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

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 SOCs are still largely written in either bare-metal C or userspace C because high-level languages running in userspace can have too much latency.

This thesis proposes a new system called G.E.R.T, the Golang Embedded Run-Time, for multi-core ARM processors. GERT is a modified version of the Go runtime for bare-metal operation on multi-core ARMv7a SOC's. It is used 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

First I would like to thank Frans Kaashoek for introducing me to the field of Operating Systems and advising me on this thesis. Next, I would like to thank Cody Cutler for helping me figure out my mistakes whenever I broke the Go runtime. I am also thankful to some of my closest friends: Tina, Ana, Hannah, and Corey for supporting me and telling me my laser projector is cool. Finally, I am grateful to my parents, Rula Coroneos and Emanuel Coroneos, for always supporting my education and motivating me to achieve more. Thank you for sending me to MIT, I have learned so much.

Contents

1 Introduction **13**

1.1 Motivation 13

1.2 GERT 15

1.3 Outline 15

2 Why Write System Code in Go? **17**

2.1 Go Runtime Organization 17

2.2 Go Memory Safety 18

2.3 Tolerating the Garbage Collector 19

3 The Go Runtime On Bare Metal **21**

3.1 Step 1 Bring Up 22

3.2 Step 2 GERT Kernel Installation 22

3.3 Step 3 Go Runtime Setup 23

3.3.1 Virtual Memory Setup 23

3.3.2 Thread Scheduling and Trapframes 24

3.3.3 Interrupts 26

3.3.4 Keeping Time 28

3.3.5 Booting Secondary CPUs 28

4 How to Use GERT **29**

4.1 API Inspiration 29

4.2 Writing Interrupt Handlers 29

4.3 Building a GERT Program 30
4.4 Design Considerations 31
4.5 Writing Drivers 31

5 Evaluation **33**

5.1 Experimental Setup 34
5.2 Microbenchmarks 34
5.2.1 Pin Toggle Frequency 34
5.2.2 Response Latency 36
5.2.3 External Event Throughput 37
5.3 Microbenchmark Conclusions 38
5.4 Case Study: Robot Sensor Platform 39
5.4.1 Overview 40
5.4.2 PWM Motor Control 41
5.4.3 Distance Sensor Reading 43
5.4.4 Encoder Reading 43
5.4.5 Complications 44
5.4.6 Result 44
5.5 Case Study: Laser Projector 45
5.5.1 Overview 45
5.5.2 Point Serialization 46
5.5.3 Path Tracing 47
5.5.4 Result 47
5.6 Evaluation Summary 48

6 Conclusion and Future Work **49**

List of Figures

2-1 Bugs That GERT Programs Can Have 19

3-1 GERT Boot Process 21
3-2 Memory Map Before and After Boot 23
3-3 Handling Go Runtime Syscalls 24
3-4 Linux Syscalls That Are Re-implemented in GERT 25
3-5 Thread state and Trapframes 26
3-6 Handling Interrupts in GERT 26

4-1 GERT Program Directory Layout 30

5-1 iMX6 Peripheral Latency 35
5-2 GPIO Toggle Rates of Different Platforms 35
5-3 Event Response Times of Different Platforms 36
5-4 Platform Event Throughput as CPU's and Events Increase 38
5-5 Code Breakdown of Robot Sensor Platform 39
5-6 Robot Event Loop 40
5-7 Higher Order Polling Function 41
5-8 Encoder Interrupt 41
5-9 Sample PWM Signals 42
5-10 PWM Register Representation 42
5-11 PWM Driver API 42
5-12 SPI Driver API 43
5-13 Motor Speed Monitor 43

5-14 Mirror Galvanometers from [6] 45
5-15 Laser Point Structure 46
5-16 Laser Point Structure 46
5-17 CSAIL Logo Generated vs Traced 47

List of Tables

Chapter 1

Introduction

This thesis investigates the feasibility of using a type-safe, high-level, garbage-collected language for bare-metal programs on a multi-core SOC. We hypothesize that the operating system provides redundant and costly isolation to a HLL program running in userspace. This can reduce execution speed and increase latency to the point where an embedded programmer is forced to use C and lose the benefits of a HLL. By running the high-level code directly on bare-metal, the program can have memory-safety and good concurrency support, but also higher performance than in userspace.

1.1 Motivation

Modern embedded systems are composed of multi-core 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 sword. 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 HLL's are that those nice features come at a cost of garbage collection and many runtime checks. Despite the shortcomings, HLL's are still preferred to C programs because fast, modern computers can neutralize the performance cost. Now that embedded devices are also very fast, the possibility of using a HLL on a performant embedded system is attractive.

