NTNU
Kunnskap for en bedre verden

Department of Computer Science

MultiGo - Implementing Parallel Execution for Bare-Metal Golang

Author:
Hans Erik Frøyland

May 2023

Abstract

This thesis looks at improving software development for bare-metal RISCV systems with MultiGo. MultiGo is a modification of Embeddedgo. MultiGo adds support for parallel execution of threads, something Embeddedgo lacks. Embeddedgo is, in turn, a modification of regular Go that adds support for running Go on bare-metal systems. Go is a relatively new and interesting programming language. Go offers many features, such as garbage collection and a novel way of handling threads, namely with goroutines. The thesis will explain how Embeddedgo works, what changes were made to create MultiGo, and compare MultiGo with C to figure out the pros and cons of both languages.

Contents

1 Introduction

Most software development today is about creating applications that are intended to run on top of an Operating System. Operating Systems, or OS for short, is a collection of software that control all the underlying physical structures of a computer and the processes that are being executed on the machine. The reasons why all personal computers today and for the last 30–40 years have come equipped with an OS is due to the fact that it makes both the usage of the computer so much more convenient and easy and also simplifies the process of software development for such systems. An OS is able to abstract away all kinds of tasks that would be tedious to deal with, such as memory management and dealing with all kinds of hardware interfaces. Operating Systems also offer more security for applications, such as by providing virtual memory, and make it so that applications do not need to worry about what architecture they're running on; they only need to interact with the OS.

Operating Systems are a great thing, so why are we not using operating systems for everything all the time on all computers? An OS is not a small piece of software. They're often several gigabytes in size and have quite a large number of processes they're running themselves to make the OS work properly. For some systems, primarily microcontrollers, there are very strict resource restrictions. These can come in the form of having a very small memory or requiring an enormously fast response time when dealing with an event. In these situations, we simply cannot have an OS installed for such systems, as it would eat up too much of the systems' resources. If we're going to create software for such systems, we cannot rely on an OS and have to develop for what is called a *bare-metal* system, namely systems without any OS.

The question is then: How do we develop software for a bare-metal system? You find a programming language that has a compiler that's able to compile your code into an executable for the targeted system. One problem here is that most programming languages are not very well suited for bare-metal software development. Languages that are suited for this are often either cumbersome to use, such as Rust[1], or they're very old, such as C.

Golang[2], or just Go for short, is a language created by Google back in the early 2010s. It offers a good mix of being easy to use while also not being so resource-demanding that it can't be used for bare-metal systems. Go is a garbage-collecting language, which is convenient and makes memory management simpler. Go presents a novel solution for dealing with concurrent programming by introducing something called goroutines. Goroutines can be thought of as very light weight threads with a dynamically sized stack. This means that Go is able to have a lot of very small threads running at the same time, even on memory-constrained systems. Go does not have support for compiling code to work on bare-metal system. Embeddedgo[3][4] is a patch for the Go runtime that adds support for execution on bare-metal systems. This means that Go is now an option for embedded software development.

One of the systems Embeddedgo has support for is RISCV[5]. RISCV is a very popular architecture for embedded software development. Because of this, it made a lot of sense to develop for this type of architecture in this project.

I'm using a version of Embeddedgo from early 2022. Embeddedgo is based on the 1.16 version of Go.

1.1 The Goal of the Thesis

The vanilla Go runtime is not able to run on bare-metal systems as it depends on system-specific code for making necessary system calls. The runtime requires specific support for both the architecture and OS to work properly. Modifying the Go runtime to work with bare-metal systems is not an easy task, as it normally relies on the OS process scheduling system. Bare-metal support for Go requires the implementation of these features in the runtime itself. The Embeddedgo project implements support for a number of different architectures running without any OS. There are, however, a lot of missing features for bare-metal systems in Embeddedgo. My primary goal for this thesis is to implement support for parallel execution for multi-core systems, as this is an important but lacking feature in Embeddedgo. A secondary goal of the thesis is to explain how Embeddedgo works and provide some documentation for it. I will also compare Go and C to figure out what is better for Embedded software development.

1.2 Contributions

I develop a new Go runtime system for bare-metal systems, namely **MultiGo**, which is based on Embeddedgo, but has the following new contributions:

- Has support for parallel execution of threads in a multi core system.
- Has increased support for features in the standard library.
- Has a light weight printing feature, suitable for embedded environments with fairly strict memory restrictions.

1.3 Structure

The structure of the thesis is as follows:

- Chapter 2 presents features in the Go runtime that are relevant for this project. The goal of this chapter is for the reader to understand how the scheduling process works.
- Chapter 3 presents the changes made to the runtime to enable support for parallel execution.
- Chapter 4 demonstrates the tools used in the project and the methodology for how I conducted the experiments in chapter 5.
- Chapter 5 evaluates the performance of MultiGo and also compares C and Go for bare-metal software development to figure out what the up and downsides with both languages are.

- Chapter 6 concludes this thesis.

2 The Go Runtime

Golang requires a very large runtime in order to provide all of the features the language offers. It is quite complicated for bare-metal systems, where the runtime needs to do the work that it would be processed by an OS. On top of this the runtime has to do all of its other tasks too, such as garbage collection and scheduling goroutines. In this chapter I will explain how the runtime starts up new threads. This is important to understand what needs to be changed for parallel execution of goroutines to be supported.

2.1 A Birds Eye View Of Go

Embeddedgo has many moving parts, and it spans many tens of thousands lines of code over many files. To help make sense of the structure of the Go runtime, I'll go through the parts most important for this project. These are the boot file rt0, the scheduler, the tasker and the traphandler. The relationship between these are show in figure 1.

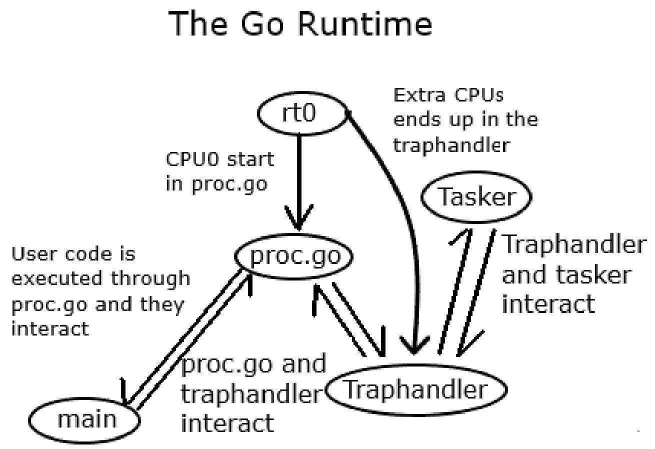

The Go Runtime
rt0
Extra CPUs ends up in the traphandler
CPU0 start in proc.go
Tasker
User code is executed through proc.go and they interact
proc.go
Traphandler and tasker interact
proc.go and traphandler interact
Traphandler
main

Figure 1: An overview of the most important parts of the runtime.

2.1.1 What Is rt0?

The entry point for execution in programming languages that use runtimes are called rt0, or runtime0. These files contain the boot sequence of the runtime and they're system specific. Being system specific means that these rt0 files care about the architecture and OS of the system they're executed in. In this project **rt0_noos_riscv64.s** is used since I'm developing on a bare-metal RISCV system. The important thing to know about rt0 is that it sets up the system in the state it's expected to be in for normal execution. CPU0 is used for running main, and additional CPUs are put to sleep in the traphandler. I go into further detail on the boot process in chapter 2.5.

2.1.2 What Is proc.go?

The scheduler is an important concept in the Go runtime. It lives inside **proc.go**. The scheduler is responsible for creating, scheduling and executing threads. This include the main thread as well. Much of the code relating to garbage collection is also in proc.go. This code is not system specific and is used in vanilla Go as well as Embeddedgo. I go into further detail about the contents of proc.go and how the scheduling system works in chapter 2.6.

2.1.3 What Is The Tasker?

The tasker is responsible for dealing with hardware specific tasks in the scheduling process. Tasks that proc.go would normally hand over to an OS are instead handed over to the tasker in Embeddedgo. The tasker is the most important addition made by Embeddego to enable bare-metal execution. Obviously the tasker is system specific, and I use the **tasker_noos_riscv64.go** tasker in this project. I write about the contents of the tasker in relation to the scheduling process in chapter 2.6.

2.1.4 What Is The Traphandler?

The traphandler is responsible for handling interrupts and environment calls. What is important to understand about the traphandler is that it is used for changing the *context* code is executing in. The traphandler is able to do this because it is written in Goassembly and it increases the machine privilege level when entered. What all of this means is explained in later chapters 2.3 and 2.4. The traphandler is system specific.

