NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4501 - Computer Science, Specialisation Project

Golang for Bare Metal RISCV Systems

Author:
Hans Erik Frøyland

June 2022

Contents

1 Introduction

Today the use of micro controllers in embedded systems are more and more wide spread in all kinds of systems, be it in vehicles, machines in all kinds of industries, the defence industry, or even in toys. It is often the case that such embedded systems have strict requirements in terms of speed of execution and reaction time. These systems may also be limited in terms of resources like memory. Because of this, embedded systems are often not suited to run operating systems. Operating systems require significant amounts of memory and may reduce the speed of executing for specific tasks, albeit some operating systems require different amounts of memory used and effect the time of execution in different manners. When we want to make a system do very specific tasks, and also set strict requirements around what the system should be capable of, we must look towards low level programming.

The most strait forward way of doing embedded programming would be to just write pure assembly, but this is slow, inefficient and prone to a lot of bugs being created under development of software. A different and better option is write the software in a programming language such as C or C++, where a compiler would help you out by optimizing your code and help you to write functioning code (due to the restrictions that the language may pose as to how code is written). The common way of doing embedded programming is often to just write your code in C or C++. You can find C and C++ compilers that will compile C and C++ code to more or less any type of system architecture that is in any form of popular use today. However, these languages are getting old and they have their shortcomings, like not having any garbage collection and making it easy for users to both use pointers and mess it up in the process. Because of this I would like to look at other options for languages to do embedded programming in. What I have decided to look at is using Golang[1], a Google made programming language, for the RISCV[2] architecture. In this project I want to compare Go and C in terms memory usage and speed of execution and evaluate if Go is a good language for bare metal software development.

2 Background

Golang, also known as just Go, is a quite new and nifty programming language which provides a garbage collector and also easy to use concurrent programming features. RISCV is a relatively new architecture and has become quite popular in recent years due to being easy to work with and an open-source project. There all ready good compilers for compiling Go code for ARM systems, but work on Go for RISCV systems is more recent. There exist an embedded Go project[3] which has made two solutions for getting Go code to work on RISCV systems. One of these solutions is to just have the compiler compile the Go code into C code and the just let GCC compile the C code to a RISCV executable. The other solution is making a compiler and associated tools for producing and running true Golang code.

When developing software you always want to make the process as convenient and efficient as possible, while at the same time have the tools necessary to make the software preform the way you want it to. With this in mind you have to make a tradeoff when deciding how you want to develop the software, because different languages provide different benefits and drawbacks. Also, when it comes to bare metal programming, there might also be limitations in what languages there are support for. Some different languages that can be used have already been mentioned, like writing pure Assembly or writing C. For RISCV processors, you often have to choose writing your code in C, if you want to develop bare metal software. But in recent years there have been development for making Golang code function on bare metal RISCV systems.

Getting Golang code to work on a bare metal system is not straight forward, as Golang is very much built for running on an operating system. Golang is very dependant often very dependant on features that operating systems provides, such as virtual memory and a scheduler. Golang both has a very sizable run time and garbage collector, which does not make it a very lean language like C. However, what this hefty run time provides is a very easy and convenient way of doing concurrent programming.

3 The Golang runtime, how it works in short

The specifics of the run time works can vary significantly depending on the system it running on. Golang usually expects to be run on a operating system, and it often rely on the memory management and the scheduler that an OS provides. If Golang is to be ran bare metal, the run time it self has to provide these capabilities, effectively turning the run time into a small OS. I will describe how the run time works for bare metal RISC-V. This is not any comprehensive explanation of the run time, but rather a good overview of how the run time work.

3.1 The p, m and g abstractions

Golang has an internal structure and abstractions that enables it to run gorutines. There are three key abstract objects which make this possible, namely p, m, and g. "p" represents an actual physical processor and will contain the actual rescuers required to execute some task. The amount of p's is determined by the system (obviously) and is equal to GOMAXPROCS. "m" is a machine thread, i.e a normal thread in a language like C. A m need to be assigned with a p in order to have the ability to execute code, as p holds the recourse's required to execute code. The run time may create new m's if it need to. The scheduling of what m to run when is left up to the OS scheduler if the OS exist, or a substitute scheduler provided by the run time if there is no OS. Then you have the "g" abstraction. This represents a grouting, which could be thought of as a thread inside a thread, or a g in an m. When invoking the "go" symbol in Go code, you create a new gorutine which run and is managed by a machine thread.

