Elastic Persistent Memory for Virtualized Environments

Master's Thesis Project Report
Submitted in partial fulfillment of the requirements
of the degree of
Master of Technology

by
Bhavesh Singh
143059003

under the guidance of
Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
India – 400076

June 2017

Approval sheet

This dissertation entitled **Elastic Persistent Memory for Virtualized Environments** by **Bhavesh Singh** is approved for the degree of Master of Technology in Computer Science and Engineering from IIT Bombay.

Examiners

Mythili. Vutukuru

Kameswari. Chebrolu

Supervisor

Chairman

Date: 28/06/2017

Place: Mumbai

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Bhavesh Singh

(Signature)

BHAVESH SINGH

(Name of the student)

14305900 3

(Roll No.)

Date: 28/06/2017

Abstract

Persistent memory is fast, non-volatile storage with performance characteristics between DRAM and hard disks. They are being used in a variety of applications as another tier in the storage hierarchy instead of a replacement of slower memory, to improve I/O throughput and provide efficient reliability. Earlier work on virtualizing persistent memory focuses only on storage class memory which is byte addressable. The use of SSDs in virtualization setups has been limited to a hypervisor managed block cache with VMs being oblivious to the presence of this device.

In this work we propose a novel approach to virtualize solid state drives and expose them as fast, dynamically sized storage devices. Our solution makes two main contributions, (i) provides an interface to dynamically change the effective storage capacity of a device and (ii) allows virtual machines to explicitly manage their fast persistent stores. We present the detailed design and implementation of our system and demonstrate its effectiveness for developing co-operative management policies with dynamically sized SSDs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtualization is a technique used widely in datacenters for server consolidation and efficient resource utilization. To increase the utilization of resources like memory, CPU and storage, multiple servers are run as virtual machines on the same physical server. Storage is typically tiered due to the varying access speeds and capacities of the devices. The fastest and most expensive memories are at the top and the slowest and least expensive memories at the bottom of the memory hierarchy (see Figure 1.1).

Two classes of fast, persistent memories have emerged in the recent times, (i) flash based solid state drives (SSDs) and, (ii) NVRAMs (Non-volatile RAM) which are the byte-addressable and attach to the memory bus. Both these types of persistent memory have access latencies and cost per gigabyte between DRAM and magnetic hard disks. Figure 1.1 shows the position of persistent memory in the typical memory hierarchy.

While flash-based devices have been around for a long time, it is only recently that such devices are being considered as a replacement of hard disks. But the cost per gigabyte of flash still remains higher than hard disks. NVRAMs, on the other hand, are based on a bunch of new technologies like ferroelectric memory, phase-change memory, magnetoresistive memory etc. They are commonly called storage class memories (SCM). Table 1.1 shows the comparison of different types of persistent memory devices along various axes.

	Access Granularity	**Read Latency**	**Write Latency**	**Erase Latency**	**Endurance**
HDD	512 B	5 ms	5 ms	N/A	$> 10^{15}$
SLC Flash	4 KB	25 μs	500 μs	2 ms	10^4 – 10^5
PCM	64 B	50 ns	500 ns	N/A	10^8 – 10^9
STT-RAM	64 B	10 ns	50 ns	N/A	$> 10^{15}$
ReRAM	64 B	10 ns	50 ns	N/A	10^{11}
DRAM	64 B	50 ns	50 ns	N/A	$> 10^{15}$

Table 1.1: Comparison of various properties of different memory types [15]