2.2 Important Go Structs: G, M and P

After having quickly gone through what the most important files are, there are several objects and structures used by the runtime that need to be introduced. It is critical to understand what these structures are and does for you to know what is going on in the scheduler. The three most important structures are *Goroutines* (G), *Machine Threads* (M) and *Processors* (P). In addition to this, bare-metal systems also use a structure *cpuctx*, or *CPU context*, to hold some additional information needed for scheduling in bare-metal environments to work. Examples of these stucts are provided in the appendix.

2.2.1 Goroutines (G)

Goroutines are used to hold information about a function which can be executed in a thread. It is also provided with a stack that the function should use when executing. The gorouinte stack is quite novel in that it starts out very small and is able to grow in case more stack space is needed. The key idea to understand about goroutines is that they contain all information about a specific function needed for execution of said function. When starting a new thread, the special `go` call is used and it requires a function as input. In proc.go, the function pointer, input variables and other relevant information is put inside a goroutine object. This object is then put in a queue, where it can later be picked up and executed. Threads able to execute a G are represented by the M struct.

2.2.2 Machine Threads (M)

A machine thread is a structure representing a regular thread. M is responsible for fetching goroutines and then executing them. A single M is able to execute several goroutines concurrently. Because of this each core only ever needs a single M, since no more is required for concurrent execution of code. The contents of stacks held by a G is subject to being garbage collected. It could be the case that sometimes certain code should not be garbage collect. Because of this M also has its own stack called the machine stack. This stack is not garbage collected, and certain sensitive code has to execute with a non-garbage collected stack. To use this stack area, each M holds a private goroutine denoted as G0. A private goroutine is required, as M has no way of holding function information by itself without a G. M use resources P for executing a function stored on G.

2.2.3 Processors (P)

P can be thought of as representing the resources that are used to execute code with, which would be the processor and it's resources. If an M does not have an associated P, it can not execute goroutines. Each P hold a local queue of ready-to-execute goroutines called `runq`. When an M wants to find a G to execute, it starts by looking in the `runq` of it's own P. P acts as a sort of lock for the right of use to each processor. If an M does not have the P representing a processor, it can't execute on that given processor. In this way P can be used as a tool to stop all executing of goroutines by taking P from all M that may have one. This is how the garbage collector stops M from executing G when memory is being garbage collected. It is called stopTheWorld when a garbage collector does this. When handing back P after garbage collection, it is referred to as startTheWorld, and normal execution of G can resume.

2.2.4 CPU Contexts (cpuctx)

CPU contexts are objects representing each physical core in the system. If no M has yet started on a given core, that core is considered to be running in a CPU context. The most important thing to note about CPU contexts is that they're an Embeddedgo structure, they're only used in bare-metal code and that they exist to make it possible for a CPU to do things when it does not possess an M yet. CPU context's responsibilities are to put the CPU to sleep, wake other CPUs up and acquire an M if awoken. Each CPU context contains a list of M objects it can use called `runnable`.

There is exactly one CPU context for a each core in use, and other CPUs can't be in the context of another core's cpuctx object. This is prohibited.

Although other cores are allowed to interact and change things about other cores cpuctx, such as adding machine threads to other run queues.

2.2.5 The Tasker and Scheduler Objects

Both the tasker and proc.go each has a global object representing them. These are `thetasker` and `scheduler` respectively. Each object holds important information that needs to be globally available in all contexts of the system. For us the most important thing to know about these objects is that `thetasker` has a list over all CPU context objects and the `scheduler` contains lists of all M and P objects, as well as having a global list of executable G objects.

2.3 Goassembly

Goassembly[6] is a proprietary assembly language specifically made to be used with the Go runtime. Goassembly code is required in the Golang runtime in order to perform low level operations on special register, such as the *g register* and CSR registers. There is no way for normal Go code to write to these registers, therefore calls to Goassembly code is required. The g register is what determines the context in which the runtime is currently executing go code. I go into further detail about execution context in chapter 2.4.1. Certain parts of the Go runtime is written in Goassembly because they need to perform tasks not possible in normal Go code, such as rt0 and the traphandler.

2.4 Runtime Context and Privilege Levels

The Go runtime contains a lot of context-sensitive code. I will explain what the different contexts in the runtime are and where they're used. In addition to this, RISCV can operate at different hardware privilege levels. The privilege level determines what sorts of instructions the system is allowed to perform. Knowing how the different Go contexts work is important for understanding the Go runtime. Knowing how the RISCV privilege levels work is not essential to understanding what is going on in this project, but I will explain it regardless because it will provide a more thorough understanding of what is going on in the traphandler.

2.4.1 Go Runtime Context

The context of the runtime is determined by whatever the g register is pointing to. The function `getg()` is used to fetch the pointer in the g register. The usage of this command is what causes the system to be context-sensitive. The g register holds a pointer to a goroutine object, and the system is defined to be running inside of the goroutine that g points to. There are three types of goroutines: regular goroutines, `m.g0` and `cpuctx.g0`. User code is always running in the context of a regular goroutine. As explained in chapters 2.2.2 and 2.2.4, M and cpuctx objects have their own private G. This G is used to execute code while in the scheduling and wake-up process. While an M is looking for another G to run, it has to temporarily be in a `m.g0` context. When a CPU is doing stuff in the tasker and finds itself outside of proc.go, it has to be in a CPU context. CPU context structs contain a G as their field, meaning that a pointer to `cpuctx.g0` is also the pointer for the cpuctx object itself. What the g register is pointing to can only be changed in Goassembly, and usually happens in the traphandler.

2.4.2 The Hardware Privilege Levels

For RISCV there exist different privilege levels, namely Machine mode(M), Supervisor mode(S) and User mode(U). In a system with an OS, the OS is running in the S mode, the kernel is running in M mode, and user applications run in U mode. On this bare-metal runtime only M mode and U mode are used, as the runtime is run as a kernel and not so much as an OS. When entering the runtime on boot, the privilege level is set to M automatically. Although at this point the control and system registers(CSR) are not set up yet. The first thing happening in the runtime is to set up the CSRs correctly, and after this, it is the state of the CSRs that determine what privilege level the runtime is in. There exist special instructions, which will switch the state in the CSRs in order to change the privilege level the runtime is in. Usually `MRET` and `ECALL` are the most common instruction's that are used to change the privilege level. `MRET` will change the privilege from M mode back to whatever mode called into M mode, in our case this would always be from U mode, and then It jumps to whatever address is specified in mepc, which is a CSR. `ECALL` is an environmental call instruction, which will change the mode into M and call into the traphandler. Where the traphandler is located is determined by what the address stored on mtvec is, which is also a CSR.

2.5 The Go Boot Process

To understand how the runtime ends up in the state it is in under normal execution, we have to go through the boot process. All of this happens in **rt0_noos_riscv64.go**. Figure 2 shows the boot process in a simple manner.

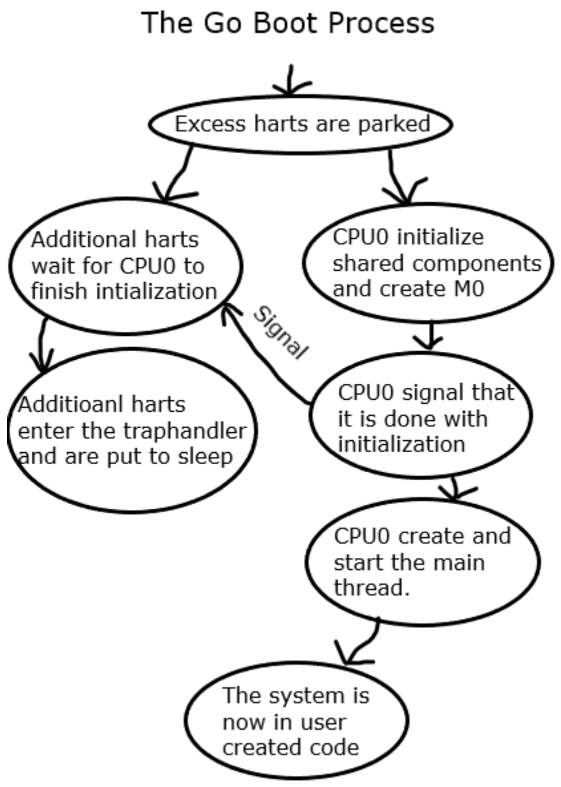

The Go Boot Process
Excess harts are parked
Additional harts wait for CPU0 to finish intialization
CPU0 initialize shared components and create M0
Signal
Additioanl harts enter the traphandler and are put to sleep
CPU0 signal that it is done with initialization
CPU0 create and start the main thread.
The system is now in user created code

Figure 2: The Go boot process.