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. Popular platforms that use this paradigm include the Raspi [8] and Beaglebone [1]. However, embedded programs that run in user space suffer from significant event latency because of OS isolation mechanisms that exist primarily for buggy C programs. Memory safe programs written in a HLL do not need to run in different address spaces and they already ensure that invalid pointers will not be dereferenced, so the OS does not need to waste more CPU time redundantly ensuring these properties. Unfortunately, the OS has no way to disable its isolation mechanisms for a single program, so embedded programmers who want to use a HLL must accept the unnecessary performance penalty.

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 multi-core SOCs. Singularity [14] and Biscuit [13] are desktop operating system kernels, written in Sing# and Go, which focus on hosting user-space programs. Copper [2] and MicroPython [5] are small embedded tool kits, written in Rust and Python, which aim to provide a high-level programming environment for single-core microcontrollers. Multi-core SOCs have, so far, been left out of the picture.

1.2 GERT

This thesis presents a new embedded toolkit, the Golang Embedded RunTime (GERT), which is specifically intended for concurrent, bare-metal embedded applications. GERT is a modified version of the Go runtime which can boot on a bare-metal ARMv7a system and execute Go code. With GERT, programmers can write embedded programs in Go and run them on multi-core bare-metal hardware. The Go language, instead of the operating system, provides safety and concurrency abstraction.

1.3 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 review of the results and their implications

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 [4].

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 stopped while the GC runs, but recent improvements to Go have enabled concurrent garbage collection [3]. 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 [10]. These issues are so commonplace in userspace programs, that OS kernels attempt to preempt the problem by inserting guard pages all over the user program's address space. In a

[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

bare-metal 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 dangling 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, one intentional and one unintentional. 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. The other way GERT unintentionally coexists with the garbage collector from the fact that many embedded programs are inherently static. Static embedded programs have a known memory use at compile time so there is no need to allocate memory at run time and count its references. In fact, some embedded platform libraries do not even have *malloc* so all memory use must be static.

Chapter 3

The Go Runtime On Bare Metal

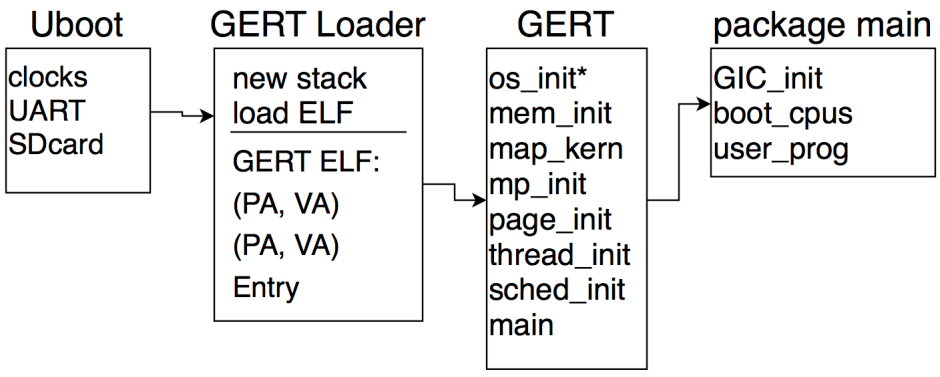

Figure 3-1: GERT Boot Process

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*() [1] during the boot process would be disastrous if the GC has not

[1] In Go, new objects are created with *make*

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 the code for 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

Unlike desktop PCs, SOCs have no BIOS, so it is entirely the programmer's job to initialize device clocks and power on essential peripherals like 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. The u-boot loader is used to initialize device clocks and execute the GERT 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; for example, the GERT loader uses this to pass the size of the GERT kernel. 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 inaccessible memory. By default, Go compiler links and loads at 0x0. This is incorrect for most SOCs because they have reserved regions near that address. Fortunately, the Go compiler can generate code

at a different link addresses by passing in a link-time flag, 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 to be used by the Go runtime, 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

