains 5 instructions and zero conditional statements.

5.5 Concurrent Events

Since the iMX6 has 4 cores, GERT should be able to concurrently register 4 interrupts. The Teensy is configured to produce 10 rising edges at 100KHz on a single pin and GERT must concurrently register them on 4 different GPIO pins. The sum total of registered edges should be $10 \times 4 = 40$.

Because polling is a blocking operation, a multithreaded C program that concurrently polls 4 pins is used for this test. The multithreaded C program is also compared to a single-threaded C program. A multithreaded Go program which uses goroutines to monitor pin states is also included. Results are shown in figure 5-4.

<talk talk talk>

Platform	Pulse Count	Pulse Rate	Min Registered	Max Registered
GERT	10	100KHz	36	42
Linux C Multithread	10	100KHz	32	33
Linux C	10	100KHz	9	13
Linux Go	10	100KHz	TBD	TBD

Figure 5-4: Concurrent Pulse Counts of Different Platforms

5.6 Benchmark Conclusions

Despite being written in a HLL, GERT can usually outperform userspace Linux C code in the benchmarks that were conducted. GERT's performance trailed Linux in the GPIO toggle test but, after changing the driver to a static model, it also beat Linux in that test too. This is a promising result because it shows that a HLL that provides the same isolations as a userspace can run on bare metal and achieve higher performance.

Unlike Linux, GERT utilizes a true interrupt handler for delivering events. In reality this doesn't seem to matter a lot because the latency difference is still just a few microseconds. Unfortunately, GERT's interrupt handling process requires a context switch so it is still too slow for meeting hard deadlines.

GERT and Linux have similar concurrent capabilities. When the frequency of external events exceeds the response time of a single core, the events can be split among multiple cores. In GERT, though, this threshold frequency is about 4μs faster so the programmer can avoid dedicating additional cores in some cases.

The Go garbage collector was never an issue during any of the tests because the benchmark programs were all static. Without memory to reclaim, the GC never had to run. However, if the GC did run, the only test that would have been affected is the pin toggle test because the GC is allowed to stop the world. In GERT, interrupts can trigger even when the world is stopped so the GC will not affect the response time and pulse counting benchmarks even though it would affect them in the Linux Go benchmarks.

5.7 Case Study: Robot Sensor Platform

Type	Line Count
Initialization	23
Event Loop	19
ADC Driver	106
Motor Driver	98
UART	41
Abstractions	20

Figure 5-5: Code Breakdown of Robot Sensor Platform

In order to evaluate GERT on a realistic workload, I put it on a robot that was donated to me from MIT's MASLAB competition. Among other things, the robot has two drive motors with encoders and also several Sharp GP2Y0A21YK infrared distance sensors on its perimeter. I wrote a program in Go using GERT to process all of these event sources at the same time and operate the robot. The working robot can move and poll the distance sensors in response to user input. It can also measure the rotation rate of its drive motors.

The code breakdown of the robot is included in fig. 5-5. The line counts include the code that must be included in addition to the base GERT system in order to produce the functional robot program. It is all written in Go.

5.7.1 Overview

The main body of the robot program is an event loop which waits for events coming out of an event channel (fig. 5-6). Independent goroutines monitor each sensor and send events into the event channel. There is a single goroutine that monitors the event channel and manipulates state in a non-blocking manner.

The robot program uses Go's higher order functions and closures in order to create a sensor polling helper function (fig. 5-7). In this paradigm, every sensor gets its own goroutine which sends data back into a central event loop.

The robot program also configured the GPIO library to use interrupts in order to count pulses on the encoder (fig. 5-8).

```
select {
    case event := <-event_chan:
        fmt.Printf("%v\n", event)
        switch event {
        case "p":
            val := adc.Read(0)
            fmt.Printf("adc reads %v\n", val)
        case "w":
            drive.Forward(0.2)
        case "s":
            drive.Backward(0.2)
        case "a":
            drive.TurnRight(0.2)
        case "d":
            drive.TurnLeft(0.2)
        case " ":
            drive.Stop()
        }
    }
```

