SymFlex: Elastic, Persistent and Symbiotic SSD Caching in Virtualization Environments

Muhammed Unais P
unaisp@cse.iitb.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Purushottam Kulkarni
puru@cse.iitb.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

ABSTRACT

Hypervisor managed SSD caching is an often used technique for improving IO performance in virtualization based hosting solutions. Such caches are either explicitly managed by the hypervisor which approximate the access semantics of the applications for improving cache utilization, or operate as statically partitioned devices (which are utilized as caches) by virtual machines. We reason that both these broad directions do not exploit the potential of SSD based IO caches to the fullest, in terms of generalized management policies and performance. We propose *SymFlex*, a novel method to perform symbiotic management of IO caches by enabling elastic SSD devices. Each virtual machine is configured with an elastic virtual SSD whose contents can be managed according to guest OS and application semantics and requirements. Furthermore, the SSD sizing is managed by the hypervisor with a ballooning-like mechanism to dynamically adjust SSD provisioning to VMs based on performance and usage fairness policies. Our primary contribution of this work is to design and engineer the mechanism for elastic SSD disks to be virtualized, and demonstrate usage models and effectiveness of the symbiotic management of SSD caches across virtual machines. Through our empirical evaluation, we show that the overhead of implementing a virtio-based elastic SSD device is minimal (within 5% of virtio based device virtualization techniques). Further, we demonstrate using dm-cache and Fatcache, the applicability and benefits of *SymFlex* for enhancing IO throughput and enforcing VM-level SSD allocation policies.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Secondary storage**.

KEYWORDS

cloud computing, storage virtualization, ssd caching

ACM Reference Format:
Muhammed Unais P and Purushottam Kulkarni. 2021. SymFlex: Elastic, Persistent and Symbiotic SSD Caching in Virtualization Environments. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3427921.3450244

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450244

1 INTRODUCTION

While the computing world is beginning to see the arrival of technology [15, 26, 27] to fuse the gap between the competing requirements of persistence and low-latency, flash memory based devices have full-filled this role in the memory-storage hierarchy. Over the past couple of decades, flash based solid-state storage devices (SSDs) [19, 27, 40] have been ubiquitously used in embedded devices [28, 29], traditional desktops and server systems [9, 21, 33, 41]. With server systems, SSDs have been used on the disk IO path for caching [32] as well as primary storage devices [13, 18, 30]. The domain of this work is the use of SSD devices for disk IO caching in virtualization based hosting services.

The commonly used setups of SSD devices in virtualization environments are, (i) as a primary storage for virtual machine image files, (ii) as attached disks for individual virtual machines, and (iii) as hosting platform managed disk IO caches. The third setup employs a hosting platform (and hypervisor) managed cache that stores objects on near-host/on-host SSD devices. Since the cache is virtual machine agnostic, cache sizing and provisioning can be performed dynamically. Past work [17, 24], has used this setup and controlled storage access performance by implementing different policies for dynamic SSD cache provisioning. While each of these three setups have benefits along the spectrum of usability and performance, they do not provide the flexibility required to exploit all the potential benefits of an auxiliary SSD device in hosting environments. The first setup does not target the caching use case, the second setup allows custom usage of SSDs by VMs but as a fixed-size device. The third setup of using SSDs for host-side caching has been widely used, but suffers from the lack of knowledge about usage semantics of applications in the virtual machines. The hosting platform can correlate aggregate disk bandwidth requirements to adjust per-VM cache sizing [17, 34] This black-box approach cannot exploit the knowledge of usage and application/VM specific semantics, e.g., meta-data blocks to be always stored in cache (irrespective of their usage rate), or providing differentiated cache usage for applications nested within a hosted virtual machine.

The *ideal* properties of a host-side cache are to simultaneously allow for application-aware management of the cache and also allow for dynamic provisioning of the cache based on policy and performance control decisions. A feature similar to this is *memory ballooning* [39], which is used to manage the memory resource in hosted platforms. Memory managed via paging (page tables) can be *provisioned* dynamically by manipulating mappings in the page tables. A fundamental problem with a ballooning like usage model

with SSDs in virtualization environments is the missing support for *elastic* resizing of block IO devices. Traditionally, block devices are initialized and configured during machine boot-up and their properties remain static for the lifetime of the system[1]. Without the resizing feature, block IO devices cannot be effectively and dynamically managed by the hypervisor/hosting platform.

With hosting platform managed caches, a possibility is to setup an explicit channel between the hypervisor and virtual machines for providing *hints* to the hypervisor based device manager. This approach has at least two possible bottlenecks— one, the framework needs to have enough expressibility to accommodate different application-level semantics; and second, the application needs to know the state of the cache to generate corresponding hints for subsequent management. We believe that this breaches the equivalence requirement for virtualization platforms [31], in the sense that para-virtualized guest operating systems will depend on knowing and using hypervisor state as part of their execution.

Our premise is that the goal of an application aware management of content on SSD caches can be met efficiently with explicit support for *elastic* SSD caching devices. Towards engineering support for an elastic virtualized block IO device and dynamic management of a persistent disk cache across several virtual machines, we make the following contributions,

- Design and implement a mechanism for provisioning elastic and persistent virtualized SSD devices.
- Design and build an adaptive disk caching framework, *SymFlex* , which utilizes the explicit elasticity of SSD devices for symbiotic cache management along with hypervisor-level policy mechanisms.
- Showcase examples of use cases that benefit from the application aware semantics used for cache management from within guest OSes.
- Demonstrate via detailed empirical evaluation the low-overhead implementation and application-level benefits of *SymFlex* .

1.1 Need for application-aware SSD elasticity

An important aspect of infrastructure hosting services is that of resource over-commitment—provisioning more resources than that can be simultaneously allocated physically. Two vital mechanisms for over commitment are dynamic allocation of resources and a policy framework for informed choices about the magnitude of allocations and de-allocations.

The adaptation policies to dynamically change the allocation levels of resources can broadly be classified as *black* box and *gray* box policies. With black box policies used for dynamic SSD cache provisioning, the hypervisor is unaware of the cache semantics and its utility for the VM. The hypervisor employs VM-agnostic policies to manage the hypervisor based cache. Gray box policies depend on information and depend on actions from software entities from within the virtual machine to influence cache provisioning decisions. Figure 1 shows the hit ratio of a hypervisor managed SSD cache using representative black box and gray box techniques. The fio tool is used to generate the workload in both the cases which access a 500 GB HDD. In the first window, the 80% of the accesses are

[1]With plug-and-play setups devices can be setup on the fly, but their properties cannot be manipulated once configured.

