The Design and Evolution of Live Storage Migration in VMware ESX

Ali Mashtizadeh *Emré Celebi* *Tal Garfinkel* *Min Cai*
{ali, emre, talg, mcai}@vmware.com
VMware, Inc.

Abstract

Live migration enables a running virtual machine to move between two physical hosts with no perceptible interruption in service. This allows customers to avoid costly downtimes associated with hardware maintenance and upgrades, and facilitates automated load-balancing. Consequently, it has become a critical feature of enterprise class virtual infrastructure.

In the past, live migration only moved the memory and device state of a VM, limiting migration to hosts with identical shared storage. Live storage migration overcomes this limitation by enabling the movement of virtual disks across storage elements, thus enabling greater VM mobility, zero downtime maintenance and upgrades of storage elements, and automatic storage load-balancing.

We describe the evolution of live storage migration in VMware ESX through three separate architectures, and explore the performance, complexity and functionality trade-offs of each.

1 Introduction

Live virtual machine migration is a key feature of enterprise virtual infrastructure, allowing maintenance and upgrades of physical hosts without service interruption and enabling manual and automated load-balancing [1].

Live migration works by copying the memory and device state of a VM from one host to another with negligible VM downtime [2]. The basic approach is as follows: we begin by copying most of the VM state from the source host to the destination host. The VM continues to run on the source, and the changes it makes are reflected to the destination , at some point the source and destination converge – generally because the source VM is momentarily suspended allowing the remaining differences to be copied to the destination. Finally, the source VM is killed, and the replica on the destination is made live.

Earlier live migration solutions did not migrate virtual disks, instead requiring that virtual disks reside on the same shared volume accessible by both the source and destination hosts. To overcome this limitation, various software and hardware solutions to enable live migrations to span volumes or distance have been developed. One such solution is live storage migration in VMware ESX.

Live storage migration has several important use cases. First, zero downtime maintenance – allowing customers to move on and off storage volumes, upgrade storage arrays, perform file-system upgrades, and service hardware. Next, manual and automatic storage load-balancing – customers in the field already manually load balance their ESX clusters to improve storage performance and automatic storage load balancing will be a major feature of the next release of the VMware vSphere platform. Finally, live storage migration increases VM mobility in that VMs are no longer pinned to the storage array they are instantiated on.

Multiple approaches to live storage migration are possible, each offering different trade-offs by way of functionality, implementation complexity and performance. We present our experience with three different approaches: Snapshotting (in ESX 3.5), Dirty Block Tracking (in ESX 4.0/4.1) and IO Mirroring (in ESX 5.0). We evaluate each approach using the following criteria:

- **Migration time:** Total migration time should be minimized, and the algorithm should guarantee convergence in the sense that the source and destination copies of the virtual disk eventually match. We show that some algorithms do not guarantee convergence and carry the risk of not completing without significant disruption to the workload. We also emphasize predictability – Live storage migrations can take a while; predictability allows end users to better plan maintenance schedules.

- **Guest Penalty:** Guest penalty is measured in application downtime and IOPS penalty on the guest workload. All live migration technologies strive to achieve zero downtime – in the sense that there is no perceptible service disruption. However, live migration of any sort always requires some downtime during the hand-off from the source to the destination machine. Most applications can handle several seconds of downtime without any network connectivity loss. Highly available applications may only handle one or two seconds of disruption before an instance is assumed down. The final approach discussed in this paper exhibits no visible downtime and a moderate performance penalty.

- **Atomicity:** The algorithm should guarantee an atomic switchover between the source and destination volumes. This increases reliability and avoids creating a dependence on multiple volumes. Atomic switchover is a requirement to make physically longer distance migrations safe, and for mission critical workloads that cannot tolerate any noticeable downtime.

We also compare how the three approaches perform when migrating between volumes with similar and differing performance, and analyze their performance with a synthetic *online transaction processing* (OLTP) and real application (Exchange 2010) workload.

2 Design

We compare three approaches to live storage migration. The first, based on snapshots was introduced in ESX 3.5, the second based on an iterative copy with a Dirty Block Tracking (DBT) mechanism was introduced in ESX 4.0 and refined in 4.1, and the most recent approach leveraging synchronous IO Mirroring, will ship with ESX 5.0.

2.1 Background

Live storage migration can take place between any two storage elements whether over fiber channel, iSCSI or NFS.

All approaches to live migration follow a similar pattern. A virtual disk(s) is migrated (copied) from a source volume to a destination volume. Initially, a running virtual machine is using the virtual disk on the source volume. As the disk on the source is copied to the destination, bits are still being modified on the source copy. These changes are reflected to the destination so that source and destination ultimately converge. Once the two copies are identical, the running VM can be retargeted to use the destination copy of the virtual disk.

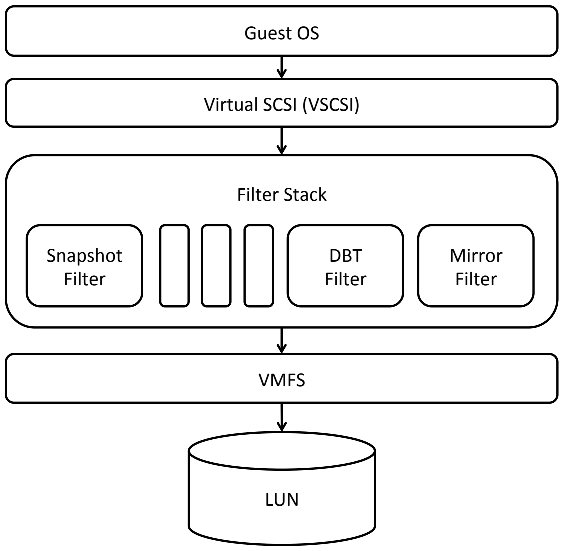

Figure 1: Simplified ESX storage architecture diagram: The guest operating system issues IO through the virtualized SCSI (VSCSI) storage stack. The IOs are passed through a stack of one or more filter drivers. As an example, this diagram shows the snapshot filter that implements the virtual disk snapshot file format. Once IOs pass through the filter stack, they are translated into file handles that are used by VMware's VMFS file-system or NFS client service.

Our migration system is built with a combination of a storage stack filter driver and a user-level thread. The user-level thread, which facilitates the migration, is a part of the VM management executive (VMX). The filter drivers are depicted in the ESX storage stack in Figure 1.

More recent architectures i.e. DBT and IO Mirroring, use the data mover (DM) copy engine that was added in ESX 4.0. The DM is a kernel service that copies disk blocks between locations with only DMAs. This eliminates user-space and kernel crossing overheads, and enables the use of copy off-load engines sometimes present in storage arrays [3].

2.2 Snapshotting

Snapshotting, our first version of live storage migration, was built to enable VMFS file system upgrades. To upgrade from VMFS version 2 to version 3, version 2 volumes were rendered read-only and virtual disks migrated onto version 3 volumes.

Snapshotting leverages virtual machine snapshots, to recap how snapshots work: when a snapshot is taken, all disk contents at snapshot time are preserved. Future modifications to the disk are logged in a separate snapshot file. Multiple levels of snapshots are possible, and multiple snapshots can be consolidated into a single disk or a snapshot by applying modifications in each to the previous snapshot or base disk. Once consolidated, intermediate snapshots can be discarded.

