Jiny kernel: Optimizing golang for Virtual Machine

Janardhana Reddy Naredula

Abstract

Virtual machines in the cloud typically run existing general-purpose traditional operating systems(OS) such as Linux. cloud's hypervisor already provides some features, such as isolation and hardware abstraction, which are duplicated by traditional OS, and that this duplication comes at a cost. But today we are using same traditional OS like linux inside the vm which is designed for the physical machine. But as the IaaS cloud becomes everywhere, this choice is make less sense. The features that made these OS desirable on physical machines, such as familiar single-machine administration interfaces and support for a large selection of hardware, are losing their relevance. At the same time, different features are becoming important. The VM's operating system needs to be fast, small, and easy to administer at large scale. Moreover, fundamental features of traditional OS are becoming overhead, as they are now duplicated by other layers of the cloud stack (illustrated in Figure-1).

We present design and implementation of a New kernel for a cloud called Jiny kernel[1]. For a unmodified golang application, We demonstrate how a golang app can run efficiently on Jiny kernel when compared to the same app running on traditional OS like linux. We demonstrate 162% improvement in throughput. By using other techniques throughput and latency can further improved.

1 Introduction

The important role of traditional operating systems is to isolate different processes from one another, and all of them from the kernel. This isolation comes at a cost. This was necessary when different users and applications ran on the same OS, but on the cloud, the hypervisor provides isolation between different VMs so mutually-untrusting applications do not need to run on the same VM. Indeed, the scale-out nature of cloud applications already resulted in a trend of focused single-application VMs. This paper describes the problems of the traditional OS on the vm, and proposed solutions for the problems. The entire software along with application written in c,java,golang,..etc running inside a VM contains the following layers as shown in figure-1:

1) Layer-5: APP: sits on top of Language VM (runs in ring-3): Java/Golang/C.
2) Layer-4: Language Virtual Machine (LVM): sits on top of Guest OS: (runs in ring-3): Examples: jvm for java, Golang runtime system for golang.

3) Layer-3: Guest OS: sits on top Hypervisor.(runs in ring-0) . Examples: Osv,Linux,Jiny,freeBSD..etc
4) Layer-2: Hypervisor(runs in ring-0): Helps to isolates the resources among vm's. Examples: kvm,xen,vmware,...etc
5) Layer-1: Host OS: This layer is not present type-1 hypervisor like xen,vmware. This is only for Type-2 hypervisor like kvm.

All these layers have clear api's between them. Some of the layers are loosely coupled, some are tightly coupled. example: Switching from layer-3 to layer-4 and vice-versa is costly because of the use of system calls.

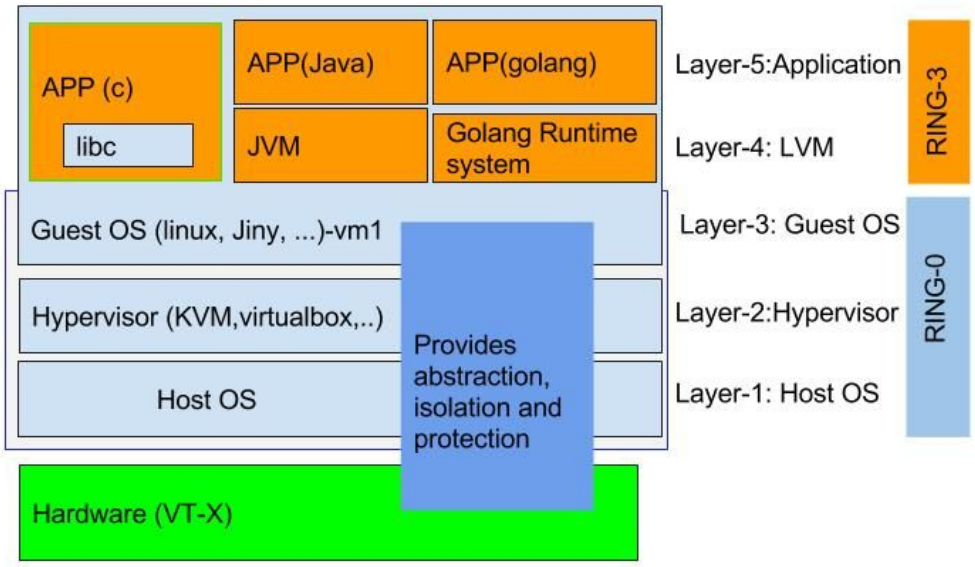

Figure-1

From Layer-1 to Layer-4, all does similar job of resource management, isolation and protection for the app. This isolation comes at a cost, in performance of system calls and context switches, and in complexity of the OS. This was necessary when different users and applications ran on the same OS, but on the cloud, the hypervisor provides isolation between different VMs with the help of virtualization hardware(vt-x..), so mutually-untrusting applications do not need to run on the same VM.