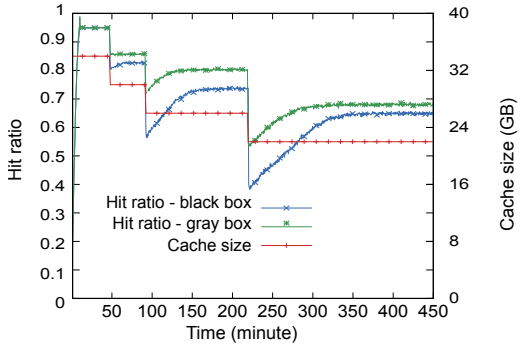


Figure 1: SSD cache efficiency with VM-agnostic and VM-managed techniques.

from the first 34GB of the device and the 20% from the remaining portion of the device. In the next window, the cache size reduced by 4GB and 80% of the accesses are from the second 34GB of the device and the 20% from the remaining portion of the device and so on. The black box technique employs an LRU cache replacement policy, while for the gray box technique, a hypothetical solutions assumes that the changing working set (of accessed disk blocks) is known to the hypervisor. As the SSD cache is resized (reduced in size) at different instances, the gray box technique can use the workload access pattern information for a better eviction policy of cached blocks and maintain higher cache efficiency levels.

As demonstrated above, the two main requirements for a high utilization caching store in hosting environments are, (i) a mechanism for dynamic resizing and (ii) usage of semantics of the software stack (the guest OS or the applications being executed in the VM) using the cache. Using hypervisor managed SSD caching for virtual machines, the main challenges of employing both the above mechanisms are, (i) lack of an elastic SSD virtualized device, and (ii) a framework for symbiotic management of the cache by the hypervisor and software entities of a virtual machine. Our work addresses both these challenges via *SymFlex*, an elastic and symbiotic IO caching framework for virtualization environments.

1.2 Related work

1.2.1 Hypervisor-based caching. With hypervisor/hosting platform based caching, the host OS or the hypervisor coordinate and manage the SSD cache. The cache itself can be organized as a *unified* cache [3] [20] across all virtual machines or applications, can be *statically* partitioned [10, 12, 16], or can be *dynamically* partitioned [8, 23].

With a unified SSD cache setup [3], IO requests from all VMs store and consume objects in a single cache and can result in performance interference side-effects based on cache usage distributions influenced by workload patterns. With statically partitioned SSD caching [10, 12, 16], the cache is partitioned based on performance or higher-level policy decisions (priority of VM etc.) considering a static set of VMs. Optimizations to the cache using this partitioning strategy include techniques to determine what resides in the cache

for maximizing the fixed size caches. Since the cache sizes are agnostic to changing load patterns and over commitment levels, they can lead to sub-par cache efficiency.

Dynamic cache partitioning techniques [8, 17, 22, 23, 35] partition the cache across virtual machines based on changing demands either from changing levels of over-commitment (addition/removal of new VMs to a host) or IO performance requirements. [22] introduced *Ratio of Effective Cache Space*, ratio of the cache size that is being effectively used to the total size of cache that has been allocated, to identify the cache demands of each virtual machines. [23] determines the cache size allocation based on run time analysis of IO access characteristics that captures both long-term locality behavior and transient locality spikes of the workloads. [8] identifies the set of blocks with high temporal locality to predict the cache requirements of different workloads and reduce the number of updates to the SSD cache. Each of these techniques adapts the size of the cache assigned to virtual machine endpoints to improve the cache utility. Other host side caching techniques [17, 35] provide control knobs to meet storage access QoS requirements. [35] aims to meet latency guarantees by periodically measuring the latency of IO requests of each virtual machine and using a feedback loop to adapt size of cache based on deviance from expected latency. Similarly, [17] also aims to minimize IO latency for a group of VMs and provide proportionate IO throughput levels based on exploiting the non-linear relation between cache sizing and miss rates.

1.2.2 Device pass-through with SSDs. A device pass-through SSD usage model enables the virtual machine to efficiently manage the allocated SSD resource from within a virtual machine. The virtual machine can use the device as a cache and implement custom policies to improve IO performance, or expose the device to applications (via an API library) to explicitly implement application-specific semantics. [1, 2, 5] use the SSD as a generic block IO cache device via a device mapper [2] setup to cache blocks of high latency hard disks and network storage devices. [4, 11, 36, 37] are examples of approaches that expose a SSD device directly to applications. These approaches are integrated with applications such as key-value stores, databases, and use the SSD as an auxiliary persistent cache to exploit the semantics of the application and its usage. [11] and [4] are flash-based key-value stores which extend the *memcached* tool to manage key-value pairs cached in SSDs. Typically, flash-based key value stores organize the key-value pairs in slabs of different sizes, and uses in-memory data structures for fast lookup and mapping operations. These applications are agnostic to the unique disk-level properties of flash storage devices and approaches in [36, 37] maximize the efficiency of key-value system by removing redundant mappings, double garbage collections, etc. in the in-memory and SSD stores.

1.2.3 Symbiotic management of SSD caches. DoubleDecker [25], a hypervisor caching framework enables multi-level provisioning of hypervisor managed in-memory and SSD caches in derivative cloud setups. This solution is integrated with the file system based `cleancache` framework and can enable differentiated provisioning of the caches for multiple entities (containers) within a nested hosting framework. While this approach is closest to symbiotic management of the SSD cache, it does not incorporate usage of application semantics for custom SSD management policies. Further, it relies on the cleancache interface and on updates to the file system operations for implementing the caching solution. *SymFlex* does not rely on file system extensions in guest operating systems and also enables custom management of caches from within virtual machines.

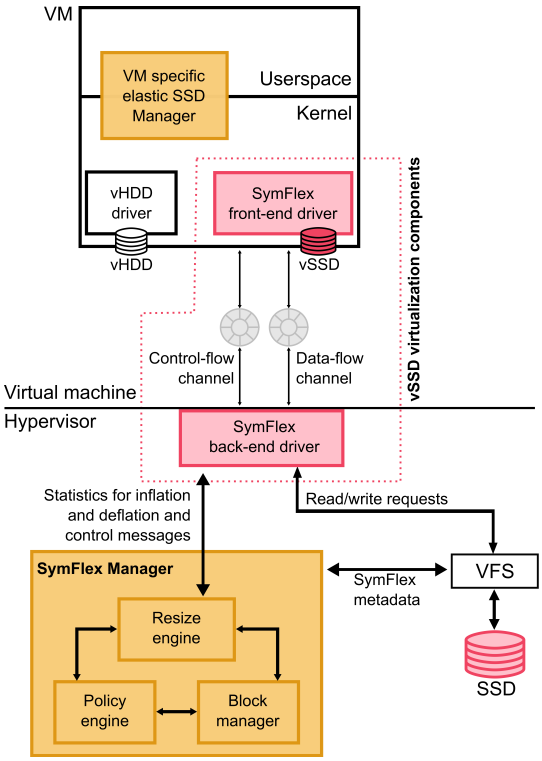


Figure 2: *SymFlex* architecture and components.

While several techniques to manage SSD caches have been discussed above, none of them exploit the caching potential of SSDs to the fullest—they either do not have access to all semantics related to application requirements, or are statically partitioned. Our domain of operation is to operate a symbiotic framework where both the hypervisor and software entities of a virtual machine cooperate to improve effective of the SSD cache. *SymFlex* also lets itself seamlessly to accommodate nested hosting setups, where custom cache management policies can be implemented from with each VM instance. Further, the flexibility provided by the *SymFlex* framework can compliment existing cache provisioning heuristics with added adaptations based on application/guest OS cache utility semantics and policies.