Figure 5-6: Robot Event Loop

```
type Pollfunc func() interface{}

func Poll(f Pollfunc, period time.Duration,
sink chan interface{}) chan bool {
    kill := make(chan bool)
    go func(kill chan bool) {
        for {
            select {
            case <-kill:
                return
            default:
                if period > 0 {
                    time.Sleep(period)
                }
                sink <- f() //sink is usually the event channel
            }
        }
    }(kill)
    return kill
}
```

Figure 5-7: Higher Order Polling Function

```
embedded.WB_JP4_10.SetInput()
embedded.WB_JP4_10.EnableIntr(embedded.INTR_FALLING)
embedded.Enable_interrupt(99, 0) //send GPIO1 interrupt to CPU0
.
//go:nosplit
//go:nowritebarrierec
func irq(irqnum uint32) {
    switch irqnum {
...
    case 99:
        inc()
        embedded.ClearIntr(1)
...
    }
}
.
func inc() {
    count += 1
}
```

Figure 5-8: Encoder Interrupt

With these powerful set of abstractions, adding events or sensors into the event loop is simple because only a Pollfunc() must be implemented. As an added bonus,

this GERT program is automatically concurrent because the Go and GERT schedulers will move idle cpus to any available goroutine. The rest of this case study explains how the sensors are interfaced with GERT.

5.7.2 PWM Motor Control

The robot has an MDD10A motor speed controller for controlling the two drive motors. This device expects a pulse-width modulated signal (PWM) on its input pins in order to direct power into the motors. A PWM signal has a constant period and the signal is a logical "on" for part of the time and "off" for the rest of the time (fig. 5-9). The ratio of "on" time vs the period is called the duty cycle. It is this percentage which the motor controller translates into a speed for the motor.

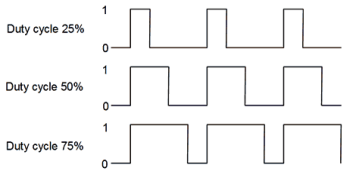

Figure 5-9: Sample PWM Signals

The iMX6Q includes an on-board PWM peripheral which can output several channels of PWM at a variety of periods and duty cycles. GERT contains a driver for this PWM peripheral in its embedded package. The PWM peripheral requires no maintenance once it is configured so the cost of outputting a PWM signal is essentially a few loads and stores every time the user changes the period or duty cycle. The API is shown in fig. 5-11.

The driver is organized in a typical C fashion where the memory map of the peripheral is represented in a structure (fig. 5-10). Go provided little benefit for writing the driver. Even though Go is a systems language, it has poor support for reading/writing arbitrary memory. In Go, the programmer cannot align or pad structs and they must use the unsafe package in order to modify memory addresses with unsafe casts. This makes for a generally unsatisfying experience when writing drivers.

```
type PWM_regs struct {
    CR   uint32
    SR   uint32
    IR   uint32
    SAR  uint32
    PR   uint32
    CNR  uint32
}
```

208_4000	PWM Control Register (PWM2_PWMCR)	32
208_4004	PWM Status Register (PWM2_PWMSR)	32
208_4008	PWM Interrupt Register (PWM2_PWMIR)	32
208_400C	PWM Sample Register (PWM2_PWMSAR)	32
208_4010	PWM Period Register (PWM2_PWMPR)	32
208_4014	PWM Counter Register (PWM2_PWMCNR)	32

Figure 5-10: PWM Register Representation

```
func (pwm *PWM_periph) Begin(freq khz)
func (pwm *PWM_periph) Stop()
func (pwm *PWM_periph) SetFreq(freq khz)
func (pwm *PWM_periph) SetDuty(dutycycle float32)
```

Figure 5-11: PWM Driver API

5.7.3 Distance Sensor Reading

The Sharp distance sensor outputs an analog voltage proportional to its distance from the nearest object. A Microchip MCP3008 8-channel ADC is used to convert this voltage into a digital signal. The MCP3008 communicates in clocked serial (SPI) with 24bit data frames so the robot program uses GERT's SPI driver (fig. 5-12). Much like the PWM peripheral, the SPI peripheral has multiple channels that can each concurrently send and receive data. The SPI driver also requires no input from the user except for the data to transmit and receive.

```
func (spi *SPI_periph) Begin(mode, freq, datalength, channel uint32)
func (spi *SPI_periph) Send(data uint32)
func (spi *SPI_periph) Exchange(data uint32) uint32
```