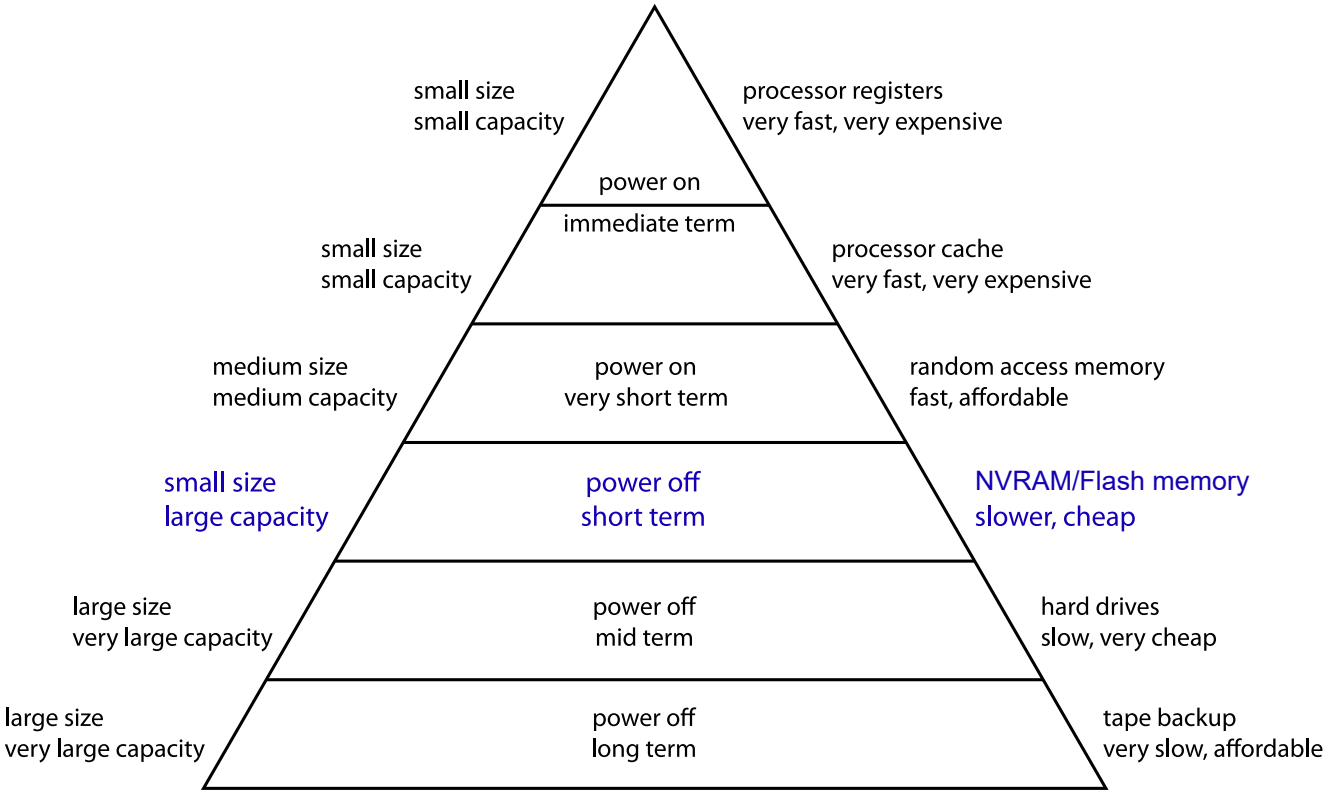

Figure 1.1: Memory hierarchy [21]

1.1 Problem Statement

Two questions can be posed for this category of fast, persistent memory as a resource

1. **Is there a case for virtualizing persistent memory to VMs?**
2. **Are there ways to over-commit/multiplex this resource across VMs?**

In this thesis we make a case for virtualizing flash-based persistent memory and show a novel approach for virtualization and symbiotic management of flash-based solid state drives (SSDs). Our solution makes two main contributions,

1. Provide an interface to dynamically change the effective storage capacity of the virtualized persistent device
2. Allows virtual machines to explicitly manage their fast persistent stores.

We present the detailed design and implementation of our system and demonstrate its effectiveness for developing co-operative management policies with dynamically sized SSDs.

1.2 Motivation

Flash devices are increasingly being used either as a replacement of the traditional spinning disks or as another tier in the storage hierarchy between the DRAM and hard disk. The former use-case stems from the better performance provided by SSDs (see Table 1.1). The latter use case comes due to two reasons,

1. Flash memory has its limitations like limited number of write-erase cycles and larger erase block granularity [15]. There has been work to alleviate these problems with solutions like wear leveling [6], applications (like file systems [13], or user-space applications) custom-built keeping the flash device characteristics in mind [20, 16].

2. The cost per gigabyte of flash memory is higher than its spinning disk counterpart.

In the virtualized environment, flash devices have only been used as another tier in the storage hierarchy, for example, as a block cache [11, 10] or an extended page cache [19]. But the device is not exposed to the VM and the guest OS is oblivious to the presence of a flash device.

Thus by virtualizing the flash device to the VM, we could allow VMs to run applications that use the flash device, in either of the two categories of use-cases mentioned above. Moreover, the higher cost per gigabyte makes it essential that innovative techniques be found to over-commit/multiplex this resource across VMs for better utilization.

Chapter 2

Related Work

This chapter covers the prior work on the applications of persistent memory as well as virtualization efforts.

2.1 Applications of Persistent Memory

We can classify the earlier work around persistent memories into two broad categories. These also include non-virtualized scenarios.

1. **Transparent to the application/guest**
2. **Visible to the application/guest**

2.1.1 Transparent to the application/guest

The application is unaware of the underlying persistent memory and just uses the facilities provided by the operating system/hypervisor.

1. **Block cache [5, 9, 10]** — flash memory is typically used as a block cache on the host. Although the other category of persistent memory, the byte-addressable persistent memories can also be used via a block interface provided by the linux kernel. In the virtualized setting, the hypervisor manages the cache and the guests are unaware of it.
2. **Second chance page cache [19]** — Venkatesan et al. suggest using persistent byte-addressable memory as an extension of the tmem interface. This is still managed by the hypervisor and not virtualized. NVM is used for frontswap while the DRAM is used for cleancache. This helps reduce the rate of swapping and thus improve performance. Due to the exclusive nature of the cache here, the eviction is still in the hands of the guest.

3. **Hybrid main memory systems [8]** — the hypervisors make dynamic decisions to map guest pages to either slower persistent memory or the faster DRAM and swap among the memories based on recency and frequency characteristics and reduce writes to persistent memory.

4. **File system for persistent memory [13]** — specialized file systems akin to log-structured file system, to cater to the asymmetry of reads and writes of some persistent memories, notably flash, but also to facilitate faster I/O directly from the persistent memory.

2.1.2 Visible to the application/guest

The application is optimized to take advantage of the persistent memory.

1. **Libraries facilitating applications to place data in persistent memory [7]** — libraries (e.g. nvmalloc) allow applications to place data in persistent memory. The application-developers can utilize this to place larger data structures (whose size grows with the application data) in persistent memory so as to keep the DRAM relatively free for more frequently accessed data.

2. **Open-channel SSDs, which expose the parallelism in SSDs to the applications [16, 20]** — applications are designed to leverage the parallelism provided by open-channel SSDs. This is usually akin to a SAN storage exposing multiple logical block devices. These SSDs do not implement the flash translation layer inside their controllers and leave it to the operating system to do it.

2.2 Virtualizing Non-volatile Memory

Liang et al. [14] talk about virtualizing byte addressable persistent memory with the minimum overhead. Their design assumes persistent memory as the only type of memory available to the VM (with access characteristics similar to DRAMs). Their solution, called VPM, tries to map more frequently accessed guest PM (persistent memory) pages to the host DRAM to improve performance. They guarantee the persistence property of guest pages, allowing crash recovery.

