Bootstrapping The Go Runtime For Use On Embedded Systems

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

Bootstrapping The Go Runtime For Use On Embedded Systems

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 SOC's (system-on-a-chip) with multiple cores, dizzying amounts of peripherals, and complicated virtual memory systems that mirror that of x86. However, performant embedded programs are still largely written from scratch in C, which leads to constant re-implementation of the same subsystems. This wastes time and increases the likelihood of bugs. Real-Time Operating Systems (RTOS's) do exist but they impose their own driver models and are also written in C, still increasing the likelyhood of bugs.

This thesis explores a system named G.E.R.T : a modified a version of the Go runtime intended to run bare-metal on 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, the Golang Embedded RunTime, provides the multiprocessor support and basic memory abstractions of a typical RTOS while also freeing the user to leverage the language features of Go in order to develop custom driver models and build more complicated concurrent programs that are easier to reason about than similar ones written in C.

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

Acknowledgments

Not so fast

Contents

List of Figures

List of Tables

Chapter 1

Why Write Low Level System Code in Go?

Modern embedded systems are composed of multicore SOCs that require careful thought about race conditions and event serialization. Like operating system kernels, most of these embedded systems are still implemented in C because it is a simplistic language that makes it good for "bare-metal" development.

outline: computers are quicker and multicore programming is scary, but everyone still uses C

C's simplicity makes it error prone for concurrent programs. Because C is very simple, the programmer must implement additional complexities in order to write concurrent programs.

try2: Low-level system code has been written in C since the 1970's because it is powerful and reliable. C can be used to express any operation a computer can do and it can also be compiled to fast byte code. Once, during an interview, an engineer even remarked: "If you can't do it in C, you can't do it". This does not mean that C is always the best choice though.

Even though multicore systems are commonplace now, kernels are still written in C.

Writing complex concurrent programs in C is too hard. Because C is very simple, the programmer must implement additional complexities in order to write concurrent

programs.

There are very few built-in abstractions so it is left to the programmer to layer additional complexities in order to accomplish a task.

try 1: Low-level system code has been written in C since the 1970's because C is powerful and reliable. C can be used to express any operation a computer can do and it doesn't come with any baggage like languages with a runtime do. C is also easy to learn because it doesn't require advanced degrees in order to comprehend, like Haskell and Coq do. The problems with C only begin to show when concurrency comes into play. C, by itself, has no idea of concurrency or concurrent programming patterns. It is really all up to the programmer to lay down these abstractions. Combined with the burden of manual memory management, concurrent programming in C almost always results in pouring over JTAG trace logs for hints of a race condition. Faced with this bleak outlook, perhaps it is reasonable to take a performance hit in exchange for faster development and less bugs. After all, computers have gotten significantly quicker in the last 20 years. This is where Go can come in. Go is meant to be a systems language that provides fundamental support for concurrency and cummincation

Chapter 2

Booting the Go Runtime

Even though Go code is compiled, it relies on a runtime to coordinate certain actions with the OS. 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 $make()$ would be disasterous if the GC has not yet been initialized. In observation of these constraints, G.E.R.T boots via a 3-stage process. The first stage is u-boot, which configures the clocks and copies the second stage off of an sdcard into memory before jumping into it. The second stage bootloader is a small C program which contains the entire G.E.R.T kernel ELF in its data section. This stage sets up the inital Go stack and loads the G.E.R.T ELF into memory before jumping to its entry point. The third stage of the bootloader lives inside G.E.R.T and is mostly written in Go, along with some Plan 9 assembly. It finishes the boot process.

Working off the initial stack from stage 2, the stage 3 bootloader enumerates all of RAM into page tables and creates an idenity mapping with a new stack before turning on the MMU. After this, a thread scheduler is setup and synchronization primitives, like $futex()$ are enabled. Additional CPU's are booted in main after the

Go runtime has finished initializing.

2.1 System Specification

G.E.R.T is written on a Freescale i.MX6 Quad SOC which implements the (32 bit) ARMv7a Cortex A9 MPCore architecture. The SOC sits on a Wandboard Quad baseboard. The i.MX6 Quad has 2GB of RAM and a wealth of memory mapped peripherals. Look at the memory map below. The rest of this chapter will discuss the implementation details of booting and running the Go runtime bare-metal on this SOC.