2 *SYMFLEX* ARCHITECTURE AND DESIGN

SymFlex provides a framework to virtualize SSD devices with dynamic resizing, and symbiotic management of the SSD-backed caches by hypervisor and VM-specific management decisions. The hypervisor is tasked with determining the size of the virtualized elastic SSD (referred to as vSSD), and the guest OS or application running in the virtual machines are tasked with managing the vSSD for caching purposes.

The architecture of *SymFlex* is as shown in Figure 2. The architecture contains three main components, (i) the virtualized elastic SSD device (vSSD), (ii) the hypervisor-based *SymFlex* manager, and (iii) the VM-specific elastic SSD manager.

The first component is involved with setting up the special device for virtual machines. The design for which includes managing run-time sizing, and setup up of control and data channels for communication between guest OS (and/or application) and the hypervisor. The hypervisor-based SSD manager is responsible for provisioning of the SSD resource across multiple virtual machines. Policies for dynamic SSD sizing and allocation to virtual machines are core parts of this component. The SSD resize decisions are handled locally by a VM-specific cache manager, which incorporates custom application-aware and guest-OS specific policies for improving utilization of the SSD-based cache.

The main operations in the *SymFlex* life cycle are, (i) registration of a virtual machine with the *SymFlex* manager, (ii) performing read/write operations on the virtualized elastic SSD device, and (iii) using the on-demand resize mechanism of the vSSD for cooperative cache management by the guest OS entities and the hypervisor.

Next, we describe the three units of *SymFlex* architecture in detail and the end-to-end flow of different operations in the *SymFlex* life cycle.

2.1 Elastic SSD Virtualization

The engineering component of *SymFlex* is the virtualization unit, which implements the virtualized elastic SSD device (vSSD). The important tasks include setting up the elastic SSD device itself for the virtual machine and the necessary virtualization support via control and data communication channels. The vSSD device virtualization includes a device specification, a front end driver in the virtual machine, and a *SymFlex* back end driver in the host (as shown in Figure 2). The two drivers coordinate configuration, access and resizing of the vSSD device via a control and a data channel. The front end driver initalizes a (virtual) block IO device (**vSSD**) which is used as a caching store by the guest operating system and/or the applications running in the virtual machine.

The vSSD device is similar to a para-virtualized `virtio-blk` device, with support for on-demand resizing of its capacity. The front end driver receives read/write requests to the vSSD device, and in turn forwards them to the back end driver. A *bitmap* is maintained at the front end to store a status bit for each (logical) block of the vSSD. Resize operations leading to change in the capacity of the vSSD set/reset bits corresponding to logical blocks of the device. The bitmap is (re)initialized on every virtual machine restart, updated on resize operations, and always consistent with the logical to physical block mapping maintained by the back end driver as part of the device virtualization. The bitmap is used to quickly verify validity of a logical block (i.e., whether mapping to physical block exists at the back end) during read/write operations originating at the front end driver. The back end driver performs the actual read/write operations over the physical SSD device, and sends appropriate responses back to the front end driver. The back end driver communicates with all the components of *SymFlex* , with the front end driver via control and data plane channels, with the manager to exchange statistics and management request/response commands and with the host file system layer to issue read/write operations to the physical device.

Details of read/write operations handling and resize operations are discussed in Sections 2.4.2 and 2.4.3, respectively.

2.2 *SymFlex* manager

The *SymFlex* manager is responsible for managing (dynamically allocate and deallocate) the SSD resource (blocks) across several virtual machines hosted on a physical machine. A *resize engine*, a *policy engine* and a *block manager* constitute the three components of the *SymFlex* manager.

The *block manager* keeps track of the physical blocks of the SSD and current allocation and usage status of each block. For *SymFlex* , a block is a sequence of contiguous logical sectors, e.g., the default block size used in our implementation is 1 MB (2048 sectors of 512 bytes each). The per block metadata managed by the block manager is divided into two categories—global state and per-VM state, and stored on the first few blocks of the SSD. The global metadata state consists of the following information, number of physical blocks available on the SSD, a list of free and allocated physical blocks, and a list of active and inactive virtual machines.

Every block can either be free, or a *SymFlex* metadata block, or a block allocated to a virtual machine. For an allocated block, *SymFlex* stores the virtual machine identifier and the logical block number (of the vSSD) to which a block is mapped. The per-VM state contains a VM identifier, maximum size of the vSSD device, current allocation (in terms of number of blocks) to the VM, and configuration flags. A particular flag of interest is the `persist` flag which indicates whether the contents of the cache should be persisted and restored across virtual machine restart.

The resize engine performs the increase and decrease operations to dynamically resize the vSSD device. The *SymFlex* policy engine implements the policies for the extent of allocation and deallocation on a per-VM basis on events of interest, e.g., change in IO performance levels, exit or start of new virtual machine etc. Currently, *SymFlex* supports a weighted-share based allocation policy and a proportionate throughput based allocation policy to distribute blocks across vSSD devices of different virtual machines.

On a vSSD resize decision, the resize engine coordinates with the *SymFlex* back end driver for resizing the vSSD. The back end driver communicates with the corresponding front end to complete the operation. On a size reduction (**deflation**) request, a set of logical block numbers are conveyed to the back end to be returned to *SymFlex* manager. The block manager changes the status of the corresponding physical blocks to *free*, and releases corresponding entries from the per-VM allocated block list. On a size increase of the vSSD (**inflation**), a size change request is forwarded to the front end via the back end driver. The front end requests for a list of new blocks, which the back end obtains from the *SymFlex* manager and forwards the logical block numbers to the front end after updating the logical blocks and physical blocks mapping information. The *SymFlex* front end driver updates the status of the corresponding bits in the bitmap to maintain validity status of the logical blocks of the vSSD during resize operations.

2.3 VM-specific vSSD cache manager

A vSSD cache manager, local to each virtual machine, is responsible for managing the vSSD based caching store. Since the size of the caching device changes dynamically, the manager has to implement a mechanism and a policy to influence the contents of the cache as as to adhere to its policy. The *SymFlex* framework currently support two mechanisms to manage cache contents, one via the block IO layer and the other where applications directly the SSD as an auxiliary device.

The vSSD cache manager implements the VM-specific cache management policy to handle cache size changes, notably to make object eviction decisions on the cache. Further, cache size increase events also need to interface with the local manager to correctly reflect the increased vSSD capacity. Section 4 presents the different components of the VM-specific cache manager.

2.4 *SymFlex* life cycle

The *SymFlex* life cycle consists of several operations like starting the *SymFlex* manager, registering virtual machines with the *SymFlex* , reading/writing data from/to the vSSD device, resize operations invoked by the policy engine, and recovery of inactive virtual machines. When the *SymFlex* manager runs for the first time, the free list is populated with all the physical blocks. The first few physical blocks in the SSD are used to persist *SymFlex* metadata and manager reserves these blocks by moving them from the free list to a metadata blocks list.

2.4.1 Registration of virtual machines with the SymFlex manager. Each virtual machine needs to be registered with manager when the virtual machine is created. The back end driver initiates the registration process by sending the registration message to the block manager. The registration message contains VM-ID, name of the VM, and the three tuple `<M, C, P>`, where `M` denotes the size of the virtualized vSSD device, `C` is the number of blocks to be allocated when the VM starts, and `persist` is a binary flag to specify if allocated blocks of the VM are to be persisted on a VM shut down (to be reused on VM restart).