The migration begins by taking a snapshot of the base disk, all new writes are sent to this snapshot. Concurrently, we copy the base disk to the destination volume. Our first snapshot may reside on the source or destination volume, though the former is preferable to minimize the time the virtual disk spans two volumes.

After we finish copying the base disk, we take another snapshot. We then consolidate the first snapshot into the base disk. By the time this consolidation is complete, we expect more writes have occurred, the result is again a delta between our source and destination.

We repeat the process until the amount of data in the snapshot becomes smaller than a threshold. Once this threshold is reached, the VM is suspended, the final snapshot is consolidated into the destination disk, and the VM is resumed, now running with the virtual disk on the destination volume.

Snapshot consolidation cannot be done using an active writable snapshot due to the risk of inconsistency. Consequently, the VM must be suspended to render it inactive, resulting in downtime in our final consolidation step. Online consolidation of a read-only snapshot is possible, allowing us to implement our iterative consolidation step with minimal down time. A threshold, that can be determined dynamically, specifies when to perform the final consolidation and what the resulting downtime will be.

Snapshotting inherits the simplicity and robustness of the existing snapshot mechanism. When compared with the next design (DBT), Snapshotting shows significant resilience in the face of differing performance characteristics on the source and destination volumes. However, it also exhibits two major limitations.

First, migration using snapshots is not atomic. Consequently, canceling a migration in progress can leave the migration in an intermediate state where multiple snapshots and virtual disks are spread on both source and destination volumes. Similarly, a storage failure on either volume necessitates termination of the VM. Snapshotting is not suitable for long distance migrations to a remote destination volume, since a network outage can cause an IO stall requiring us to halt the VM. We attempt to create the initial snapshot on the source volume to help mitigate this issue.

Second, there are performance and space costs associated with running a VM with several levels of snapshots. More specifically, when iteratively consolidating snapshots there are multiple outstanding snapshots, a writable snapshot that is absorbing all new disk modifications and a read-only snapshot that is being consolidated. Using both snapshots concurrently increases memory and IO overheads during the migration. In the worst case, assuming both levels of snapshots grow to the size of the full disk, the VM may temporarily use three times its normal disk space.

2.3 Dirty Block Tracking

Our next design sought to overcome the limitations of Snapshotting, including downtime penalties from consolidation overhead and the lack of atomic switches from source to destination volumes for failure robustness.

Our approach, informed by our experience with live VM migration, uses a very similar architecture. *Dirty Block Tracking* (DBT) uses a bitmap to track modified aka dirty blocks on the source disk, and iteratively copy those blocks to the destination disk.

With DBT, we begin by copying the disk to the destination volume, while concurrently tracking dirty blocks on the source disk in the DBT kernel filter. At the end of the first copy iteration, we atomically get and clear the bitmap from the kernel filter, blocks marked in the bitmap are copied to the destination. This process is repeated until the number of dirty blocks remaining at each cycle stabilizes i.e. no forward progress is being made or a threshold based on a target downtime is reached. At this point, the VM is suspended and the remaining dirty blocks are copied.

DBT is done concurrently with bulk copying the disk contents to the destination. If a block is dirtied but not yet copied, we do not need to track that block, as it will later be bulk copied. Using this technique results in a roughly 50% speedup for the first copy iteration, assuming a workload consisting of uniformly distributed random writes, leading to an optimization we call *incremental DBT*.

DBT has several attractive properties. Operating at the block instead of snapshot granularity makes new optimizations possible. Also, DBT guarantees atomic switch-over between the source and destination volumes i.e. a VM on the source can continue running even if the destination hardware or link fail, improving reliability and making DBT suitable for migrating in less reliable conditions, such as over the WAN.

DBT also introduces new challenges. Migrations may take longer to converge if the guest workload is write intensive. If the workload on the source dirties blocks at a rate greater than the copy throughput then the migration cannot converge. The only remedy is to quiesce the guest, imposing significant downtime, or to cancel the migration.

2.3.1 Hot Block Avoidance

We present an optimization that detects frequently written blocks and defers copying them. We discuss the motivations for this optimization in section 2.3.2. In Sections 2.3.3 and 2.3.4 we present the implementation and some preliminary results. Finally, in Sections 2.3.5 we explore some of the challenges we encountered implementing this solution. Due to these challenges, this feature was never enabled by default in a shipping release. While we present these optimizations in the context of DBT, we hope to apply them to future versions of IO Mirroring.

2.3.2 Distribution of Disk IO Repetition

Real-world disk IO often exhibits temporal and spatial locality. To help us better understand locality in a common enterprise workload, we analyzed VSCSI traces from an Exchange 2003 workload with 100 users.

Our workload was generated using the Exchange Load Generator [4] in a VM configured with three disks: a system disk of 12GB with Windows 2003 server, a mailbox disk of 20GB, and a log disk of 10GB. Exchange is configured to use circular logs.

Our results are shown in Figure 2 that plots logical block numbers (LBNs) sorted by decreasing popularity i.e. most to least frequently written blocks, for all three disks. All traces follow a zipf-like distribution. Once hot blocks are identified, we can ignore them during the iterative copy phase, and defer copying of hot blocks to the end of the migration. This eliminates numerous repeated copies that reduces IO costs and overall migration time.

2.3.3 Multi-stage Filter for Hot Blocks

We collect data from the DBT filter to identify candidate hot blocks. Using a hash table to index the repetition counters for large disks would be quite memory intensive. Therefore, we use a multi-stage filter [5] to identify blocks that have been written at least t times, where t is a threshold. The multi-stage filter is similar to a counting version of bloom filter, which can accurately estimate the dirty blocks. Multi-stage filters provide a compact representation of this data.

Our multi-stage filter has n stages. Each stage includes an array of m counters and a uniform hash function H_i, where $0 \leq i \leq n$. When a block with LBN x gets modified, n different hash values of a block are calculated, and the corresponding counters in all stages are increased. The hotness of a block can be determined by checking the counters of the block in all stages. When all counters are greater than a threshold t, the corresponding block is considered to be hot. Since the collision probability of all n counters decrease exponentially with n, the multi-stage filter is able to filter out hot blocks accurately with limited memory.

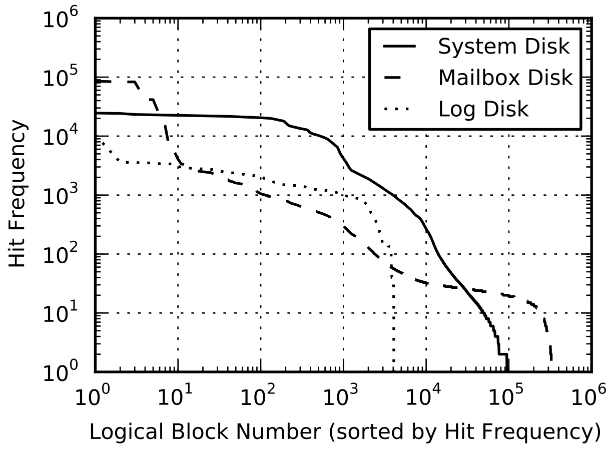

Figure 2: Distribution of Disk Write Repetition of Exchange Server Workload. The x-axis shows the logical block number (LBN) of the disk sorted by hit frequency. The y-axis shows the hit frequency for disk blocks from hottest to coldest. The writes follow a zipf-like distribution.

