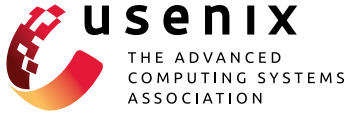

A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment

Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar, *PernixData, Inc.*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/bhagwat

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment

Deepavali Bhagwat Mahesh Patil Michal Ostrowski Murali Vilayannur
Woon Jung Chethan Kumar

PernixData, Inc.

Abstract

Host-side flash storage opens up an exciting avenue for accelerating Virtual Machine (VM) writes in virtualized datacenters. The key challenge with implementing such an acceleration layer is to do so without breaking live VM migration which is essential for providing distributed resource management and high availability. High availability also powers-on VMs on new host when the previous host crashes. We introduce FVP, a fault tolerant host-side flash write acceleration layer that seamlessly integrates with the virtualized environment while preserving dynamic resource management and high availability, the holy tenets of a virtualized environment. FVP integrates with the VMware ESX hypervisor kernel to intercept VM I/O and redirects the I/O to host-side flash devices. VMs experience flash latencies instead of SAN latencies and write intensive applications such as databases and email servers benefit from predictable write throughput. No changes are required to the VM guest operating systems so VM applications can continue to function seamlessly without any modifications. FVP pools together all the host-side flash devices in the cluster so every host can access another host's flash device preserving VM mobility. By replicating VM writes onto peer host-side flash devices, FVP is able to tolerate multiple cascading host and flash failures. Failure recovery is distributed, requiring no central co-ordination. We describe the workings of the FVP key components and demonstrate how FVP reduces VM latencies to accelerate VM writes, improves performance predictability, and increases virtualized datacenter efficiency.

1 Introduction

Virtualization has revolutionized how we build and operate data centers today. Large cost savings and increased operational efficiencies have made virtualization the biggest trend in data centers. However, as more applications become virtualized, and Virtual Machine (VM) density increases, shared storage performance does not scale with the high volumes of cumulative I/O generated by the VMs. I/O bottlenecks in the storage array add significant latency to virtual applications, resulting in slow response times at best and unusable applications at worst [1–3].

Provisioning better or more storage hardware is one way to address this problem. Some of these strategies include provisioning faster disks, improving the Storage Area Network (SAN) interconnect speeds, deploying flash caches in storage controllers [4–8], and replacing spinning disks with an all flash array [9, 10]. Replacing spinning disks with an all flash array is disruptive, incurring system downtime which may not be viable. Also, upgrading the SAN is usually a temporary fix in that even the new storage array will reach peak performance at some point resulting in the need for constant upgrades. Adding more CPUs or hosts is less disruptive.

An alternative approach is to install a flash device on the host and use it to cache VM writes. By co-locating the application's working set close to the application at the beginning of the I/O path, applications experience response times of the order of microseconds as opposed to milliseconds for shared storage. Host-side flash is thus used to *accelerate* applications and decouple storage performance from storage capacity [11–16].

Host-side flash has been typically used to cache recently accessed data to accelerate reads alone. VM reads are first issued to the flash device and, in case of a cache miss, issued to the SAN. The newly read data are also cached. VM writes are issued to the flash device and to the SAN. This approach of using the host-side flash device to accelerate reads is called *write-through* (*wt*) acceleration [12, 14]. Because a significant number of reads are offloaded from the SAN, it frees up the SAN's resources to service writes and un-cached reads. Therefore, write-through yields some improvement in write

performance as well.

Another approach is to use the host-side flash to accelerate both writes *and* reads. This is called write-back (*wb*) acceleration [11]. In *wb*, VM writes are issued to flash and on flash write completion, acknowledged back to the VM without issuing them to the SAN. This accelerates the writes since writes complete at flash speed, not SAN speed. In the background, writes on the flash device are batched and then issued periodically to the SAN to flush dirty VM data and to free up flash space for future writes. This process of issuing batched writes to the SAN is called *destaging*.

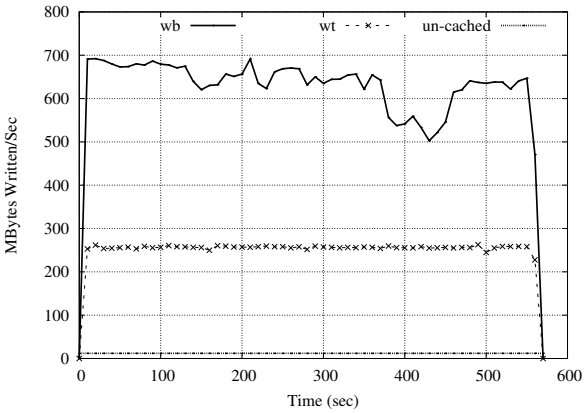

Figure 1: Combined throughput of Microsoft Exchange Server JetStress and fio, in *wb*, *wt*, un-cached

Write-back acceleration has a significant impact on write performance, because writes are acknowledged to the VM as soon as they are committed to the flash device alone. This creates a very short I/O path, resulting in very low latencies (typically microseconds). Write-through, in contrast, requires writes to cross the storage fabric to get to the SAN and be acknowledged before completion. Hence, VM writes in *wt* incurs SAN latencies. Figure 1 depicts the combined throughput of two VMs, one running Microsoft Exchange Server Jetstress [17] and the other running fio [18]. JetStress simulates the workload of an Exchange database consisting of transactions (reads and writes) by issuing 32 KB random reads and writes, and 14 KB sequential writes. fio was setup to issue 64 K sequential writes. Clearly, *wb* yields much better throughput than *wt*. Though the theory is sound and these benefits are appealing, accelerating writes using host-side flash in a clustered virtualized environment is not trivial.

The first challenge using host side flash in virtualized environments is that it must be done without breaking VM mobility [19]. For Distributed Resource Scheduling and Power Management (DRS) [20, 21], to balance resources utilization, VMs are live migrated or redistributed across hosts. For High Availability (HA) [22], in case of a host failure, VMs are migrated away from the failed host. VM mobility, therefore, must be preserved.

To ensure VM mobility, *all* the hosts *must* have access to the VM's data. Virtualized datacenters achieve this by consolidating storage arrays under a shared SAN. Host-side flash, in contrast, is a local resource so hosts cannot access each others flash. If a VM with data on a host's flash device gets migrated to another host, the VM loses access to its data. This precludes VM mobility and consequently breaks DRS and HA. Byan *et al.* [12] and Holland *et al.* [13] cite these reasons for why host-side flash cannot be used for accelerating writes.

The second challenge using host-side flash for accelerating writes is that it creates potential fault tolerance and consistency problems. In case of a flash device failure VM writes which were not yet destaged are not retrievable resulting in data loss. Koller *et al.* [11] argue that the only way to *prevent* data loss is to retrench to *wt*.

The third challenge using host-side flash for accelerating writes is that write-heavy applications may fill up the flash device at a faster rate than the rate at which those writes can be destaged to the SAN. This happens if SAN latency is high. If there is no space left on the flash device, the application cannot be allowed to write to the SAN because the SAN has stale data and overwriting stale data would cause data inconsistency and corruption. In this scenario, the application will stall until space can be made available on the flash device. A stalled application is clearly unacceptable; it would be preferable not to accelerate writes at all.