On receiving a registration request, the block manager creates a per-vm block list and allocates the blocks to the new virtual machine. Depending on the policy, the block manager may invoke the policy engine if the free list is not enough to satisfy the number of blocks requested at the time of registration. After, the block manager reserves the blocks for the new VM adds the blocks from the free list to the per-vm block list, and updates the per physical-block entry with the corresponding VM-ID, and replies to the back end driver with the list of physical blocks allocated. The back end driver updates the logical to physical block mapping table entry with the physical block number, starting from the logical block number `0` to `N-1` where `N` is the number blocks allocated by the block manager. Next, the back end driver forwards the list of mapping table entries corresponding to the newly allocated block to the manager, and the manager updates the per physical-block entry with corresponding logical block numbers. During initialization, the front end driver reads the list of valid logical block numbers(`lbn`) from the back end driver and initializes the block status bitmap. The block status bitmap maintained by the front end driver has to be reinitialized after each VM restart.

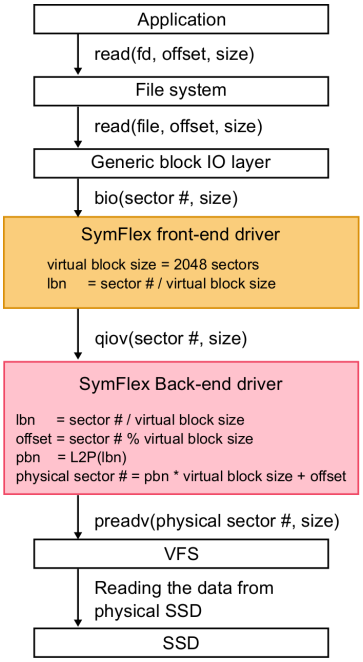


Figure 3: Logical to physical block translation during read/write operations.

When a virtual machine restarts (which had a persist flag for the vSSD device) and re-registers with *SymFlex* , the block-manager replies with list of `<pbn, lbn>` tuples for previously allocated blocks to the back end driver.

2.4.2 vSSD read/write operations. When a virtual machine is started with the virtualized elastic SSD device, a special block IO device, `vssda`, is available for the VM. Like any other block IO device, any application can perform read/write operations on the device. Figure 3 illustrates the flow of requests of the read/write operations, and logical to physical block translation. The front end driver receives all read/write requests, and a single read/write request can contain multiple vSSD blocks. If a physical block is not allocated to one of the logical blocks in the request, the request is considered as an invalid request. For an invalid request, the front end driver immediately returns the error to the higher layer. For a valid request, the front end driver forwards the request to back end driver. The back end driver divides read/write requests into sub requests if the request crosses the vSSD block boundary. For instance, a request to read 2 MB of data from an offset of 256 KB will be divided into 3 sub requests, (i) sub request to read 768 KB from 256 KB, (ii) a sub request to read 1 MB from 1 MB offset, (iii) and sub request to read 256 KB from 2 MB offset. After, the back end driver identifies the physical vSSD block number of each sub requests using the logical to physical block mappings that it maintains, it determines the physical sector numbers correspond to each sub requests, and issues the read/write requests to the physical SSD. When all sub requests are finished the responses are sent to the back end driver, which merges the sub responses into a single response and forwards to front end driver.

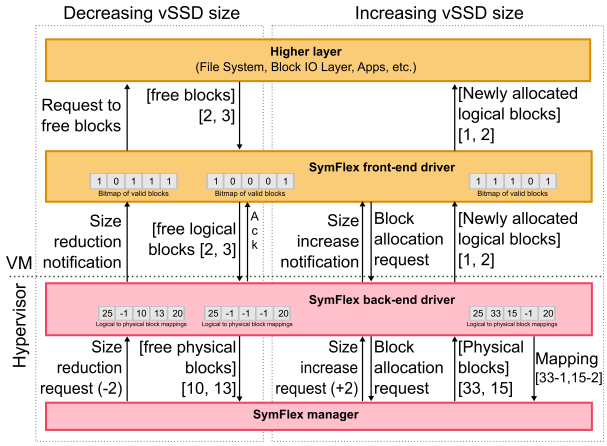


Figure 4: Messages involved in the vSSD resize operations of *SymFlex*.

2.4.3 vSSD resize operations. The manager initiates all resize operations by sending a resize request to the back end driver. A resize request contains VM-I D, the number of blocks to be inflated/deflated, and type of the resize operation. During the vSSD size reduction operation, the manager deallocates the vSSD blocks from a virtual machine, and hence the effective size of vSSD reduces. During the size increase operation, the manager allocates addditional vSSD blocks to the virtual machine. On receiving the resize requests from the manager, the back end driver immediately forwards the resize request to the front end driver.

Sequence of operations performed during a size increase operation are different from the size reduction operation. When the front end driver receives the size increase operation request, it sends an *block-allocation-request* to the back end driver, and the back end driver forwards this request to the *SymFlex* manager. The manager selects the requested number of physical vSSD blocks from the free list, allocates the selected blocks to the VM, and replies to the back end driver with a list of allocated physical block numbers. The back end driver maps these physical blocks to logical blocks (which were previously invalidate) and forwards the list of the new valid logical blocks to the front end driver. The front end driver updates the corresponding bits of the block bitmap by setting the bits corresponding to the new valid logical blocks. In addition to sending the list of logical blocks to the front end driver, the back end driver also sends the same list to the manager. The manager updates the per physical block entry with logical block number and the VM ID of the newly allocated blocks.

During a vSSD size reduction operation, the front-driver requests the cache-management unit of the VM to release blocks. The VM/application specific cache management unit executes its own policy to decide the blocks to be freed, and forwards the list of selected blocks to the front end driver. The front end driver resets the corresponding bits in the bitmap, and forwards the list of selected logical block numbers to the back end driver. The back end driver invalidates the logical-to-physical block mapping for the corresponding logical blocks, and forwards the list of freed physical blocks to the *SymFlex* manager. The manager moves those blocks from the VM-specific list to the free list of physical blocks.

2.4.4 Recovery of blocks. To provide over-commitment of the vSSD device, the *SymFlex* manager may opt to swap blocks allocated to a stopped virtual machine to secondary storage. The manager periodically detects stopped virtual machines and recovers physical blocks in the following manner:

- De-allocation of non-persistent blocks: The non-persistent blocks allocated to a VM are moved to the free list of physical blocks.
- Swapping of persistent blocks: Physical blocks associated with blocks with persist flag set are copied to a secondary storage, along with persisting their physical to logical block mappings. Subsequently, these physical blocks are moved to the free list.

When a virtual machine restarts and re-registers with the *SymFlex* , the manager allocates physical blocks from the SSD, copies contents of the persistent blocks to the allocated blocks, and updates the physical to logical block mappings, and coordinates with the back end driver to update its logical to physical block mappings.

3 IMPLEMENTATION DETAILS

We have implemented *SymFlex* using a Linux + QEMU-KVM platform. The *SymFlex* front end driver is implemented with Linux kernel version of 4.9.35 and the back end driver as extensions to QEMU 2.9.0. The *SymFlex* manager is implemented as a multi-threaded application which communicates with the back end driver using shared memory. The following subsections discuss implementation details of each of these components.