Amazon Elastic Compute Cloud (EC2) provides fixed-sized virtual disks, backed by SSDs, called Amazon Instance Store [3]. The Instance Store disks have ephemeral semantics and might be used by the guest OS for expendable purposes like caching.

While our solution is similar to Amazon Instance Store [3] in terms of virtualizing a flash device, our approach is conceptually similar to the memory ballooning approach used by Liang et al. [14] offering a co-operative management opportunity for SSDs in virtualized environments.

Chapter 3

Background

3.1 QEMU and `kvm`

QEMU (Quick EMUlator) [17] is a full system emulator that supports running any guest operating system on a linux host via dynamic binary patching. It also provides hosted virtualization support with the help of `kvm` (Kernel-based Virtual Machine) [12] which enables the use of hardware virtualization features to turn the linux operating system into a hypervisor.

3.2 Virtio

Virtio [18] is a paravirtualized device driver model for common drivers like disk, network etc. in virtualized environments. As opposed to a full virtualization setup, where the devices are emulated, it is based on the principle of cooperation between guest and hypervisor to get better I/O performance. Device emulation is complex and highly inefficient. Paravirtualized drivers, on the other hand, work on the principle that the guest operating system is aware of the presence of a virtual device and can employ drivers specially written for such devices. These devices have highly simplified interfaces and usually have two parts, (i) a simplified frontend driver, and (ii) a backend driver which interacts with the actual device in the hypervisor. Figure 3.1 shows the split driver model along with the simplified interface for communication called the virtio ring.

3.2.1 Virtio Ring: Communication Between Frontend and Backend

Virtio Ring is a circular queue abstraction commonly used in the paravirtualized driver designs including the Xen IO ring [4]. It allows for a simple abstraction for communication between the guest and the host. The guest shares some memory with the hypervisor and creates a descriptor which refers to the memory region, and pushes it into the virtio ring. The hypervisor pops the event from the queue and processes it and then sends data back to the guest.

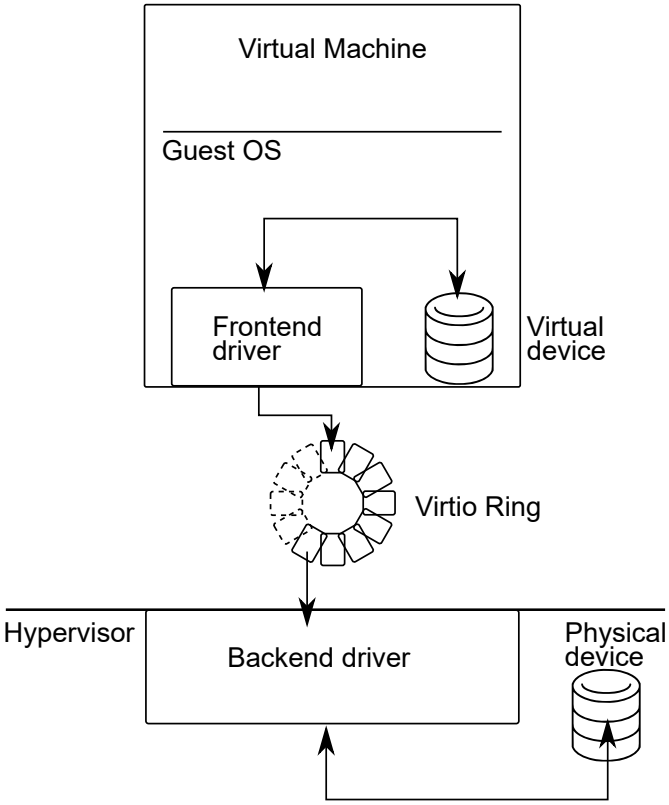

Figure 3.1: Virtio split drivers and virtio ring; icons taken from [1] and [2]

The communication happens by the guest earmarking some memory region as a buffer in which data is to be shared. A descriptor with this information is pushed onto the virtio ring and the hypervisor is notified. The memory region can be labeled as either an input buffer or output buffer depending on the direction of flow of data. Figure 3.1 shows the flow of data in only one direction. But essentially after this first step, the hypervisor can fill in the data in the buffer and notify the guest via an interrupt.

3.2.2 Virtio Balloon Driver

As an example of the virtio device drivers, and also as a reference implementation which we will refer to later while explaining our own device model, we look at the Virtio Balloon driver.

The balloon driver is a special driver residing inside the guest that can help the hypervisor reclaim memory frames from the guest. The idea is that the hypervisor issues a call to the balloon driver to reclaim certain number of pages. The balloon driver allocates that much memory and pins those pages to memory. Then it communicates to the hypervisors the page numbers so that they can be reclaimed.

Figure 3.2 shows the working of virtio balloon driver

(1) Hypervisor sends a message to guest balloon driver to release `n` pages

(2) The balloon driver increases its memory footprint by allocating more memory; called **balloon inflation**

(3) The balloon driver replies back to the hypervisor with the `n` page numbers.