Figure 5-12: SPI Driver API

5.7.4 Encoder Reading

The robot program also includes a motor speed monitor which uses the encoders. Encoders emit a pulse every time the motor rotates a known amount. This amount is variable depending on the encoder resolution. The encoders on the robot motors emit pulses at a max rate of 4KHz, corresponding to maximum motor speed. GERT had no difficulty picking up these pulses because this frequency is far less than the max pulse frequency in fig. 5-3.

The robot program asynchronously reads encoders with a special goroutine which computes the pulse difference every second (fig. 5-13). The result is sent into the event channel.

```
1  //count is updated by the interrupt routine
2  //and it is the amount of encoder pulses
3  go func() {
4    for {
5      old := count
6      time.Sleep(1 * time.Second)
7      new := count
8      event_chan <- new - old
9    }
10 }()
```

Figure 5-13: Motor Speed Monitor

5.7.5 Complications

Systems do not work perfectly, and this robot is no exception. The switching motor controller used on this robot emits a lot of noise. The 5v noise spikes measured on the oscilloscope wreaked havoc on the 3.3v single-ended signals that the iMX6 operates with, causing serial communication failures and spurious interrupts. To deal with this, the robot's motors are connected to an external power supply before taking encoder measurements in order to remove noise from the digital circuits. Consequently, the physical robot cannot move when the motors are connected to an external power supply.

5.7.6 Result

GERT is a plausible embedded toolkit to use for robots that incorporate many sensor systems. By utilizing Go's language features, an embedded firmware engineer can implement a complicated sensor integration platform on top of GERT without worrying about issues like scheduling or shared memory. Because it is written in Go, the robot sensor platform never experienced a single use-after-free, index out of range, or memory safety bug. As an added bonus, the robot sensor platform also does not contain a single lock despite the fact that every sensor runs in its own thread, deadlock will never occur.

The most painful part about using GERT is writing drivers. Interacting with MMIO peripherals inherently requires unsafe writes and reads to arbitrary memory, but Go tries very hard to stop the programmer from doing this. Successful drivers for MMIO peripherals must defeat the type system. In the end, a GERT driver looks like the equivalent C driver but with many more explicit casts. If a future version of Go deprecates the unsafe package, it will be catastrophic for GERT's current implementation.

5.8 Case Study: Laser Projector

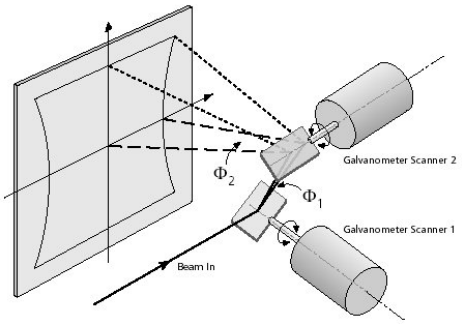

Figure 5-14: Mirror Galvanometers

A scanning-mirror galvanometer laser projector is a device that deflects a laser beam off of several mirrors in order to draw an image on another surface, as shown in fig. 5-14. If the entire image can be scanned faster than 24Hz, then the light appears to blend and the human brain perceives it as a single image rather than many points. The maximum rate at which the projector can trace points is bounded below by the speed of the galvos and bounded above by the speed of the software. In this case study, GERT is used to implement a laser projector with a red laser.

5.8.1 Overview

I selected a laser scanner for this case study because I have a lot of experience programming them in C. It is interesting to observe how GERT can alter the experience.

Points for the scanner are generated from a vector graphics file on a desktop computer and stored on an sdcard before GERT loads them and traces the image. The laser projector, unlike the robot sensor platform, does not have to process any external events or manage concurrency so it is a relatively simple program.

The only challenge for the laser program is to trace points fast enough so that the image appears smooth. To do this, the laser program runs a dedicated goroutine, *lasermon* which loops over all of the points in a circular buffer and sends them in order to a Microchip MCP4922 DAC. The DAC converts the digital position signal into an analog voltage and then sends that voltage signal into an analog servo circuit, which sets the position of the mirrors.