This paper explores the question of what an operating system would look like if we designed it today with the sole purpose of running on virtual machines on the cloud, and not on physical machines. Here we pick a specific high level language golang and cloud OS Jiny for demonstrating the performance improvements on virtual machine. The choice of computer language for application have also impact. In Higher level languages like jvm in java, runtime system in golang also provides some isolation and protection. The positive impact of High level language also described in this paper. Higher level language have advantage of faster development when compared to low level languages like c.

1.1 Summary of Problems

1. **Problem-1:** Too many layers doing similar job: providing abstraction, isolation and protection are provided from layer-1 to layer-4. This duplication causes throughput and latency loss.
2. **Problem-2:** Traditional OS in vm : In the virtual environment, we are using the same traditional OS inside the vm, these os's(linux, bsd, windows) are designed for the physical machines.
3. **Problem-3:** Lack of Non-Posix API's for throughput improvement: To improve the throughput of the app, new posix API will help, but this need changes to the APP/LVM(Language Virtual Machine) and guest OS. addition or changes to existing posix API for the generic kernel like linux will be a difficult task.

2 Jiny Kernel

2.1 Cloud-OS and Language Virtual Machine(LVM):

In this paper, combination of guest OS and LVM are used to solve the above problems. There are some opensource kernels designed for cloud to address the above problems. In this paper we discuss one such kernel called Jiny kernel[1], along with kernel golang runtime system(LVM) is used for prototyping and describing the problems and solutions. Golang runtime system is similar to jvm for java. Other cloud OS's are osv, click os, libra,..etc. Other languages are java,c,..etc. Golang and Java are very similar, but golang is picked for easy prototyping when compared to java. What ever changes we describe related golang will be equally applicable to java

2.2 Design and Implementation of Jiny

Jiny is a Thin kernel designed for cloud. It runs in two modes, First is Normal mode to run linux apps as it is, and second mode is High Priority app mode(HP) to run single app inside the vm. Jiny follows library OS design especially for HP mode. Library OS design attempts to address performance and functionality limitations in applications or LVM that are caused by traditional OS abstractions. It moves resource management to the application/LVM level, exports hardware directly to the application via safe APIs, and reduces abstraction and protection levels. This type of paradigm is very well suited for high level languages like java and golang, since there is a layer LVM(jvm/golang runtime layer) that does abstraction and protection role. Role of LVM layer is minimum in low level languages like c,c++.

Normal mode: Jiny kernel emulates most of the posix calls(system calls) compatible to linux, so that linux binaries can run as it is. Multiple applications can be run at the same time similar to linux OS.

HP mode: In this mode, kernel functions like library os, design with low overhead. Only one app can be launched in the OS, and app runs in the same ring as that of os(i.e ring-0). The functions which are considered "system calls" on Linux(e.g., futex,read ,..etc) in Jiny are ordinary function calls and do not incur special call overheads, nor do they incur the cost of user to kernel parameter copying which is unnecessary in single-application OS. fork and exec posix calls are not supported in this mode. App is launched in a different way. Apart from posix calls, Non posix API's between LVM and Guest OS will further decreases latency and improves throughput of the app. Currently golang and C apps are supported in this mode. In this mode, as application runs in ring-0, LVM uses function calls instead of resource intensive system calls. Here app,LVM and Jiny kernel all three layers runs in ring-0 instead of ring-3. This is as shown in figure-2.

2.3 Design of Golang

go(often referred as golang)is relatively new open source programming language,Go is a compiled, concurrent, garbage-collected, statically typed language developed at Google. Created at google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is used in some of Google's production systems, as well as by other firms. Go is high level language similar to Java, here the "runtime support" is similar to jvm for java. This is called LVM at the layer-4. LVM contain system specific or platform dependent software to interface with various os's like linux,jiny,freebsd,..etc. In this prototype, golang runtime system platform code is modified to run on Jiny kernel in HP mode. Golang app does not need any changes.

3. Solution to the Problems:

Problem-1 solution: Too many layers doing similar job: HP mode in Jiny kernel contains only single app, so kernel does not need to protect, isolate the app. The app(golang app), LVM(runsystem of golang) and guest OS(jiny kernel), all the three layers are combined and made it to run in the same address space as one monolithic piece, but while developing all three are developed independently, and interface between them are same as before, New non-posix api's are added to the existing interface between LVM and Guest OS for improving throughput. Since LVM does the job of app isolation and protection, so kernel spends less cpu cycles to avoid duplication. while combining all the three layers, only LVM and kernel need to change the interface between them without changing the application, this reduces huge cost for application developer. Jiny kernel is compiled as one binary, golang app and runtime system(LVM) both compiled as static second binary, second binary(in elf format) is loaded on the jiny kernel when it boots. After that, all the three layers runs in ring-0 in one address space.