2.2 Stage 1 Bring Up

The u-boot loader is used to initialize device clocks and load the G.E.R.T bootloader into RAM. When the SOC is powered on, the program counter starts executing from ROM. The code in the ROM reads u-boot into RAM and jumps into it. U-boot programs the myriad of frequency dividers which are required to run the i.MX6 at a frequency of 792MHz per core. After this, u-boot is instructed to load the G.E.R.T kernel off the sdcard and into RAM at address 0x50000000. This address is specifically chosen because it does not overlap with any ELF program headers of the G.E.R.T kernel which are loaded in stage 2. After the stage 2 bootloader is in RAM, uboot jumps into it.

2.3 Stage 2 Decompression

The G.E.R.T bootloader sets up the initial Go stack and decompresses the Go kernel ELF into RAM. Much like Linux, the kernel of G.E.R.T is wrapped in a custom bootloader stage. This is necessary because G.E.R.T is compiled as a user space Linux program which expects a stack and the standard ELF auxiliary vectors. By default, the Go compiler links all programs at address 0x0. This would normally be

a disaster for the i.MX6 because the first megabyte of RAM is either inaccessable or reserved for peripherals. One solution around this is to simply turn on the MMU in the stage 2 bootloader but this creates a headache with preserving page tables across the transition to Go land. An alternative, and much simpler, solution is to just change the link address of the Go ELF. This is the preferred approach so the link address was changed to 0x10000000. After loading the Go binary into RAM, the stage 2 bootloader reserves 4kb of initial stack and jumps into G.E.R.T.

2.4 Stage 3 Go Abstraction Synthesis

The thread scheduler and virtual memory system are statically initialized in order to prevent Go runtime subsystems from running before the environment is ready. At the beginning of execution, G.E.R.T is in a really constrained spot: Linux is not there but the Go runtime thinks it is. Specifically, there is no scheduler, no virtual memory, no syscalls. Nothing but a 4kb stack and the program counter. This is clearly inadequate for the Go runtime to do anything but crash, but G.E.R.T creates all of these missing subsystems (in Go) before the runtime actually uses them.

Chapter 3

Evaluation

G.E.R.T is evaluated for usability and performance. Usability refers to the ability of an engineer to easily express their desires for system behavior in the framework that G.E.R.T provides. These constraints essentially boil down to the feature set of the Go language. Performance is based on G.E.R.T's ability on synthetic timing benchmarks as well as its measured capability on two case studies: a mobile sensor platform robot and a mirror galvanometer laser projector.

Writing enough drivers to make an SOC functional constitutes a huge pain so, for its initial implementation, G.E.R.T is written specifically for the Freescale i.MX6 Quad which implements the ARM Cortex A-9 MPCore architecture. This SOC was chosen because it has four cores, 2GB of RAM, and a very well written datasheet. It is the author's opinion that more manufacturers release complete datasheets for their SOCs so that researchers and hobbysists aren't limited to such a small set. Drivers were written for the UART, ECSPI, USDHC, PWM, GPT, IOMUX, and GPIO peripherals.

All timing benchmarks were also run on Debian Linux userspace using the sysfs drivers that the Linux kernel provides.

3.1 Usability Evaluation

Talk about new sensor integration primitives and concurrent state machines that go makes possible and nearly effortless to implement.

3.2 Synthetic Timing Evaluation

In order for G.E.R.T to be a feasible solution for performant embedded systems it must be able to quickly receive and respond to external events, such as interrupts. Two benchmarks were constructed to measure the interrupt latency. The first is a pulse counter where an FPGA is programmed to send n pulses at f frequency and G.E.R.T must count them. The second benchmark is a simple interrupt latency test where G.E.R.T simply toggles a pin upon noticing a rising edge on a different pin.

Another timing benchmark was constructed to measure the maximum frequency that G.E.R.T can toggle a pin at. Large software systems often introduce an unecessary code "tax" that occurs when performing basic operations and this must be quantified. An output pin on the iMX6 can be toggled with only 3 lines of bare-metal assembly, but the Go and C compiler emit something much more complicated for the same basic operation (quantify). For this test, a pin is toggled as fast as possible and the frequency is measured.