2.3.4 Analysis of Hot Block Avoidance

Our hot block avoidance algorithm uses the heat map from our multi-stage filter to determine which blocks are hot. Sampling is done during the initial copy phase. In the iterative copy phase, we query the multi-stage filter and defer copying of the hot blocks. At the end of the migration, hot blocks are copied, prior to the last copy iteration.

To appreciate the potential benefits, consider the distribution of write frequencies for the Exchange workload shown in Figure 2. Several hundred megabytes of blocks are hot, ignoring these until the final copy iteration can yield substantial benefits.

These benefits can be seen in Figure 3, a migration using same Exchange workload described previously, with and without our hot block and incremental DBT optimizations. The initial copy phase is not shown, as it is independent of optimizations. Shorter bars on the left represent a migration with optimizations, the taller bars on the right, without. Iterations 5 through 10 are not present for our optimized case, since incremental DBT and hot block avoidance eliminates the need for those iterations.

The consistent height of the bars with the red hatch pattern shows the hot block avoidance algorithm detected the approximate working set correctly. Note that the blocks labeled with the red hatch pattern are *not copied*

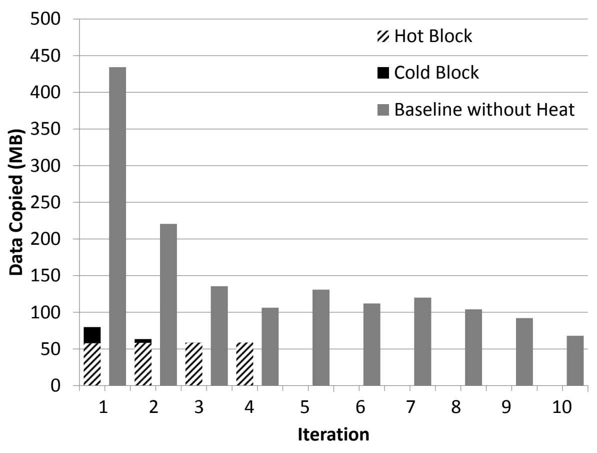

Figure 3: Dirty blocks copied vs. Iteration number. Exchange workload migration with and without our hot block and incremental DBT optimizations. Shorter bars on the left represent a migration with optimizations, the taller bars on the right, without. The initial copy phase is not shown, as it is independent of optimizations. Iterations 5 through 10 are not present for our optimized case, since hot block avoidance eliminates the need for those iterations.

for the first two iterations. We displayed the graph like this to make it easier to see the remaining blocks that need to be copied. Most of the blocks copied in third iteration are so hot that they are necessarily copied again during the switchover and completion of the virtual machine. In our implementation we attempted to copy the hot blocks in next to last iteration in order to reduce the remaining blocks for the final iteration. This is critical because the final iteration occurs while the VM execution is suspended and accounts for our downtime.

Incremental DBT saves more than 50% on the first iteration. Our workload issues more writes towards the end of the disk, and incremental DBT allows us to ignore those blocks during our iterative copy phase, allowing them to be taken care of by the bulk copy of the base disk that is happening concurrently.

2.3.5 Problems with Hot Block Avoidance

Our hot block avoidance algorithm performed well in most scenarios, however, we encountered several problems. First, we found hot block avoidance complex to tune. Success was hard to reason about and we were concerned about harming the performance of untested workloads.

Hot block avoidance also consumed significant amounts of memory. Even with the multi-stage filter, we could envision large VMs consuming upwards of a gigabyte of memory. Our customers, many of whom run with memory overcommitted, could find additional memory pressure problematic.

Finally, some workloads e.g. the OLTP workload discussed in our evaluation, have little or no temporal locality, and thus receive minimal benefit from this optimization.

2.4 IO Mirroring

DBT is an adaptation of the technique used for live virtual machine migration, namely, iterative pre-copying of virtual machine memory pages. While DBT has benefits over Snapshotting, they come at the cost of complexity. To improve on DBT, we note a critical distinction between virtual memory and storage systems.

Virtual machine memory accesses are usually transparent to the hypervisor and write traps are quite expensive. Consequently, write traps are used only to note if an already copied page has again been dirtied – i.e. only the first write to a copied page is trapped – necessitating an iterative copying approach where all the writes to a page of a given "generation" are captured by copying the entire dirty page. In contrast, intercepting all storage writes is relatively cheap. Our next approach leverages this observation, using a much simpler architecture based on synchronous write mirroring.

IO Mirroring, our most recent architecture, works by mirroring all new writes from the source to the destination concurrent with a bulk copy of the base disk. We again use a filter driver as shown in Figure 1. Our bulk copy process is implemented using the VMKernel data mover (DM). We drive the copy process from user-level by issuing DM operations. The DM issues reads and writes directly to the underlying file without the intervention of the filter driver. Thus, if a VM could issue a write while a DM read operation is in progress, without a synchronization mechanism we would copy an outdated version of the disk block. To prevent this situation, the filter driver implements a synchronization mechanism to prevent DM and VM IOs to the same region. When a DM operation is issued first the filter acquires a lock on the region in question, and then releases it on completion.

Locking in the IO Mirroring filter driver works by classifying all VM writes into one of three types: writes to a region that has been copied by the DM, writes to a region being copied by the DM, and writes to a region that will be copied by the DM. Two integers are used to maintain the disk offsets that delineate these three regions.

Writes to a region that has already been copied will be mirrored to the source and destination – as the DM has already passed this area and any new updates must be reflected by the IO Mirror. Writes to the region currently being copied (in between the two offsets) will be deferred and placed into a queue. Once the DM IO completes we enqueue those writes and unlock the region by updating the offsets. As part of the updating operation we wait for inflight writes to complete. The final region is not mirrored and all writes are issued to the source only – as eventually any changes to this area will be copied over by the DM. Reads are only issued to the source disk.

IO Mirroring fulfills all of our criteria, guaranteeing an atomic switchover between the source and destination volumes, making this method viable for long distance migrations. It also guarantees convergence as the mirrored VM IOs are naturally slowed down to the speed of the slower volume.

2.5 Implementation Experience

Snapshotting benefited from leveraging the existing snapshot mechanism, making it simpler to implement, and easier to bring into a hardened production quality state. Further, it was the only approach that was feasible for the file system upgrade use case that originally motivated its creation. For this use case, the source virtual disk lives on an older read-only file system, and both DBT and IO Mirroring require a writable source disk. Finally, it required no complex tuning. Unfortunately, it also inherited substantial limitations from leveraging snapshots, most notably the atomicity, and the performance limitations of snapshots.

DBT overcame most of those performance inadequacies, but introduced many parameters that required significant engineering effort to tune. Specifically, for the convergence detection logic that determines whether a migration needs additional copy iterations, is safe to complete, or needs to be terminated.

This logic required several threshold values to determine whether the remaining dirty blocks at the end of the iteration seem to be getting smaller. If the algorithm detects a significant reduction in the remaining dirty blocks, it continues for another iteration. If there is no discernible reduction or possibly an increase, the algorithm determines whether to complete or abort the migration.