2.5.1 Booting CPU0

The first thing that happens is that all excess cores that have an ID higher than `maxHarts` are parked. These cores are irretrievable and cannot be used by any following code. `maxHarts` is a constant defined in the tasker that sets the upper limit of cores the system is able to make available.

Afterwards, all CPUs that are not CPU0 will wait for CPU0 to initialize the system. CPU0 will clear memory and start to put itself together. It will create its cpuctx object and link it to the g register. Doing this means that CPU0 is now running inside a defined context within the runtime.

CPU0 will then have to set up `M0` manually. M0 is a special type of machine thread that runs the background processes in the runtime, such as the garbage collector and `sysmon()`. M0 is what tells other threads that it is time to yield their goroutine and find another one to work on, or to stop execution to allow garbage collection. M0 is the only thread that is allowed to execute code without needing a P. In practice, it will always be running on CPU0, since this is the CPU that does the initialization of shared components.

After M0 is created and hooked up to the cpuctx, CPU0 will be able to initialize the tasker and scheduler. M0 will have to borrow the cpuctx's G because the `taskerinit()` will assume that the system is running in a CPU context, and `schedinit()` assumes the system is running in a machine thread context. Setting M0's G0 to be the cpuctx's G0 works due to the fact that the scheduler and tasker figures out the association between G0 and their respective owners in different ways. The tasker figures out the association through casting G0 to a cpuctx object, and the scheduler finds the association between a G and an M by looking at `g.m`. In this way, G0 can be associated with M0 and the cpuctx at the same time. A proper G0 for M0 is then created, allowing the system to run in a machine thread context, without being in a cpuctx. Doing this is important because it frees up the cpuctx for use when creating the main thread. Remember that two different threads can not exist in the same context at the same time, this is disallowed.

After initialization, CPU0 will signal other harts that they may continue their own boot process. After doing this, CPU0 will start to build the main thread. This is done by creating a goroutine containing the main function, followed by a call to `startm()`. This will create a new normal M that picks up the main goroutine and executes it. Execution of user-created code then starts, and the boot process is complete.

2.5.2 Booting Other Harts

All harts/CPUs with an id higher or equal to `maxHarts` are permanently parked and not made available. Harts with an id lower than `maxHarts`, but higher than 0 have to wait for CPU0 to finish initialization before booting.

After initialization, other CPUs get a signal to continue their own boot sequence. Additional CPUs does not create a thread yet, as only the main thread will be running at boot time. These extra CPUs will be sent to the tarphandler, where there will be an attempt to find an M for them in the tasker. Since there is no work for them at boot time, they will all be put to sleep and be in an idle state.

2.6 The Go Scheduling System

When starting a new goroutine in Go, it can end up on any available CPU on the system. In the event that there are no idle CPUs in the system, all that needs to be done to create the goroutine object and hand it to one of the queues it should be in. The scheduler will be able to pick it up whenever an M becomes available to pick up new work. In a situation where a CPU is sleeping, it has to be awoken before it can pick up and execute a goroutine. I will first present the situation where a new CPU has to be awakened before it can start executing a gorutine. When the system arrives at the `scheduling()` function, the course of events is the same in both scenarios and will therefore be presented afterwards. Figure 3 shows the execution flow of this process.

2.6.1 How Machine Threads (M) Are Started

Starting new threads in Go code is done by using the `go` call. The `go` call has to be followed by a function that is to be run by the new goroutine. The compiler will translate the `go` key word to be a call to the `newproc()` function found in the proc.go file. In `newproc()`, a new goroutine is created containing a pointer to the function provided to the `newproc()` function call. The newly created goroutine will then be put in the `runq` of the current P, which is the P with id 0 if we call `go` from CPU0. All of this happens in the context of the G, which is called `go` to begin with.

Inside proc.go

After a new G has been created and put in the current `P.runq`, wakep() is called. This function will try to wake an idling P, if there exist one. It will return without doing anything if either there is no idle P or if there is a spinning M that can pick up and execute the newly created G. If there is no sleeping P, there is no need to do anything more since CPU's are able to schedule themselves as long as they're awake, and it will return back to user code in this case.

If a new CPU has to be awoken, `startm()` is called. This function will find an idle P, or use one assigned to it in the function call, and create a new M to be used with that P. `startM()` will call `newm()`. This will allocate memory, create a new M object, and connect it to the input P.

Afterwards, `newm()` will call `newosproc` which is a system call function located in the **asm_riscv64.go** file. The newosproc call is going to do an `ECALL` into the traphandler. After entering the traphandler we're going to find ourselves in a trapped context. In a trapped context, the g register is going to point to G0 of the current cpuctx object. This context switch will make calls to tasker functions valid, as they require the system to be running in a CPU context to be valid.

Going to the tasker through the traphandler

The traphandler will handle the newosproc `ECALL` and send the program to the `sysnewosproc()` function in the tasker. All of these calls have forwarded the new M through arguments, and `sysnewosproc()` will also have been provided with the new M as an argument. Since the proc.go file is not architectural or OS specific, there are still some things that have to be done for M to function in a bare-metal setting. This is done through the `newarchm()` function, which will add a few more fields to add information about relevant return and stack pointers. These are used to put the CPU back to sleep.

After M has been turned into something capable of running in a bare-metal setting, `taskerSetrunnable()` will be called. This is going to put M into the run queue of the cpuctx associated with the P that M is pointing to. Meaning that the CPU that is about to be woken up is going to get linked with the newly created machine thread object. If the CPU context assigned with M is different than the current CPU context we're in, a signal is sent to that CPU context. If the CPU is sleeping, this signal will wake it up. This is done by calling `newwork()`. After the singal has been sent, The current CPU will return back to user code. Now we will see what happens in the sleeping CPU.

The old CPU returns back to the user code, and we enter a new CPU

Each CPU starts off in a sleeping state, where it is sleeping on a wait for interrupt(`WFI`) instruction. The call stack for the sleeping CPU at this time is as follows: the current CPU context has called the EnterSchedule function into the traphandler (this is done directly on boot or through a system call otherwise). From within the trap handler, `curcpuRunScheduler()` is called. `curcpuRunScheduler()` tries to find an M to run. If no M's are found, as is the case at boot time, `curcpuSleep()` is called from within the scheduling function. When the CPU is awoken, it returns from `curcpuSleep()`, it will jump back up to the start of the scheduling process within `curcpuRunScheduler()` and then finds an M from its runnable list. Then it will set M as `cpuctx.exe =` M. This means that M is now the next M that will be running on the CPU. It will now return from `curcpuRunScheduler()` back to the traphandler.

The CPU now has the M it needs for executing gorutines, and all that needs to be done is to start the M. The M provided contains the function `mstart()`. This is the entry point for any newly created M. After having switched over to an M context, `mstart()` will be called.

2.6.2 How Gorutines (G) Are Scheduled

At the end of `mstart()`, `schedule()` will be called. At this point, the system is in the same state as it would be if it had already been running goroutines and needed to schedule a new goroutine. The wake-up process is complete at this point, and regular scheduling of goroutines has started.

`schedule()` will try to schedule a G to run on M. There are three areas in which goroutines can be stored. Either it is on the local queue of the associated P, or it is on the global queue held by the scheduler object, or it is on the local queue of another P. The priority for where to look is to first try to look in the local queue of our own P. If nothing is found there, try to look at the global queue. If there is nothing found there either, `findrunnable()` is called.

`findrunnable()` will try to steal a G from another processor's local queue of goroutines. There will be a number of attempts (this can be tweaked, but is currently 4) to go through a list of all P and look for a G on their local queue. If no goroutine is found, there are some further attempts to look for work on the global queue for goroutines. While M is looking for work this way, it is defined as being in a spinning state, where it is ready for work and actively looking for it, but it is not executing any G. If a G is found, `findrunnable()` will return to `schedule()`. If no G is found, M will eventually start to tear itself down, which is its exiting process. When exiting, it will yield P, which will set P in an idle state. After this, it will make some last-ditch attempts to see if a new G is available at the last moment, but if there are none, M will stop being in a spinning state, and it will become a dead M. It then unwinds, and the memory used by M is freed.