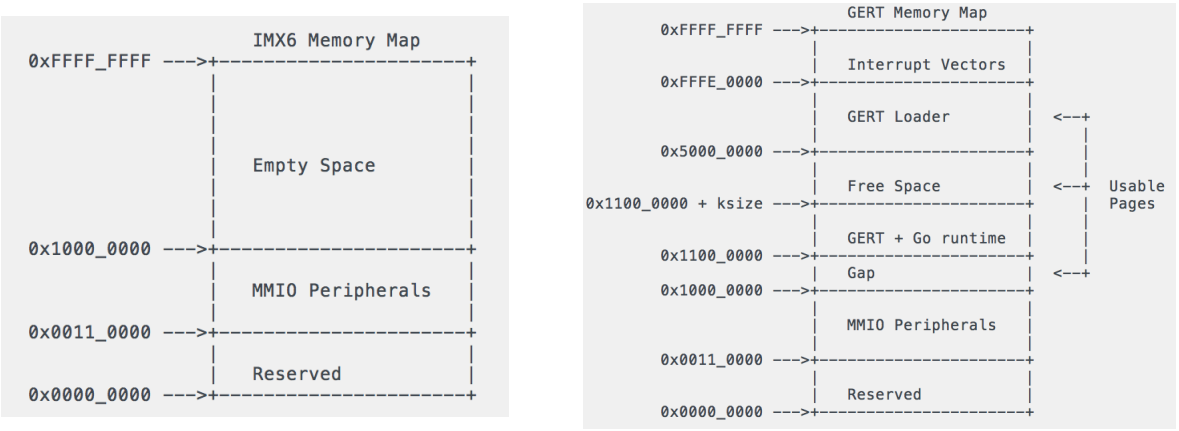

Figure 3-2: Memory Map Before and After Boot

GERT needs to have virtual memory enabled so the Go runtime can function properly and so GERT can manage available memory. Fig. 3-2 shows the physical memory map of an i.MX6 SOC before GERT is booted and after GERT has booted. Even though the runtime is linked at a high address, Go still requests memory inside the reserved regions of physical memory so a virtual address must be mapped there. GERT also uses paging to create a virtually contiguous address space which is easier to manage than a physically discontiguous address space. For example, the Go runtime sometimes requests large chunks of continuous memory. It is possible that there is not

a large enough chunk of physical memory to return due to fragmentation of MMIO peripherals, but there will be a big enough chunk in virtual memory. GERT also recycles the bootloader using virtual memory by simply marking its occupied area as usable pages.

GERT uses 1MB page tables and it rarely has to reload them. GERT has no userspace or programs that must be isolated from each other and the Go runtime only allocates memory infrequently and in large chunks. This is the only time that GERT has to reload the page tables. This static nature of GERT's memory space means that 4kb pages are unnecessary and even costly because they incur a 2-level page translation [2] , unlike 1MB pages which just require a 1-level walk. By the time user code starts running in GERT, the runtime has nearly completed all of its memory manipulation.

3.3.2 Thread Scheduling and Trapframes

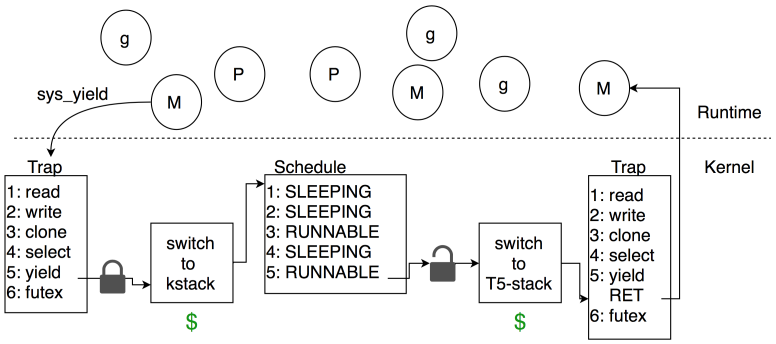

Figure 3-3: Handling Go Runtime Syscalls

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 along with the control flow of *sys_ yield*, which invokes the GERT scheduler. To be clear, there is no such thing as a syscall in GERT; it is a single-address space application that runs in privileged mode. Whenever the Go runtime makes a syscall, the processor mode

[2] 4kb paging in ARMv7a requires $2^11 = 4096$ entries in the L1 table and $2^8 = 256$ entries in each L2 table

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*, the entry point into the GERT kernel. While execution is in *trap*, the Go runtime believes that the OS is servicing its syscall, but in actuality the blocked thread is still running Go code inside the GERT kernel. The list of Linux syscalls that has to be re-implemented is shown in fig. 3-4. All syscalls with an * are expensive because they cause a stack switch to the scheduler's stack.

Syscall	Used For
exit	crash
read	read from a file descriptor. All reads go to UART
write	write to a file descriptor. All writes go to UART
clone	spawn a new M
*select	blocking read operation
*yield	yield to scheduler
mmap	allocate large chunks of memory
*futex	wait for a condition to become true
clock_gettime	goroutine scheduling and time package
getpid	runtime asks its pid

Figure 3-4: Linux Syscalls That Are Re-implemented in GERT

GERT also 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, futex address, and trapframe associated with it (3-5). The trapframe records the state of all the registers and the location of the stack at the time that the Go runtime made a syscall.

When threads go to sleep, the CPU stores their execution context in a trapframe. When a thread is scheduled, the CPU restores the contents of the thread's last trap frame and continues running the thread.

