FreeBSD Virtualization - Improving block I/O compatibility in bhyve

Sergiu Weisz
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: sergiu121@gmail.com

Mihai Carabaș
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
Email: mihai.carabas@gmail.com

***Abstract*—**

In a world where cloud computing and cloud infrastructures have become a mainstay, virtualization technologies have enabled a secure way to share resources with different users. A snapshoting mechanism is of great use in the area of virtualization, as it enables the backup of virtual machines, or the creation of templates for machine state replication. These virtual machines have many different virtual devices connected to them that need to have their state saved and restored for a system to be truly useful; examples include block devices, USB devices, or system time. For block I/O a virtual machine may use different types of files depending on its use case. This leads to a greater flexibility in terms of features; for example, one can use a file type that enables saving the state of the hard disk in order to be used later, or as a backup. Hypervisors like VirtualBox, VMWare and Hyper-V already have support for multiple disk file formats. This paper will present a way to implement support for the devices mentioned above, and fill readers in on the procedure of saving device states.

I. Introduction

FreeBSD is an open source operating system that is designed with the goal of being a successor to the BSD operating system, and it is the most popular OS in the BSD family. The reason in part is because of its BSD licence, which allows companies to fork the code and modify it without needing to push the modifications upstream or make them public. This makes it useful for companies like Netflix or Sony to use it in their systems, because of security concerns or financial reasons. Another reason for the popularity of FreeBSD is the performance of the network stack, that outperformes competing OS's [1].

bhyve is the hypervisor that comes packed in with FreeBSD. It is a type-2 hypervisor, so it runs over the operating system. In corporate environments bhyve is used by companies such as Joyent or iXsystems because of its support for legacy operating systems and relative small code base compared to other popular hypervisors.

We have a special interest in the snapshoting feature, because we wish to implement a fully featured checkpoint system for the bhyve hypervisor, which comes with the FreeBSD operating system. Our current implementation of the system will stop the execution of the virtual machine. These features are being worked on in an ongoing project at the University POLITEHNICA of Bucharest. This is the only such feature in the FreeBSD project [2].

This paper will present in depth the state of virtualization in the bhyve virtualization, how the snapshot and restore mechanism works, along with its stong points and flaws, and I will present the improvements I have implelementented in this mechanism, along with what problems I have had along the way.

II. State of the Art

Virtualization is the process of running an operating system over an already existing operating system. The operating system that "runs" directly on the hardware is called a host OS, while the operating system being run over the host is called a guest operating system.

The application that manages the interaction between the host and guest operating systems is called a hypervisor. Depending on the implementation, and the level of optimization, the hypervisor might have components implemented at a kernel/driver level, or it can be fully implemented in user space.

Hypervisors can be split in two major categories by the connection they have with hardware:

- Type 1 hypervisors, which communicate with the hardware directly. ex: Xen
- Type 2 hypervisors, which communicate with the hardware through a fully fledged operating system, like FreeBSD, GNU/Linux or Wwindows. ex: Hyper-V, bhyve, KVM

Snapshoting is the act of saving the state of the virtual machine while it is running. This is done in order to make a backup, or a checkpoint, of the machine that one could roll back the machine to. Another application of this is migrating the virtual machines without shutting them off. By saving their state, moving them to another site, and starting them from the backup, you can make it look, from the point of the virtual machine, that it hasn't been turned off. Hypervisors

that implement these features are Hyper-V, VirtualBox, VMWare, qemu, and others.

In bhyve the act of saving the virtual machine is made of the following steps:

1) Stop virtual CPUs' instruction execution
2) Iterate through all the kernel structures and save their context to a file
3) Iterate through all the used devices and save their contexts to a file
4) Dump the VM memory to a file

For restoring the virtual machine state the hypervisor goes through the following actions:

1) Copy to memory the "old" memory content
2) Copy device information from the restore file
3) Copy kernel structure information stored in restore file
4) Start virtual CPUs' instruction execution

A. Block devices virtualization

A block device is a type of hardware device that is used for I/O operations. Its special characteristic is that reads and writes from it are made in discrete chunks and random access to the address space of the device. Because of the access to all the random access property, they are used for large storage devices, such as HDD's or SSD's.

In bhyve a virtualized block device is either a physical block device that is passed through to the virtual machine, or a file hosted on a physical block device. When a VM needs to access the disk, a request is filled and passed to the hypervisor. The hypervisor in turn receives the request, puts it in a worker pool, and when the time comes, it satisfies the request by calling a read or a write system call.

Currently, the only virtual disk type, also called a disk image, is the "raw" type, which acts like a normal hard drive.