We introduce FVP, a write acceleration layer that uses host-side flash devices to accelerate VM writes in a clustered virtualized environment, while tolerating multiple cascading flash and host failures. FVP is a kernel module installed inside the ESX hypervisor [23] which intercepts VM I/Os and forwards them to the flash device. Since FVP sits inside the hypervisor, the I/O path from the guest OS to the flash device is short, resulting in good application performance. No change is required to the guest VM operating system.

FVP pools together the flash devices from all the hosts such that each host can access the data on another host's flash device. When a VM migrates to another host, its data on the previous host's flash device can be accessed by the new host eliminating inconsistency and data loss. VM mobility, DRS and HA are preserved.

VM writes are replicated to other hosts or peers. Therefore, in the event of a host or flash failure, other hosts co-ordinate with each other to destage their copy of VM writes to the SAN and restore VM consistency. By replicating VM writes, FVP can tolerate multiple host and flash failures. Recovery is distributed and happens without any central co-ordination. Our contributions are as follows:

1. FVP is a host-side write acceleration layer which supports transparent VM migration and seamlessly integrates into the virtualized environment.

2. FVP can handle multiple, cascading host and flash failures. To the best of our knowledge, ours is the only solution that provides fault tolerance. Recovery is distributed without requiring any coordination.

3. We introduce *Flow Control*, an I/O control mechanism designed to prevent write heavy applications from running out of flash space.

Though FVP has been implemented for ESX, the design principles are hypervisor independent and can be applied to any well-known hypervisor architecture, including Xen [24], Hyper-V [25], and KVM [26]. In the subsequent sections, we describe FVP in more detail.

2 Overview

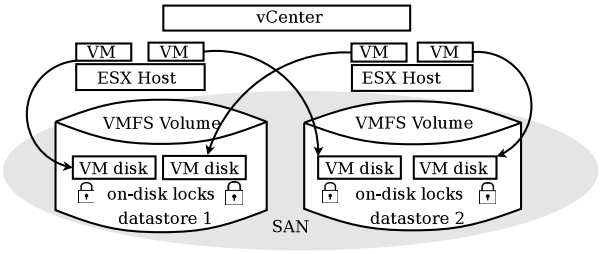

Figure 2: Virtualized Environment

Figure 2 depicts a typical virtualized environment with physical machines/hosts running the ESX hypervisor. A Storage Area Network (SAN) is used to consolidate storage and provide hosts shared access to it. A clustered file system, VMFS [27], provides multiple hosts simultaneous access to file system volumes or *datastores*. A datastore is a VMFS (block) based volume that houses VM virtual disks. A hypervisor cluster is a group of such hosts sharing one or more datastores via SAN. All the components and operations in the cluster are managed by a central entity, the *vCenter*.

A VM can only migrate to a host that has access to its datastore. Though a VM's files are accessible to every host in the cluster, VMFS moderates this access to one host alone. This is done using on-disk locks co-located on the same datastore as the VM's virtual disk. Only one host can acquire the VM's on-disk lock and this is the host that can write to the datastore on behalf of that VM. When DRS live migrates a VM, from one host to another, VMFS releases the on-disk lock for that VM. The new host now acquires the lock and runs the VM.

A FVP cluster is a hypervisor cluster in which every host runs FVP. VMs, thus, migrate only among FVP hosts. Henceforth, the term 'host' assumes that it is a FVP host, and 'cluster' assumes that all hosts in the cluster are FVP hosts.

3 VM Acceleration Policies

Each VM on an FVP host is configured with an *acceleration policy*. The acceleration policy dictates how VM writes and/or reads should be cached. Data center administrators can configure VMs with an appropriate write policy depending on VM workload.

Write Through (*wt*) policy: In *wt* [12, 13, 28] mode, FVP intercepts VM writes and issues them simultaneously to SAN and to flash. The write is acknowledged to the VM after it has been acknowledged by both the flash and the SAN. Typically, the longest time to acknowledgment is from the SAN, and the VM sees SAN latencies for writes.

Write-Back (no peers) (*wb*): For a VM in *wb*, FVP intercepts VM writes and issues them to the host's local flash alone. The write is acknowledged to the VM after flash completion. As writes accumulate on the flash device, they are batched and destaged/issued in batches to the SAN. Destaging is completely transparent to the VM. The VM, thus, sees flash latencies instead of SAN latencies for writes.

Write-Back with Peering (*wbp*): For a VM in *wbp*, FVP synchronously duplicates every write and sends it to another host, or *peer*, in the cluster while simultaneously writing it to local flash. When the peer writes VM data to its flash, an acknowledgment is sent back to the primary host housing the VM. FVP on the primary host then acknowledges the write to the VM. The peer holds on to the write until the primary has destaged it. Typically, data transmission over the network to the peer takes the longest. VM writes experience network *and* peer flash latencies.

FVP selects peer hosts based on rules/policies setup by administrators.
Datacenter administrators can setup fault domains that group together hosts based on the datacenter topology such as, hosts on a rack belong to one fault domain *etc.* Further, FVP also allows administrators to choose that hosts be configured with local peers within their fault domain and/or remote peers in other fault domains.

Un-cached: For a VM that is un-cached, FVP does not cache its data. Neither reads nor writes are accelerated.

For *wt*, *wb*, and *wbp* VMs reads are first issued to

flash. In case of a cache miss, the read is issued to SAN and the data written to flash. Evictions, irrespective of caching policy are done on a LRU basis.

4 Seamless VM Migration

To balance resource utilization in datacenters, VMs are live migrated, or re-distributed across hosts [20]. An eligible host is one that has access to the VM's datastore. After migration, VMs continue to have access to their data because datacenters consolidate storage arrays under a shared SAN. However, host side flash is local to the host. If a VM having dirty data on its host's flash is migrated to another host, it loses access to that data. FVP solves this problem by enabling the new host to access the previous host's flash.

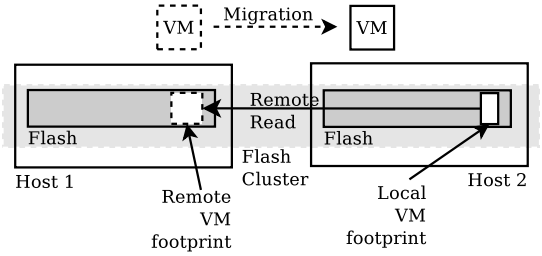

Figure 3: Virtual Machine Migration

Figure 3 illustrates how FVP orchestrates VM migration between two hosts. When a VM migrates from Host 1 to Host 2, its cached data on Host 1's flash, *i. e.*, its footprint, remains on Host 1. Host 1 is free to evict the VM's cache if it so requires.

FVP maintains state about the VM's previous host. This obviates the need for Host 2 to poll every other host to locate that VM's footprint. So, when the VM issues a read that causes a cache miss on Host 2's flash, Host 2 re-transmits the read to Host 1. Such a read is called a *remote read*. If the read request causes a cache miss on Host 1's flash (possibly because Host 1 has already evicted that data to reclaim flash space) Host 2 re-issues the read to the SAN.