5.8.2 Point Serialization

The laser program uses Go's native Gob library in order to serialize and deserialize points. The laser projector is currently limited to two dimensional images with a single red laser, so only three properties must be stored for each point: X position, Y position, and Color. The DAC only has a 12bit resolution so 16bit integers are used to store each point. The struct is shown below in fig. 5-15

```
1 type CompactPoint struct {
2     X     uint16
3     Y     uint16
4     Color uint8 //either 0 or 1
5 }
```

Figure 5-15: Laser Point Structure

In order to store points, a seperate encoding program running on a desktop computer encodes an array of *CompactPoint* structs into a Gob object and writes them to a file. Next, a utility called *go-bindata* is used to embed the gob file into the laser-scanner program. Then the laserscanner reads the gob'ed data during initialization. Code is shown in 5-16.

```
var points []CompactPoint
    contents, err := Asset("bindata.gob")
    if err != nil {
        panic("bindata not found")
    }
    r := bytes.NewBuffer(contents)
    d := gob.NewDecoder(r)
    err = d.Decode(&points)
    if err != nil {
        fmt.Printf("error de-GOBing:\n")
        panic(err)
    }
```

Figure 5-16: Laser Point Structure

5.8.3 Path Tracing

Lasermon draws points by looping through a circular buffer and transmitting each point to the DAC. If lasermon sends points too fast, then the scanner cannot keep up and it displays junk. If lasermon sends points too slow, then the scanner does not trace the image fast enough and it does not look static.

Lasermon attempts to mitigate these issues by heuristically adjusting how long it should wait between sending successive points to the scanner. If the norm-2 distance of the current point is far from the last point, *lasermon* waits longer in a busy loop before transmitting the next point. With tuning, this approach can work for a specific image, but it does not work very well in general because the laser scanners are not an LTI system. A better solution to the scanner problem is a state-space feedback controller which includes *lasermon* in the loop. That work is out of scope for this thesis though.

5.8.4 Result

The laser scanner programmed with GERT is able to successfully trace an SVG of the CSAIL logo at 24Hz (fig 5-17). The scan rate is not a limitation of GERT, but the mirror galvos themselves. GERT is capable of transmitting points at several megaherts - the rate of its SPI peripheral - but the galvos cannot keep up with it.

The laser scanner takes no user input or external event input so it is not a very complicated program. The most useful feature of Go, for the laser scanner, is the Gob library for point serialization. Based on my experience programming other laser

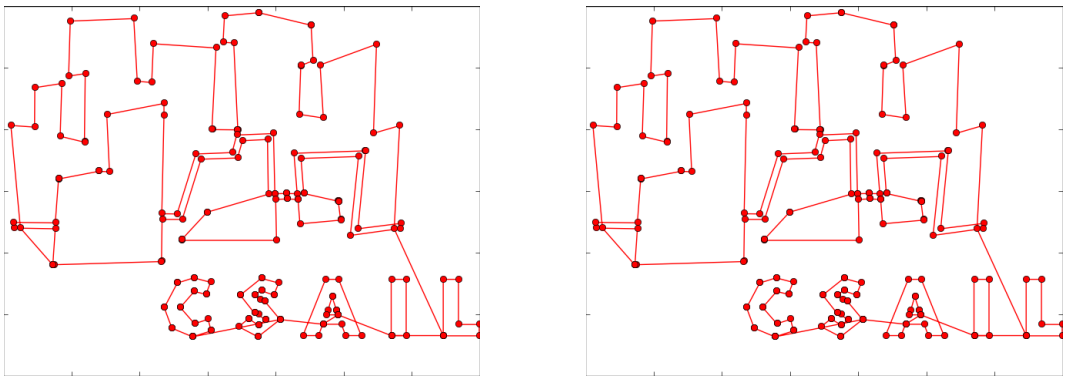

Figure 5-17: CSAIL Logo Generated vs Traced

scanners in C and 8051 assembly, this one was much easier to implement because Go caught every out-of-bounds index error and also prevented memory use errors from ever occuring.