Problem-2 solution: Traditional OS in vm: Jiny kernel is designed from scratch for the cloud. it does not have any constraints of traditional OS like linux,freebsd to run on physical machines. Like traditional OS it does not need

to be generic, since it designed only for cloud, so it is easy to adapt to cloud requirements. Also, it need to be Thin kernel and functions like a library OS especially in HP mode where LVM does lot of OS functionalities. Currently Jiny kernel contain both normal and HP mode, If it is only for HP mode then kernel can become much more thinner.

Unlike traditional OS, Jiny is different in the following ways:

1. Jiny is thin since it is catered only for the cloud requirement and not needed to run on the physical machine and need not be generic. It is designed like a library OS for the single app in HP mode.
2. It is very tightly coupled with LVM: lightweight functional calls with LVM instead of expensive syscall. Non-posix API calls between LVM and kernel on top of the existing posix calls, this Non-posix is only to improve and latency of the app.
3. Performance improvement without changing the application code: This is possible by changing the LVM and guest OS. This may cause additional code for LVM specific, like golang , java. But the size of this codes will much less when compare to the generic nature of traditional os.

Problem-3 solution: **Non-Posix API's for throughput improvement:**

The interface between Layer-3 and Layer-4 is standard posix calls. New non-posix calls will be added to improve the throughput and reduce latency of the app. Adding non-posix calls will be difficult in traditional os, but in cloud OS it is will easy because it not generic and it is designed for cloud.

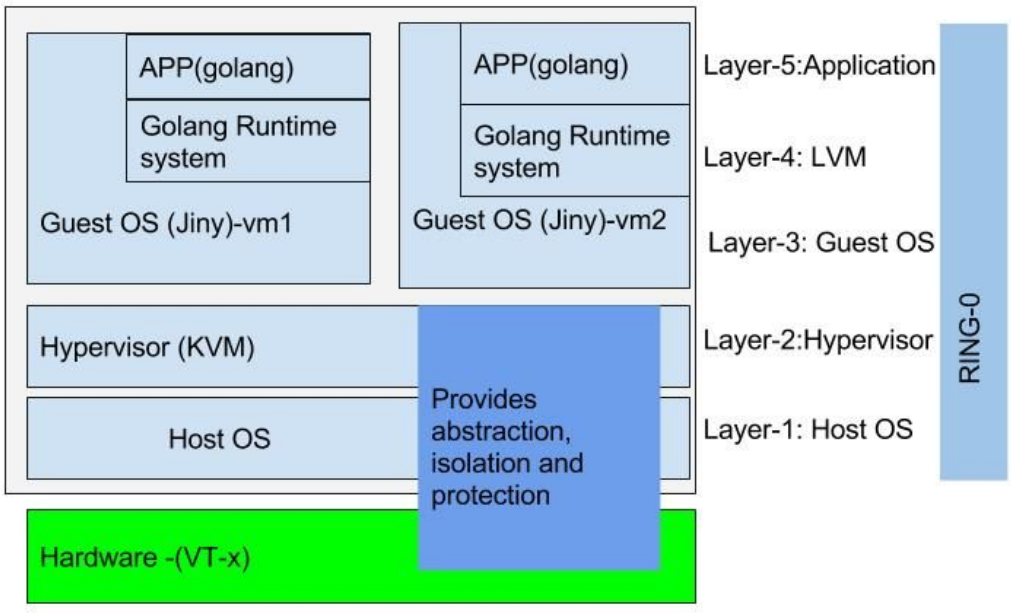

Figure-2

3.1 Details of Prototype:

In the prototype, golang app is chosen to run on a Jiny kernel in HP mode, for this golang runtime system and jiny kernel code related to HP mode need to

modify accordingly. The below two sections describe the code changes at the high level.

Summary of Changes in golang Runtime system:(layer-4:LVM)

Golang 1.6 is used for the prototype, the diff is mentioned in reference[2]. The changes are as follows:

1. All Syscall are converted to function call. Golang runtime system does not uses libc for making syscall, it is having its own set of code to interface with OS. this as made easy to convert all the syscall to function calls.
2. Default Stack size is increased to 8K, since it start with the small size(2k) and the same stack is used when in the kernel code also, this is a temporary change till it is fixed in the Jiny kernel. This change will be undone once it is fixed in Jiny.