When M has found a G to execute, `execute()` is called. `execute()` will make a system call to the gogo function. gogo will change the context of the system by making the g register point to the G that is about to execute. Afterwards, the program will jump to the function pointer held by G, and the function provided to the `go` call as an argument will then start to run in this new thread. We now find ourselves back in user code on a different goroutine.

2.6.3 Quick Explanation

As can be seen, the process of starting a new goroutine is quite messy and all over the place. A very quick explanation of what happens is that when calling `go` a new goroutine will be created in any case. If there are no idle P in the system, nothing more will happen, and the goroutine will be picked up and executed at a later time.

If there is an idle P, a new M will be crated, and a sleeping CPU will be awoken. The new M will then start to run on the newly awakened CPU and go looking for goroutines to execute. The problem with Embeddedgo is that it only provides a single P, meaning that only one CPU can run at a time. This will be changed in MultiGo.

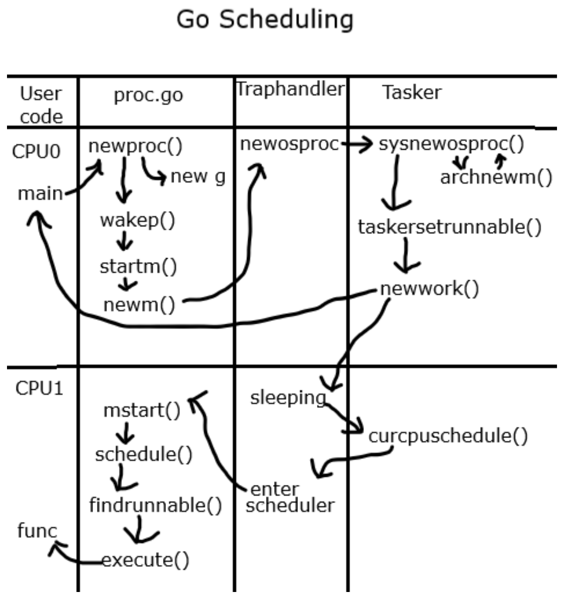


Figure 3: The process of starting and scheduling a new thread.

3 MultiGo

MultiGo supports parallel execution of goroutines, which normal Embeddedgo does not. In addition, there has been more support added for printing on embedded systems. The task proved to be more about knowing where to tweak the existing code than writing a lot of new code. In this chapter, I will explain in detail what non-trivial modifications I have made to MultiGo to solve the issues.

3.1 Supporting Parallel Execution

The bare-metal RISCV runtime would originally only allow for a single P object to be created. This is no problem if only one CPU is running, as having a single P for the entire system would reflect the underlying structure. However, if we try to run a multi-core system with only a single P, things will not work as they should. If there is only one P, the system would only be able to execute code on one CPU at a given moment. M's require a P to execute a G; therefore, if an M running on CPU 1 has the only P, no other M's are allowed to execute at that time, even if there is an idle CPU. In Embeddedgo, only a single P is created, allowing the system to only use one CPU at a time.

3.1.1 Changes In rt0

The solution to this problem is, of course, to have the number of Ps reflect the exact number of CPUs available for executing code. Normally the Go runtime relies on system calls to the OS for figuring out how many CPUs are available, but since there are no OS in bare-metal RISCV, the runtime has to figure out by itself at boot time how many CPUs are used in the system.

Near the start of **rt0_noos_riscv64.s**, all CPUs are counted in a sequential manner. This happens after excess harts have been parked, as they're not going to be used anyway. The counted number of CPUs is stored in a global variable `numberOfCPU` defined in **rt0_noos_riscv64.s**. What happens in the code below is that the address of `numberofCPU` is loaded into register `A0`. Then the content that is on that address, that is, the current number of counted CPUs, is loaded into register `S1`. Earlier in the runtime, all CPU's write their id into register `S0`. To ensure that CPUs are counted sequentially in order to avoid a race condition, an id check is added. If the number of counted CPU's is equal to the id of this CPU, it can be counted; otherwise, do the check until that time. The counting is done by adding 1 to the value in `S1`, which contains the number of already counted CPUs. `S1` is then written back to the address in `A0`, which holds the address of `numberOfCPU` at this time.

```
MOV   $numberOfCPU(SB), A0
MOVW  (A0), S1
BNE   S0, S1, -1(PC)
ADD   $1, S1
MOVW  S1, (A0)
```

Listing 1: Number of CPU's in use are counted.

All of this is done in Goassembly, but normal Go code can't directly use variables defined here. It is therefore required to have a get function. This `TEXT get_numberOfCPU(SB),NOSPLIT|NOFRAME,$0-0` function is at the end of the rt0 file. An interface function written in a regular .go file is required to make a function call to a Goassembly file. The function signature needs to be the same for the interface function and the corresponding Goassembly function. The interface function in this case is placed in **tasker_noos_riscv64.go**.

```
TEXT get_numberOfCPU(SB),NOSPLIT|NOFRAME,$0-0
  MOV   $numberOfCPU(SB), A0
  MOV   $runtime.numbcpu(SB), A1
  MOVW  (A0), S1
  MOVW  S1, (A1)
  RET
```

Listing 2: Get function for numberOfCPU.

3.1.2 Changes In proc.go

There are a few things that need to be changed in proc.go for all of this to work. The only change is that the number of counted Ps has to be called in `schedinit()` right before `procresize()` is called. The two things that are important here are that `ncpu` is set to the counted number of available CPUs and that `procresize()` is provided with the same number as input. In `procresize()` all of the scheduling structures that use P's are set up, and if it does not get the correct number of CPU's as input, the system will not initialize correctly. These things need to happen at this point. If these things are done afterwards, the system will already have moved past its initialization phase and will just break.

```
if noos {
  // getNumbcpu is a function in rt0_noos_riscv64.s which gets the
    number of registered CPU's
  procs = getNumbcpu()
  ncpu = procs
}
if procresize(procs) != nil {
  throw("unknown runnable goroutine during bootstrap")
}
```

Listing 3: The number of avlaible CPU's are fetched from rt0 by calling getNumbcpu() and provied to procresize().

3.1.3 Why This Works

There are two big questions that should be asked about the changes made, and those would be number one: ”Does this work?” and number two: ”If it works, then why does it work?”. It does in fact work, as can be seen in figure 4 in the appendix. These relatively small changes allow the runtime to start up and use multiple CPUs at the same time.

This leaves that last question to be answered: why does it work? It works because if the system has an idle P when creating a new goroutine, it will cause `wakep()` to call `startm()`. In `startm()`, there is going to be an attempt at fetching an idle P if one exists, but if there are no available P, `startm()` will just return and prevent any new Ms from being created. However, if there is an idle P, there are no problems with creating new Ms, as there are Ps that they can use for execution. When creating the main thread, rt0 calls `startm()` to make the main thread; therefore, all of the necessary changes made to the process of creating and starting up new M's are already implemented by Embeddedgo. Because of this, MultiGo can just use the existing procedure for creating and starting M's as Embeddedgo does; all it has to do is make more P's available and have the number of P's reflect the number of CPU's in the system.

3.1.4 Possible Problems

MulitGo has only been tested for a 1 and 2 core system. It is possible that there are more bugs if more than two CPUs are in use at the same time.

CPUs rely on the MSIP registers in the CLINT to send wakeup calls to other sleeping CPUs. If there are multiple CPUs in use, there has to be at least one MSIP register available for each of them. In total, there are 4096 such registers, which makes this number the absolute hard cap for the number of CPUs that can be in use at the same time. Make sure that `maxHarts` is not lower than the desired amount of CPUs. The runtime will only allow `maxHarts` number of CPUs to be available and will park any excess harts.

MultiGo is set up to associate a P with a specific CPU, where the id of P should match the id of the CPU it is used on. When a thread exits and concedes P, the CPU must either go to sleep or acquire the specific P it yielded if it decides it wants to start doing some more work.

3.2 Issues with curcpu()

`curcpu()` is the tasker's method for fetching a pointer reference to the currently running CPU context object. What `curcpu()` does is call `getg()`, which will fetch the G object referenced in the g register. The G pointer returned from `getg()` is then cast into a cpuctx pointer. This works as the cpuctx struct has a G as its first data type. Casting a G in this way is only valid if that G is the G0 of a CPU context; otherwise, it is invalid. Anywhere `curcpu()` is called, it's assumed that the G register is pointing to the G0 of the currently running CPU context. The g register is always set to point to the G0 associated with the

CPU we're on when entering a trapped context. When making changes to the tasker or the trap handler, getting the g register context right is very important for making the runtime work correctly.