Two scenarios occur often enough that may cause a noticeable increase in the dirty blocks remaining. First, the workload may make a burst of changes e.g. a database may flush its buffer cache, causing the migration progress to temporarily regress. To handle this, the algorithm monitors progress for the last two copy iterations. Analyzing the last two iterations prevents nearly all migration aborts due to workload spikes. The second cause of failure to converge is a slow destination. If the workload running in a VM is too fast for the destination the migration will terminate. There is no solution other than to ask the user to quiesce such workloads manually.

IO Mirroring removed all of the tunable parameters and convergence logic. Using a synchronous mirror naturally throttles the workload to the speed of the destination volume. Switching to this approach eliminated significant engineering and performance testing effort. To our surprise, customers seemed most interested in the predictability aspect of IO Mirroring, as it allows them to better plan their maintenance schedules.

3 Evaluation

We evaluated total migration time, downtime, guest performance penalty and convergence, using synthetic (Iometer [6]) and real application (Exchange 2010) workloads for each of our architectures. We also present the IOPS profile for each architecture over the duration of a migration.

Our synthetic workload uses Iometer in a VM running Windows Server 2003 Enterprise, we varied disk size and outstanding IOs (OIOs) to simulate workloads of varying size and intensity. The IO pattern simulates an OLTP workload with 30% write, 70% read of 8KB IO commands with a 32GB preallocated virtual disk. We used Exchange 2010 for our application workload with loads of 44 tasks/sec and 22 tasks/sec.

Snapshotting and DBT were evaluated using ESX 4.1. IO Mirroring was evaluated using a pre-release version of ESX 5.0. Our snapshot implementation was first available in ESX version 3.5 however, we used the version in ESX 4.1 that included support for the DM and other major performance improvements to get a more fair comparison with DBT.

Our synthetic workload ran on a Dell Poweredge R710 server with dual Intel Xeon X5570 2.93 GHz processors, and two EMC CX4-120 arrays connected to the server via 8Gb Fibre Channel(FC) links. We created 450GB sized VMFS version 3 volumes on each array. Our test VM has a 6GB system disk running Windows Server 2003 and Iometer, and a separate data disk. The Snapshotting implementation requires all disks to move together to the same destination. For a fair comparison we migrated the system and data disk for all architectures.

Our application workload ran on a Dell PE R910 4 socket 8-core Intel Nehalem-EX processor with 256GB of memory. Migration is done with 6 disks in RAID-0 configuration on the same EMC CX3-40 Clariion array with separate spindles. Our Exchange 2010 VM is configured with 8-vCPU with 28GB of memory and contains multiple virtual disks. We only migrated the 350GB Mailbox disk containing 2000 user mailboxes. We omitted Snapshotting from this workload because it requires all disks to be migrated together.

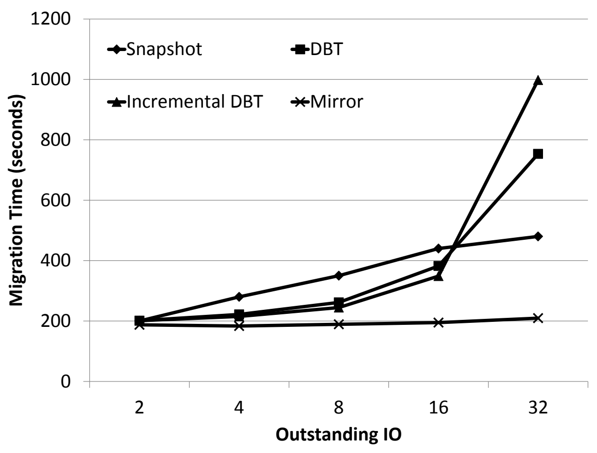

Figure 4: Migration Time vs. OIO. IO Mirroring exhibits the smallest increase by 11.8% at most. DBT variants exhibit the largest increase up to 2.74x and 3.96x. At 32 OIOs, the DM struggles to keep up with the workload's dirty rate. Snapshotting exhibits a 1.4x increase.

3.1 Migration Time

Minimizing total migration time reduces the impact on guest workloads, decreases response time for maintenance events, and makes load balancing more responsive. Ideally, migration time should not vary when workload size and intensity change, as this makes migrations more predictable, easing manual and automated planning.

Total migration time vs. OIOs for our synthetic workload is shown in Figure 4. For OIOs less than 16, each architecture performs better than the previous one. Incremental DBT does marginally better than DBT, because the incremental dirty block tracking improvement reduces the number of dirty blocks copied in the first iteration. The VMKernel data mover (DM), used by both DBT variants, supports a maximum of 16 OIOs. Consequently, for workloads with more than 16 OIOs, Snapshotting outperforms both DBT variants, which has a bottleneck on the DM. IO Mirroring consistently offers the lowest total migration time.

IO Mirroring also offers the smallest change in migration time under increasing load, as we see in Figure 4. Migration time only grows by 11.8% when changing OIOs from 2 to 32, a 4.9x increase in guest write and read throughputs. In contrast, migration time increases by 1.4x for Snapshotting, and 2.74x and 3.96x for DBT and incremental DBT. IO Mirroring is less sensitive to OIOs because it implements a single pass copy operation. The increased IO slows that single pass copy rather than inducing additional copy iterations.

For comparison with the ideal case, we performed an off-line disk copy with a 32GB virtual disk and 6GB system disk, it took 176 seconds on the same hardware setup. Migration with IO Mirroring with 2 OIO and 32 OIO OLTP workloads took only 5.8% and 15.7% longer to complete.

Total migration time vs. disk size for our synthetic workload is shown in Figure 5. Again, each architecture migrates faster than the previous one. Generally migration time grows linearly with disk size however, for 64GB disks, Snapshotting performs worse than expected. This occurs because all subsequent snapshots grow in size leading to an increase in the number of snapshot creation and consolidation iterations. IO Mirroring exhibits minimal change as the disk size increases, with migration time growing less than 4%. Our figures include the 6GB system disk's migration time. Thus, the migration time for the 4GB and 64GB data disk tests corresponds to a seven fold increase.

Our Exchange workloads are depicted in Table 1. For the initial run, Incremental DBT offers a 11.2% reduction

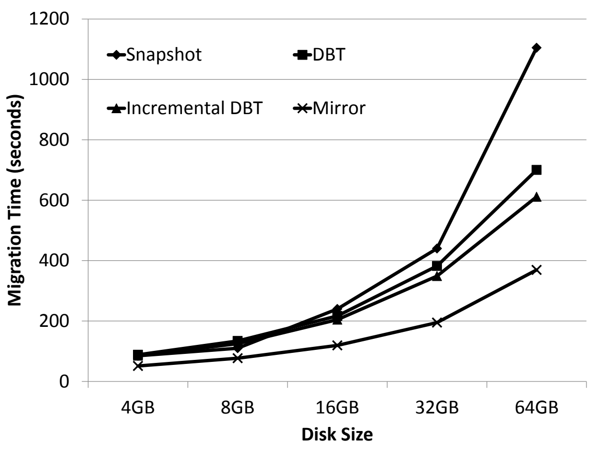

Figure 5: Migration Time vs. Disk Size. The x-axis denotes only the data disk size. Migration time includes the additional 6GB system disk. The migration times of IO Mirroring grows less than 3.4% with increasing disk size. DBT, Incremental DBT and Snapshotting takes 13%, 2.4% and 85% longer than expected.