3.1 The elastic vSSD device and its drivers

We modified the Linux kernel to add a new block IO device, `virt_vssd`, which provides the elastic sizing feature. Corresponding *SymFlex* front end and back end drivers are implemented virtio_vsdd_driver's for the host and the guest Linux kernel. Implementation of the `virt_vssd` device is similar to existing `virt_blkio` device, but with an additional virtio ring and corresponding call back functions for *SymFlex* specific communication between the front-driver and back end drivers. The read/write data operations on the device are performed using the data-flow ring/channel and *SymFlex* specific control messages are conveyed through control-flow channel (refer to Figure 2). To send a read/write block IO request or a *SymFlex* control message, the *SymFlex* front end driver formats the request as virtio queue element and adds the element to the queue attached the corresponding ring. For each queue, a call back function is registered at both ends of the IO ring. Pushing an element to a `virtio_vssd` queue from the front end invokes the call back function registered at the back end. The back end driver dequeues the element from the corresponding queue, converts it to a vSSD IO request, and process the request/message based on the type of the request. The back end driver initiates a similar sequence in reverse to communicate with the front end driver.

3.1.1 SymFlex read/write operations. Semantics and functionalities of the IO rings used for read/write operations, and the data flow channel, are the same as the rings used in the existing `virtio_block`

technique. With *SymFlex*, the front end driver first checks whether the logical block number exists in the vSSD device (as the device size can change dynamically) before forwarding the request to the back end. If the logical block does not have a valid mapping, i.e., the corresponding bit in the block status bitmap is unset, the front end driver returns an error (to be handled by the block IO layer or higher layers). For a valid read/write request, the request is converted to a virtio queue element and pushed to the data flow queue (IO for data request and response). The back end driver on retrieving the request, determines the type of operation to be performed, the logical sector number, the number of bytes to read/write and a list of IO vectors. Each IO vector contains information regarding the starting memory address and the size of memory region for the corresponding block IO operation. Further, the back end driver maps the logical sector number of the vSSD device to a logical block (based on number of logical sectors per block), then maps the logical block to a physical block and offsets it with the sector number, to obtain the physical sector for the IO operation.

Read/write requests may span over multiple logical blocks which may not be contiguous in the physical range, and such requests are issued as multiple block IO requests by the back end driver. The back end determines the starting physical sector number for each of the sub-requests and also maintains a count and status of issued sub-request. On completion of a sub-request, its status is updated to reflect completion. On completion of all the sub-requests, a response (along with data in case of a read operation) is sent to the front end driver via the data flow channel.

As an optimization, QEMU dequeues multiple read/write requests at a time from a virtio ring, sorts the requests based on the starting sector number, merges the sequential requests of same type (i.e., it merges two or more continuous read requests into a single request). Requests are merged by copying the IO vectors to a single request and updating size of the request. With the vSSD this optimization operates on logical block numbers, and may lead to fragmented block IO requests on the physical device. To counter this situation and to amortize the cost of SSD access for caching, *SymFlex* operates with large physical block sizes (e.g., vSSD blocks are of size 1 MB in our prototype implementation).

3.2 *SymFlex* Manager

The *SymFlex* Manager is implemented as a multi-threaded user space application and has three main threads, (i) a listener thread to communicate with the *SymFlex* back end drivers of virtual machines, (ii) a thread to receive registration and configuration inputs from the user, and (iii) a recovery thread to recover blocks allocated to inactive virtual machines. In addition, the listener thread creates a separate thread for processing each message it receives from the back end drivers. The recovery thread periodically identifies new inactive VMs, and recovers the blocks of those VMs. The communication channel between the *SymFlex* manager and the back end drivers is a shared memory region which is setup by the manager when it starts. The shared region is used for transferring the list of blocks during inflation, deflation and initial registration of the vSSD device. The list of blocks are exchanged in an iterative manner, with number of entities dependent on the size of the shared region. The *SymFlex* manager and the back end drivers synchronize using a set of condition variables to signal events for processing by each entity. The shared region has a small header that encodes the type of operation (inflate, deflate etc.) and related data for processing.

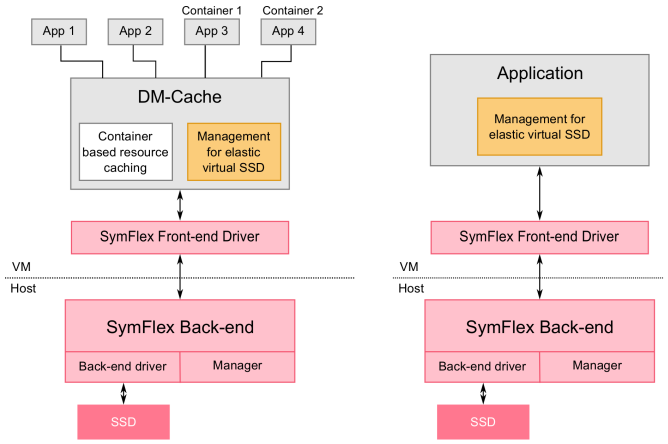


Figure 5: Block layer IO caching and application managed caching use cases of *SymFlex*.

4 USE CASES

To demonstrate the efficacy of *SymFlex*, we modified two existing applications to operate with an elastic SSD device. Figure 5 illustrates the integration of *SymFlex* and usage of the vSSD device for the two use cases.

4.1 *dm-cache* with *SymFlex*

dm-cache[2] is a cache to store disk blocks and is implemented via the device mapper setup. The device mapper framework requires three devices to create a dm-cache, an origin device (with large storage capacity, possibly slower and/or on the network), a cache device (closer to the disk accesses), and a small metadata device (used to persist the dm-cache metadata information). With *SymFlex*, we employ the elastic vSSD device as the cache device. We configured the granularity of read/write operations of the dm-cache and the vSSD block size to 1 MB. A *SymFlex* enhanced dm-cache first registers with the vSSD *SymFlex* front end driver, and as a part of the registration process the front end driver forwards to dm-cache a bitmap representing the valid logical blocks of the vSSD device. dm-cache maintains a list of free blocks, clean blocks, and dirty blocks. As part of *SymFlex* extensions, we modified dm-cache to maintain an additional list, the *invalid* block list. The blocks that we relinquish as part of run-time vSSD size reduction are maintained in this list. While servicing a cache miss, the *SymFlex* dm-cache selects a block from the free list and copies data from the origin disk. Once the list is empty, blocks are evicted either from the clean list or the dirty list based on different eviction policies.

During the vSSD size reduction process, the front end driver forwards the size reduction request to the *SymFlex* dm-cache. If the free list has enough blocks to satisfy the reduction operation, dm-cache selects blocks from the free list, and moves those blocks into the invalid list. If the free list does not have enough blocks, blocks

either from the clean list or dirty list are evicted (based on a pre-configured dm-cache policy) and moved to the invalid list. Finally, the block numbers (moved to the invalid list of dm-cache) are sent to *SymFlex* front end driver (corresponding bits in the bitmap are reset to mark invalidity) and forwarded the *SymFlex* manager via the *SymFlex* back end driver.