This machine thread can contain several gorutines which run concurrently. How these g's are manage both when it comes to what is scheduled to run when and how their stacks are put together is managed by the m (machine thread). The machine thread's stack consists of its own m stack and g stacks, which is used for gorutines. The size of m stack is fixed and not garbage collected. M stack is used when some task should not be allowed to grow the stack, or when context switching from one gorutine to another. The size of g stacks is not fixed and the placement and size may be changed by the run time. The g stack will start out small, and grow as more memory is needed. This process is more straight forward when running on an OS, as each m will have their own virtual memory and most of the heavy lifting of memory management is done by the memory management unit. When there is no OS this becomes more complicated.

The patch used for this project does not support more than one m for that the user may use. In reality there are two m's running, but one is running a sudo OS which provides the tools necessary for Golang to function. This master m, which is called m0, is not available for the user to use and runs in the background. However gorutines are supported, which means that running code concurrently is still supported, even if only one machine level thread is supported.

3.2 The garbage collector

Golangs garbage collector is a non-generational concurrent mark and sweep tri-colour garbage collector[4][5]. I will go through what all of these tearms means.

A generational garbage collector focuses on short lived variables and focuses on checking weather recent allocations needs to be garbage collected. Generational garbage collection is not needed for Go code, as the Go compiler performs escape analysis of global variables. The Go compiler will store global variables in the stack frame of a function, if that global variable is only used for that function. This means that it looks at the global variables and if they are effectively used as just local variables, and they are treated as such if this is the case. This means that the benefits of a generational garbage collector is negated over the fact that what it would collect is just put on the stack. Thus these short lived variables does not need to be garbage collected.

A concurrent garbage collector simply means that the garbage collection runs currently with executing code, where the running code and the GC is scheduled by the run time. The run time uses a method which is called stop the world to make this work. What stop the world does is to wait for all running gorutines to get to a garbage collection safe spot, and then they are halted. The run time then run the garbage collector which free unreferenced allocations.

Mark and sweep tri-colour garbage collection means that memory is iterated over by the GC (3 times for this type of GC). From the start all data is marked as white. Then the the GC iterates over all data which there exists a reference to, marking them as grey. After this the iteration happens once more to confirm that the grey data are indeed referenced, marking them black. After this, all data that is not black is freed.

Go provides some options for garbage collection, specifically for when to garbage collect. This can be done by changing the environmental GOGC. It is set to 100 by default, which means that when we use 100% more heap space than what we used when the previous GC happened, the GC should run again.

3.3 The start up pattern for the run time

The run time starts by setting up the mode status flags to be put in machine mode. After this, the address of where the trap handler starts are written to MTVEC, this makes sure that the location of the trap handler is known. The bootstrap thread with a temporary gorutine is initialized, and stack boundaries and segments are calculated and declared. The scheduler is then initialized and runs on the bootstrap thread. The tasker is initialized, it is here PLIC and CLIC is created and set up. A new gorutine is created, this will be used as the main gorutine running on the main thread. The system is changed to run in user mode. Newporc() is called and the main thread is created. Main starts running on m.g, the main gorutine of the main thread, and normal execution of user created code starts. This run time execution pattern is illustrated in figure 1 bellow.