Type	Migration Time	Downtime
DBT	2935.5s	13.297s
Incremental DBT	2638.9s	7.557s
IO Mirroring	1922.2s	0.220s
DBT (2x)	Failed	-
Incremental DBT (2x)	Failed	-
IO Mirroring (2x)	1824.3s	0.186s

Table 1: Migration time and downtime for DBT, Incremental DBT, and IO Mirroring with the Exchange workload. The double intensity version only completes with IO Mirroring.

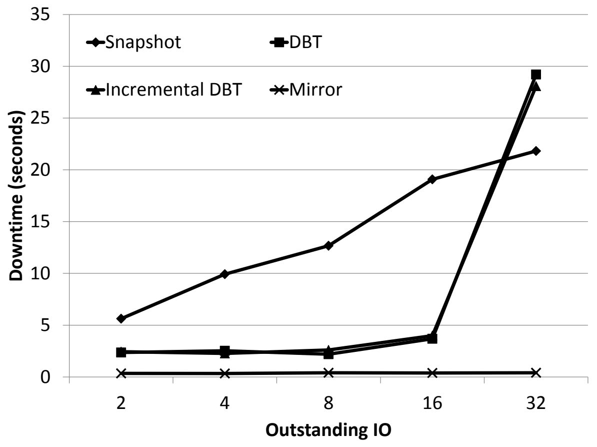

Figure 6: Downtime vs. OIO. IO Mirroring exhibits near constant downtime below 0.5s. DBT variants exhibit moderate downtime until OIO is 32 when the DM becomes a bottleneck. Snapshotting exhibits the worst downtimes except when OIO is 32.

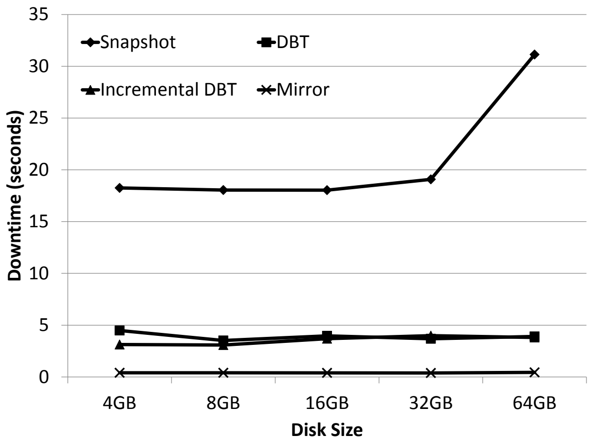

Figure 7: Downtime vs. Disk Size. IO Mirroring exhibits near constant downtime under 0.5s. DBT variants exhibit downtimes below the 5s convergence logic threshold. Snapshotting shows significant degradation beyond 32GB as snapshot overheads become prominent.

in migration time compared to DBT. IO Mirroring reduces the migration time by 52.7% compared to DBT. IO Mirroring is the only architecture to complete the double intensity workload successfully. The migration time appears lower, but the 5% difference is within the noise margin.

Overall, IO Mirroring exhibits the least change across workload intensity and disk size variations, while DBT and Snapshotting migration times increase significantly with such variations.

3.2 Downtime

While outages up to five seconds and beyond can be tolerated by some applications, others such as highly available applications, audio and video make even sub-second outages noticeable. Therefore we prefer to minimize downtime.

Downtime vs. OIOs is shown in Figure 6. With IO Mirroring downtime increases with increasing OIO by one tenth of a millisecond. This slight increase is due to the additional time required to quiesce the VM IO. There is no other downtime dependence on OIO for this architecture.

Both DBT variants choose their final copy thresholds with the intention of keeping downtimes under five seconds. Usually the algorithms overestimate, putting downtime consistently in the two to three second range for OIO under 8. From 8 to 16 we see that the downtime increases slightly as the DM begins to struggle to keep up. From 16 to 32, we see that the downtime jumps to values greater than 28 seconds.

The reasons for this are two fold, first the DM becomes a serious bottleneck however, our convergence logic also takes into account total migration time. If we were willing to wait an additional, potentially much longer time to converge, we might end up with a smaller final dirty page set, resulting in a shorter downtime.

Snapshotting shows a near linear growth in downtime as the workload increases, better than the DBT variants for the OIO equal to 32 case because the snapshot approach is consolidating the final snapshots on the destination volume.

Figure 7 shows the downtime as a function of disk size. Both DBT and IO Mirroring scale well with disk size. Snapshotting shows significant growth in downtime and total migration time when moving 64GB disks. As disk size increases, each snapshot create and consolidate iteration takes longer since there is increased disk fragmentation, increased virtual disk block lookup overhead, and other overheads related to the implementation of snapshots that accrue. Migration times also increase because the number of snapshot create and consolidate operations has to increase to keep downtime low.

Our Exchange workload shown in Table 1 exhibits larger downtimes for DBT and incremental DBT of roughly 8 and 13 seconds. For the double intensity workload, the DBT variants do not converge. IO Mirroring completes the migration for both the normal and double intensity workloads with roughly 0.1s and 0.2s of downtime.

IO Mirroring guarantees small constant downtimes. DBT variants offer low downtimes if the DM can keep up with the workload, but a slow destination volume or high

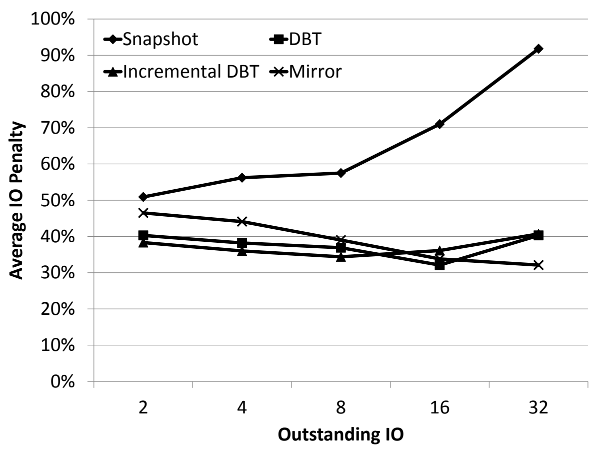

Figure 8: Average Guest IO Penalty vs. OIO. IO Mirroring exhibits lower penalties as OIOs increase as guest IOs come to dominate DM IOs. DBT variants exhibit similar pattern until 32 OIO when iterative copy times become an increasing portion of the overall cost. Snapshotting costs increase consistently as snapshot overheads worsen with increased OIO.