I found out that using `curcpu()` often did not work correctly when used in tasker code. For whatever reason, the compiler would compile the code related to the `curcpu()` function to not be where it should be in relation to the .go file. Compilers rewrite code all of the time, and this is no issue as long as you get the same behavior as your written code. The issue here is that `curcpu()` related code would end up executing after the pointer it returns was used elsewhere. The workaround to this problem was to try to avoid using `curcpu()` where possible. An example is shown in the appendix.

3.3 Issues With The Standard Library

Several of the packages that are part of Go's standard library contain code that is system-specific. This means that if we want to add support for more systems, code from the standard library might not support those systems. Packages such as `io` and `os` rely on making system calls to the OS for them to function. There is some support added by the Embeddedgo patch for bare-metal RISCV systems, but not that much.

3.3.1 Implementing noos_fmt

I've added support for printing on bare-metal systems. The fmt package originally relied on the OS to tell it where to write to for the printed text to end up in stdout. Since there is no OS to ask for in bare-metal RISCV, it has to somehow get this information elsewhere. The solution is to add a configuration file with the UART address. These config files are located in the embedded package under arch_const. In noos_fmt, these architecture-specific addresses are used to figure out where to write to.

3.3.2 Memory Limitations

Knowing where to write to is not the only problem. Go code uses an awful lot of memory, to the point where it becomes a real problem for embedded software developing. If all of fmt's functionality is going to be supported, the executable it would create would simply be too big to work on the system I'm using in this project. I have therefore created a more lightweight version of fmt, in addition to the bare-metal modifications. noos_fmt has a subpackage called simple. This package only supports the `simple.Println()` call. It only offers basic printing of strings and nothing more. It does not have support for utf8 characters, only ASCII, and it does not support formatting. This makes it far lighter in terms of memory usage and, therefore, a lot more usable in a bare-metal system.

4 Tools and Method

When working on an embedded project, it is always a question of whether to work on a physical microcontroller or just develop on a virtual machine. I chose to go with the virtual machine option. In this project, I used Qemu[7] as the VM platform and GDB[8] as the debugging tool. In this section, I will explain how to use all of these tools, including Go itself.

4.1 Qemu

Out of convenience, Qemu is used to simulate a RISCV 64-bit Virtio[9] machine, which is used as the development platform for this project. To start the right kind of machine, use this command:

qemu-7.0.0-rc3/build/qemu-system-riscv64 -serial stdio -machine virt -smp 2 -m 128M -bios none -kernel test_code -s -S

The particular command in this case is **qemu-system-riscv64**, found under the **bulid** directory. This will start a RISCV64 machine, which is what we want. In addition to this, there are some extra flags that are required.

- serial stdio This flag will make the virtual machine forward what is output to its own stdio to the stdio of the machine that started the virtual machine.

-machine virt This flag will start a Virtio machine. RISCV does not define everything about the layout of RISCV machines, such as the memory regions. A specific RISCV machine must therefore be chosen, which is a Virtio machine in this case.

-smp 2 SMP is an abbreviation for Symetric MultiProccesing. This means that the system is simulated with several cores that all run the same executable as their kernel. The number after the flag is the number of cores to be simulated.

-m 128M This flag specifies the amount of memory the system has. 128M means that the system is simulated with 128 megabytes of memory.

-bios none This will disable the default BIOS for the Virtio machine. We want to run the user-made code as the kernel in a completely bare-metal system.

- kernel test_code This will start the user-created executable as the system kernel. Here **test_code** is just used as a place holder name, the executable will of course have whatever name it was compiled as. The executable must obviously be compiled for the system that is simulated; otherwise, it will not work. An x86 binary can't run on a RISCV system.

-s -S These flags are used for debugging. The **-s** flag will start a GDB server for the virtual machine on localhost:1234. The **-S** flag will prevent the kernel from immediately starting execution, enabling the user to start GDB and control execution from there.

4.2 GDB

Usually, Delve[10] is used for debugging Go code. Delve understands what gorutines are better than GDB; however, Delve cannot be used for debugging the runtime itself. GDB is the best debugger for working with the runtime. Qemu has support for setting up a GDB server that can connect a host machine that runs GDB to the virtual machine where the Go code is being run. SiFive[11] offers a version of GDB that has support for debugging on a bare metal RISCV system, and this version of GDB is what I'm using for this project. To start debugging, use this command:

riscv-tools/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin/riscv64-unknown-elf-gdb test_code

It is important that the version of GDB being used supports the setup you want to use it on. In this case, I'm running GDB on an x86 machine using Linux as the OS, and I want to debug a RISCV64 binary. This means that GDB has to be able to be executed on a x86 Linux system while also being able to work with a RISCV64 binary. Not all versions of GDB can do this, and therefore it is important to use one that can, like SiFive's mentioned earlier. When using GDB to remotely debug on the virtual machine, use the command **target remote:1234**. This command will connect it to the GDB server on the virtual machine if Qemu is started with the **-s** flag. After all of this, GDB should find itself at the start of the runtime. From here, normal GDB commands can be used to navigate the code.

GDB is not in any way perfect for debugging Go code. There are several calls and objects in Go that confuse GDB and cause it to be somewhat awkward to use sometimes. If you try to step through certain calls in Goassembly code, it causes GDB to lose track of where the program pointer is. If the **CALL** or **MRET** instructions are used, this happens. One solution to overcoming this problem is to look at the jump address in the instruction, followed by setting a break point at that address. Then use the **c** command to continue to that specific break point, as opposed to just stepping through the instruction.

4.3 Go

Golang is a compiled language, much like C. This means that you write your code in a .go file, which is essentially just a text file. If the file has a .go at the end, the Go compiler knows that this file is to be compiled; otherwise, it will ignore it. Go can either be asked to just compile your file by calling **goroot/bin/go bulid "your .go file"**. This will create an executable target for whatever system the compiler was called from. The other option is to call **goroot/bin/go run "your .go file"**. This will compile and start executing the code immediately on the machine the compiler was called from. To compile the type of executable that is needed in this project, use this command:

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

As can be seen, to compile the .go file into an executable that will work on the correct target system, quite a lot of information has to be provided to the compiler. This is what each of the inputs means:

GOOS=noos GOOS is an environment variable used by the runtime to know what OS it is running on. If the GOOS is set to be noos, it means that the runtime is running on a bare-metal system, hence no OS.

GOARCH=riscv64 GOARCH is also an environment variable, but it specifies the architecture of the system instead of the OS. Setting GOARCH=riscv64 means that the code is compiled for a RISCV64 machine.

goroot/bin/go build The **go build** command is what starts the compiler. When called with build, it will just compile the provided .go file into a binary, but does not run it. The key word run can be used instead of build to both compile and run the code on the same system that calls the compiler.

-a This flag will just force a recompilation of all packages used by the .go file, even if there are no changes made to them.

-tags "k210 noos" The **-tags** flag is used to provide the compiler with certain tags that it will use to decide which files to compile. In Go, you can add a command like this at the top of the file: `// +build "tag_name"`. This will only compile that file if the tag specified in the `+build` field is provided to the compiler. A file may have several such tags that are required for the compiler to compile it. In this case, we provide the tags `"k210"` and `"noos"` to the compiler. The `"noos"` tag is used since it is a bare metal system, and `"k210"` means that we compile for a Kendryte machine. I use a Virtio machine in this project, but Kendryte has a similar layout to Virtio machines, so using this layout works fine.

-ldflags "-M 0x80000000:128M" This tells the linker that the stack starts at `0x80000000` and is of size 128 megabytes. The start of the stack is something that is not defined by RISCV and is therefore vendor-specific. In a Virtio machine, memory starts at `0x80000000`.

-gcflags "-N -m -l" Using **-gcflags** is optional. It provides the garbage collector and Go's memory system with flags that can change their behavior. The **"-m"** flag will print all variables that end up on the heap. The Go memory management system performs an operation known as escape analysis[12]. This means that at compile time, everything that is placed in the heap, such as global, variables are analysed to see whether or not they escape the scope in which they were first used. If the variable does not escape, it can effectively be used as just a local variable for that specific scope, and the variable is placed on the stack instead of the heap. Using **"-m"** is useful when trying to understand what is going on in Go's memory management system, and it can be used to see if something in fact ends up on the heap or if it is just put on the stack. The **"-N"** and **"-l"** flags turn off optimization. This can be useful when running simple tests.

test_code.go This is the user-crated .go file that is to be compiled.