In case of a cache hit on Host 1's flash, Host 1 transmits that data to Host 2. When Host 2 receives the data, it copies the data onto its own local flash. Thus, Host 2 begins building the VM's footprint locally. Owing to remote reads, VM reads that are a cache hit are faster, seeing only network and flash latency than if they were issued to SAN. Remote reads also alleviate traffic to the storage array while VM footprint is rebuilt on Host 2. By issuing remote reads for only what the VM requires, instead of eagerly copying the entire VM footprint, Host 2 conserves network bandwidth and flash wear.

The VM's footprint eventually rebuilds on Host 2's flash. When the number of cache misses for remote reads hits a pre-defined value, Host 2 stops issuing remote reads to Host 1. VM mobility is thus transparently preserved.

It would be ideal if a *wbp* VM were to migrate to one of its peers. The VM would then run off of its own footprint on the peer obviating the need to issue remote reads. However, FVP has no say over which host is chosen as the destination host for a VM. This is a decision made by the DRS process into which FVP has no visibility.

5 The Destager

The destager is a process that collects dirty VM writes cached on the flash device and issues them to the SAN. The writes are batched and all writes in a batch are issued concurrently. Batching is used to improve performance and ensure correctness as is described in this section. When the SAN acknowledges the writes, FVP marks those writes as destaged. Those writes can now be evicted from the cache. We discuss how the underlying storage fabric drives the design and behavior of the destager.

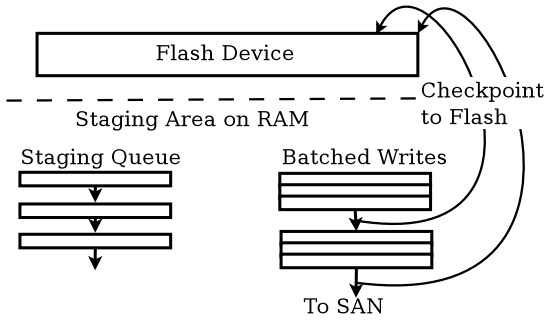

Figure 4: Workings of the destager

The sequence in which a storage controller may order concurrent writes is opaque to FVP, and indeed, to the application issuing such writes. This is not an issue for concurrent *non-overlapping* writes. However, concurrent overlapping writes, if not handled correctly, can cause application inconsistency and data corruption. Consider a write X at offset Y, denoted as X(Y). Consider two concurrent overlapping writes A(O) and B(O). FVP construes an ordering of A(O) followed by B(O) and issues them to flash. FVP only indexes the last write and therefore records B for offset O. So, when the VM issues a read for offset O it receives B. Meanwhile, the destager begins destaging the writes and both (A, O) and (B, O) being concurrent writes, issues them concurrently to the VM's datastore. The storage controller orders them (B, O) followed by (A, O), overwriting B with A. The datas-

tore is now inconsistent with the VM.

To avoid such inconsistencies, a *gatekeeper*, assigns a monotonically increasing serial number (`ser#`) to *every* write. Non-overlapping writes tagged with a `ser#` are issued simultaneously to the flash. However, concurrent overlapping writes are serialized, *i. e.*, the gatekeeper waits for the flash device to acknowledge the write before issuing the next overlapping write. The writes are then enqueued in a staging queue in the *staging area*.

Figure 4 depicts the workings of the destager. The staging area is kept in RAM to save flash space and wear. It consists of a *staging* queue where writes are enqueued in order of their `ser#`. Note that the queue contains metadata only. The data resides on the flash device. The destager scans these writes, batches them up such that no two writes in a batch overlap. All writes in a batch are issued concurrently to the SAN. After the SAN acknowledges *all* those writes, the destager issues a `checkPoint` to the flash. A `checkPoint` is a special record consisting of the VM UUID, and `ser#` of the VM's last write record acknowledged by the SAN. FVP uses `checkPoint` records during failure recovery described in Section 7.

After the destager receives acknowledgment from the flash that it has completed writing the `checkPoint`, the destager proceeds to scan the staging queue again. If the VM was configured with peers, the `checkPoint` record is also transmitted to the peer/s. Each peer commits the `checkPoint` record to its flash. FVP on the primary and peer hosts is now free to evict from its flash device all records for that VM whose `ser#` $\leq$ `checkPoint` (`ser#`).

The storage controller may or may not order the writes in `ser#` order (indeed, it is unaware of any such metadata), but, since these writes are non-overlapping, writing them in any order it wishes does not lead to data corruption. The justification for such behavior is that even in the absence of FVP, for concurrent writes the storage controller gives the same consistency guarantees and therefore after the entire batch of writes has been committed to the SAN, the SAN is consistent with VM, though lagging by some time.

The staging queue is maintained on a per VM basis. The destager cycles through staging queues in a round robin fashion destaging one batch per queue before proceeding to the next VM's queue. To allow each VM a fair share of the SAN's bandwidth for servicing writes and reads, the size of each batch is capped to a configurable value. The maximum batch size, defined in the number of writes, is configured to match the queue depth of the storage's host bus adapter.

As a rule of thumb, issuing fewer but larger sets of concurrent writes to storage yields better throughput than issuing smaller and frequent concurrent writes. Therefore, deferring and consolidating writes while destaging yields better flash acceleration. A VM that issues frequent overlapping writes does not see the level of flash acceleration that another VM issuing fewer overlapping writes would see. If such writes are bursty, the flash is able to absorb the burst, simultaneously destage writes and catch up with the VM a few moments after the burst of writes has stabilized without degrading application throughput.

6 Flow Control

The flash space on an ESX host could potentially be shared by hundreds of VMs. Though every VM may have different space requirements which change over time [29], FVP implements a fair share policy when carving out flash space for individual VMs. Fair share, though an early implementation, has advantages such that it isolates VMs from any noisy neighbors. Noisy neighbors are VMs that claim a high proportion of flash space due to their large working sets. Consequentially, other VMs experience degraded performance. FVP implements fair share for its simplicity and to insulate VMs.

For sustained write bursts, a VM's writes accumulate on flash at a high rate filling up its quota of flash space. The destager works in tandem with the VM to clear up that flash space to accommodate new writes. However, the destager's throughput is predicated on the latency of the SAN. If SAN latency is high, during a sustained write burst, a VM's flash space fills up before the destager has a chance to catch up. The VM, then, would have to be stalled, *i. e.*, it cannot be allowed to issue I/Os until (a) the destager is able to reclaim space by flushing out accumulated writes and/or (b) FVP is able to evict cached *reads* from the flash device. When space is reclaimed, the VM can continue to issue I/Os, but if the write burst continues, the VM would have to be stalled once again to allow the destager to make more space and so on. At such times, the VM would experience degraded performance or SAN latencies.

The VM cannot be allowed to write to SAN because SAN has stale data. If the VM were allowed to write, and those writes overlapped with writes that are yet to be destaged, data corruption would occur.