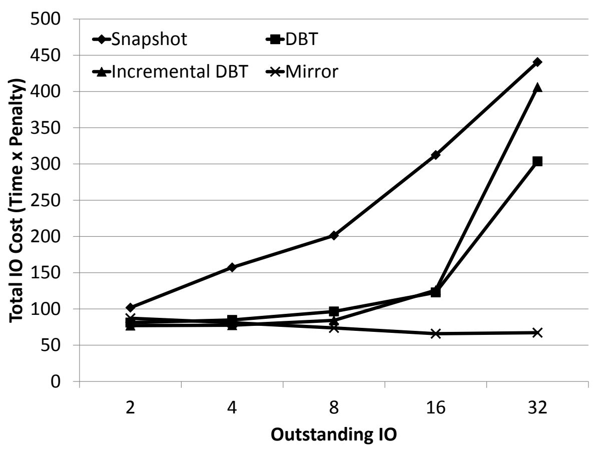

Figure 9: Aggregate migration cost (*total migration time* $\times$ *instantaneous penalty*) vs. OIO. IO Mirroring exhibits the lowest aggregate migration costs as OIO increases. As migration times stay near constant, average penalty decreases. DBT variants exhibit higher costs especially for 32 OIOs since migration times increase significantly. Snapshotting exhibits the greatest penalty as migration time and instantaneous penalty grow with OIO.

intensity workload may make that impossible. Snapshotting tends to have the highest downtimes.

3.3 Guest Performance Penalty

Minimizing the guest IO penalty and total cost during a migration improves user experience and lessens the impact on application IO performance. Average guest IO penalty vs. OIO for our synthetic workload is shown in Figure 8. The percentage IO penalty is identical for read and write IOs.

Snapshotting incurs the largest penalty because it uses snapshots, where IOs issued to new blocks require allocation and maintenance of indirection data structures. As OIOs increase, these overheads increase proportionally. If the workload runs long enough, these penalties will eventually start to amortize. However, since migration times are relatively short, these penalties remain high.

Penalties for DBT variants hover around 39%, due to guest IO competing with IO induced by the DM. With increased OIO, guest IO takes an increasing share of IO away from the DM. This causes slightly longer migrations. We observe slightly less IO penalty on the guest up until 16, since the DM has a maximum of 16 OIOs.

At 32 OIO the DBT variants show an increased IO penalty. The average instantaneous penalty is a combination of the penalty during the initial copy and the subsequent iterative copy phases. In the iterative copy phase, the IO penalty tends to be much higher than in the initial copy phase, because the DM is inducing a random IO workload on both volumes. In the 32 OIO case, the migration time increases significantly as shown in Figure 4. Consequently, the iterative copy phase contributes substantially more to the instantaneous penalty.

IO Mirroring begins with a penalty greater than both DBT variants but less than Snapshotting. Figure 8 shows that, as the workload OIOs increase the IO penalty becomes smaller than that of the DBT. The IO Mirroring architecture does not contain an iterative copy phase, thus it does not suffer from the higher performance penalties that DBT does with 32 OIOs. The only impact on the workload, assuming the destination is not a bottleneck, is due to the DM, we have a monotonically decreasing plot for performance penalty, because the workload consumes a proportionally larger share of the throughput.

Figure 9, shows the total guest penalty vs. OIOs. Penalty is measured in units of time, the total IOPS lost as if the workload was not executing for that many seconds. This shows that Snapshotting incurs the worst penalty amongst other migration implementations. The DBT variants both have similar migration penalties. Finally, IO Mirroring starts off with a penalty very similar to that of the DBT variants and gradually decreases as OIOs of the workload increase.

Exchange workload runs initially did not appear to show any degradation in throughput or IOPS. A closer look revealed that while average read latency remained the same, there was a slight increase in average write latency values per IO. However, write latency did not im-

Type	Latency	Variance	Penalty
No migration	0.777 ms	0.106	-
DBT	3.622 ms	0.363	4.7x
Incremental DBT	3.571 ms	0.544	3.6x
Mirror	3.338 ms	0.362	3.3x
Iterative Copy Phases			
DBT	5.922 ms	1.550	6.6x
Incremental DBT	5.171 ms	1.468	5.7x

Table 2: Comparison of changes in write latency observed on the Exchange 2010 workload.

pact IOPS or throughput, because the array could still consume the increased IO rate during the migration operation.

In Table 2, we list the write latency values observed during the migration of Exchange 2010 workload. The first number corresponds to the no migration case. The next three numbers corresponding to the migration architectures are obtained from the same amount of samples, during the copy phase of the disk. IO Mirroring completes as soon as the copy phase completes. The last two latency values are observed during the iterative copy phase of the DBT and incremental DBT methods, where the incremental DBT completes sooner than the DBT as shown in Table 1.

3.4 Convergence

We define convergence as an architecture's ability to reduce the number of blocks that differ between a source and destination to a level where a migration can complete with an acceptable downtime. We discuss convergence in our previous experiments, then examine a migration from a faster to slower volume, which is a challenging case for DBT variants. We omit Snapshotting from our discussion, as it always converges, but may still incur significant downtimes.

Every migration of the synthetic workload in the previous sections completes successfully however, when run with 32 OIOs, both DBT variants shows excessive downtimes and migration times. Downtimes larger than five seconds imply that the DBT convergence detection algorithm notices when the workload is not converging fast enough or at all, and considers aborting the migration. When the expected downtime is short enough the convergence logic forces the migration to complete, inflicting that downtime on the guest.

We know that DBT variants are not guaranteed to converge. For DBT to converge, it requires that the workload dirties blocks at a slower rate than the copy rate, and the destination volume should not be slower than the source.

An example of our DBT variants being overwhelmed

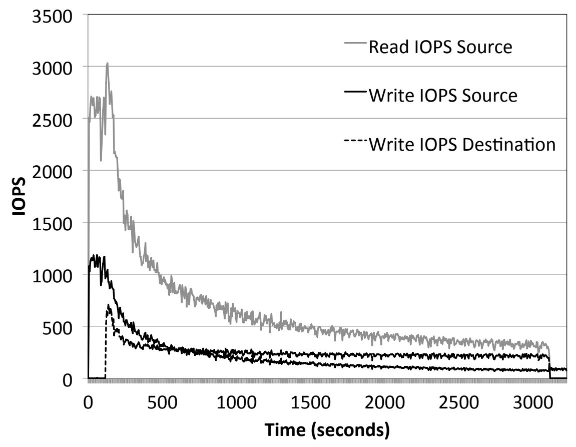

Figure 10: Graceful throttling of the 16 OIO synthetic OLTP workload during a migration from an FC to a slower NFS volume. The workload begins with approximately 2600 read and 1100 write IOPS and at the destination the workload runs at 200 write and 95 read IOPS.

is found among the 2x intensity Exchange workloads in Table 1, where neither variant completes successfully. Nevertheless, IO Mirroring completes the migration with negligible downtime. When running the normal intensity workload, all architectures complete their migrations successfully, but with significant downtime for the DBT variants.

In contrast, IO Mirroring converges even with a 10x slower destination volume. This is shown in Figure 10, where the total source and destination read and write IOPS observed during a migration from a Fibre Channel attached volume to a slower NFS attached volume, using our synthetic OLTP workload with 16 OIOs. The graph also shows that when IO Mirroring is involved, the gradual slow down is not linear but instead governed by the saturation of the OIO on the destination.