4.4 Method

Embeddedgo has very limited support for packages that rely on system-specific functions. Vanilla Go offers quite a lot of different tools to benchmark code with, such as tracing of what has been executed and profiling with pprof[13]. None of this is supported in bare-metal RISCV. At the start of the project, not even the fmt package used for simply printing to stdout had support for bare-metal RISCV. There was also no support for the Time package, meaning that the program could not provide you with the execution time itself.

As has already been written about in chapter 3.3, I implemented a simple form of printing that works for MultiGo. This will at least allow for printing, but it does not solve the timing issue. The solution for this problem was to record the computer screen when benchmarks were executed, followed by analyzing the footage. Print statements were used to determine when execution of the benchmark both started and stopped. To figure out the timing between start and stop, the frames between the print statements were counted. Knowing that the recording was done at 10 frames per second, the execution time of the benchmarks can be calculated from the number of frames between the beginning and the end of execution.

For this project, I used SimpleScreenRecoder[14] to record my screen and DJV[15] to analyze the footage. Both programs can be freely downloaded for Linux. Any other software that can do screen recording and step between frames can also be used to perform these benchmarks.

5 Benchmarks and Results

I will be looking at the performance differences between Go in a 1 versus 2 core system as well as the performance differences between Embeddedgo and C. C is the most obvious language for embedded software development, and it will be interesting to see how Embeddedgo compares to it as an alternative language.

5.1 The Benchmark

The benchmarking between a 1 vs. 2 core system was done with the benchmark **benchmark_simple_arithmatic.go** found in the **benchmarks** directory. This benchmark is quite simple in that it starts a specified number of goroutines, each of which runs a for-loop a specified number of times, where all that happens is a simple addition.

```
for i:=0; i<numbGorutines; i++{
      wg.Add(1)
      go func(){
        temp := 0
        for j:=0; j<numbAdditionsBase*test; j++ {
          temp += 1
        }
        result += temp
        wg.Done()
      }()
    }
wg.Wait()
```

Listing 4: Contents inside the gortuine in simple_arithmatic.

It is easy and simple, and it can test and benchmark the most interesting differences between a 1 and 2 core system. There are four variables that can be tweaked to test different aspects of MultiGo. The first one is **numbAdditionsBase**, this will decide how many times the addition instruction inside the gorutine happens. It is used to scale up the number of instructions in the benchmark such that the execution time takes enough time to be observable in a screen recording. The addition should happen over one hundred million times by each core for the execution of the benchmark to last over one second and be observable through screen recording.

The second variable is **numbGorutines**. This specifies the number of goroutines that are to be started. It can be increased to start more goroutines. If the benchmark is running on a two-core system, increasing the number of goroutines to more than two will not increase the performance. However, it can be turned up to test how many goroutines can exist at the same time.

The two last variables are **numbTests** and **iterations**. These are not used at the same time, and one will be set to 1 if the other is not 1. The **numbTests** variable is used to run the benchmark a number of times equal to **numbTests**. For each time the benchmark is run, the number of additions is increased by the number of tests that have been run multiplied by **numbAdditionsBase**. This means that if **numbTests** is 3, then the benchmark is run 3 times, first doing 1 times **numbAdditionsBase** number of additions, then doing 2 times as many additions, and lastly doing 3 times as many additions. This can be useful when gathering data for the execution time of different numbers of instructions.

The **iterations** variable can be changed to decide how many times the benchmark will be run without changing the number of additions done in the benchmark. It can be used to determine the average execution time of a given number of additions, as the execution time for the same benchmark is not always the same for a multi core system.

5.2 1 Versus 2 Core Go

To test the performance between a 1 core and a 2 core system, I run the simple_arithmetic benchmark with **numbAdditionsBase** set to one hundred million, **numbGorutines** set to 2, **numbTests** set to 10, and **iterations** set to 1. I run the benchmark first with **-smp 2** in Qemu to simulate a 2-core system and with **-smp 1** to simulate a 1-core system. The video recording of this benchmark can be found in **Recordings/Go_simple_arithmetic_benchmark_test1.mp4**. I use the notation $a \to b = c$. This means that the measurement starts at frame a and ends at frame b and has an execution time of c frames. The measurement of one round of additions is considered to start at the moment "new test" is printed and end when either "correct result" or "wrong result" is printed.

5.2.1 Results

Number of Additions	**2 Core Exec Time In Frames At 10 FPS**	**1 Core Exec Time In Frames At 10 FPS**
1×10^8	$155 \to 171 = 16$ frames	$1078 \to 1093 = 15$ frames
2×10^8	$171 \to 202 = 31$ frames	$1093 \to 1124 = 31$ frames
3×10^8	$202 \to 248 = 46$ frames	$1124 \to 1171 = 47$ frames
4×10^8	$248 \to 310 = 62$ frames	$1171 \to 1233 = 62$ frames
5×10^8	$310 \to 388 = 78$ frames	$1233 \to 1310 = 77$ frames
6×10^8	$388 \to 480 = 92$ frames	$1310 \to 1403 = 93$ frames
7×10^8	$480 \to 589 = 109$ frames	$1403 \to 1511 = 108$ frames
8×10^8	$589 \to 712 = 123$ frames	$1511 \to 1634 = 123$ frames
9×10^8	$712 \to 851 = 139$ frames	$1634 \to 1774 = 140$ frames
10×10^8	$851 \to 1007 = 156$ frames	$1774 \to 1928 = 154$ frames

Table 1: Benchmark results from test1.

From looking at the data in table 1 it can be concluded that both a 1 core and 2 core system have the same execution time for this benchmark. The expected result would be that the 2-core system would have half the execution time of a 1-core system. How can this result be explained?

When a new goroutine is started, it has the possibility of being executed by any machine thread using any available core. A goroutine is not tied to a particular core. It is therefore possible that several goroutines are executed by the same core. This is seemingly what has happened in this run of the benchmark, providing the same execution time in both the 1 and 2 core systems.

Taking another look at the **Go_simple_arithmetic_benchmark_test1.mp4** footage, it can be observed that there was an initial attempt to run the benchmark on a 2-core system. The first test with 1×10^8 additions is successful and has an execution time of $47 \rightarrow 55 = 8$ frames. This is half the execution time for the same amount of additions in a 1-core system, and it is what would be expected when running a 2-core system. Immediately afterwards, though, the program crashes, and I have to make a second attempt to run the benchmark. From all of this, there seem to be a few insights to gather.

- **In a multi-core system, goroutines can be executed by *any* core.**
- **If multiple cores are used, the program may crash.**
- **Execution is seemingly more stable if only a single core is used.**
- **The longer the amount of time spent using two cores at the same time, the greater the chance of the program crashing.**

5.2.2 Reliability

Since where goroutines end up is not deterministic, it would be interesting to look at the average execution time of a benchmark with a given number of additions. By changing the **iterations** variable, the benchmark can be run multiple times with the same number of additions. For this test, **iterations** is set to one hundred and **numbAdditionsBase** is set to one million. This will give us a total number of additions equal to one hundred million, which is the starting value in the previous measurements. From the footage in **Recordings/Go_simple_arithmetic_benchmark_test2.mp4** it can be seen that there are two successful attempts at running the benchmark from start to finish. The result from these are $215 \rightarrow 232 = 17$ frames and $280 \rightarrow 297 = 17$ frames. The execution time is about the same as one core doing 1×10^8 additions. This indicates that through the entire test, only one core was used. This shows that the system can end up in a situation where only one single core is used all of the time, regardless of how many new gorotuines are created. This behavior is not deterministic, though. Sometimes both cores are used, which is demonstrated at the beginning of the test1 footage.

Through experience, I have found that MulitGo runs more reliably if it is executed with GDB. For whatever reason, setting a break point at the start of the program and at the end of the program usually increases the success rate of a complete execution of the benchmark. Originally, it was suspected that maybe the garbage collection could be the cause of the crashing, but turning this off will not make the crashing go away. Therefore, the most likely reason for the crashing are race conditions arising between code executed by multiple cores. The nondeterministic nature of this happening also supports this hypothesis.

In the case that it is a race condition happening, tracking it down and solving it would be very difficult, as it happens once every few ten million instructions. On top of this, Embeddedgo does not print any error messages when it crashes, making it extremely hard to debug.