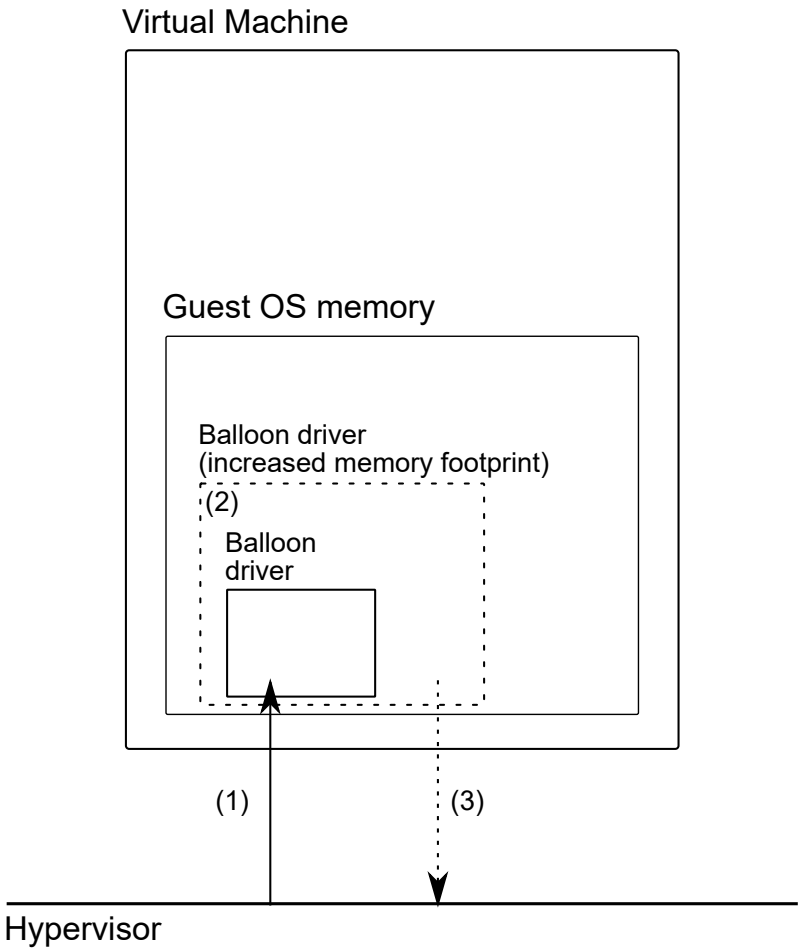

Figure 3.2: Functioning of Virtio Balloon driver

Thus instead of the hypervisor having to choose pages to reclaim from the guest at random, it reclaims pages effectively given by the guest.

Chapter 4

Design and Implementation

4.1 Design

Figure 4.1 depicts our proposed solution. An easy choice is to have static partitions in the underlying SSD and map them to the VMs. But this essentially defeats the purpose of virtualization. Our solution has the following key components

- Hypervisor-managed partitioning and resizing
- Guest-managed block evictions

The possible abstractions to expose this device in the guest OS were

- An abstract caching device which takes an inode, offset and length as parameters
- A generic block device just like the underlying device

We chose the block device interface as it is more generic and could allow a wider range of applications on top. In fact, the former abstraction can be built on top of the generic block device.

4.2 The Resize Interface

The device can be resized in a way conceptually similar to memory ballooning (see Section 3.2.2). The resize interface of the device is depicted in Figure 4.2.

(1) The hypervisor sends a command to shrink the device size.

(2) The vSSD [1] driver asks the above layer [2] about the sector numbers to evict from the device.

[1] Our device shows up in the guest as a block device called `vssd`.
[2] We have given example applications of block caching layer or the file system.

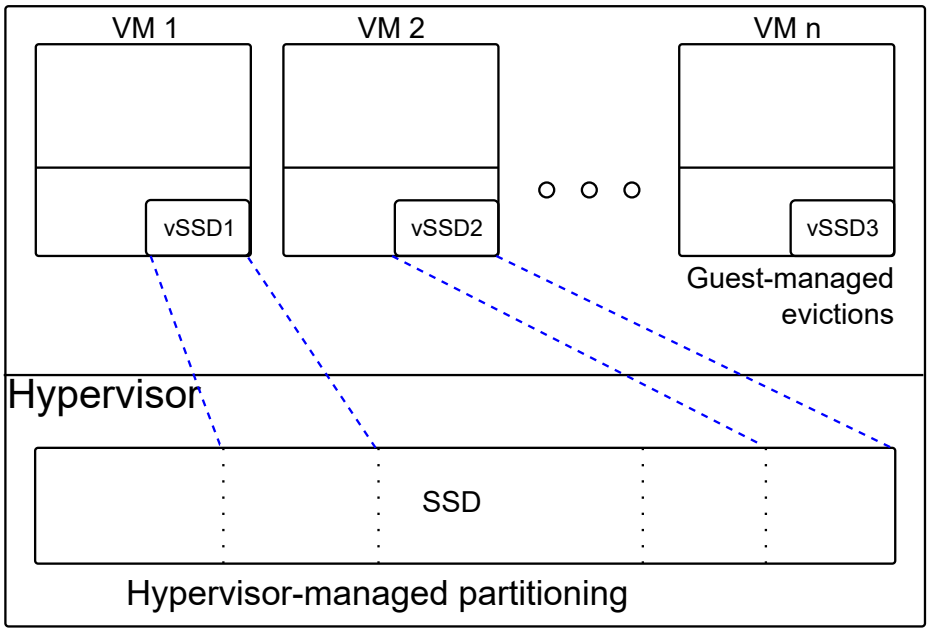

Figure 4.1: Our Proposed Soltion

(3) The application responds with the sectors that should be removed.

(4) The driver passes this information to the hypervisor.

(5) (Not shown in figure) The hypervisor acknowledges the receipt of the sectors.

The corresponding command to grow the device size back is similar. There is a minor difference in the number of steps due to the way virtio device communications happen.

4.3 Implementation

A virtio device is an abstract device that is attached to an abstract virtio bus. To create a new virtio device, we need to extend virtio device abstraction and define functionality specific to our device.

Since virtio bus is an abstract bus, it is encapsulated in a real bus interface called virtio pci bus. This allows the guest OS to discover the virtio device as a pci device and register the front end driver for it. The virtio pci device encapsulates the virtio device functionality and exposes it to the guest OS as a PCI device which can be discovered. We have implemented a new virtio backend device driver in QEMU 2.8.0.