Similarly, during the vSSD size increase operation, the front end driver sends dm-cache a list of new valid logical blocks and dm-cache moves the corresponding cache blocks from the invalid list to the free list. We use the default dm-cache stochastic multi-queue [2] policy for eviction of cached objects. Further, the *SymFlex* enabled dm-cache can also provide differentiated provisioning of the vSSD cache across applications. To provide this functionality, we leverage the `cgroups` subsystem of Linux. Each application is assumed to belong to a separate cgroup and a new cgroup variable is added to the subsystem in order to specify weights for proportionate sharing of the vSSD cache. To differentiate and identify IO traffic across applications, we modified the VFS layer to attach cgroup information of the end point process as part of the `bio` request. These cgroup tags are used by the *SymFlex* dm-cache to manage cache provisioning across multiple (nested) applications.

4.2 Fatcache with *SymFlex*

Fatcache [4], a key-value store (similar to *memcached* [6]), is our second use case which uses an vSSD device to extend its capacity. Fatcache uses a slab-based management strategy for managing the available SSD storage. Each slab (a contiguous region in memory or on the SSD) is divided into equal sized slots, with different slot sizes are part of different slab classes. A value is stored in a slot that is closest to size of the value. Each value is first in the memory and then drained to the SSD. The metadata for resolving keys to the slabs is stored in memory. Updates to the in-memory slabs are drained in batches to the SSD device.

As part of *SymFlex* extensions to Fatcache, we configure the slab size to be the same as the vSSD block size. Similar to *SymFlex* dm-cache, *SymFlex* Fatcache registers itself with *SymFlex* front end driver when the application starts, and the front end driver returns a bitmap corresponding to the valid logical blocks. After registration, *SymFlex* Fatcache moves the the valid vSSD slabs (slabs with valid blocks) to a free-ssd-slab list, and those corresponding to invalid blocks to a invalid-ssd-slab list (this list is part of Symflex related changes). During reduction in the vSSD size, Fatcache selects slabs from the free-ssd-slab list to evict, and if these slabs are not enough to satisfy the deflation operation, slabs from the fully-filled-ssd-slab list (list of densely packed slabs) are freed up. The selected slabs are moved to invalid-ssd-slab list. As part of vSSD increase, the *SymFlex* front end driver forwards to Fatcache a list of new valid logical blocks, and and the corresponding vSSD slabs are moved from the invalid-ssd-slab list to free-ssd-slab list.

5 EVALUATION

5.1 vSSD IO performance benchmarking

Towards benchmarking vSSD characteristics, we first compared the raw IO performance of the elastic vSSD device with *SymFlex* virtio drivers and existing block IO virtualization techniques. The following experiments were carried out on a machine with Intel i7-4790K processor and 16GB memory. We configured a virtual machine with 2 GB RAM, 2 VCPUs, a vSSD of capacity 20 GB, and used the fio benchmarking tool to issue file IO requests from the virtual machine. A single fio thread issued read and write requests (of size 2MB) to files on the vSSD device, for a cumulative read/write of 20GB worth of data. Table 1 reports the performance of different virtualization techniques including 2 vSSD configurations and 2 existing *virtio_blk* configurations. [14]. In the first virtio_blk configuration (virtio_blk[raw]), the SSD is directly exposed to the virtual machine as raw device. With virtio_blk[ext2+qcow2], the SSD is formatted with the ext4 file system, a virtual disk image with the qcow2 format is created, and the image is exposed to the virtual machine. As can been seen from the table, the average reported fio throughput achieved by the vSSD configurations is always within 4% of the best performing virtio_blk configuration. In fact, it was slightly better as compared to the virtio variant which used qcow disk. The performance of vSSD with block sizes of 512 KB was slightly less than with that 1 MB blocks, and were was expected as the fio read-write granularity was 2 MB in size. The throughput similarities hold across different file level read granularity issued by `fio`.

read: write ratio	Throughput (MBps)			
	vSSD (1MB)	vSSD (512KB)	virtio (raw)	virtio (qcow2+ext4)
1:0	518.45	518.45	529.31	517.62
1:1	445.24	403.14	462.15	443.10
0:1	331.48	318.45	342.84	332.27

Table 1: Performance comparison of different vSSD and virtio-blk configurations.

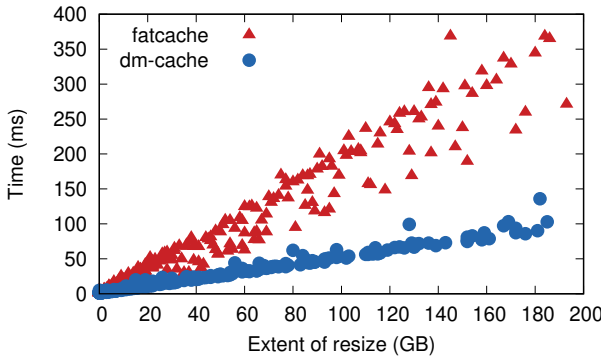


Figure 6: vSSD resize latency characteristics.

We use this result to conclude that our vSSD block device virtualization implementation has performance comparable to the current block virtualization solutions.

5.1.1 Resize latency characteristics. Next, we characterized latency of the vSSD size increase and decrease operations of *SymFlex*. The setup was a virtual machine running one of the use-cases, dm-cache or Fatcache, and a series of vSSD size increase and decrease requests were issued. The corresponding *SymFlex* block management layers,

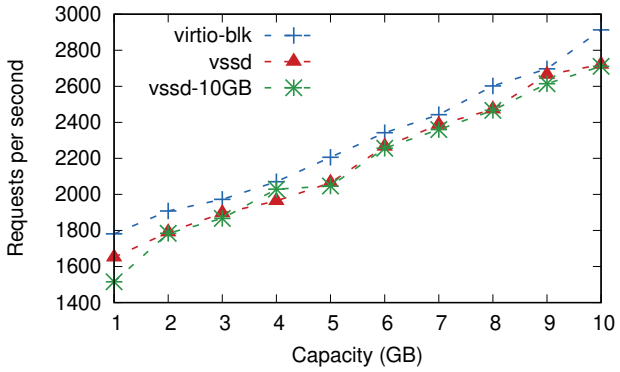


Figure 7: Throughput of set operations off the Fatcache application.

one at the block IO layer and the other at application layer coordinated with the *SymFlex* front end driver for resizing the vSSD device.

We performed experiments on both dm-cache and Fatcache setups by varying the granularity of resize. For each of the experiment, we measure the total time taken to finish a resize operation, the time spent in the front end, and the time spent in the back end. One of the time consuming operations during resize is persisting metadata. As an optimization, we persist metadata periodically, not at the end of each resize operation. Figure 6 shows the latency for vSSD size reductions for dm-cache and the Fatcache setups. The latency with Fatcache is 2-3 times more than that of dm-cache, as each resize request has to cross the user-kernel boundary in the VM and be serviced in the user space. For both increase and decrease in vSSD size, the time spent in the backend driver is less than 10% of the duration for resize. Further, we found that the front end driver consumes most of the time as part of vSSD reduction to identify blocks to free and coordinate communication with the back-end driver.