The left hand side of the graph shows the starting IOPS of the workload, which is approximately 2500 for read and 1000 for write. The shape of the gradual diminishing of IOPS may not be immediately clear. One may think that as an increasing percentage of IOs are mirrored the workload should also slow down linearly. That is not the case, because when the latency of the destination is more than an order of magnitude slower, the IOs on the destination will saturate the device. The mirror driver waits for write IOs on the destination to complete, while IOs on the source get acknowledged much faster. If we were to examine the number of OIOs at each device we would see that the source has only a few IOs while the destination is full. Equation 1, allows us to compute the approximate IOPS as a function of the OIOs, latency, and percentage of IOs sent to the destination (*DestIOPct*).

$$IOPS \approx Min(\frac{OIO}{DestIOPct \times Lat_{Dst}}, \frac{OIO}{Lat_{Src}}) \qquad (1)$$

The migration starts after the first two minutes of stable IO in the graph, with a dip due to the IO Mirroring instantiation, followed by a rapid increase in read IOPS on the source. The read IOPS consists of the aggregate VM and migration read IOs. Similarly, the destination IOPS consists of storage migration and VM mirror write IOs. At the end of the migration, the VM IO workload stabilizes at approximately 200 read and 95 write IOPS.

When we run this benchmark using DBT, the migration fails after several iterations because it is unable to converge as the workload dirties blocks at a rate faster than the copy rate. Using IO Mirroring the workload is naturally throttled to the performance of the destination volume.

3.5 Anatomy of a Migration

Looking at IOPS over time provides unique insights about migration behavior. Figure 11, shows Time vs. IOPS seen by the host for all three migration architectures. The labels *S* and *D* refer to the region where the VM is the sole IO source. The initial disk copy is marked by the region *M*. In all three graphs the throughput for the initial disk copy is roughly the same. This suggests that the time it takes to clone a virtual disk has a constant cost, where enhancements such as hardware off-loading may be helpful.

With Snapshotting, while the disk is being copied, an initial snapshot is created on the source volume. Label C_S corresponds to the process of consolidating this initial source snapshot into the base disk. In region C_S the VM is running on a destination snapshot. In region C_D, we are creating and consolidating multiple snapshots on the destination. Each spike in IOPS marks the consolidation of the previous snapshot, and we see the slight degradation in random read/write IO as we access deeper offsets of a snapshot, until the next snapshot is created. Overall IOPS also increases due to the use of smaller snapshot sizes, resulting in few disk seeks.

DBT has a slightly better utilization of IO during the disk copy, because only a dirty block tracking bitmap is maintained in the kernel. In the region marked *DBT*, the iterative copy process begins. We see a reduction in throughput because the iterative copy process consists of random access to dirtied blocks. Together with the random IO workload, the total disk access pattern on the source becomes more random. Incremental DBT, which is not shown in Figure 11, shrinks the region *DBT* considerably with the optimization previously explained in Section 2.3.

IO Mirroring consists of a single pass copy, represented by region *M*. In this region, IO Mirroring shows a slightly lower utilization towards the end of the migration, because within the copy process, write IO is also mirrored to the destination volume. By the end of the migration all IOs will be mirrored. This method is shorter since there is no copy process that follows.

4 Related Work

Clark et al. implemented VM live migration on top of the Xen virtual machine monitor (VMM) [7]. Xen uses a pre-copy approach to iteratively copy memory pages from source host to destination host. It supports both managed and self migrations. Managed migration is performed by migration daemons running in the management VMs on the source and destination hosts, while self migration places the majority of the implementation within the guest OS of the VM being migrated. The guest OSes running on Xen are para-virtualized and are aware that a migration is in progress. The guest OS is able to improve migration performance by stunning rogue processes and freeing page cache pages. The authors mention, as possible future work, leveraging remote mirroring or replication to enable long distance migrations.

VMware VMotion [2] was first introduced in VMware ESX hypervisor and VirtualCenter suite in 2003. It supports the live migration of VMs among physical hosts that have access to shared storage, such as *storage area network* (SAN) or *network attached storage* (NAS). In VMotion, only VM memory pages are transferred from source host to destination. The virtual disks of a VM have to reside on shared storage and cannot move. VMware VMotion is the basic building block for VMware's Distributed Resource Scheduler (DRS) that dynamically balances the workloads between a set of hosts in a cluster, and for Distributed Power Management, that uses migration to consolidate workloads to reduces power consumption during off-peak hours [1].

In addition to the pre-copy approaches, Hines et al. proposed a post-copy approach for live VM migration [8] for write-intensive workload by trading off fast migration time with atomic switchover between source and destination hosts. Pre-paging and self-ballooning mechanisms are also used to facilitate the post-copy approach. Pre-paging utilizes locality of access to reduce the number of network page faults. Self-ballooning is technique to remove unneeded buffer cache pages from the guest so they will not be transfered.

VM live migration has also been extended from *local-area networks* (LAN) to *wide-area networks* (WAN) for various use cases, including data center load balance and disaster recovery. Bradford et al. extends the live migration in Xen to support the migration of a VM with

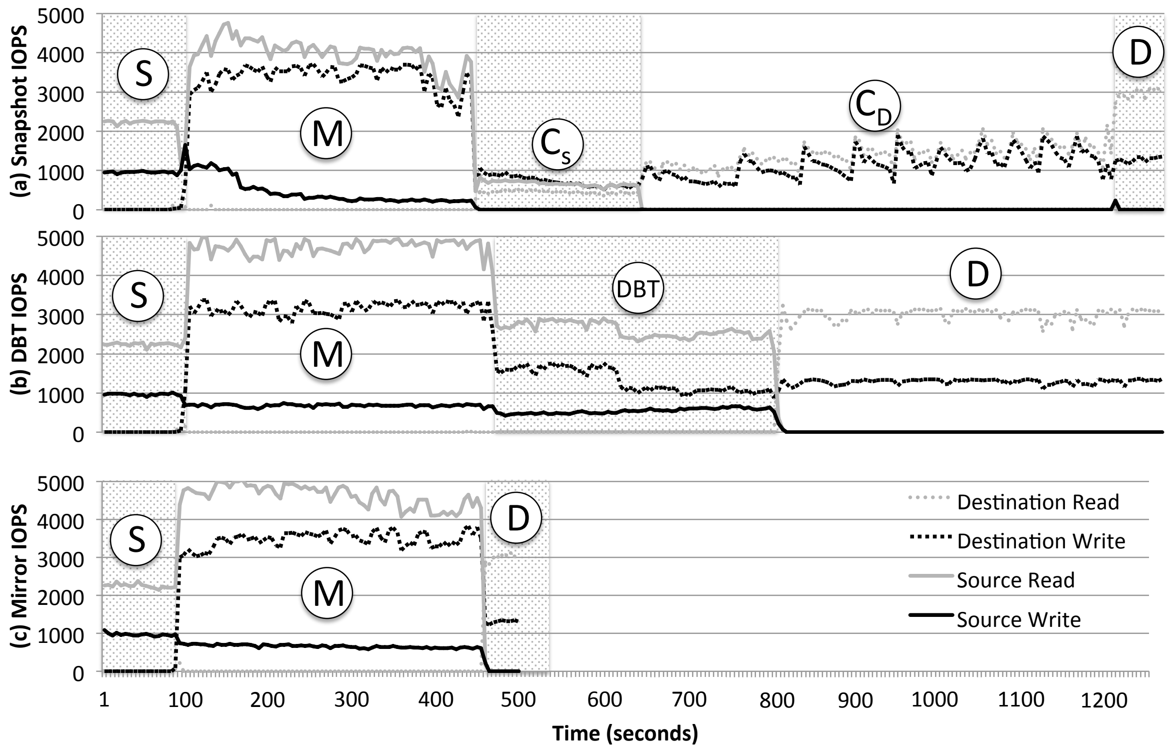

Figure 11: The IOPS observed for the duration of the three main architectures. region *S*, is prior to the migration, and region *D*, corresponds to the post migration. Region *M* in all graphs refers to the copying process of the virtual disks. In the Snapshot graph, the area denoted as C_S and C_D correspond to the consolidation of the source and destination snapshots. Finally, the shaded region *DBT* refers to the iterative copy operation.