The frontend virtio driver (implemented for linux 4.9.14) registers itself for this PCI device. It then registers a generic block device of size that it reads from the PCI configuration which is set by the hypervisor. It also sets up a `proc` file called `/proc/vssd_balloon_blknum` to get the block numbers to be evicted from the userspace application.

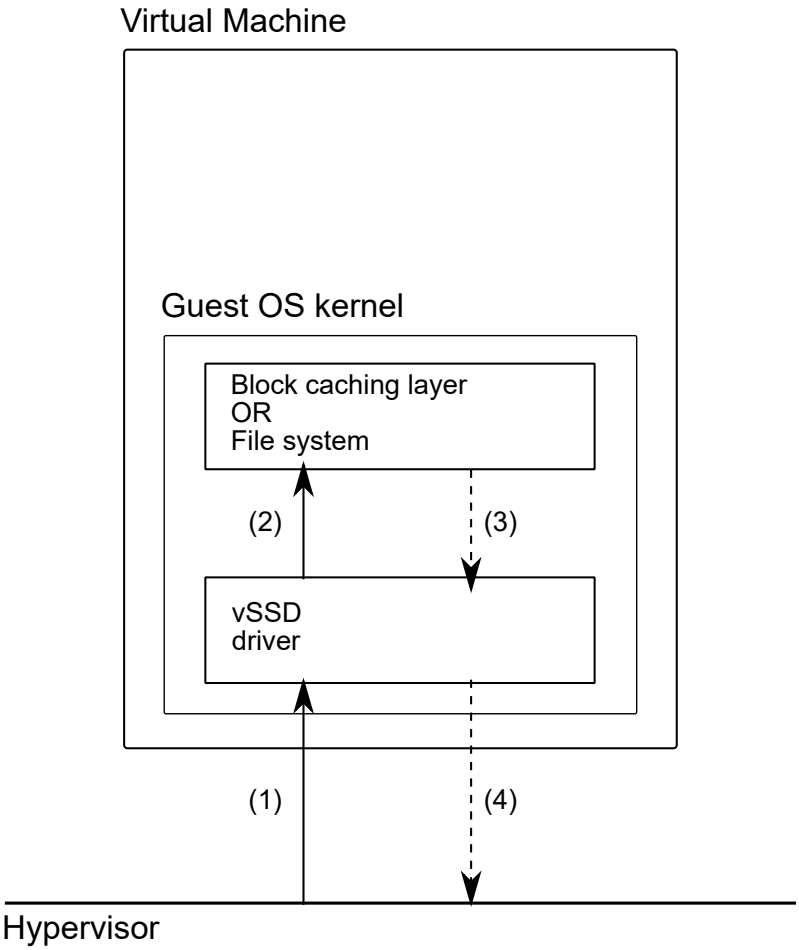

Figure 4.2: The resize interface

We have also implemented a QEMU monitor [3] command called `ssd-balloon` which can be used to trigger vSSD resize in the VM. The source code is available on github.com. See Appendix A for more details.

The `ssd-balloon` command is implemented a bit differently for balloon inflation and deflation. We list the steps for both the cases separately. The syntax for `ssd-balloon` is

```
ssd-balloon ±<number of sectors>
```

The **+** sign denotes balloon deflation i.e. the vSSD will be given back sectors by the hypervisor.

The **−** sign denotes balloon inflation i.e. the vSSD will give sectors to the hypervisor.

[3] QEMU Monitor gives a command line interface to the user to interact with the VM

The procedure for balloon inflation is as follows

- The command from the user is handled in the backend. The backend maintains a configuration of the device whose size now needs to change.
- The backend notifies the frontend via a `config_changed` event that it needs to free `n` sectors.
- The frontend allocates memory for an array of 4096 sector numbers (unsigned 64-bit integers). It then marks the buffer as OUT (direction of data flow) because it needs to free sectors and tell the backend.
- The frontend populates the sector number array with the numbers obtained from the application and notifies the backend.
- The backend reads data from the buffer shared by the frontend, updates its own records of the sector numbers taken from the guest; and acknowledges the number of sectors obtained, to the guest via another notification.

Balloon deflation works in a similar way but with a slight difference. Its steps are outlined below.

- The command from the user is handled in the backend.
- The backend notifies the frontend via a `config_changed` event that it will be given back `n` sectors.
- The frontend allocates memory for an array of 4096 sector numbers (unsigned 64-bit integers). It then marks the buffer as IN (direction of data flow) because it needs to receive sector list from the backend.
- The frontend notifies the backend.
- The backend fills the buffer with the sector numbers and notifies the frontend.
- The frontend reads data from the buffer filled by the backend, updates its records of valid sector numbers on the vSSD; and acknowledges the number of sectors obtained, to the backend via another notification.

There are two important points to note here, (i) there is an extra step of communication involved between frontend and backend when the balloon is deflated, and (ii) the frontend always shares a fixed array of 4096 sector numbers. This means that for balloon commands greater than this size, the backend code has to repeat the steps outlined above till the desired size has not been reached.

4.3.1 Challenges

We need to find more efficient ways to increase the size of a shrunk disk. We can balloon sectors out more quickly but giving back takes a very long time (see Section 5.4 in experiments).

Possible solutions could be to work at a block granularity instead of sector granularity. In fact we could increase the granularity to 2MB block sizes or more. But we need to check the impact of this on the applications using the disk. A caching application might be okay with this but not a file system which would need to move larger amount of data around.

Another major issue that came up while performing experiments was that the host operating system was caching the access to the SSD as well as the VM image file, even after taking care to access the SSD device file in direct I/O mode, and specifying the `cache=none` option for the QEMU VM image file. We could not find the solution to this problem and it hampered our performance numbers as all the accesses were being serviced from within the host cache.