5.3 Go Versus C

C is a very popular choice of language when developing software for embedded systems. C is very light in terms of memory usage; it is a compiled language, and it does not require any additional software at runtime to work. It also has a very fast execution time. It is therefore interesting to see how Embeddedgo compares to C in terms of what the languages offer. To compile a C file for bare-metal RISCV, use this command:

```
riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14
/bin/riscv64-unknown-elf-gcc -g -ffreestanding -O0 -Wl,–gc-sections
-nostartfiles -nostdlib -nodefaultlibs -Wl,-T,riscv64-virt.ld -o test_c
crt0.s test_c.c
```

It assumes you're doing this from the **cstuff** directory. This directory contains everything needed for compiling a RISCV64 binary. There are many great internet resources[16] that explain every part of this command.

5.3.1 Execution Time

For testing and comparing the execution times, I created two identical matrix multiplication benchmarks, both in C and in Go. The C benchmark can be found under **cstuff/test_c.c**, and the Go benchmark can be found in **tests/test_matrixmultiplication.go**. When compiling C for bare-metal RISCV, the standard library is omitted, as it is not supported for bare-metal execution. Because of this, the benchmarks cannot be printing. To then be able to know when they finish execution, I start both benchmarks in GDB and set a break point when `main()` is about to exit. As before, I record my screen and count the frames from start to finish to figure out the execution time. The footage of this test can be found in **Recordings/Go_vs_C_test.mp4**. I consider the start of the execution when **continue** is prompted in GDB, and the end of execution is when the break point is hit. The execution time of the Go benchmark is $591 \rightarrow 660 = 69$ frames, and for C it is $1331 \rightarrow 1373 = 42$ frames. The footage was recorded at 10 FPS. From this, it can be concluded that C is quite a bit faster than Go, with Go using 64% more time. The difference is not huge, though, with C being less than twice as fast as Go.

5.3.2 Features

Both C and Go are compiled languages, and neither requires additional software to work. This is probably the most important attribute for a language to have to be suitable for bare-metal software development to begin with. In this regard, they are the same.

C's standard library does not have native support for bare-metal systems and is reliant on an OS. You can, of course, find libraries that work with bare-metal RISCV, but they are not shipped with the compiler. This is an area where Go has an advantage. Go has native support for many features, such as garbage collection and thread support. With MultiGo, it even has support for parallel execution and basic printing. In terms of features and ease of use, Go is clearly superior to C. C, however, provides a bit more access to low-level functionality. C and Go do not treat pointers in the same way. Go pointers do not normally support pointer arithmetic and are more strict when it comes to casting. There is a workaround in Go, though, where, by using the **unsafe** package, pointers can be cast into an `unsafe.Pointer`. These types of pointers are effectively the same as a C pointer, and they allow for pointer arithmetic. Function pointers cannot be held by a variable in Go, though.

5.3.3 Memory Usage

Probably the biggest difference between Go and C is memory usage. Go uses far more memory than C, to the point where running bare-metal Go can be a big problem in most embedded systems. Comparing the test_matrix and test_c executable, it can be seen that test_c use 9.1 kilobytes of instruction memory and test_matrix use 1.0 megabytes of instruction memory. This is over a one hundred time difference, even though the code these binaries are compiled from is pretty much identical.

Most of the packages in Go's standard library add several hundred kilobytes to the size of the executable if they are used. All of this becomes a real problem when using Go for embedded software development and will outright make Go unusable for this if the target system is too memory-restricted. Go requires at least several megabytes of instruction memory to work well. If the system has less than a megabyte of instruction memory, Go will not work for it at all, as the runtime requires at least that much. In the field of memory usage, C is a clear winner over Go, and in fact, it is the only option of the two in very small systems.

6 Conclusion

The clear limitation for software development with Go is the heavy memory usage by Go's runtime and its libraries. Go was never really designed with embedded development in mind. Although in situations where memory is less restricted, using Go for embedded development works very well.

Go comes with far more features than a language such as C. Go has native support for concurrent programming, and with MultiGo, it also has support for parallel programming. MultiGo is, however, far from perfect. It works well if the gorutines are not too large, but because of weird race conditions, it can unexpectedly crash if provided with too large a work load. There is still work to be done with MultiGo for it to run reliably under all circumstances.

References

[1] "Rust home page." https://www.rust-lang.org/. Sourced 12.5.2023.

[2] Google, "The go home page." https://go.dev/. Sourced 14.5.2023.

[3] michalderkacz, "embeddedgo/patch." https://github.com/embeddedgo/patch. Sourced 14.5.2023.

[4] 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 14.5.2023.

[5] RISCV, "Riscv home page." https://riscv.org/. Sourced 14.05.2023.

[6] "A quick guide to go's assembler." https://go.dev/doc/asm. Sourced 12.5.2023.

[7] Qemu, "Qemu home page." https://www.qemu.org/. Sourced 12.5.2023.

[8] "Gdb: The gnu project debugger." https://www.sourceware.org/gdb/. Sourced 12.5.2023.

[9] "Virtual i/o device (virtio) version 1.1." https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html. Sourced 12.5.2023.

[10] "Delve git page." https://github.com/go-delve/delve. Sourced 14.5.2023.

[11] SiFive, "Sifive_riscv_gdb." https://github.com/sifive/freedom-gdb-metal. Sourced 12.5.2023.

[12] S. Gangemi, "An overview of memory management in go." https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8. Sourced 3.5.2023.

[13] "pprof git page." https://github.com/google/pprof. Sourced 15.5.2023.

[14] "Simple screen recorder." https://linuxhint.com/install_simple_screen_recorder_ubuntu/. Sourced 14.5.2023.

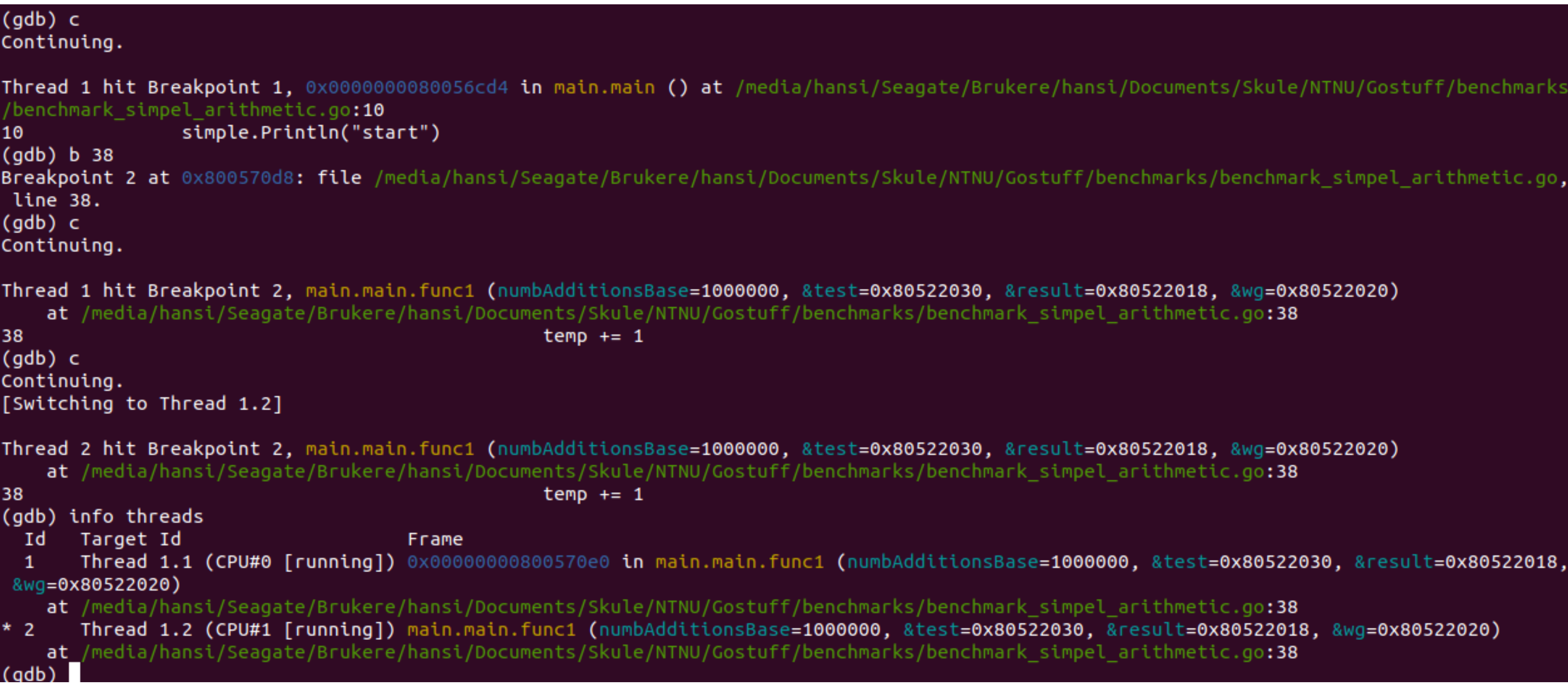
[15] "Djv." https://darbyjohnston.github.io/DJV/. Sourced 14.5.2023.