The futex, which stands for *fast user space mutex*, is a useful building block that the Linux kernel (and now GERT) provides to help with locking. Linux user space

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

Figure 3-5: Thread state and Trapframes

programs can use it to wait until a certain condition becomes true, and the Go runtime uses it extensively to monitor elapsed time and wake sleeping threads.

3.3.3 Interrupts

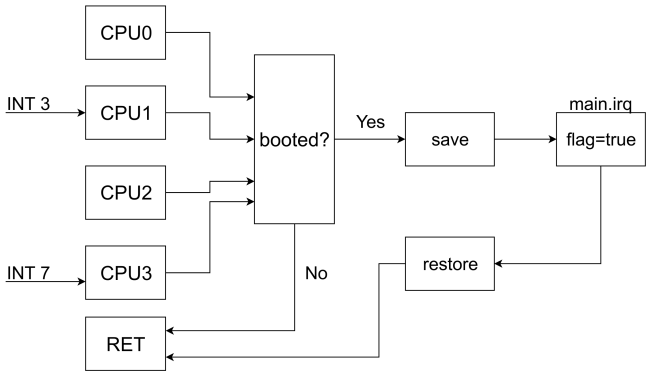

Figure 3-6: Handling Interrupts in GERT

The interrupt handling process in GERT is explicitly designed so that interrupts can always be serviced, even when Go's garbage collector is running. To achieve this design, GERT puts some restrictions on the Go code in an interrupt handler, as described below.

When a GC cycle runs, it can potentially stop the world and prevent any Go code from executing. On a desktop OS it might be OK to disable interrupts during a GC cycle because it is infrequent, but on an embedded system interrupts could occur at any time, even while the world is stopped. GERT allows interrupts to trigger

at all times by assuming that the interrupt handler does not perform any blocking operations. A block diagram of the GERT interrupt process is shown in fig. 3-6.

GERT's interrupt handlers are written in Go and they execute on dedicated interrupt stacks. When GERT receives an interrupt, the target processor automatically switches its stack to the interrupt stack and then the GERT interrupt handler saves its old execution context onto the new stack. Before proceeding any further, the interrupt handler checks a boolean flag to see if GERT has fully booted yet. This is important for preventing unintentional Go code from executing while the memory map is still undefined. While secretly executing on the interrupt stack, the processor can run any Go code as long as it does not invoke the Go scheduler. This is because the Go runtime is unaware of any interrupt code that can possibly run so it does not maintain any G's or M's to run it. Thus, from the view of the Go runtime, the interrupt handler is running on an unknown secret stack which attaches itself to whichever M was interrupted.

Interrupt handlers in GERT should not execute blocking operations or allocate memory from the Go heap [3]. The garbage collector is unaware of the hidden interrupt stacks so any heap-allocated objects on the stacks will become memory leaks because the GC will never scan it. Fortunately, allocating heap objects in the interrupt handler will cause a panic because *make()*, and many other blocking operations, need to acquire a lock from the runtime. This is dangerous because the runtime keeps track of which Ms are holding locks and interrupt code can violate its beliefs and cause an unrecoverable panic.

Interrupt routines should be short and execute quickly in order to avoid missing sequential interrupts. This limits acceptable operations in an interrupt handler to toggling global boolean flags and non-blocking reads/writes to memory. A typical GERT program is unaffected by these constraints because it can use a goroutine to monitor a flag which is only toggled in the interrupt handler.

[3]Go maintains a pool of free memory from which it allocates new objects. Most computer programs have a heap, not just Go programs.

3.3.4 Keeping Time

Go needs a timer to schedule goroutines properly. GERT uses the 64bit ARM global timer which is part of the general ARMv7a architecture [11]. In the timer, each tick is about $2ns$ so this means it will not overflow for 1169 years. This means that GERT does not need a special timer interrupt to know when the counter rolls over. The current value of the timer is returned whenever the Go runtime calls the *clock_gettime()* syscall by reading from its MMIO address.

3.3.5 Booting Secondary CPUs

GERT boots auxiliary CPU's in two stages. After power on, the primary CPU starts executing at address 0x0 but the secondary CPU's are held in a *wait for interrupt* state. GERT creates an initial 4kb stack for each additional CPU, which also doubles as its interrupt stack, before instructing them into a holding pen. The secondary CPU's stay in the holding pen while GERT and the Go runtime finish booting because the virtual memory map is in a state of flux. After GERT has booted, the user must call *Release()* which causes the secondary CPU's to reload their page tables and exit the holding pen into the GERT scheduler. Now the secondary CPU's can each run Go threads normally.

Chapter 4

How to Use GERT

It would be silly for this thesis to present a software toolkit and then neglect to show how to use it.