To prevent VM performance degradation, FVP triggers a process called *flow control*. Flow control throttles the VM by introducing an artificial delay before a write is acknowledged back to the VM. Though this slows down the VM, it gives the destager some extra time to make space for new writes. The delay is calculated as a moving average over SAN latencies observed in the near past. FVP uses three heuristics to trigger flow control per VM:

1. The number of dirty VM writes.
2. The cumulative size of the dirty writes.

3. The expected time to destage those writes.

These were chosen because they are indicative of how fast the destager would be able make progress given its pending workload. A high number of writes in proportion to their cumulative size indicates that the VM is primarily issuing small writes. The destager's batch size being fixed, this means the destager has to cycle through a larger number of batches, and consequentially, engage in those many conversations with the SAN. A higher cumulative size of dirty writes in proportion to the number of dirty writes indicates that the VM is issuing large writes which typically incur higher SAN latencies. The expected time to destage all the writes is calculated using a moving average of SAN latencies seen in the past. Together, these three heuristics indicate the probability of whether the destager would be able to clear up flash space in time to accommodate VM writes. If any of the above heuristic counters cross pre-defined triggering values, flow control kicks in, progressively injecting increasing amounts of delay when acknowledging VM writes. Flow control slows down the incoming writes by introducing delays of the order of $1\times$ to up to $4\times$ the SAN latency to reduce the pressure on the staging area. However, if VM writes fill up its flash space because the destager is not able to make progress on account of a very slow SAN, the VM performance degrades to match that of the SAN.

Stalling is extremely rare. Only in the case of workloads that are write intensive over a prolonged period of time coupled with a very slow SAN would the VM be stalled.

Most VMs, however, are not continuously write intensive, but rather bursty [30]. Short write bursts are absorbed by the flash device and the VM throughput remains unaltered during the bursty period. Sustained write bursts may need to be throttled for the latter part of the bursty period, and in majority of the cases, continue to run while experiencing SAN latencies instead of flash latencies. Also, once the sustained burst of writes has lapsed the destager once again regains ground and the VM is freed from flow control.

Besides undesirable degraded VM performance, holding on to a large size of dirty VM data on the flash device increases the impact of data loss to a *wb* VM in case of a flash failure and the window of vulnerability in the case of host failure. Flow control, thus, prevents VM performance degradation, mitigates the consequences of failures, and speeds up VM migration.

7 Distributed Fault Tolerance

For *wb* VMs, in the event of a failure the SAN is left in an inconsistent state with respect to the VM in that some of the VM's writes have not yet been persisted to the SAN but have been acknowledged to the VM right after they were persisted to the host-side flash. The key challenge towards achieving crash consistent fault tolerance when using host-side flash devices for write back acceleration is ensuring that at the end of failure recovery cached VM writes are destaged correctly and completely to the SAN.

In the case of a host failure, for *wb* VMs, after the host recovers the destager flushes VM writes from its flash to the SAN from the last `checkPoint` onwards in order of their `ser#`. The `ser#` is persisted to flash as part of other metadata for every write. This ensures correctness, *viz.*, the writes are replayed in the order in which they were received by FVP.

However, HA may migrate affected *wb* VMs to another host while the failed host is recovering. If those VMs issue I/Os that overlap with previous I/Os that have not yet made it to the SAN, data corruption will occur.

To prevent data corruption, and co-ordinate the recovery process between two hosts as a *wb* VM migrates between them, FVP uses an on-disk lock file called `vault.lock`. For every VM, FVP persists its current acceleration policy and `checkPoint` record in the `vault.lock` file. The `vault.lock` file is located on the VM's datastore. Through atomic `vault.lock` access, VMFS arbitrates ownership of a VM's datastore between hosts such that the `vault.lock` file is locked and kept open by one host only. Only this host is eligible to execute I/Os on behalf of the VM to the SAN. The lock is held by the recovering host until recovery is complete.

In the case of a flash failure the cached writes are lost. FVP solves this problem by replicating VM writes onto peer hosts' flash devices. The peers can now flush those replicated writes to the SAN to complete recovery.

When a host loses connectivity to the storage fabric, the cached writes cannot be flushed to the SAN resulting in data loss. If any of the peers has access to the SAN, it can flush the replicated writes from its flash.

In addition to on-disk locks, hosts also monitor network, storage, and, peer health via regular heart beats. Thus, atomic access to the on-disk `vault.lock` file, read-only access to the `vault` file contents and heart beats together form the basis of FVP's failure recovery mechanism. FVP is designed to tolerate multiple flash, host, and network failures. Recovery is distributed; there is no master-slave protocol between hosts. We now discuss how this distributed failure recovery is instrumented for each of the failure scenarios.

7.1 Flash Failure

For a *wt* VM, in the event of a flash failure there is no data loss.

For a *wb* VM, in the event of a flash failure, if there was dirty data on the flash device, FVP stalls the VM. To

Failure	*wt*	*wb*	*wbp*
Flash	*wt* →un-cached No data loss	Data loss	Peer online replay *wb* →*wt* →un-cached
Host	No recovery required	Offline replay *wb* →*wt* →*wb*	Peer online replay *wbp* →*wt* →*wbp*
Network	NA	NA	Primary online replay *wbp* →*wt* →*wbp*
Multiple Failures	NA	For host failures, offline replay, or surrogate offline replay	Online/offline replay by primary, peer, or surrogate hosts

Table 1: Recovery from flash, host, network and multiple failures

avoid data corruption the VM cannot be allowed to write to the SAN. A flash failure for a *wb* VM results in data loss.

For a VM in *wbp*, in the event of a flash failure, to avoid data corruption FVP stalls the VM, and relinquishes lock on `vault.lock`. The peers, who periodically poll `vault.lock`, attempt to acquire the lock, contending against each other, where contention is resolved by VMFS. The peer that acquires the lock begins destaging VM writes starting from the last `checkPoint` that it had received from the primary host, regularly updating the `checkPoint` in `vault.lock`. After destaging is complete the peer updates `vault.lock` with the latest `checkPoint`, a cache policy of *wt*, and relinquishes lock on `vault.lock`. The process of destaging writes by a *live* host in the event of a failure is called *Online Replay*.

The VM policy of *wt* indicates to the peers that the VMs dirty data has been flushed. They drop all of the VM's writes from their flash and stop polling `vault`. The host that destages VM writes updates `vault.lock` `checkPoint` regularly so as to communicate to the remaining peers, via `vault`, that they can clear up writes that have already been destaged. Also, in case of peer failure or peer flash failure, the remaining peers can pick up where the last peer left off by starting with the `checkPoint` in `vault.lock` minimizing recovery time.

When the primary host detects from `vault` that the VM has transitioned to *wt*, it reacquires `vault.lock`, updates `vault.lock` with an acceleration policy of 'uncached', and, un-stalls the VM.

7.2 Host Failure

There is no data loss for a *wt* VM in the case of a host failure.

In the event of a host failure all the `vault.lock` files for the affected VMs are released as part of the failure detection process. After the host recovers, the host reacquires those locks. For a *wb* VM, after the failed host recovers, FVP scans all the cached writes on the flash, building an inventory of resident *wb* VMs and their dirty data. This scan is necessary because as a consequence of host failure, the 'in RAM' staging data structures used during normal destaging and online replay operations are no longer available.