5.1.2 Performance of SymFlex Fatcache. To analyze the impact of validation of the requests and logical to physical address translation in the read/write path on application throughput we have performed following experiments. The main aim of these experiments is to determine performance impact on Fatcache and dm-cache when they are configured with a fixed size of SSD as compared to they are inflated or deflated to that particular size. Figure 7, illustrates the performance of set operations of Fatcache running in three different configurations. *virtio-blk* represents Fatcache running over existing virtio-blk device, and *vssd* represents Fatcache running with a vSSD device. *vssd-10GB* represents Fatcache using a vSSD device of size 10GB, with 10 GB being achieved by a series of resize operations. *mcperf*, a key value store load generator, is used to generate load to Fatcache. mcperf opens 100 connects to Fatcache server and each connection sends 1000 set operation requests of size 100KB to the server. The figure plots performance of Fatcache for set operations in terms of requests processed per second against the capacity of SSD allocated for the fatcache. A performance degradation of 4% to 5% is observed in *vssd* as compared *virtio-blk*, and it is due the additional operations performed by *vssd* in the read/write path. In addition, a 0.5% to 1% of performance degradation is observed when the 10GB vssd is deflated to a size as compared to a pre-configured vSSD of the same size.

We performed another experiment to identify the overhead of vssd configurations on the performance of fatcache get operations. Get requests are generated by `mcperf` which is configured to create 100 connections with fatcache server and each connection is sending 1000 get requests, which are meant to read the values stored as a part of set operation experiment explained above. We observed a performance overhead of 2% to 3% in the get operations as compared to `virtio-blk` configuration.

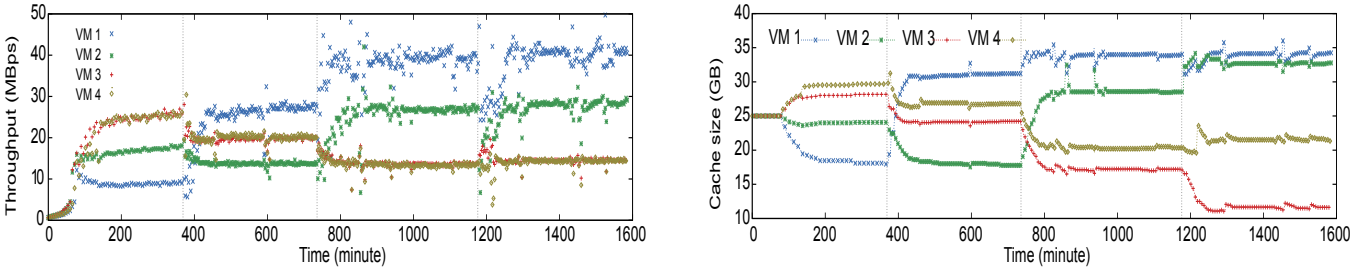
5.2 Utility of vSSD elasticity

To demonstrate the utility of elastic vSSD caches, we implemented two cache provisioning policies in *SymFlex*—weighted fair share based provisioning and proportionate IO throughput based provisioning. Here we describe an experiment using the IO throughput based provisioning approach.

The aim of this experiment was to demonstrate the efficacy of *SymFlex* of SLA based provisioning of the SSD resource across virtual machines in an adaptive manner. Figure 8a illustrates a proportionate throughput cache provisioning solution using *SymFlex*and figure 8b illustrates the cache size across different VMs during the experiment. *SymFlex* is configured to allocate a maximum of 100 GB of SSD space across four virtual machines. The experiment consists of four windows, and they start at 0m, 370m, 735m, and 1175m. The VMs are configured to operate with vSSD IO throughput in the ratio of 1:2:3:3. The *fio* tool was running in each virtual machine, and it reads the entire dm-cache origin device where the 80% of the accesses are from the first 40GB of the device and the 20% from the remaining portion of the device. Initially, an equal amount of vSSD is allocated across all VMs. When all the caches are warmed up, the policy engine starts enforcing the configured ratios of IO throughput by resizing the vSSD associated with each virtual machine. In the second window, we changed the ratio of VMs to 4:2:3:3 and hence the throughput of the VM1 is increased, and throughput of remaining VMs are reduced. Similarly, we changed the ratio to 4:2:1:1 in the third window. In the fourth window, we changed the workload of VM3 with less cache requirements (access pattern is changed as 80% of the accesses from the first 25GB of the device) and that of VM2 with high cache requirements (first 80% of the accesses are from first 50GB of the device). This resulted to allocate less blocks to VM3 and more blocks to VM2 to maintain the same ratio on throughput values.

5.3 Utility of vSSD persistence

To demonstrate the utility of persistent caching support of *SymFlex*, we setup a machine configured with *SymFlex*-enabled dm-cache with the persistent block flag set. dm-cache was configured with a 10GB vSSD and 30GB of origin disk. We used the flexible IO benchmarking tool *fio* to generate load to stress dm-cache. The fio application perform non-sequential IO operations worth of 25GB on the dm-cache device where the first 75% of the accesses are from first 30% of the device and the 25% from the remaining portion of the device. The system is forced to restart at different time instances, and the dm-cache hit ratio, reported at every 10 seconds, during



(a) Throughput achieved over time.

(b) Cache size variations with time.

Figure 8: Proportionate throughput based vSSD cache provisioning.

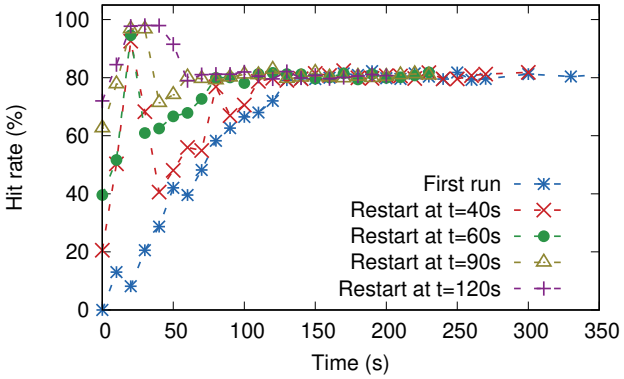


Figure 9: Impact of warmed up caches via persistent vSSD caching.

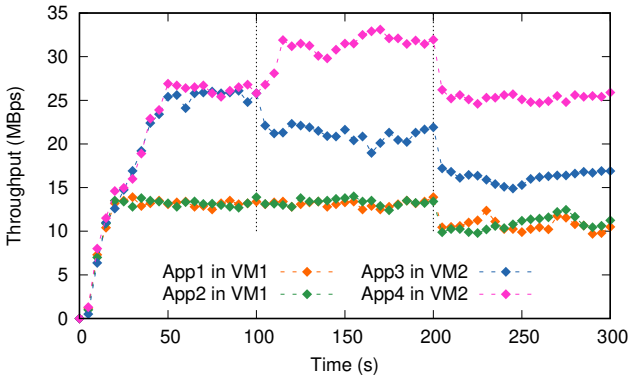


Figure 10: Dynamic cache provisioning for applications nested in VMs with *SymFlex* and dm-cache using the vSSD.

each run is which as shown in Figure 9. During the first run, fio took 360 seconds to complete the operation and the hit ratio saturated at 80% after 130s.