local disk states across WAN [9]. When a VM is being migrated, its local disks are transferred to destination volume using a block level disk pre-copy. The write IO workload from the guest OS is also throttled to reduce the dirty block rate. Existing network connections to the source VM are transparently forwarded to the destination by using IP tunneling, while new connections are redirected to the destination network using Dynamic DNS. Further optimizations for wide area storage migration are explored by Zheng et al. [10]

VM Turntable demonstrates live VM migration over WAN links for Grid computing [11]. In this system, dedicated paths are dynamically setup between distant sites. However, no local disk state is transferred over the WAN in VM Turntable. CloudNet builds a private network across different data centers using MPLS based VPN for live WAN migration of VMs [12]. Disk state for VMs is replicated to the destination volume using both asynchronous and synchronous schemes in different migration stages.

Storage mirroring or disk shadowing is an existing concept that was first explored for building redundant disks [13]. Mirroring for redundancy and performance reasons is very common in volume managers such as LVM [14] and Vinum [15]. Network based mirroring for replication and virtually shared volumes that span multiple physical hosts is done by distributed block devices like DRBD [16] and HAST [17].

Live storage migration of VMs using IO Mirroring is discussed in Meyer et al. [18]. This work presents a modular storage system that allows different storage systems to be implemented by configuring a set of components.

SAN vendors provide volume level mirroring and replication for redundancy such as NetApp's SnapMirror product [19] and EMC's Synchronous and Asynchronous SRDF [20]. The concept of LUN migration is also provided by some of the storage array vendors that is similar to the problem of virtual disk migration. For example, Data Motion [21] from NetApp uses asynchronous SnapMirror to initiate the copy and achieve a small target *recovery point objective* (RPO) time. During the final switchover the NetApp LUN may be unavailable for up to 120 seconds. If the Data Motion is unable to complete within that time it cancels the LUN migration [21]. The NetApp Data Motion product functions similar to dirty block copy approach discussed in Section 2.3 and does not appear to meet our goals of no downtime and guaranteed success.

5 Conclusions

We present our experience with the design and implementation of three different approaches to live storage migration: Snapshotting (in ESX 3.5), Dirty block tracking (in ESX 4.0/4.1) and IO Mirroring (in ESX 5.0). Each design highlights different trade-offs by way of impact on guest performance, overall migration time and atomicity.

The first two approaches exhibit several shortcomings that motivated our current design. Snapshotting imposes substantial overheads and lacks atomicity, this hinders reliability and makes long distance migrations fragile. DBT adds atomicity, and by working at the block level allows a number of new optimizations, but also cannot guarantee convergence and zero downtime for every migration.

While the IOPS penalty caused by Snapshotting to the OLTP workload is around 70%, DBT and IO Mirroring reduce this penalty to around 32% and 34%. The total penalty for IO Mirroring is approximately 2x better than DBT.

Our latest approach based on IO Mirroring offers guaranteed convergence, atomicity and zero downtime with only a slightly higher IOPS penalty than DBT. When moving a virtual disk to a slower volume, IO Mirroring exhibited a graceful transition and completion. We achieved consistent reduction in total migration time, bringing the total live migration duration close to that of a plain disk copy. For the OLTP workload, IO Mirroring, takes less than half the time of Snapshotting and only 9.7% longer than an off-line virtual disk copy. We showed that under varying OIO and disk size, IO Mirroring offered very low variation in migration time, downtime, and guest performance penalty.

Acknowledgements

We would like to thank Vincent Lin for running Exchange workloads; Irfan Ahmad, Joel Baxter, Dilpreet Bindra, Kit Colbert, Christian Czezatke, Ajay Gulati, Jairam Ranganathan, Mayank Rawat, Yan Tang, Jayant Kulkarni, Murali Vilayannur, Swathi Koundinya and Chethan Kumar for their valuable discussions and feedback with the original work.

References

[1] "VMware Infrastructure: Resource Management with VMware DRS," Aug. 2010. `http://www.vmware.com/pdf/vmware_drs_wp.pdf`.

[2] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2005.

[3] "vstorage api for array integration and vmware ready," `http://www.vmware.com/partners/programs/vmware-ready/vstorage-api-arrays.html`.

[4] "Exchange Load Generator." `http://technet.microsoft.com/en-us/library/bb508866(EXCHG.80).aspx`.

[5] Cristian Estan, et. al., "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice," *ACM Transactions on Computer Systems*, 2003.

[6] "Iometer." http://www.iometer.org.

[7] C. Clark, et. al., "Live Migration of Virtual Machines," in *NSDI*, Oct 2005.

[8] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 14–26, 2009.

[9] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, (New York, NY, USA), pp. 169–179, ACM, 2007.

[10] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, (New York, NY, USA), pp. 133–144, ACM, 2011.

[11] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y. Wang, "Seamless live migration of virtual machines over the man/wan," *Future Gener. Comput. Syst.*, vol. 22, no. 8, pp. 901–907, 2006.

[12] T. Wood, K. Ramakrishnan, J. van der Merwe, and P. Shenoy, "Cloudnet: A platform for optimized wan migration of virtual machines," *University of Massachusetts Technical Report TR-2010-002*, January 2010.

[13] D. S. Dina, D. Bitton, and J. Gray, "Tandem tr 88.5," 1988.

[14] "Logical volume manager (linux)." `http://en.wikipedia.org/wiki/Logical_Volume_Manager_(Linux)`.

[15] "The vinum volume manager." `http://www.vinumvm.org/`.

[16] "Distributed Replicated Block Device (DRBD)." `http://www.drbd.org/`.

[17] "Hast - highly available storage." `http://wiki.freebsd.org/HAST`.

[18] D. Meyer, B. Cully, J. Wires, N. Hutchinson, and A. Warfield, "Block mason," in *WIOV '08: First Workshop on I/O Virtualization*, 2008.

[19] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara, “Snapmirror: File-system-based asynchronous mirroring for disaster recovery,” in *FAST ’02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), p. 9, USENIX Association, 2002.

[20] “EMC SRDF Datasheet.” `http://www.emc.com/collateral/software/data-sheet/1523-emc-srdf.pdf`.

[21] L. Touchette, R. Weeks, and P. Goswami, “Netapp data motion,” *NetApp Technical Report TR-3814*, March 2010. `http://media.netapp.com/documents/tr-3814.pdf`.