As can be seen from above there is no step where a copy of the virtual storage disk is made during the checkpoint process. If you would want to use the saved machine state for backup purposes, you would have to make a copy of the whole disk image. This can be a problem when you have a large scale service where users backup their VMs like Amazon AWS, or a deployment of Openstack. This makes the need for more complex disk formats quite apparent.

III. Related work

There exist many implementations of block device abstraction. These have all been implemented in various hypervisors in order to offer a more robust and flexible interface to the users, and offer interoperability between hypervisors.

A. QEMU

QEMU [4] (Quick Emulator) is a free and open source emulator. Besides hardware virtualization, it can also do hardware-accelerated virtualization to obtain less overhead than full emulation thanks to the KVM project.

It is the emulator that has provided us with the QEMU Copy-On-Write, file format, which we have used in this paper as a proof of concept for libvdsk. This format has gone through many iterations over the years, and the code that pertains to it is the most complex for this format. It includes complex caching mechanisms that have lead to it being one of the most popular image formats.

B. Palacios

Palacios [5] is an open source VMM that is targeted towards embedded computers. It is built in such a way as to enable its integration into multiple OSs. As of the time of writing, there have not been any new commits in the Palacios main branch since 09 January 2017. It has support for multiple disk file formats, such as RAM disks, or netdisks and QCOW disks, so this present an interest to us.

Palacious VMM's block device abstraction layer works in a similar way to libvdsk. It has a system where a device registers a read, write, open and others, and depending on the file format it calls the specific implementation.

C. VMD

VMD [3] (Virtual Machine Daemon) is the OpenBSD hypervisor. It has a similar approach to the Palacios VMM of using a structure where you register callbacks to read, write and close files. Our implementation for QCOW2 operations were inspired by the implementation in this hypervisor.

IV. Implementation

The work in this paper is based on the libvdsk library, first developed by Marcel Molenaar, and brought up to date by Marcelo Araujo. It is used as an abstraction layer that enables the use of block I/O requests without being concerned about the format of the backing file used by the virtual machine. At the time of the project's start libvdsk only had support for raw image files, files that act exactly like a hard disk.

For each block I/O operation libvdsk implements a function that will be called in the block I/O interface. These functions cover common file operations like open, close, read, write, trim, flush and probe. All of these in turn use a callback to fulfill the operation received. Each file format, be it a raw disk file or

qcow2, has specific functions implemented for all the previously mentioned operations.

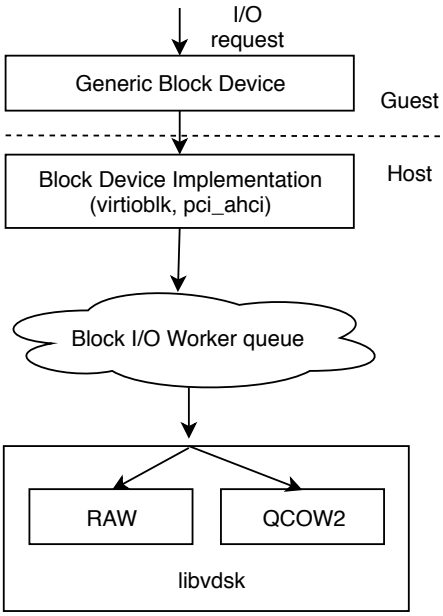

Figure 1. Libvdsk workflow

Figure 1 conveys the whole path of an I/O request through the block devices stack to reach to the libvdsk implementation of a given function. When a request is sent to a block device in the guest VM the emulator implementation (virtio-block, ahci) fills a block I/O request on behalf of the host, and it puts it in a worker queue until it is picked up by a worker thread. After a worker thread runs the job, it calls a generic function in libvdsk which will call the implemented operation for a particular disk file type.

We have been working on implementing these operations for the qcow2 image file format. This format was initially developed for the qemu emulator, but it has since gained a significant following because of features like sparse image files, snapshoting, encryption, and compression. As this is a widely used format with different applications, we have found it the perfect candidate to test libvdsk's capabilities on more complex use cases, other than the raw format that was supported natively in bhyve too.

There are multiple open source hypervisors that have integrated support for qcow2. All of these have a similar implementation to what we have put together, since there aren't many new ways in which one can read data from a file that has a well defined structure. A difference that one may observe is that the workflows for different hyervisors are different. For example, qemu implements a caching mechanism that vmd (the OpenBSD hypervisor), and bhyve do not implement.

Since libvdsk was built, and abandoned, without implicit support for more complex disk types, we needed to add necessary code to it in order to make its design more modular, like adding an additional pointer field to the structure that holds data about the disk that points to an area which has data specific to a disk implementation. This allowed us to store an extra structure related to the qcow2 internal data structures, but it can be used to store structures for any format.