4.1 API Inspiration

GERT's API is based on Arduino's because of its remarkable simplicity. In order to get started with GERT on a Freescale iMX6 SOC, the programmer must only implement three functions: *user_init*, *user_loop* and *irq*. *user_init* should contain code that is run only once on startup, *user_loop* should contain the main event loop of the embedded program, and *irq* is the IRQ handler.

4.2 Writing Interrupt Handlers

Since interrupts can happen even while the world is stopped, the programmer must make sure that *irq* never executes a blocking Go function or allocates memory on the Go heap with *make()*. These constraints are easily manageable by breaking the program interrupt logic into two pieces: a monitoring goroutine which watches a boolean flag, and the interrupt service routine which sets the flag. For example, a NIC (Network Interface Controller) driver may use interrupts to determine when a new frame should be retrieved from the NIC. Rather than reading the new frame in

the interrupt routine, the NIC driver can set a flag in the interrupt routine so that a seperate thread inside the NIC driver can retrieve the new frame. This seperate thread is just an ordinary goroutine with no constraints on the type of code it can execute.

4.3 Building a GERT Program

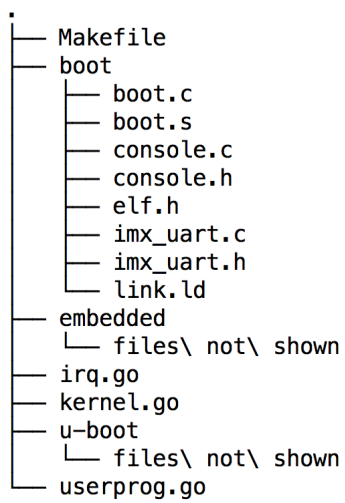

Figure 4-1: GERT Program Directory Layout

GERT needs a bootloader and entry point to start on bare metal. The directory structure of a GERT program is shown in fig. 4-1. The boot directory contains the GERT bootloader as well as linker script for it. Userprog.go and irq.go contain the user-implemented functions as well as the user-implemented irq handler. Kernel.go does three mundane things: it defines a new entry point for the Go runtime (which just sets a flag so the runtime knows it is booting on bare metal), it also finishes booting additional CPU's, and it configures the ARM Generic Interrupt Controller [12] for normal operation. The makefile is responsible for stitching the bootloader and GERT program together into an SD card image for u-boot. It uses *go build* to build the GERT program with the modified Go runtime and then inserts the GERT program as a binary blob into the bootloader's data section. The makefile also includes a target which writes u-boot and the final GERT binary to an SD card.

4.4 Design Considerations

Every SOC has a different memory map and peripherals, so GERT must be adjusted accordingly. In order to change the link address of GERT, pass "-T <link address>" as a link flag into *go build*. The link address of the bootloader also needs to be changed too inside *link.ld*. It is good practice to link the bootloader in an area of RAM that can be reclaimed once GERT enables paging.

4.5 Writing Drivers

GERT imposes no driver model on the programmer. All drivers in GERT should be written as normal Go code in the best style for the intended application. GERT exposes no safe methods for reading or writing device memory so any MMIO peripherals in the SOC must be carefully programmed using the *unsafe* package. Most MMIO peripherals arrange their registers contiguously in memory so they can be represented with a Go struct, which requires only one unsafe cast for initial assignment.

GERT currently comes with an example driver library in the form of a package called *embedded*. 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 only functions for the Freescale i.MX6 and includes drivers for the UART, SPI, PWM, GIC, USDHC, GPIO, and GPT peripherals. The package also includes a generic implementation of the FAT32 file system, which is layered on top of a *read* and *write* function that the programmer can define.

Chapter 5

Evaluation

With GERT, the programmer should be able to implement concurrent embedded programs which are as performant as the equivalent C implementation without having to worry about concurrency 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 microbenchmarks which measure GERT's speed in creating and responding to external events. Pin toggle (5.2.1) measures the maximum pin toggle frequency. Response latency (5.2.2) measures the minimum time it takes to respond to an external interrupt. Event throughput (5.2.3) measures the maximum number of total events that each platform can reliably detect, as well as how this number scales with additional CPU's monitoring more simultaneous events.

The second question is harder to answer because difficulty is a subjective measure. This thesis attempts to show that GERT presents a better framework for concurrency through two case studies: a robot sensor platform (5.4), which runs motors and reads sensors, and a galvo laser projector (5.5) 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, which sits on a Wandboard platform, 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 [9] uses a Cortex M4, which is specifically intended for microcontroller applications, and has good real-time performance. Even though the Cortex M4 has a vastly different architecture and purpose than the iMX6, its event response times provide 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.