As soon as the scan is complete, the host and FVP come back online, and the VMs are powered on. VMs having dirty data on flash are stalled until their data is destaged, but FVP is ready to service VMs not affected by the host failure. For VMs with dirty data, FVP tries to lock `vault.lock`. For those VMs whose `vault.lock` was acquired, their dirty writes are destaged in order of `ser#` while regularly updating `vault.lock` `checkPoint`. At the end of replay, each `vault.lock` is updated with a cache policy of *wt*. This process of destaging records by a host recovering from a failure is called *Offline Replay*.

If HA has migrated any *wb* VMs to another host before the previous host has recovered, the new host is now able to acquire a lock on their `vault.lock` file and is, therefore, now eligible to issue I/Os on behalf of the migrated VM. This could cause data corruption. To prevent this from happening, FVP persists VM acceleration policy in `vault.lock`. When the new host acquires `vault.lock` for a migrated VM, it gleans that the VM was in *wb* on the previous host. This indicates to the new host that the VM has pending writes on the previous host's flash which have not yet been flushed to the SAN. It releases `vault.lock` and stalls the VM.

A read only copy of `vault.lock` is kept in another file called `vault`. The new host polls `vault` periodically so it can re-acquire `vault.lock` when the previous host is done destaging.

For *wbp* VMs, while the primary host is down, the peer/s detect host failure due to missing heart beats. Either that, or the peer/s detect host failure because one of them is able to acquire `vault.lock` for affected VMs. The peer that succeeds, executes an online replay on behalf of those VMs regularly updating the `checkPoint` in `vault.lock`. At the end of online replay `vault.lock` is updated with the last VM `checkPoint`, and *wt* cache policy. The other peers, on gleaning transition of the VM to *wt*, drop all data belonging to that VM from their flash.

The peers no longer participate in providing fault tolerance for the VM.

Meanwhile, FVP on the recovering primary host tries to acquire `vault.lock`, and fails. When it eventually acquires `vault.lock` it detects that the VM has transitioned to *wt*. No offline replay is required for this VM. The VM is un-stalled, new peers are configured and it transitions back to running in *wbp*.

7.3 Network Failure

A network failure is one when peers lose connectivity with each other. This only affects VMs in *wbp*. In case of a network failure, writes can no longer be replicated to peers. When FVP on the primary host detects a network failure, it initiates a $wb \rightarrow wt$ transition of the affected VMs. The destager flushes all the dirty data for those VMs. Flow control is invoked to allow the destager to make quick progress. During this stage, the affected VMs continue running in *wb*, but without the desired level of write redundancy. When a very small number of writes remain to be destaged, the VM is stalled, the remaining writes are destaged, the VM is un-stalled, released from flow control, and the transition to *wt* is complete.

The peers glean the transition to *wt* from `vault` and drop all cached data for the concerned VMs. For every affected VM, FVP chooses new peers from a list of other candidate hosts. Once the desired number of peers are re-established the VM is transitioned back to *wbp*.

It is possible that after the failed peer comes back up, the VM has already transitioned to *wb*/*wbp*. The peer may then incorrectly deduce that it should continue holding on to the VMs dirty data on its flash. To prevent this, FVP maintains a generation number (`gen#`) for every VM which is also persisted in `vault.lock`. The `gen#` is a monotonically increasing number that keeps track of a VM's transitions through cache policies. It is incremented every time a VM transitions from $wb \rightarrow wt$ or $wbp \rightarrow wt$. The incremented `gen#` indicates to the freshly recovered peer that the VM's data it contains on its flash has already been replayed. It is now free to evict that data.

Lastly, a network failure, could be misconstrued by peers, as a primary host failure. They try to acquire `vault.lock` while monitoring `vault` and take over replay if the primary host fails.

7.4 SAN Failure

A SAN failure is when hosts are unable to connect to the storage array. It is possible that such disruption affects only partial hosts. If FVP on the primary host has lost connectivity to the storage array, it stalls the VMs. For VMs in *wbp*, the primary host depends on the peers to replay dirty data. If only the peers have lost connectivity to storage, they begin to fail remote writes sent for replication by the primary host. This indicates to the primary host that the peers are no longer able to keep replicas. The VMs are flow controlled, their data is destaged and then transitioned to *wb*. To summarize, any host that has access to the storage array can acquire `vault.lock` and replay dirty data. Note that even if all hosts have experienced SAN failure, data on the flash device is still intact.

7.5 Multiple Cascading Failures

If during an online or offline replay, the concerned host fails, the remaining replay can be completed either by its peers or itself on recovery. If the primary host *and* peers fail and are unable to recover, the flash device can be re-installed on another host that has access to the VM's datastore. This is possible because all necessary metadata required to conduct replay is persisted on the flash device itself. The surrogate host would scan the flash device, acquire the necessary `vault.lock` from the VM's datastore and then complete replay for affected VMs. To minimize recovery time in the case of cascading failures, `vault.lock` (and consequently, `vault`) are regularly updated with the last `checkPoint`.

7.6 Distributed Recovery

The key to distributed recovery is access to shared storage and exclusive ownership of `vault.lock` among participating hosts. Shared storage access enables primary and peer hosts to monitor VM acceleration policy transitions and destaging progress. VMFS ensures access to the on-disk lock is atomic preventing any split-brain scenarios allowing hosts to co-ordinate failure recovery one at a time via online/offline replay picking up from where the last host left off. For instance, in the case of primary host failure, one of the peers takes over online replay. If this peer fails, the next peer may take over. If both peers fail and the primary host comes back online, the remaining data, is destaged by the primary host via offline replay.

Table 1 summarizes how FVP recovers from various failure scenarios. FVP recovery is crash consistent. Data corruption is prevented by use of `checkPoint`, `gen#` and `ser#`. Recovery is complete when every write acknowledged to the VM before failure/crash is committed to the SAN. Hence, after recovery, affected VMs return to a crash consistent state. VMs in *wt* are always crash consistent. For VM in *wb*, in case of a flash failure, there is data loss. VMs in *wbp* are protected against p flash, and $p+1$ host failures, where p is the number of peers.

8 Evaluation

Our setup consists of two hosts, each a HP Proliant DL 380 G6, 8 cores, ®Intel Xeon CPU E5540 at 2.533 GHz, 55 GB RAM, and 120 GB flash drive. Each host runs VMware vSphere 5.5 Enterprise Plus. The shared storage is a storage appliance with disk spindles and a flash cache. We will show that even when used with this faster than average SAN, FVP provides significant speed and latency improvement. With a slower HDD-based SAN, latency improvements will be more significant.