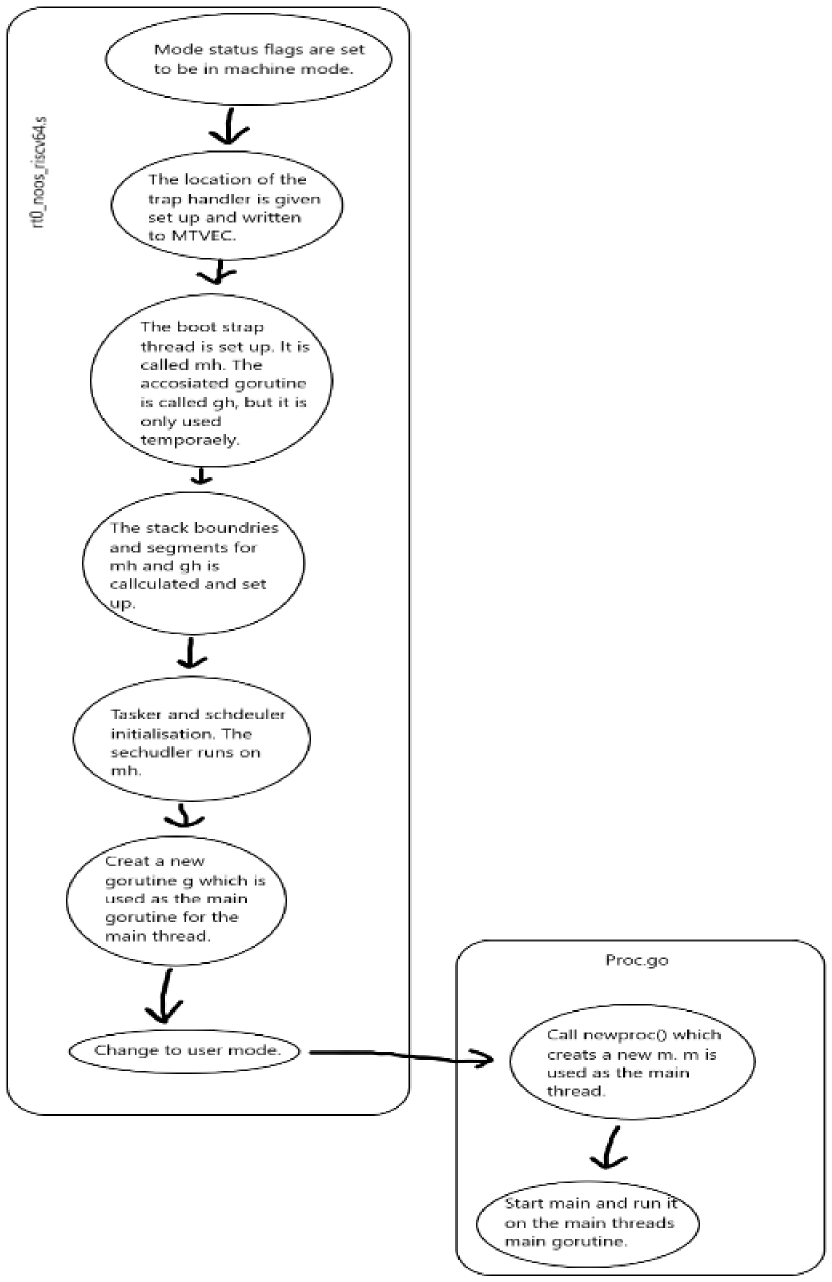
rt0_noos_riscv64.s
Mode status flags are set to be in machine mode.
The location of the trap handler is given set up and written to MTVEC.
The boot strap thread is set up. It is called mh. The accosiated gorutine is called gh, but it is only used temporaely.
The stack boundries and segments for mh and gh is callculated and set up.
Tasker and schdeuler initialisation. The sechudler runs on mh.
Creat a new gorutine g which is used as the main gorutine for the main thread.
Change to user mode.
Proc.go
Call newproc() which creats a new m. m is used as the main thread.
Start main and run it on the main threads main gorutine.

Figure 1: This is the execution flow of Go run time.

4 Implementation of the compiler on the Qemu-virtio system and how to use it

I'm using a patch made by git user "the git name of the guy who made the patch[6] which apply a patch to the 1.16 version of the Golang compiler. Here i will go through how to use each of the different software used in this project.

4.1 How to use the Go Patch

The instruction for compiling the go code correctly is:

GOOS=noos GOARCH=riscv64 goroot/bin/go build -a -tags "k210 noos" -gcflags "-N -l" -ldflags "-M 0x80000000:128M" test.go

This this instruction will compile the test.go file into an executable for a bare metal RISCV64 system, where the stack starts at 0x8000000 and is of size 128 Megabyte.

The run time needs to know what sort of OS and architecture the code will be running on, therefor this has to be provided as input to the compiler. The compiler read the GOOS and GOARCH environmental variables to know what OS and architecture the system should run on. However, there are small differences between specific implementations of the RISCV architecture, so more information has to be provided. There exist different configurations for different sorts of specific RISCV systems. We compile the code for a Kendryte machine, which is similar enough to the Qemu-Virtio system that I'm running it on that it will work. Go can use build tags in different files, where the file will only compile if you tell the compiler to explicitly do so. When i invoke -tags "k210 noos" the compiler compile for a Kendryte system with no OS. -gcflags "-N -l" turns of garbage collection, which is not supported, and turn off optimization. -ldflags "-M 0x80000000:128M" tells the the linker that the stack starts at 0x80000000 and is of a size 128 megabytes. -a just force a rebuild of all of the packages that is to be compiled.