5.2 Microbenchmarks

5.2.1 Pin Toggle Frequency

This test measures the speed at which GERT can toggle a simple GPIO pin on the iMX6 Quad SOC. GPIO pins are an MMIO peripheral on the iMX6 which requires many clock domain crossings to produce an output. Additionally, MMIO regions are marked as strongly-ordered device memory in the page tables. This means that all accesses to MMIO regions must occur in explicit program order. After the toggle program sets the state of a GPIO pin, this information must pass through the memory bus, GPIO peripheral, and IO multiplexer peripheral before arriving to the output (5-1). The latency of this whole process is determined by the 66MHz peripheral clock, which is the slowest clock in the pipeline. Therefore, the maximum frequency of the square wave that can generated by the toggle program is approximately $\frac{\frac{66MHz}{3}}{2} = 11MHz$. The presence of additional pipeline stages inside the GPIO or IO multiplexer blocks will further reduce this frequency.

In ARM assembly, pin toggle 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

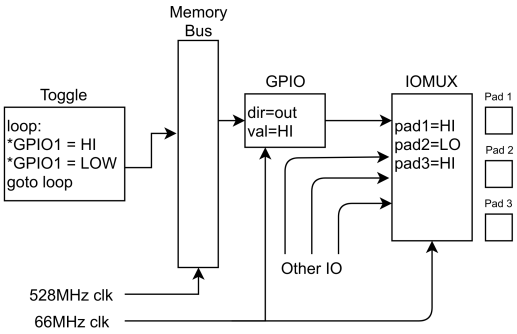

Figure 5-1: iMX6 Peripheral Latency

with the GPIO peripheral on the iMX6, but userspace Linux code must use the sysfs driver. Results are shown in figure 5-2.

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

Figure 5-2: GPIO Toggle Rates of Different Platforms

The results of the pin toggle initially show that GERT under performs 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 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 5-2. With a static driver, GERT is able to toggle a pin faster than user-space C code, but it is slower compared to assembly.

5.2.2 Response Latency

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 or medical equipment. 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 5-3.

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

Figure 5-3: Event Response Times of Different Platforms

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

5.2.3 External Event Throughput

Embedded systems sometimes have to monitor multiple sources of external events whose frequencies exceed the capabilities of a single cpu. In this situation, additional CPU's must be dedicated for the embedded system to reach its throughput target. This benchmark attempts to simulate such a scenario by delivering a clock signal simultaneously to 4 external GPIO pins on the iMX6. The total event throughput should be $n \cdot frequency$ where n is the number of GPIO pins being monitored and $frequency$ is the input clock frequency. The frequency of the clock is increased until each platform starts producing incorrect count totals.

Unfortunately, due to the semantics of the ARM Generic Interrupt Controller and the implications of its 1-N model [12], it is not possible to dedicate multiple CPU's to a single interrupt without also having expensive program logic in the interrupt handler to ensure that only one CPU can service the interrupt. Additionally, this concept does not even exist in Linux userspace because the state of each pin is represented as a file descriptor and concurrent reads are undefined for file descriptors. Instead, this test program dedicates one additional CPU for each event that must be processed, up to four CPU's and events. Doing this is easy in GERT because the target CPU of an interrupt event can be specified during attachment. In Linux, this is accomplished by dedicating each open file descriptor to a single reader thread.

The purpose of this benchmark is to observe how the total number of serviceable external events can scale with the number of dedicated CPU's. The results for each platform platform are shown in fig. 5-4.

The event throughput of every platform scales approximately linearly, with GERT achieving the highest throughput. This is an unsurprising result again because Linux file descriptors are slower at delivering events than a true interrupt handler. GERT's higher event throughput per core means that potentially fewer CPU's can be used to process the same amount of events. This reduces power and space requirements.