Chapter 5

Experimental Evaluation

5.1 Setup

Our setup included a single VM with 2 GB RAM, 40 GB Hard disk, 2 CPUs and a 1 GB or 32 GB vSSD device. We conducted three categories of experiments on the VM

1. Correctness
2. Proof of concept
3. Virtualization overhead

5.2 Correctness

We verified that the VM resize is happening correctly and that the desired number of sectors are marked invalid/valid respectively when the balloon inflation/deflation command is issued. We used a 1 GB vSSD for this. We created a userspace program that read all the blocks in the vSSD sequentially. On trying to access a block that has been ballooned out (marked invalid), the frontend driver returns an error.[1] The error returned is counted as a cache miss. Graph 5.1 shows the number of cache misses alongwith the respective balloon sizes. The balloon was inflated to the given size and again deflated to zero for each run.

[1]The frontend and backend drivers maintain state in the form of a bitmap of valid sector numbers. 1 sector = 512 bytes. 1 block = 4096 bytes.

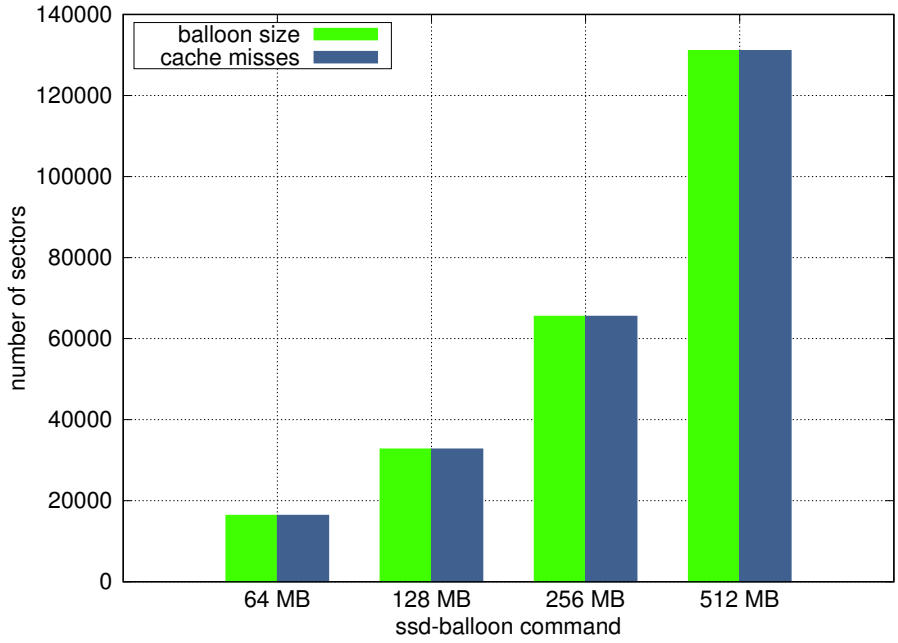

Figure 5.1: Balloon sizes for `ssd-balloon` command

5.3 Case for Guest-managed Evictions

In this experiment we show that guest-managed evictions have chances of performing better than hypervisor-managed evictions. The setup of this experiment includes a 1 GB vSSD in the guest and a userspace application that generates block accesses based on Gaussian distribution with mean 20000 and standard deviation 8192. Graph 5.2 shows the access frequencies for about 130,000 block accesses.

We run this userspace application for two scenarios,

1. Hypervisor evicts blocks according to guest access pattern

2. Hypervisor evicts blocks oblivious to the guest access pattern

We keep the balloon size at 256 MB in both cases. To figure out which blocks to evict, we generate block numbers to access the remaining 768 MB using a random number generator according to the (i) Gaussian distribution (same as the earlier userspace application) and (ii) Uniform distribution. The block numbers not accessed are written to the proc file of our frontend device for ballooning out.

In the first case, since the evictions are based on the knowledge of the access pattern, the number of misses are less, while in the latter case, the number of cache misses are more. In case of a cache miss, the read fails for that block but it continues generating other random block reads on the device. Graphs 5.3 and 5.4 demonstrate our point.