[16] Twilco, "Risc-v from scratch 2: Hardware layouts, linker scripts, and c runtimes." https://twilco.github.io/riscv-from-scratch/2019/04/27/riscv-from-scratch-2.html. Sourced 14.5.2023.

A Appendix

In appendix, I show some results and code.

A.1 Screenshots



```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, 0x0000000080056cd4 in main.main () at /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks
/benchmark_simpel_arithmetic.go:10
10              simple.Println("start")
(gdb) b 38
Breakpoint 2 at 0x800570d8: file /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks/benchmark_simpel_arithmetic.go,
 line 38.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, main.main.func1 (numbAdditionsBase=1000000, &test=0x80522030, &result=0x80522018, &wg=0x80522020)
    at /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks/benchmark_simpel_arithmetic.go:38
38                                              temp += 1
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 2, main.main.func1 (numbAdditionsBase=1000000, &test=0x80522030, &result=0x80522018, &wg=0x80522020)
    at /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks/benchmark_simpel_arithmetic.go:38
38                                              temp += 1
(gdb) info threads
  Id   Target Id                    Frame
  1    Thread 1.1 (CPU#0 [running]) 0x00000000800570e0 in main.main.func1 (numbAdditionsBase=1000000, &test=0x80522030, &result=0x80522018,
 &wg=0x80522020)
    at /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks/benchmark_simpel_arithmetic.go:38
* 2    Thread 1.2 (CPU#1 [running]) main.main.func1 (numbAdditionsBase=1000000, &test=0x80522030, &result=0x80522018, &wg=0x80522020)
    at /media/hansi/Seagate/Brukere/hansi/Documents/Skule/NTNU/Gostuff/benchmarks/benchmark_simpel_arithmetic.go:38
(gdb)
```

Figure 4: Example of multiple CPUs being used at once.

A.2 Code

```go
type m struct {
  g0      *g
  morebuf gobuf
  divmod  uint32

  procid        uint64
  gsignal       *g
  goSigStack    gsignalStack
  sigmask       sigset
  tls           [6]uintptr
  mstartfn      func()
  curg          *g
```

```
caughtsig     guintptr
p             puintptr
nextp         puintptr
oldp          puintptr
id            int64
mallocing     int32
throwing      int32
preemptoff    string
locks         int32
dying         int32
profilehz     int32
spinning      bool
blocked       bool
newSigstack   bool
printlock     int8
incgo         bool
freeWait      uint32
fastrand      [2]uint32
needextram    bool
traceback     uint8
ncgocall      uint64
ncgo          int32
cgoCallersUse uint32
cgoCallers    *cgoCallers
doesPark      bool
park          note
alllink       *m
schedlink     muintptr
lockedg       guintptr
createstack   [32]uintptr
lockedExt     uint32
lockedInt     uint32
nextwaitm     muintptr
waitunlockf   func(*g, unsafe.Pointer) bool
waitlock      unsafe.Pointer
waittraceev   byte
waittraceskip int
startingtrace bool
syscalltick   uint32
freelink      *m

mFixup struct {
  lock mutex
  used uint32
  fn   func(bool) bool
}

libcall   libcall
libcallpc uintptr
libcallsp uintptr
libcallg  guintptr
syscall   libcall

vdsoSP uintptr
vdsoPC uintptr

mOS
```

```
	mqkey uintptr

	preemptGen uint32

	signalPending uint32

	dlogPerM

	locksHeldLen int
	locksHeld    [10]heldLockInfo
}
```

Listing 5: The entire M struct

```
type g struct {
	stack       stack
	stackguard0 uintptr
	stackguard1 uintptr

	_panic       *_panic
	_defer       *_defer
	m            *m
	sched        gobuf
	syscallsp    uintptr
	syscallpc    uintptr
	stktopsp     uintptr
	param        unsafe.Pointer
	atomicstatus uint32
	stackLock    uint32
	goid         int64
	schedlink    guintptr
	waitsince    int64
	waitreason   waitReason

	preempt       bool
	preemptStop   bool
	preemptShrink bool

	asyncSafePoint bool

	paniconfault bool
	gcscandone   bool
	throwsplit   bool

	activeStackChans bool

	parkingOnChan uint8

	raceignore     int8
	sysblocktraced bool
	sysexitticks   int64
	traceseq       uint64
	tracelastp     puintptr
	lockedm        muintptr
	sig            uint32
	writebuf       []byte
	sigcode0       uintptr
	sigcode1       uintptr
```

```
  sigpc          uintptr
  gopc           uintptr
  ancestors      *[]ancestorInfo
  startpc        uintptr
  racectx        uintptr
  waiting        *sudog
  cgoCtxt        []uintptr
  labels         unsafe.Pointer
  timer          *timer
  selectDone     uint32

  gcAssistBytes int64
}
```
Listing 6: The entire G sturct.

```
type p struct {
  id          int32
  status      uint32
  link        puintptr
  schedtick   uint32
  syscalltick uint32
  sysmontick  sysmontick
  m           muintptr
  mcache      *mcache
  pcache      pageCache
  raceprocctx uintptr

  deferpool    [5][]*_defer
  deferpoolbuf [5][32 / noosScaleDown]*_defer

  goidcache    uint64
  goidcacheend uint64

  runqhead uint32
  runqtail uint32
  runq     [256 / noosScaleDown]guintptr

  runnext guintptr

  gFree struct {
    gList
    n int32
  }

  sudogcache []*sudog
  sudogbuf   [128 / noosScaleDown]*sudog

  mspancache struct {

    len int
    buf [128 / noosScaleDown]*mspan
  }

  tracebuf traceBufPtr

  traceSweep bool

  traceSwept, traceReclaimed uintptr

  palloc persistentAlloc

  _ uint32

  timer0When uint64

  timerModifiedEarliest uint64

  gcAssistTime         int64
  gcFractionalMarkTime int64

  gcMarkWorkerMode gcMarkWorkerMode

  gcMarkWorkerStartTime int64

  gcw gcWork

  wbBuf wbBuf

  runSafePointFn uint32

  statsSeq uint32

  timersLock mutex

  timers []*timer

  numTimers uint32

  deletedTimers uint32

  timerRaceCtx uintptr

  preempt bool

  pad cpu.CacheLinePad
}
```

Listing 7: The entire P struct.

```
type cpuctx struct {
  gh       g
  t        *tasker
  exe      muintptr
  newexe   bool
  schedule bool
  runnable mq
  waitingt msl
  wakerq   [fbnum]notelist
  pp       puintptr
  mh       m
}
```

Listing 8: The entire cpuctx struct.

```go
var (
    bestcpu *cpuctx
    bestn   int
  )
  // bug: bestcpu is set to curcpu, and then it refuses to chage no
     matter what.
  // curcpu := curcpu()
  allcpu := thetasker.allcpu
  var a int
  var b int


  p := m.nextp
  if p != 0 { // for some reason, bestcpu is set to be the same as
    curcpu at this point.
    goto byid
  }
  p = m.p
  if p != 0 {
    goto byid
  }
  p = m.oldp
  if p != 0 {
    goto byid
  }
  // naive search for the less loaded cpu
  bestcpu = curcpu()
  bestn = bestcpu.runnable.atomicLen()
  // this should not be done incase p is 0 and there are idle p's.
    (as it would wake a cpu without having awoken a p)
  for _, cpu := range allcpu {
    if n := cpu.runnable.atomicLen(); n < bestn { // just declear
    the cpu with the shortest runnable queue to be the best cpu to
    run m.
      bestcpu = cpu
      bestn = n
    }
  }
  goto end
byid:
  //bestcpu = allcpu[int(p.ptr().id)%len(allcpu)]  // use the cpu
    with the same id as the any p attached to m.
  a = int(p.ptr().id)
  b = len(allcpu)


  bestcpu = allcpu[a%b]
```
Listing 9: Example of omiting the use of curcpu() at the start of taskerSetrunnable() in tasker_noos.go

Link to the MultiGo Git repository: https://github.com/BondeKing/MultiGo