Summary of Changes in Jiny Kernel:(layer-3)

The summary of Jiny kernel[1] changes are as follow:

1. New API are added for Launching Golang app, fork and exec calls cannot be used to launch the app, so introduced the new api similar to fork to launch the golang app from the kernel shell. This New API can make it generic, and can be used for other languages like java,..etc.
2. Syscall interface: For HP mode, the entry point to kernel is plain function call instead of system calls.
3. Updating Stack: When app enter into kernel mode, it need to use kernel stack instead of user stack, this is modified accordingly.

Test App: syscall intensive golang app like writing data continuously into /dev/null file[3] is chosen for the test. With Jiny kernel test completes in 8 seconds as against 21 seconds in traditional OS like linux. There is improvement around 162% in execution time. This is mainly due to saving of cpu cycles from converting expensive syscall to lightweight functional call. Non-posix calls are not yet implemented to unlock further performance improvements.

Similar Tests in other cloud OS:

The following are similar type of test done in other cloud OS with different languages and workload:

Guest OS	Application	Improvements
Osv	memcached(c lang)[5]	On linux- 104K - baseline On OSV-posix - 161K (154%)

		On OSV-non-posix -406K (400%)
Osv	Java app	Not collected - There is a improvement from the baseline
Jiny	Golang app with file i/o intensive[3]	On linux: 21sec - baseline On Jiny-posix(HP mode): 8 sec (162%) On Jiny-non-posix: -NA-
Jiny	C syscall intensive test app[6]	Same as above
Jiny	C app: Memcached-udp based	On Linux: 116K - baseline On Jiny-posix(normal mode) : 160K (38%) On Jiny-posix(HP mode): -NA- On Jiny-non-posix : -NA-

Jiny-non-posix: currently, Non-posix Api's are not yet added to the Jiny and golang runtime system(LVM). Non Posix Api's will unlock big throughput and latency. Non-posix api's can be added to the following possible areas:

1. **Networking:**Jiny uses Network scheduler based on Van Jacobson's net channel paper[7]. Non-posix socket API's will be used to improve the network throughput further.
2. **Garbage collection:** memory related Api's can help to improve the throughput of garbage collection in golang. The MMU already tracks write access to memory. By giving the golang runtime system access to the MMU, It can track reference modifications without a separate card table or write barriers.
3. **File System api's:** The test app used is a file intensive app, but it uses only syscall modification to unlock the throughput.
4. **Cpu scheduling:** golang is having cpu scheduler for threads, there will be room to unlock throughput using non-posix api's.
5. **Memory Management:** golang uses tcmalloc based memory allocation. Non-posix api' can used to shrink and expand the memory based on the available memory in the system.

4 Virtualization Models

Virtualizing the resources in the cloud can be doneusing three different models as below:

1. **Container based approach:** uses software inside the host kernel to provide isolation and protection among the vm's.
2. **Traditional Guest os on Hypervisor like KVM,vmware:** uses vt-x hardware to provide the isolation and protection among the vm's.
3. **Cloud Guest OS on hypervisors like KVM,vmware:** uses vt-x hardware to provide the isolation and protection among the vm's. The guest os inside vm function like library os. This model is suitable for single app vm to provide high throughput.

Advantages of cloud OS when compared to others:

1. **Uses vt-x for isolation and protection:** with hardware assist vt-x, cost of isolation is less when compared to container based.
2. **Runs apps in Ring-0:** syscalls and same address space consume less resources when compared to rest of the 2 models. The app here will runs faster then the app in container based.
3. **Non-posix API's provides big throughput improvement:** For memcached in OSV, the improvement is 400% from 154%, this is due to additions of non-posix api.
4. **Boot up time will be less when compare to Traditional guest os.**
5. **Well suited for High level languages:** cloud OS is well suited for High level languages like java, golang,etc. because performance improvement can be done without changing app, only LVM need to change.
6. **Managing complete software stack together in same address space:** App+LVM+OS all together in the same address space, can be managed using uniform REST based API and other tools. Example: The performance metrics of APP,LVM and OS for the prototype are captured uniformly in[3].

References:

[1] Jiny Kernel: https://github.com/naredula-jana/Jiny-Kernel

[2] Golang runtime system changes for Jiny kernel: https://github.com/naredula-jana/Jiny-Kernel/tree/master/modules/HP_golang_changes/

[3] Golang test program: https://github.com/naredula-jana/Jiny-Kernel/blob/master/test/golang_test/file.go

[4] OSV : http://osv.io/

[5] OSV memcached test Results: http://osv.io/benchmarks/

[6] Jiny C app in HP mode: https://github.com/naredula-jana/Jiny-Kernel/blob/master/modules/test_file/test_file.c

[7] Van Jocbson paper : https://github.com/naredula-jana/Jiny-Kernel/blob/master/doc/externel_docs/van_jacbson.pdf