Software-defined Consistency Group Abstractions for Virtual Machines

Muntasir Raihan Rahman, Sudarsan Piduri, Ilya Languev, Rean Griffith, Indranil Gupta
University of Illinois Urbana-Champaign, VMware Inc.
{mrahman2,indy}@illinois.edu,{sudarsan,ilanguev,rean}@vmware.com

ABSTRACT

In this paper we propose a practical scalable software-level mechanism for taking crash-consistent snapshots of a group of virtual machines. The group is dynamically defined at the software virtualization layer allowing us to move the consistency group abstraction from the hardware array layer into the hypervisor with very low overhead ($\sim$ 50 msecs VM freeze time). This low overhead allows us to take crash-consistent snapshots of large software-defined consistency groups at a reasonable frequency, guaranteeing low data loss for disaster recovery. To demonstrate practicality, we use our mechanism to take crash-consistent snapshots of multi-disk virtual machines running two database applications: PostgreSQL, and Apache Cassandra. Deployment experiments confirm that our mechanism scales well with number of VMs, and snapshot times remain invariant of virtual disk size and usage.

CCS Concepts

•**Information systems** → **Storage recovery strategies;**

1. INTRODUCTION

Virtualization service providers are gradually transitioning towards software defined storage architectures in their datacenters [11]. With prominent examples like VMware's virtual volumes (vVol) abstraction [13], and Openstack Swift [6] the software control plane for storage operations can be clearly separated from the underlying hardware (e.g., storage arrays).

In traditional array-based storage, a logical unit number (LUN), a unique identifier that maps to one or more hard disks, is used as the unit of storage resource management. Object storage allows virtual machine disks to become the unit of management instead. This results in flexible software level *policy* management of storage, while delegating snapshot and replication *mechanisms* transparently to hardware storage arrays.

For applications running across multiple VMs, storage arrays provide a *consistency group* abstraction for crash-consistent snapshots of all virtual disks attached to the VMs. The number of hardware consistency groups available for a storage array is typically limited (order of tens of groups) and administrators must manually map virtual disks to appropriate LUNs to ensure that they are snapshotted or replicated as a group (see Figure 1). The number of VMs (and virtual disks) that can benefit from array-defined consistency groups are constrained by (1) the small fixed number of hardware/LUN-based consistency groups, and (2) the manual mapping of virtual disks to LUNs that administrators must (re-)do as VMs are provisioned and decommissioned.

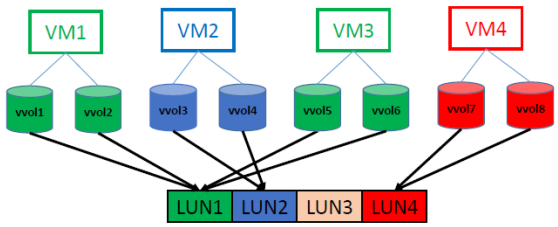

Figure 1: To manually configure a consistency group containing VM1 and VM3, the administrator has to ensure that their VMDKs are placed on storage backed by LUN1 in this example.

We posit that the consistency group abstraction can be defined and managed in the hypervisor while leveraging the snapshot capabilities of an underlying storage array. We present a practical, scalable, mechanism to take crash-consistent snapshots of a group of virtual machines, each with multiple disks (devices) attached.

By carefully managing short ($\sim 50msec$), targeted, pauses of write I/O in hypervisors managing the virtual disks in a consistency group we provide a low-overhead, scalable and robust way to realize the consistency group abstraction. Using our mechanism, we demonstrate the successful recovery of two non-trivial applications – PostgreSQL and Apache Cassandra. Detailed microbenchmarks and experiments on a hardware array confirm that our mechanism scales well with number of VMs, and the total snapshot time for a group remains invariant of virtual disk size and usage.

2. PROBLEM FORMULATION

Consider an application A running across n virtual machines (VM) $v_1, v_2, \ldots, v_n$. VM v_i has attached to it n_i virtual machine disks (VMDK) $d_1^i, \ldots, d_{n_i}^i$. The consistency group across all the VMs consist of $\sum_{i=1}^{n} n_i$ devices (objects). The application interacts with the VMs by issuing write operations to any VMDK in the consistency group. Our objective is to take a snapshot of the consistency group so that we can later recover the application state from VM crash failures by attaching the snapshotted VMDKs to a set of remote VMs.