8.1 Short Write Bursts

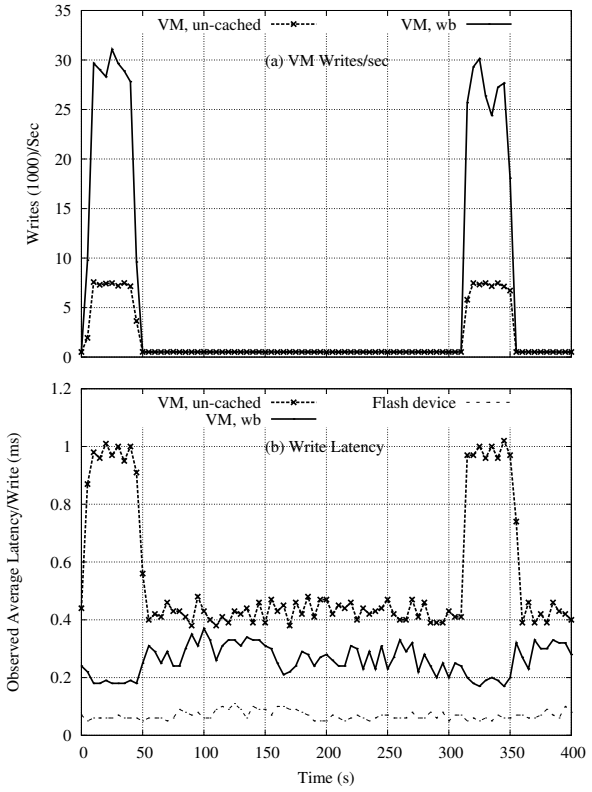

Figure 5: Short Write Bursts: (a) VM Writes/sec, (b) Write Latency

The objective of the first experiment is to demonstrate how FVP absorbs *short* write bursts thereby masking the VM from latency spikes of the SAN. To do this, we generated a workload resembling writes issued by a database servicing an OLTP application [31]. The workload, generated using Iometer [32], issued a burst of sequential writes for a short duration (40 seconds) followed by a slow and steady pace of random writes for 280 seconds. The short bursts of sequential writes simulated writes generated by a database as a result of multiple transaction commits, and log flushing. The steady stream of random writes simulated inserts/updates to support short database transactions. All writes were 8 KB in size, and 8 KB aligned.

To illustrate the benefits of using FVP to accelerate such workloads, we compared VM performance in *wb* and in an un-cached mode. The workload was run twice, once with the VM configured in *wb* and next with the VM in un-cached mode. In the un-cached mode, writes were not cached on the flash device but were issued directly to the SAN.

Figure 5 (a) compares the rate at which writes were acknowledged to the VM when running in *wb* with that when the VM ran in un-cached mode. In un-cached mode, writes were acknowledged to the VM after they were written to the SAN, at about 7,500 writes/second. In *wb*, during the first write burst (between 0 and 50 seconds) FVP acknowledged the writes to the VM as soon as the writes issued to the flash device were completed, at the rate of about 30,000 writes/second. Even though the SAN was slower to acknowledge writes during the bursty period, the VM did not see a degradation in performance when running in *wb*. This is because the write burst was absorbed by the flash device and the VM writes were acknowledged at flash speed. In the background, the destager issued those accumulated writes to the SAN.

Next, Iometer issued a steady stream of random writes at a slower rate for 280 seconds. During this period, as seen in Figure 5 (a), VM writes were acknowledged at the same rate in *wb* and in the un-cached mode. This is because, the incoming rate of writes was slow enough so that FVP and the SAN were able to service all the writes in a batch before the next batch of writes was issued.

This cycle of short bursts followed by steady writes was repeated once more.

Figure 5 (b) plots the average write latency, *i. e.*, the time from write issue to write completion, as observed by the VM. The figure also plots the flash write latency when VM writes were cached (*wb*). Together, the Figures 5 (a) and (b) demonstrate two key strengths of FVP: the first being that during the bursty period, the VM latency in *wb* tracked flash latency, *not* SAN latency. This allowed the VM to issue 4× the number of writes during the bursty period in *wb* than when un-cached. The second, is that the write latencies in *wb* were steady *and* low. In contrast, write latencies observed by the un-cached VM varied from 0.4 ms at best, to 1 ms during the bursty period.

8.2 Sustained Write Bursts

Figures 6 (a) and (b), demonstrate how FVP handles *sustained* write bursts. Figure 6 (a) depicts the rate at which writes were acknowledged to the VM by FVP and to the destager by the SAN, while Figure 6 (b) depicts VM, and

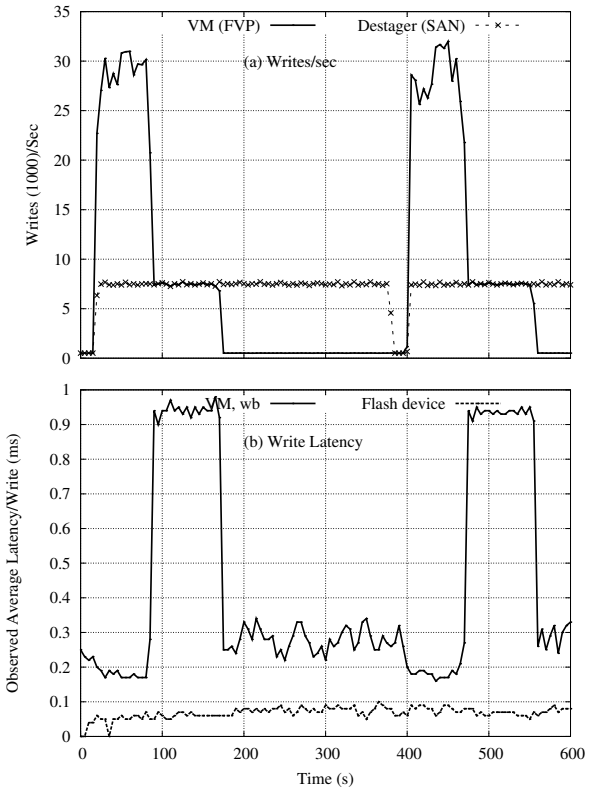

Figure 6: Sustained Write Bursts: (a) Writes/sec, (b) Write Latency

flash write latencies for those corresponding times.

To demonstrate how flow control manifests, we generated a workload similar to the OLTP workload, but quadrupled the write burst period (160 seconds). For the first 80 seconds, writes were acknowledged to the VM at flash speed. After 80 seconds, due to the volume and rate at which writes accumulated on flash, FVP triggered flow control. Flow control introduced delays in the write completion path before the writes were acknowledged to the VM. The induced delays were equal to the SAN's latency. Hence, VM writes experienced SAN latencies. These additional delays are shown in the Figure 6 (b) where VM latencies increased over and above flash latencies during the flow control period. Flash latencies were always steady at about 70 μseconds.

Once the sustained burst was over, the rate of incoming writes dropped. This allowed the destager to catch up and destage the pending records. The VM was then released from flow control. We know it was released, because at the beginning of the next burst, the VM acknowledged writes at flash speed again.

SAN latencies may increase if/when the SAN is overloaded because of the cumulative I/Os issued by several VMs or due to SAN administrative tasks. This too can trigger flow control. By using flow control, FVP avoids stalling applications/VMs which would unacceptably interrupt business processes. Instead VMs continue to run gracefully at SAN speed.