4.2 How to use Qemu

As mentioned before, the Go code is compiled for a Qemu-Vertio["link to this here"] system. You can run the Vertio machine with this command:

qemu-7.0.0-rc3/build/qemu-system-riscv64 -serial stdio -machine virt -m 128M -bios none -kernel test -s -S

It is important that a qemu-system-riscv64 is called, as we want to use a RISCV64 system. The flags for the command do the following:

-serial stdio: This sends output from the VM to stdio on the terminal you called the qemu-system-riscv64 command.

-machine virt: This tells Qemu that we want to simulate a Virtio machine. Virtio is a general purpose virtual machine that can be used for many different architectures.

-m 128M: The system memory should be of a size of 128 megabyte.

-bios none: Disable the default BIOS that is provided. We want a completely bare metal system, so there should be no BIOS at all.

-kernel test: Run the test executable as the system kernel.

-s and -S: these flags are both for debugging purposes. -s opens a GDB server at localhost:1234. -S prevents the CPU from immediately start executing code, allowing us to connect with a GDB debugger.

4.3 How to use GDB

I have used GDB to debug the run time and to see that the works as it should. Golang code does not have full support for being debugged by GDB. Usually Delve["link to delve"] is used for debugging Go code, but there is not support for delve to debug Go code that run on a RISCV system, and it does not work on any bare metal system either. This leaves me with GDB as the only option.

GDB works alright when debugging the run time, though there are some issues when calling functions from other files. GDB does not work well for debugging actual user written code however. It does not understand what a gortuine is, and it has trouble dealing with the run time. However it works sometimes, which makes it better than nothing. GDB can also accept plugins to make it behave a bit differently. The Go compiler do provide you with a Python script to make GDB work a little better for Go code. Use the -d flag and provide the plugin script as input.

The command for running GDB with the plugin is:

riscv-tools/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin/riscv64-unknown-elf-gdb test -d ./goroot/src/runtime/runtime-gdb.py

4.4 Modifications for the Qemu-Virtio sytem

4.4.1 PLIC spec problems

When getting the patch to work for the Virtio machine, there had to be some changes to the runtime in order to get this to work. One of which was to the platform level interrupt controller(PLIC). The PLIC is defined as a object in the run time. It contains a bunch of different list that has different kind of control flags. One of these types of list contains Enable flags for 1024 different sources for 15872 different contexts as specified by the RISCV spec[7]. However there where a issue in the spec description which also existed in the patch implementation. There was a mistake where each address was thought to hold 32 bit(4byte) instead of just 8 bit(1byte). Because of this, the other lists in the PILC object where allocated in the wrong places and there where memory issues because of this. I just reduced the number of possible contexts to fix this issue, and it worked fine. However the PLIC should maybe get rewritten with the correct memory address, as the spec is now updated to assume that each memory address only hold 8 bit of data, as is the case. There might be some issues related to the reduced number of context if ever a lot of peripherals are simulated for the system.

4.4.2 Mstatus problems

There where issues with how the task handler returned to execute normal code. The the return address put on MEPC where wrong, where the last bit where set to 1 a lot of the time. An instruction never start on an address where the least significant bit is set to 1. Also the state of control bits which determines the privilege level of the system where set incorrectly. After these issues where fixed, my code executed correctly.

5 Evaluation

I'm comparing the performance between C and Golang. I will look at both the actual speed of execution and also the memory requirements of both C and Golang in regards to the size of the run time. It is important to note that the Golang run time used in this evaluation lacks several features which affects the performance of Go. The results of Go's performance in this evaluation is not the same as what you would get if the run time where fully implemented.

5.1 Execution time

There is not rely good bench marking tools to use on C and Go on bare metal RISCV. I ended up making a test program, which does matrix multiplication and I tried to write the test and the way the multiplication is preform to be as similar as possible. To test the actual execution time of both test i recorded my screen with SimpleScreenRecorder[8] and viewed the recording in djv [9]. Golang's garbage collector is not implemented for bare metal Go, so there will be issues about the amount of matrix's that can be created. There is not any way to manually clear memory in Go since it is supposed to be garbage collected. When running the test 100 thousand times, these memory restrictions arose. I found that running it 50 thousand times worked fine, so that was the number i landed on. I did not do any freeing of memory on the C test either, as this does not happen on the Go side. These where the results I got:

performance GO: When running Go_matrix_multiply_test 50k times I get a time of 97 frames - 82 frames = 15 frames @30fps.

performance C: When running C_matrix_multiply_test 50k times I get a time of 163 frames - 125 frames = 38 frames @30fps.