Crash Consistency Defined: Let $w_1, w_2, \ldots, w_k$ be the ordered

DCC'16, July 25-28, 2016, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/2955193.2955198

sequence of consecutive writes issued to a consistency group CG. Let $w \in W^{CG}$ denote that write w is persisted to a VMDK within CG, where W^{CG} denotes the set of all writes to CG. A snapshot S_{CG} of CG is said to be crash-consistent, if it captures a prefix $w_1, w_2, \ldots, w_p$ of the set of all writes $w_1, w_2, \ldots, w_k$, where $p \leq k$. Formally, S_{CG} is crash-consistent if $W^{S_{CG}}$ is a prefix of W^{CG}.

For correctness, we assume that the application follows the *dependent write principle* [25] where new writes are only issued after previous writes have been acknowledged as completed. This model of operation is common in systems with Write-Ahead-Logs (WAL) such as databases (e.g., PostgreSQL and Apache Cassandra) and journaling file systems (e.g., ext4).

3. DESIGN

Our VM hypervisor managed (software level) mechanism for consistency groups builds upon three primitives.

First, we require a mechanism for stopping (and later resuming) the I/O activity of a VM to its virtual disks (VMDKs). Standard virtualization platforms (e.g., VMware ESX, Xen, Hyper-V, QEMU) typically have mechanisms for stopping a VM's I/O activity – *Pause* (ESX-only), *Stun* and *VM quiesce* [17, 4, 10]. *Pause* stops the virtual CPUs (vCPUs) of the VM preventing any further I/Os from being issued. Pause returns immediately. *Stun* flushes any I/O that has made it to the Virtual SCSI (vSCSI) layer. Stun waits for the I/O flushes to complete before returning. *VM Quiesce* causes VMware Tools to quiesce the file system in the virtual machine. Quiesce waits for I/Os buffered at the guest file system (and the layers below) to be flushed before returning. Thus in terms of time to complete, we have: *Quiesce* > *Stun* > *Pause*. Thus we use the relatively low-cost Pause mechanism in our work.

Second, we need a mechanism for taking a snapshot of the current state of a VMDK. This mechanism is delegated to the storage backend. The efficacy of our approach – specifically the amount of time VMs need to remain stopped – strongly depends on the performance of the snapshot provider[1]. Snapshot providers that employ technology that allows for efficient snapshots, e.g., constant-time snapshots [20, 28], light-weight diffs/deltas etc. allow us to significantly reduce the absolute time that a VM must remain paused.

The final mechanism we need is one for replicating crash-consistent snapshots to remote sites. For performance reasons snapshot replication needs to be asynchronous [16]. We rely on storage arrays to replicate to a remote site. In our experiments (in Section 4), we mount the snapshot of the consistency group on a different VM to test for correctness. In a production setting, the consistency group snapshot would be asynchronously replicated to a remote site, instead of a separate VM, for disaster recovery.

Using these primitives, our consistency group mechanism works as follows. A centralized coordinator (we use VMware vSphere in our implementation) issues Pause directives to all the servers running VMs that comprise the consistency group. Once I/O has been paused for the target VMs, the coordinator issues snapshot directives to all disks in the group in parallel. Once the snapshot completes, the coordinator issues UnPause directives to all the servers. A two-phase commit (2PC) [30] protocol is used to robustly realize this interaction sequence. The correctness of our approach depends on the *coordinated pause* across all the VMs that are part of a consistency group and the *Dependent Write Principle (Section 2)* [25].

4. EVALUATION

Our evaluation consists of two stages: 1) micro-benchmarks, and 2) deployment experiments. The microbenchmark experiments demonstrate the correctness of our proposed mechanism using a toy writer program which sequentially writes integers to disk using a sample vendor provider. The deployment experiments show that our crash-consistent snapshot mechanism can recover non-trivial applications, and that it scales well.