8.3 Read Latencies during VM migration

The objective of the next experiment is to demonstrate how FVP preserves VM mobility with minimal impact to VM performance. Figure 7 shows latencies observed by a VM running Iometer while issuing random 4K reads. For the first 700 seconds, the VM experiences millisecond latencies as the data is fetched from SAN. This was because, none of the data was cached before the workload was started. As the cache was populated and hits increased, read latencies gradually reduced. Once the working set is cached, the VM experiences flash latencies of the order of 450 μseconds. Then the VM is migrated to another host. In response to cache misses the new host begins to issue reads to the previous host and gradually builds up the VM's footprint on its local flash. The increased read latencies in the graph after VM migration when the new host issues remote reads are due to the additional latency incurred in transmitting the read data over the network from the previous host to the new one. As the new host gradually builds up the VM's footprint, fewer remote reads are issued. This can be seen from the graph where read latencies gradually reduce after migration. Once the VM footprint on the new host's flash is complete, remote reads are no longer issued and the VM experiences flash latencies once again.

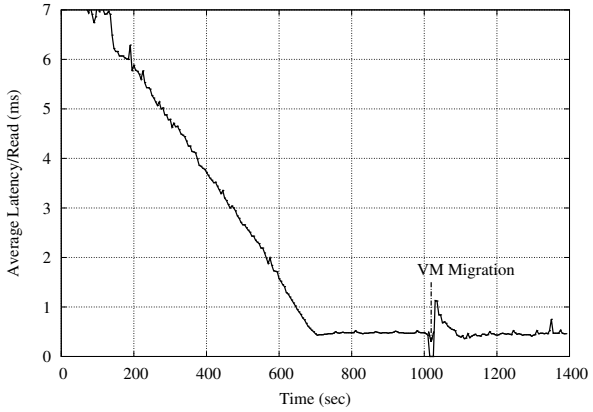

Figure 7: VM observed read latencies after migration

8.4 Fault Tolerance Cost vs Benefit

The objective of the final experiment is to analyze the trade-off between fault tolerance and performance. To do this, we compare VM throughput in *wt* with that in

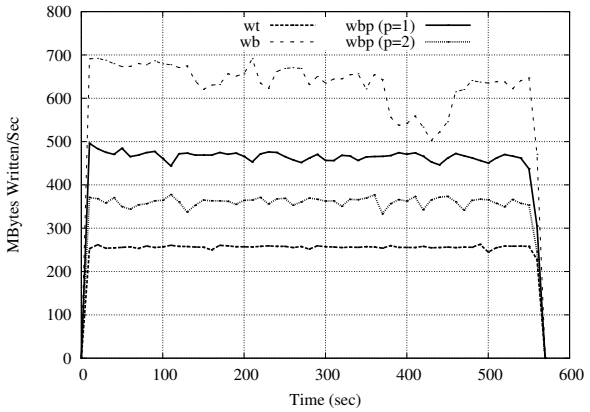

Figure 8: Combined throughput of two VMs in *wt*, *wb*, *wbp* (p=1), *wbp* (p=2). Workloads: Microsoft Exchange Server Jetstress and fio

	fio Throughput	JetStress Transactional IOPs
wb	468.81 MB/s	9325
wbp (p=1)	322.48 MB/s	7677
wbp (p=2)	235.32 MB/s	6605
wt	149.44 MB/s	5514

Table 2: Application performance in *wt*, *wb*, *wbp* (p=1), *wbp* (p=2)

wb, *wbp* ($p = 1$), and *wbp* ($p = 2$). Figure 8 depicts the combined throughput of two VMs. The guest OS of one VM was Windows 2008, running Microsoft Exchange Server Jetstress [17]. Jetstress simulates the workload of an Exchange database consisting of database transactions (reads and writes), log writes, and maintenance tasks such as database compaction, defragmentation and checksums. Jetstress was configured with 150 mailboxes allocated over 40 GB. The guest OS of the second VM was Ubuntu running fio [18], an IO workload generator. fio was configured to simulate a throughput intensive workload with two threads, each thread issuing 64 KB sequential writes with 8 writes in flight.

The first observation from Figure 8 is that write caching (*wb*) has clearly accelerated VM throughput, a definite gain over *wt*. The second observation is that when writes were replicated to the peer (*wbp*), the VM throughput slowed down compared to that in *wb*. In *wb*, the average throughput of the VMs was around 600 MB/s, while in *wbp* ($p = 1$) it was 475 MB/s. This difference was because in *wbp*, every write incurred an additional latency when it was replicated across the network onto the peer's flash.

DRS and HA use the network interconnect between hosts to migrate VMs between them. FVP uses the same network to transmit writes between a host and its peers for fault tolerance. The cumulative throughput for *wbp* ($p = 2$) was lesser than that with *wbp* ($p = 1$) because currently the FVP network stack instrumentation does not exploit multiple NICs. This work is, however, planned for the coming future. With multiple NIC support, VM throughput would track that of the slowest peer network.

Table 2 lists the performance of the individual applications in *wt*, *wb*, and *wbp* ($p = 1, p = 2$). For fio, the table lists the average throughput whereas for JetStress, the table lists the transaction rate (IOPs). Figure 8 and Table 2, together illustrate that the cost of replicating writes is reduced VM throughput. However, the throughput of a *wbp* VM is still better than a *wt* VM with the added advantage of fault tolerance. To summarize, FVP solves fault tolerance by replicating writes, *and* achieves write acceleration by using host-side flash.

9 Related Work

Holland *et al.* [13] and Byan *et al.* [12] prescribe using host side flash for *wt* only because *wb* causes consistency issues with VM migration. Byan *et al.* explore various options for deploying host-side flash: integrated with the storage controller, the network, the hypervisor *etc.*, and choose to deploy their solution within the hypervisor.

Koller *et al.* [11] discuss the trade-offs of using host-side flash with respect to data consistency, staleness and performance for *wt* and *wb*. They propose ordered and journaled destaging. Ordered destaging evicts writes in the order in which they were issued, like FVP. In addition, Koller *et al.* parallelize evictions for unrelated writes. Journaled destaging coalesces writes to absorb write bursts. Using application specified hints Journaled destaging provides application level consistency. FVP also offers a best-effort application level consistent destager, but for the sake of brevity, and to focus on our key contributions, we have not elaborated on it.

Qin *et al.* [14] use application specified write barriers to achieve application consistency when using host-side flash to accelerate writes. Application specified write barriers, or hints are not distinguishable to the FVP kernel module. This was a deliberate decision; one that allows FVP to seamlessly integrate into the virtualized environment. Further, in an enterprise environment third party softwares [33] are employed to perform backups. These software quiesce the guest OS to allow for application consistent snapshots. FVP seamlessly detects this activity and transitions those *wb* VMs to *wt* for the duration of the snapshot operation. The details of this mechanism have not been discussed in the paper since that is not the main focus. In addition FVP also provides datacenter administrators a switch to initiate a $wb \rightarrow wt$ tran-

sition on VMs for the duration of the backup/snapshot window.

Koller *et al.* also discuss the cost in terms of the potential data loss that might be incurred in the case of failure for applications when they use *wb* acceleration. With *wbp* in FVP VMs are protected against p flash failures and $p+1$ host failures to minimize the probability of a data loss. The cost, however, is increased recovery time for when the affected VMs are stalled until their *wb* data has been flushed to the storage.