As can be seen above, the Go code is actually faster than the C code, however this could be because the Go run time is not doing much here. There are no garbage collection, and I'm not using any gorutines. There are not rely any activity from the run time because of this. In a more complete implementation of Go, it would be slower due to more run time activity. However these results show that the current version of the Go implementation is not out performed by C. The only problem is that it is not a complete implementation, with critical features missing that would slow down the performance, namely garbage collection.

5.2 Run time memory usage

To figure out the size of the runt time I just looked at where the main.main file started using the GDB i line feature. Both the C and Go code where compiled without using any libraries at all. This is what i found.

Memory usage for go code: run time starts at 0x80000000 and stop at 0x80050e50. The Amount of memory used by the run time is then 0x50e50 (when there are no imports in the code). This is 331344 addresses, each holding one byte each this would be the same as 323.6 * 2$\hat{1}$0, or 324.6KB of memory dedicated to the run time.

Memory usage for c code: "run time" starts at 0x80000000 and ends at 0x80000018. The Amount of memory used by the c "run time" is 0x18 which is just 24Bytes and is just a few lines of initialization. C's run time does not do much, other than calling main.main.

As seen above, the Go run time is somewhat large, and a more complete implementation of Go would mean that it would get even larger. Because of this, Go is not a good option if the system is very limited on the amount of available memory. C does not have a proper run time and it is very small, so there is not this memory constraint with C.

6 Conclusion and ideas for further work

Golang provides a easy to use language, which both provides features like gorutines to make it easy to write concurrent programs and a garbage collector is also provided. At the same time it is a compiled language which provides a fast running executable, much like C. Golang has the benefits of being fast, easy to use and more secure than C. However it has to make a sacrifice of demanding a lot of memory for to achieve this. This makes Golang not suited for systems with strict memory requirements as the Golang run time is measured in hundreds of kilobytes. On the other hand, for systems that has a few megabytes to available, Go can be a very good option for software development on such systems. Even better, if such a system is expected to run lots of code concurrently, then go excel at this area in particular. Golang provides a level of security and ease of use which is not present in C, and at the same time reach a similar speed of execution.

The compiler patch which is currently provided lacks several important features which needs to be implemented in order to compile completely functional Go code. The two most important features it lacks is support for running multiple machine threads, which means that it can't run code in parallel. It does however support gorutines, but gorutines are only "Go run time threads". This means that the current implementation can run code concurrently, but not concurrently in parallel. The other big feature that is missing is the garbage collector. The garbage collector absolutely needs to be implemented in order to have a Go compiler that works. Without a garbage collector, there is no way of freeing memory stored on the heap, and this will obviously be a big problem if there are lots of things that is stored on the heap and you run out of memory that way. First and foremost these two features should be implemented if there is to be further development on this Go compiler. These features are absolutely necessary to get properly functioning Go code. In addition to this there should be developed a debugger for debugging Go code on bare metal RISCV systems, as a properly functioning debugger does not already exist.

References

[1] Google, "The go home page." https://go.dev/. Sourced 9.6.2022.

[2] RISCV, "Riscv home page." https://riscv.org/. Sourced 9.6.2022.

[3] M. Derkacz, "Bare metal risc-v programming in go." https://embeddedgo.github.io/2020/05/31/bare_metal_programming_risc-v_in_go.html. Sourced 9.6.2022.

[4] S. Gangemi, "An overview of memory management in go." https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8. Sourced 9.6.2022.

[5] R. Hudson, "Getting to go: The journey of go's garbage collector." https://go.dev/blog/ismmkeynote. Sourced 9.6.2022.

[6] michalderkacz, "embeddedgo/patch." https://github.com/embeddedgo/patch. Sourced 9.6.2022.

[7] r. t. P. p.-d. d. b. changab, SKTT1Ryze, "riscv-plic-spec." https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc. Sourced 9.6.2022.

[8] "Simple screen recorder." https://linuxhint.com/install_simple_screen_recorder_ubuntu/. Sourced 9.6.2022.

[9] "Djv." https://darbyjohnston.github.io/DJV/. Sourced 9.6.2022.

Link to the related git repository:
https://github.com/BondeKing/TDT4501_BM_RISCV_GO/tree/main