During the second round, the virtual machine is forced to restart at 40 seconds, and fio generates the same workload load, i.e., same request sequence starting the beginning. On restart, the persistent vSSD cache was restored and the dm-cache already had the blocks before the virtual machine get restarted at 40 seconds. Hence, dm-cache benefits from a warm cache after the restarting —the hit rate saturates earlier, at 110 second, and fio completes its IO operations in 300 seconds (compared to 360 seconds with a cold cache). On subsequent rounds, the VM was restarted after progressively increasing the duration before restart and replaying the original IO sequence. As expected, the benefits of a persistent warm cache are larger and can very quickly hit peak performance.

5.4 Differentiated vSSD caching for nested applications

Figure 10 shows the usecase of *SymFlex* used to provision the vSSD cache across applications running inside VMs. Like in the previous experiments, *SymFlex* manager is configured to allocate maximum of 100GB SSD across virtual machines. Initially, two virtual machines, with each running two instances of the filebench file system benchmarking tool, share the vSSD in the ratio of 1:2. The 4 applications are labelled App1 and App2 (executing in VM1) and App3 and App4 (executing in VM2) and each of them belonged to a separate cgroup, which was passed on to the *SymFlex* dm-cache block manager. All the four applications create 50 threads and are configured to run file-server workload with 100000 files with mean directory width of 50 and mean file size of 13840 bytes. For the first 100 seconds the ratio of the vSSD cache within each VM is set to 1:1 for both the applications. As is shown in Figure 10, the IO throughput values for applications in the same VM are identical, and are approximately twice for applications in the second VM. After 100 seconds, the ratio of the vSSD for applications in VM2 is changed to 2:3 and after 200 seconds vSSD sizes available to both VMs was reduced by 20 GB. In both these cases, the IO throughput of the applications behaved in the same ratios, with the overall ratios across VMs still adhering to the 1:2 configuration.

6 CONCLUSION

SymFlex targets to virtualize persistent storage devices and symbiotically manage them with cooperation of the hypervisor and the software entities of the virtual machine. We defined a new type of elastic virtual SSD (vSSD) for VMs by extending the existing `virtio-blk` virtualization design. We showed that the virtualization overheads of *SymFlex* result in IO performance to be within

5% of conventional virtio based device virtualization. We also implemented frameworks and interfaces to symbiotically manage the vSSD at the block IO layer and directly by applications. The features related to dynamic proportionate sharing of the cache were demonstrated via extensions to the existing dm-cache and Fatcache applications.

Currently, *SymFlex* only supports host local SSDs. An immediate extension is to provide support for network-attached SSDs, which will increase applicability of *SymFlex* for other hosting features like VM migration and stop-and-copy. Further, *SymFlex* supports a single physical SSD for virtualization and also supports a single vSSD device per VM. We intend to generalize our approach to accommodate and manage multiple SSD devices to implement a software-defined hypervisor managed caching layer. *SymFlex* can also be integrated with virtual machine specific storage systems like VMFS Datastores [7, 38] that manage blocks over a cluster of machines and target optimizations specific to virtual machines — de-duplication, sharing, snapshots etc.

REFERENCES

[1] bcache. https://bcache.evilpiepirate.org.
[2] device-mapper. https://www.kernel.org/doc/Documentation/device-mapper.
[3] Emc corporation. 2012. emc vfcache. https://www.emc.com.
[4] Fatcache. https://github.com/twitter/fatcache.
[5] lvm-cache. http://man7.org/linux/man-pages/man7/lvmcache.7.html.
[6] memcached. https://github.com/memcached/memcached.
[7] VMFS Datastores. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.storage.doc/GUID-5EE84941-366D-4D37-8B7B-767D08928888.html.
[8] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.
[9] A. Badam and V. S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. pages 16–16, 03 2011.
[10] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.
[11] A. Gartrell. MCDipper: A key value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/101513470904239203.
[12] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.
[13] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. Dfs: A file system for virtualized flash storage. *ACM Trans. Storage*, 6(3).
[14] G. Joshi, S. T. Shingade, and M. R. Shirole. Empirical study of virtual disks performance with kvm on das. In *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*, 2014.
[15] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam. Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011.
[16] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.
[17] R. Koller, A. J. Mashtizadeh, and R. Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, 2015.
[18] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.
[19] J. Lee, J. Jang, J. Lim, Y. G. Shin, K. Lee, and E. Jung. A new ruler on the storage market: 3d-nand flash for high-density memory and its technology evolutions and challenges on the future. In *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016.
[20] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo. Vfrm: Flash resource manager in vmware esx server. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014.
[21] X. Liu and K. Salem. Hybrid storage management for database systems. *Proc. VLDB Endow.*, 6(8).
[22] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-cave: Effective ssd caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.
[23] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.
[24] D. Mishra, P. Kulkarni, and R. Rangaswami. Synergy: A hypervisor managed holistic caching system. In *IEEE Transactions on Cloud Computing*, 2017.
[25] D. Mishra, Prashanth, and P. Kulkarni. Doubledecker: A cooperative disk caching framework for derivative clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, 2017.
[26] S. Mittal and J. S. Vetter. 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), 2016.
[27] C. Monzio Compagnoni, A. Goda, A. S. Spinelli, P. Feeley, A. L. Lacaita, and A. Visconti. Reviewing the evolution of the nand flash technology. *Proceedings of the IEEE*, 105(9), 2017.
[28] C. Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim, and Bumsoo Kim. A low-cost memory architecture with nand xip for mobile embedded systems. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, 2003.
[29] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. In *Proceedings 21st International Conference on Computer Design*, 2003.
[30] Po-Liang Wu, Yuan-Hao Chang, and T. Kuo. A file-system-aware ftl design for flash-memory storage systems. In *2009 Design, Automation Test in Europe Conference Exhibition*, 2009.
[31] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7), July 1974.
[32] T. Pritchett and M. Thottethodi. Sievestore: a highly-selective, ensemble-level disk cache for cost-performance. In *37th Annual International Symposium on Computer architecture*, 2010.
[33] L. Sang-Won, M. Bongki, P. Chanik, K. Jae-Myung, and K. Sang-Woo. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, 2008.
[34] P. Sehgal, K. Voruganti, and R. Sundaram. Slo-aware hybrid store. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.
[35] P. Sehgal, K. Voruganti, and R. Sundaram. Slo-aware hybrid store. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.
[36] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.
[37] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didacache: A deep integration of device and application for flash based key-value caching. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.
[38] S. B. Vaghani. Virtual machine file system. *SIGOPS Operating System Review.*, 44(4):57–70, Dec. 2010.
[39] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.
[40] M. Wei, R. Banerjee, L. Zhang, A. Masad, S. Reidy, J. Ahn, H. Chao, C. Lim, T. Castro, O. Karpenko, M. Ru, R. Fastow, A. Brand, X. Guo, J. Gorman, W. J. McMahon, B. J. Woo, and A. Fazio. A scalable self-aligned contact nor flash technology. In *2007 IEEE Symposium on VLSI Technology*, 2007.
[41] Q. Yang and J. Ren. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.