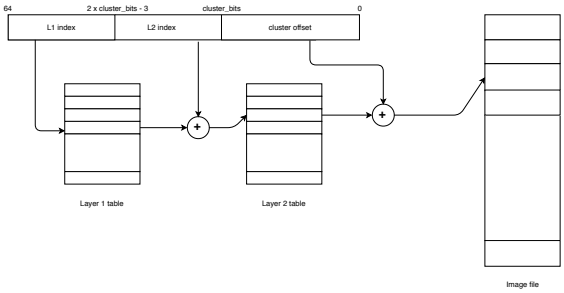

Figure 2. qcow addressing

The qcow2 image file uses a 2 layer addressing scheme, similar to the multi-level page table memory scheme, and it is also uses clusters which are effective physical storage space (analogue to memory pages). A physical address is made up of:

1) An index in the first table, named Layer 1 table (L1), that stores the address of a Layer 2 table
2) An index in the second table, named Layer 2 table (L2), that stores the address of a cluster
3) An offset in the cluster retrieved from an L2 table

This addressing scheme can also be seen in Figure 2. As can be observed in the figure, the cluster size is not a fixed value, it can vary from file to file depending on the configuration. Another detail is the fact that one L2 table is exactly one cluster long, and an entry in it is 8 bytes long.

The layered approach in qcow has de advantage of not needing a large memory allocation when creating the file. Further more, this scheme allows the existence of a Copy On Write mechanism, which is used for maintaining a backup of a disk, since all writes in a backup file will be done on a cluster by cluster basis, so these are the only storage areas that will be duplicated.

Using libvdsk we have implemented the open, probe, read and write operations on the qcow2 image format, with more support to be added in the near future, as this project is of high priority for providing the save/restore functionality with a file backup mechanism.

The read operation on qcow2 follows step by step Figure 2 in order to retrieve the cluster where the desired information is located. If the size being read is more that the cluster size, we try to determine in which part of the read operation the full cluster is located.

The write operation requires the ability to make new clusters on demand if the operation tries to write in

a space where the cluster is not allocated. This will increase the footprint of the image file on the disk as more and more clusters, and eventually L2 tables when the existing one is filled up.

V. Results

The results of the project so far are that we are able to read and write from a disk image that is in the QCOW2 format. In order to test, we converted an existing disk that had information on in to the QCOW format. After, we tested the reads at first reading trying to read the partition table of a disk. We did is by using the fdisk tool. After he did this, we read from different areas from the disk, and we checked the result with the original disk file.

For testing writes, we first tried to write in the partition table, because it is situated in the first sector, so it would rule out offset problems. After we were able to add partitions to the partition table, we checked them using the fdisk tool. After this, we formatted the new partition. We chose to format it at first with the ext2 file system, since it is a simple format that keeps its metadata at the beginning of the disk, but it doesn't have complex mechanisms, such as journalizing. After formatting the partition, we mounted it, created a file, unmounted the partition, and remounted it, to check if the file still had the written data.

As the project is still in development, the only features that have been fully tested are the hooks that go into the libvdsk. We tested this by appending printing functions to the callbacks and checking whether something in printed to the terminal.

Another feature working at the time of writing is the probe function, which prints the header of the disk image format, if it has one.

VI. Conclusion and Further Work

Furthermore, by implementing qcow2 support in bhyve using libvdsk, we aim to show a proof of concept for the library, as it aims to provide a universal way of working with and on virtual disk through an easily extensible API. This will lead in turn to possibilities for implementing varying features for working with disk image files.

Going forward, we aim to implement support for Copy on Write and disk snapshoting in our implementation of qcow2 and integrate this with the checkpointing mechanism outlined in this paper in order to allow for efficient virtual machine backup. Adding to this, we could also add a caching mechanism, similar to the implementation integrated in qemu in order to offload the overhead of translating virtual disk addresses to offsets in the disk vile.

Another avenue worth exploring is implementing support for other virtual disk files, and comparing them to the qcow2 implementation in bhyve.

Acknowledgment

The authors would like to thank Matthew Grooms for hisfinancial support in form of scholarship for Sergiu Weisz. We would also like to address a special "thank you" to Elena Mihailescu an Darius Mihai for their help with debugging various issues that we have encountered, and to Marcelo Araujo who revived the libvdsk project and who maintained continued support for this project.

References

[1] Serving 100 Gbps from an Open Connect Appliance. https://medium.com/netflix-techblog/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99.

[2] University POLITEHNICA of Bucharest, Save and Restore Project. https://github.com/FreeBSD-UPB/freebsd/.

[3] Virtual Machine Daemon man page, vmd(8). http://man.openbsd.org/vmd.8.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 2005.

[5] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, and P. Bridges. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. *24th IEEE International Parallel & Distributed Processing Symposium*, 2010.