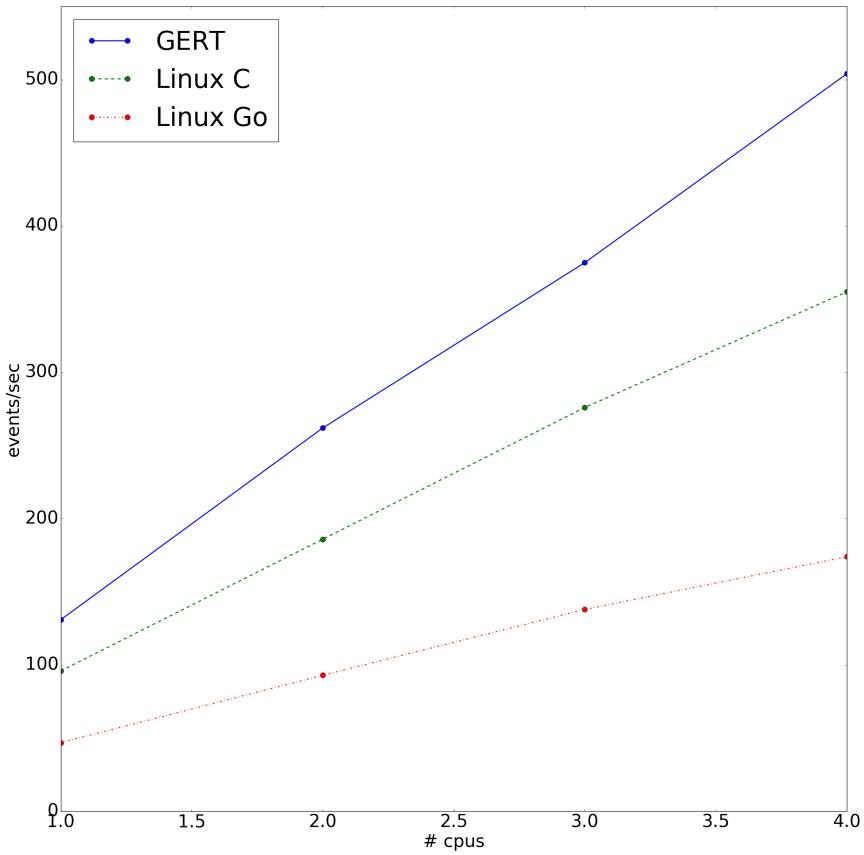

Figure 5-4: Platform Event Throughput as CPU's and Events Increase

5.3 Microbenchmark 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, which provides the same isolations as an OS kernel, can run on bare metal and achieve higher performance than a user space C program.

Unlike Linux, GERT utilizes a true interrupt handler for delivering events. This does not seem to matter very much for event latency, but it can explain the throughput

differences between GERT and Linux. In GERT, the interrupt handler can directly increment the event count, but in Linux the userspace program must indirectly observe the event by reading a file descriptor. If the user space program reads at a bad time, it can miss a few events before reading again.

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 35KHz 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 event latency and throughput benchmarks, but a GC cycle will affect these benchmarks in the Linux Go program since it polls.

5.4 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 [7] 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.

5.4.1 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 configures 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

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

```
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

will move idle CPU's to any available goroutine.

The rest of this case study explains how the sensors are interfaced with GERT and gives examples of their API's in Go.

5.4.2 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.

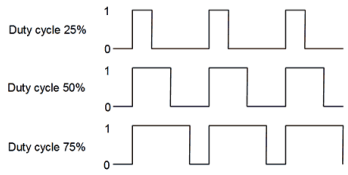

Figure 5-9: Sample PWM Signals

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, but it is no worse than C.

```
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

5.4.3 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.

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

Figure 5-12: SPI Driver API

5.4.4 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 latency is far higher than GERT's latency 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

5.4.5 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.

5.4.6 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. Go channels can still cause a deadlock though if they are used incorrectly. The Go runtime will report total deadlock when all goroutines are blocked waiting but it cannot detect partial deadlock.

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.

5.5 Case Study: Laser Projector

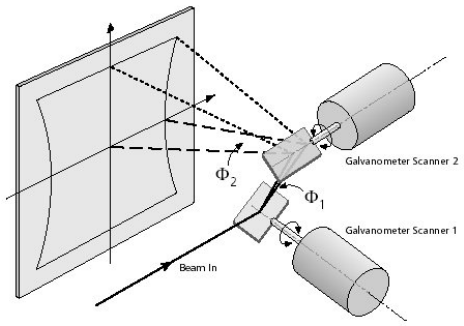

Figure 5-14: Mirror Galvanometers from [6]

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.

5.5.1 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 see 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.

5.5.2 Point Serialization

The laser program uses Go's native Gob library in order to serialize and de-serialize 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

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

Figure 5-15: Laser Point Structure

In order to store points, a separate 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 laser scanner 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

5.5.3 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.