How long a VM is kept stalled depends on several multi-dimensional factors. The most critical being the volume of dirty writes. The time required to destage those writes, and therefore, the period for which the VM is kept stalled is proportional to the size of staged data. The other factors being the SAN speed/performance, the saturation of the SCSI network, the load on the disk array, the number of VMs being hosted, the workload those VMs are generating *etc.* These factors, other than staged data size, change dynamically depending on the workload supported by the datacenter. If these were steady, then the time for a which a VM needs to be stalled is proportional to the amount of staged data that needs to be flushed. To mitigate the cost of recovery, FVP caps the size of dirty data that can accumulate for a VM. As discussed in 6, flow control is invoked to moderate VM write footprint. To tune this further, datacenter administrators can configure the staging size based on the RPO requirements for applications.

Also, in the case of FVP, recovery speeds up when peers are involved. When a host fails, the peers can finish destaging so that when the VMs are brought back up most or all of their data has been already flushed to SAN. Thus, peering reduces the cost of recovery for *wbp* VMs.

10 Future Work

We are working on building more intelligence into FVP; an adaptive resource manager that detects VM workload characteristics and priority, and in response, tunes VM flash space usage, acceleration policy, eviction policies, and destager behavior for that VM.

11 Conclusion

FVP brings seamless, fault tolerant write acceleration using host-side flash to HA and DRS enabled virtualized datacenters. Failure recovery is distributed and, with p peers, is $p+1$ host/flash failure tolerant. FVP absorbs short write bursts so VMs see flash latencies instead of degraded SAN latencies during the bursty period. This masks VMs from SAN latency spikes, improves VM performance predictability to help deliver SLA objectives allowing IT teams to accelerate write heavy applications, such as databases and Virtual Desktop Infrastructure [34]. For sustained write bursts FVP uses flow control; write intensive applications continue without stalling. The storage can now be provisioned linearly with new hosts, instead of being provisioned for bursty workloads. FVP helps increase VM density allowing existing hosts to support more VMs without having to provision additional storage. This increase in performance means happier end users and, consequently, fewer support calls relating to poor application performance. This allows IT organizations to consolidate more hosts and economize.

12 Acknowledgments

Heiko Köhler, Venkatesh Kothakota, and Kaustubh Patil helped implement crash consistent write back acceleration. Sameer Narkhede implemented flow control. Satyam Vaghani, Akhilesh Joshi, Eric Cohen, Alex Depoutovitch, Nakul Dhotre, and Bala Narasimhan provided feedback during the preparation of this paper. Govindarajan Soundararajan conducted experiments to evaluate the performance of remote reads after VM migration. We thank our anonymous reviewers and our shepherd, Ashvin Goel for valuable feedback.

References

[1] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-metal Performance for I/O Virtualization," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (London, England, UK), pp. 411–422, ACM, 2012.

[2] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 92–105, Jan. 2010.

[3] J. Shafer, "I/O Virtualization Bottlenecks in Cloud Computing Today," in *Proceedings of the 2Nd Conference on I/O Virtualization*, WIOV'10, (Pittsburgh, PA, USA), pp. 5–5, USENIX Association, 2010.

[4] NetApp, Inc., "Flash Cache - Clear Flash Cache - Improve Storage Performance." {http:

//www.netapp.com/us/products/storage-systems/flash-cache/}.

[5] SanDisk Corporation, "Hybrid Flash Storage Appliance :: ioControl :: Fusion-io." {http://www.fusionio.com/products/iocontrol/}.

[6] EMC Corporation, "EMC VNXe3200 Hybrid Storage - EMC Store." https://store.emc.com/Product-Family/EMC-VNXe-Products/EMC-VNXe3200-Hybrid-Storage/p/VNE-VNXe3200-Hybrid-Storage.

[7] Nimble Storage, "Data Storage Solutions — Enterprise Information Management — Flash-Based Storage - Nimble Storage." http://www.nimblestorage.com/.

[8] Saxena, Mohit and Swift, Michael M. and Zhang, Yiying, "FlashTier: A Lightweight, Consistent and Durable Storage Cache," in *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys '12)*, (Bern, Switzerland), pp. 267–280, 2012.

[9] Pure Storage Inc., "Flash array storage - all-flash enterprise array." {http://www.purestorage.com/flash-array/}.

[10] EMC Corporation, "XtremIO — All-Flash Scale-Out Enterprise Storage Arrays." http://www.xtremio.com/.

[11] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write Policies for Host-side Flash Caches," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, (San Jose, CA), pp. 45–58, USENIX, 2013.

[12] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, (Pacific Grove, CA), pp. 1–12, 2012.

[13] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 127–138, USENIX, 2013.

[14] D. Qin, A. D. Brown, and A. Goel, "Reliable write-back for client-side flash caches," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 451–462, USENIX Association, June 2014.

[15] SanDisk Corporation, "ioTurbine :: Fusion-io." {http://www.fusionio.com/products/ioturbine/}.

[16] EMC Corporation, "EMC XtremCache Server Flash Cache." {http://www.emc.com/storage/xtrem/xtremcache.htm}.

[17] Microsoft Corporation, "Microsoft Exchange Server Jetstress 2013 Tool." http://www.microsoft.com/en-us/download/details.aspx?id=36849.

[18] Jens Axboe, "fio - Flexible I/O Tester Synthetic Benchmark." http://freecode.com/projects/fio.

[19] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proceedings of the 2005 USENIX Annual Conference (USENIX ATC 2005)*, (Anaheim, CA), pp. 391–394, USENIX, 2005.

[20] VMWare, Inc., "VMWare Distributed Resource Scheduler (DRS)." {http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf}.

[21] VMWare, Inc., "VMWare Distributed Power Management." {http://www.vmware.com/files/pdf/Distributed-Power-Management-vSphere.pdf}.

[22] VMWare, Inc., "VMWare High Availability (HA)." {http://www.vmware.com/files/pdf/VMware-High-Availability-DS-EN.pdf}.

[23] VMWare, Inc., "VMWare ESX and VMWare ESXi." http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf.

[24] The Linux Foundation, "The Xen Project, the powerful open source industry standard for virtualization." http://www.xenproject.org/.

[25] Microsoft Inc., "Virtualization for your modern datacenter and hybrid cloud." http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[26] RedHat , "Kernel-based Virtual Machine." http://www.linux-kvm.org/page/Main_Page.

[27] S. B. Vaghani, "Virtual Machine File System," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 57–70, December 2010.

[28] M. Srinivasan, "Flashcache : A Write Back Block Cache for Linux." {https://github.com/facebook/flashcache/}.

[29] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Operating System Review*, vol. 36, pp. 181–194, Dec. 2002.

[30] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-scale Network File System Workloads," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, (Berkeley, CA, USA), pp. 213–226, USENIX Association, 2008.

[31] J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[32] "Iometer." {http://www.iometer.org/}.

[33] Veeam, Inc., "Veeam: Availability for the Modern Data Center." http://www.veeam.com/.

[34] Wikipedia, "Desktop Virtualization." http://en.wikipedia.org/wiki/Desktop_virtualization.