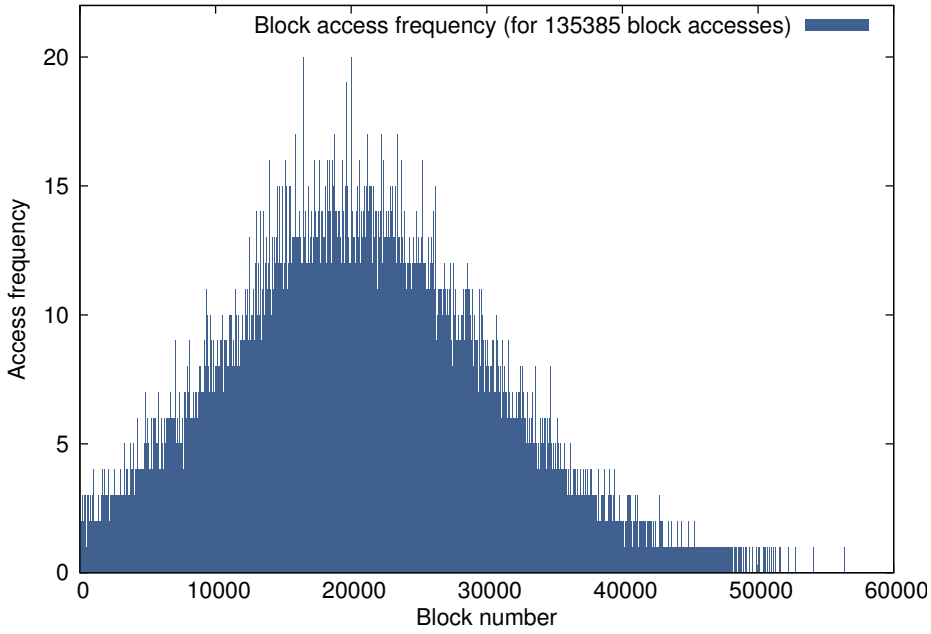

Figure 5.2: Block access pattern for the userspace application

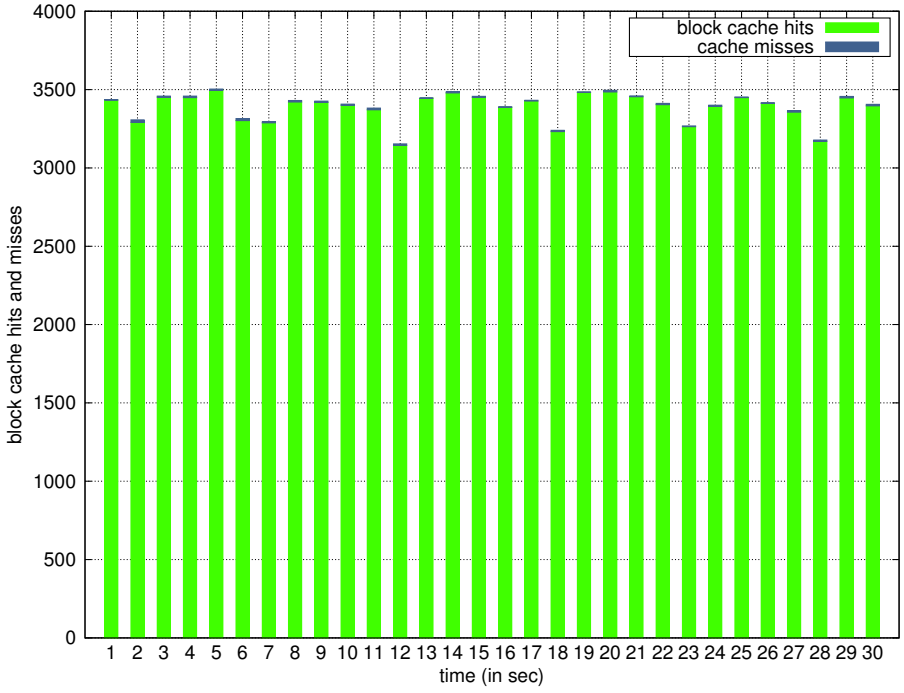

Figure 5.3: Block cache hits and misses with guest-aware evictions

5.4 Virtualization Overhead

In this experiment, we measured the time taken for inflating and deflating the balloon inside the VM. Balloon inflation means that some sectors are freed from the VM and given back to the hypervisor. Thus

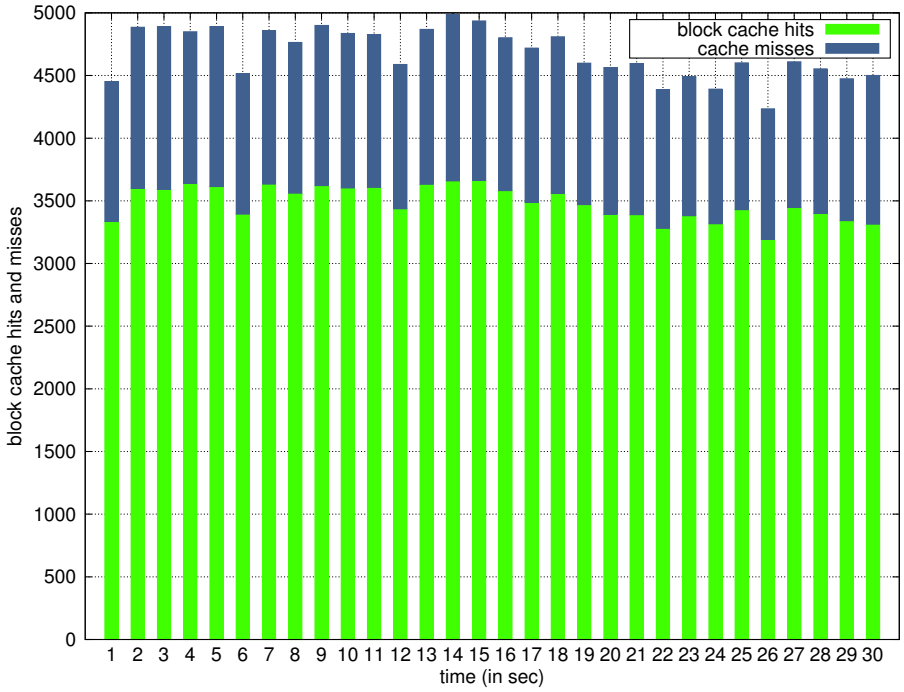

Figure 5.4: Block cache hits and misses with random evictions

any access to those sector numbers should result in an I/O error in the VM. Similarly balloon deflation means that the hypervisor returns back sectors to the VM which marks those sector numbers as valid again. Accesses to these sectors in the VM should be successful now. Graph 5.5 shows the time taken for resizing for different balloon sizes.