5.5.4 Result

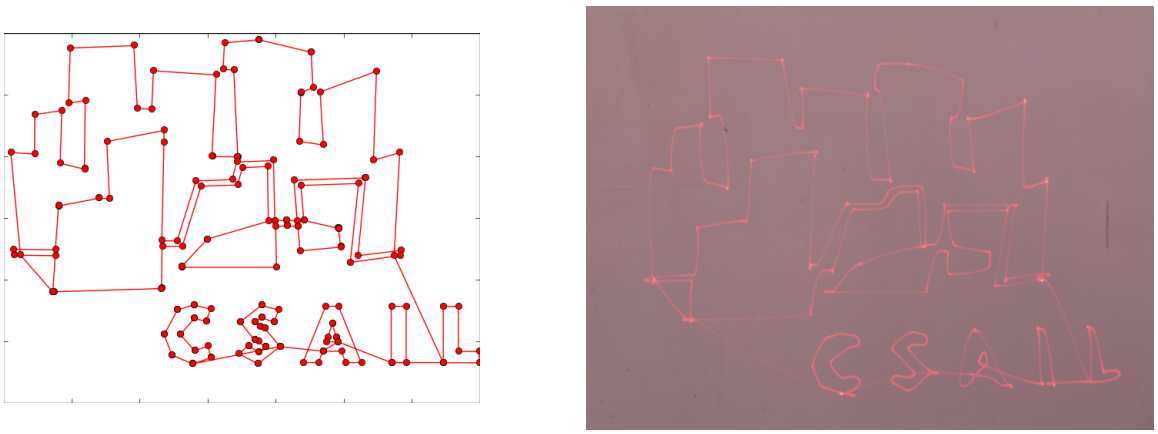

Figure 5-17: CSAIL Logo Generated vs Traced

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 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 occurring.

5.6 Evaluation Summary

GERT was evaluated both quantitatively in the benchmark section and qualitatively in the case study section. The benchmarks show that GERT can achieve lower event latency and higher event throughput than the equivalent programs written in user space C and Go code. This is a great result because it indicates that any embedded programs which are written for user space can achieve potentially higher performance in GERT.

Writing embedded programs in GERT is also easier than writing them in C. Channels and goroutines are excellent concurrency primitives. With these two primitives, program architecture can be very moduler and performant. A good example is the sensor paradigm from the Robot Sensor Platform (5.4) where each sensor became associated with a goroutine and a channel. Since GERT inherits the Go standard libraries, GERT programs are also more portable than bare-metal C programs. If GERT ever gets ported to different platforms, existing GERT applications will need few modifications to run on the new platform because the Go runtime presents the same abstractions everywhere that it runs.

Chapter 6

Conclusion and Future Work

This thesis has presented GERT as part of an effort to investigate the efficacy of using a type-safe, garbage-collected, high-level language on a multi-core embedded system. The premise of this idea is that OS kernels provide redundant and costly isolation to programs which do not require it because they are written in a HLL. GERT demonstrates, through micro benchmarks, that removing the OS from an embedded platform and running the high-level code directly on bare metal, can be more performant than even a user space C program in Linux. Writing embedded programs with GERT is also relatively painless because of Go's concurrency patterns and its portable standard library. The goal of this thesis is not to convince everyone to use GERT, but rather to show that performant embedded programs do not have to be written in C anymore. HLLs can provide memory safety and concurrency abstractions even while running on bare metal.

The GERT project still leaves a few questions unanswered. First of all, GERT was never bench marked against bare metal C running on the iMX6. This is because there was not enough time to develop an RTOS that could provide similar semantics to Go. Instead, a Teensy with a cortex M4 processor was used to establish an ideal latency baseline for an embedded system. GERT is not likely to outperform a well-written RTOS, but GERT programs will still be more portable.

It is also possible that GERT can exist as a kernel module inside of Linux, instead of a stand-alone bootable binary. Since GERT is already memory-safe, it should be

able to run in kernel mode without crashing the whole system. This is an especially desirable avenue to approach because it means that an embedded programmer can have the convenience of developing a GERT program in user space before installing it as a kernel module where it can run more efficiently. Re-spinning GERT as a kernel module may also allow it to hook into existing Linux drivers for the network card and the file system, greatly reducing the number of drivers that have to be written for each platform.

Bibliography

[1] Beagle bone main site. `http://beagleboard.org/bone`. Accessed: 2017-08-24.

[2] Copper. `https://github.com/japaric/copper`. Accessed: 2017-08-24.

[3] Go gc: Prioritizing low latency and simplicity. `https://blog.golang.org/go15gc`. Accessed: 2017-08-24.

[4] Go standard libraries. `https://golang.org/pkg/#stdlib`. Accessed: 2017-08-24.

[5] Micropython. `https://micropython.org/`. Accessed: 2017-08-24.

[6] Mirror galvanomoter image. `http://www.zamisel.com/SSpostavka2.html`. Accessed: 2017-08-24.

[7] Mit maslab. `http://maslab.mit.edu/2017/`. Accessed: 2017-08-24.

[8] Raspberry pi main site. `https://www.raspberrypi.org`. Accessed: 2017-08-24.

[9] Teensy usb development board. `https://www.pjrc.com/store/teensy32.html`. Accessed: 2017-08-24.

[10] Periklis Akritidis. Practical memory safety for c. Technical Report UCAM-CL-TR-798.

[11] ARM. *ARM Architecture Reference Manual*, ddi 0406 edition, 2012.

[12] ARM. *ARM Generic Interrupt Controller*, ihi 0048b.b edition, 2013. section 3.2.3.

[13] Frans Kaashoek Cody Cutler, Robert Morris. The performance of a kernel written in a high-level garbage-collected language. in submission.

[14] Galen Hunt and Jim Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41/2:37–49, April 2007.