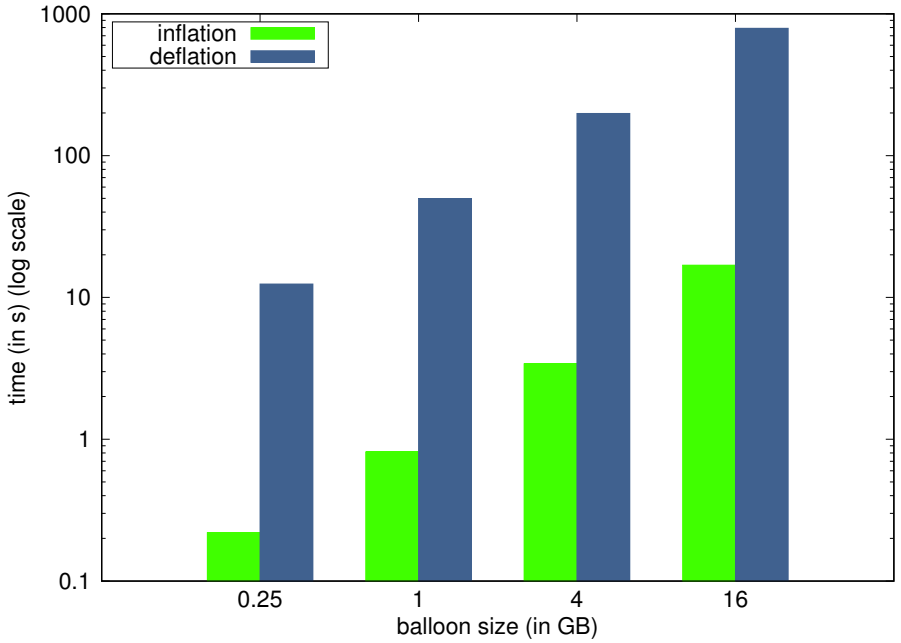

Figure 5.5: Time taken for dynamic resizing

We see that balloon inflation took very less time as compared to balloon deflation. We attribute this to the fact that there is an extra communication step between VM and hypervisor involved in giving back sectors.

Chapter 6

Conclusion

This thesis was an effort to explore one possible approach to virtualize and symbiotically manage SSDs. As part of this work, we defined a new type of resizable virtual SSD for the VMs running in QEMU, using the Virtio framework. We also defined a novel framework for resizing the disk dynamically with inputs from the guest.

6.1 Future Work

We have built a new dynamically resizable block device and an interface for the resize to happen with the guest OS' cooperation. But it needs to be extended in the following ways to be usable for real world applications:

- Each VM in QEMU is a separate QEMU user process running on the linux host. To be able to truly share an SSD among VMs we need to have a way to share state [1] among these different QEMU processes.
- The setup needs to be tested with real applications like dm-cache (a block caching layer in the linux kernel). This would need modification in the dm-cache code to handle the disk resize events coming from the hypervisor via the frontend virtio driver.

Some other small code level modifications to make it more efficient are as follows

- The resize is done iteratively in chunks of 4096 sectors at a time. This design could be improved upon to make it faster.
- The IO is being done synchronously now. But it could be done asynchronously using the QEMU's multithreading support called `coroutines`.

[1] Shared state here means the mapping of logical block numbers of the device in the VM to physical block numbers in the underlying SSD.

6.2 Acknowledgments

The idea for exploring this topic was suggested by Debadatta Mishra. He is always willing to help and his deep insight and valuable inputs and comments throughout the project helped me a lot. He was especially helpful with the inputs on linux kernel whenever I used to get stuck. The long discussions with my guide, Prof. Purushottam Kulkarni, were helpful in getting clarity and direction. I am grateful to both of them.

Bibliography

[1] Circular buffer image. https://gnuradio.org/wp-content/uploads/2017/01/ring-buffers_web.svg. [Online; Last accessed: 22-June-2017].

[2] Disk image. http://cdn.onlinewebfonts.com/svg/img_504033.svg. [Online; Last accessed: 22-June-2017].

[3] AMAZON. Amazon ec2 instance store. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html. [Online; Last accessed: 22-June-2017].

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[5] BYAN, S., LENTINI, J., MADAN, A., AND PABÓN, L. Mercury: Host-side flash caching for the data center. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012* (apr 2012), pp. 1–12.

[6] CHANG, L.-P., AND HUANG, L.-C. A low-cost wear-leveling algorithm for block-mapping solid-state disks. *SIGPLAN Not. 46*, 5 (Apr. 2011), 31–40.

[7] DULLOOR, S. R., ROY, A., ZHAO, Z., SUNDARAM, N., SATISH, N., SANKARAN, R., JACKSON, J., AND SCHWAN, K. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 15:1—-15:16.

[8] HIROFUCHI, T., AND TAKANO, R. RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 112–125.

[9] HOLLAND, D. A., ANGELINO, E., WALD, G., AND SELTZER, M. I. Flash Caching on the Storage Client. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 127–138.

[10] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write Policies for Host-side Flash Caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 45–58.

[11] KOLLER, R., MASHTIZADEH, A. J., AND RANGASWAMI, R. Centaur: Host-Side SSD Caching for Storage Performance Control. In *IEEE International Conference on Autonomic Computing (ICAC), 2015* (jul 2015), pp. 51–60.

[12] KVM. Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. [Online; Last accessed: 22-June-2017].

[13] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 273–286.

[14] LIANG, L., CHEN, R., CHEN, H., XIA, Y., PARK, K., ZANG, B., AND GUAN, H. A case for virtualizing persistent memory. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 126–140.

[15] MITTAL, S., AND VETTER, J. S. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems 27*, 5 (may 2016), 1537–1550.

[16] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 471–484.

[17] QEMU. Qemu. http://www.qemu.org/. [Online; Last accessed: 22-June-2017].

[18] RUSSELL, R. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 95–103.

[19] VENKATESAN, V., QINGSONG, W., AND TAY, Y. C. Ex-Tmem: Extending Transcendent Memory with Non-volatile Memory for Virtual Machines. In *IEEE International Conferance on High Performance Computing and Communications, 2014* (aug 2014), pp. 966–973.

[20] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 16:1—-16:14.

[21] WIKIPEDIA. Memory hierarchy - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory_hierarchy. [Online; Last accessed 14-April-2016].

Appendices

Appendix A

Source Code

The source code is available as a patch for linux 4.9.14 and qemu 2.8.0 at https://github.com/bhaveshsingh/MTP.