4.1 Experiment Setup

In our experiments we use as the virtualization platform an object build of ESXi 6.0 (vSphere 6.0) and a version of the CloudVM (a self-contained virtual appliance that contains Virtual Center – the application used to manage and configure VMware ESXi hypervisors). For correctness (but not performance testing) we use a sample *Vendor Provider* [15] – the sample Vendor Provider is a Linux appliance that behaves like a vVol-enabled array[2]. This allows us to configure it as a storage provider for the CloudVM and for ESXi and create VMDKs on it. Later for performance testing, we use a NetApp storage array as a real vendor provider.

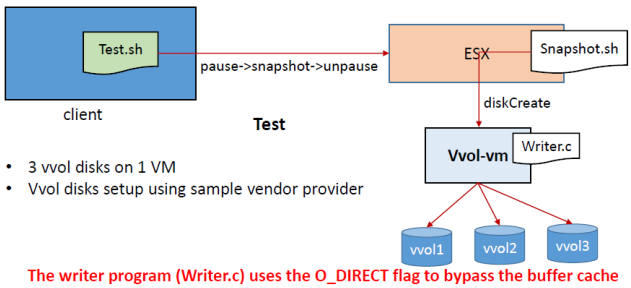

Figure 2: Simple, single-VM, multi-vVol setup.

4.2 Microbenchmarks

4.2.1 Single VM Multiple Disk Consistency Groups

The first question we wish to answer is whether we can take a crash-consistent snapshot of a single VM with multiple (3) VMDKs stored on vVols. Figure 2 shows the setup. A small script initiates the calls to pause, snapshot and unpause. A short C-program (Writer.c) uses Direct I/O (the O_DIRECT flag) to write fixed-length records (512 bytes) containing integer data while bypassing the Operating System buffer cache to files stored on each VMDK/vVol. Figure 3 shows that our test program correctly stripes writes across the 3 vVols. To verify the contents of the snapshot we mount the snapshots of each vVol on a separate virtual machine and read back the records written to each data file. An epoch represents a round of writes across all (three) attached vVols.

- Read from 1st vvol snapshot
  - 1, 4, 7, 10, 13, 16, 19, 22, 25
- Read from 2nd vvol snapshot
  - 2, 5, 8, 11, 14, 17, 20, 23, 26
- Read from 3rd vvol snapshot
  - 3, 6, 9, 12, 15, 17, 21, 24, 27

epoch	vvol1	vvol2	vvol3
0	1	2	3
1	4	5	6
.....	.....	.....	.....
8	25	26	27

Figure 3: Simple, single-VM, multi-vVol results. Our snapshot contains writes of integers striped across the 3 vVols.

[1]Having VMs paused for too long may result in VM unavailability reports by monitoring systems.

[2]A real Vendor Provider acts as the interface between Virtual Center and a specific array.

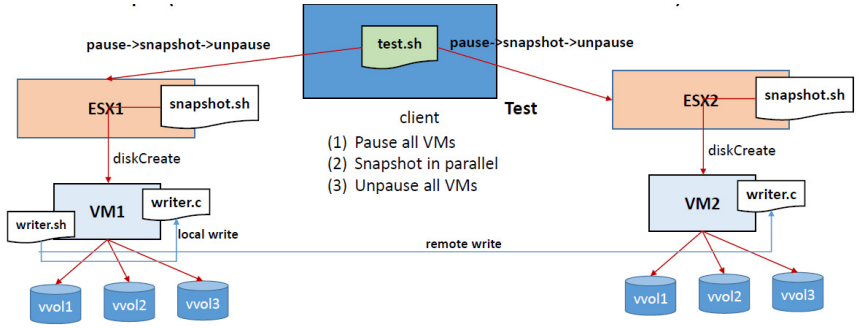

Figure 4: Multi-VM, multi-vVol, and multi-ESXi setup. In this setup our writer program alternates writing across "local" vVols and "remote" vVols.

4.2.2 *Multiple VM Multiple Disk Consistency Groups*

The second question we wish to answer is whether we can take a crash-consistent snapshot across multiple VMs on multiple ESXi hosts, each with multiple vVols. This is shown in Figure 4. An extended writer program alternates writes across the vVols of VM1 (local) and VM2 (remote) – a write to a vVol of VM1 is followed by a write to a vVol of VM2 before writing again to a vVol of VM1 etc. Each epoch represents a write across all the vVols for all the VMs. Figure 5 shows that a crash-consistent snapshot taken across the vVols of VM1 and VM2 contains a prefix of the writes.

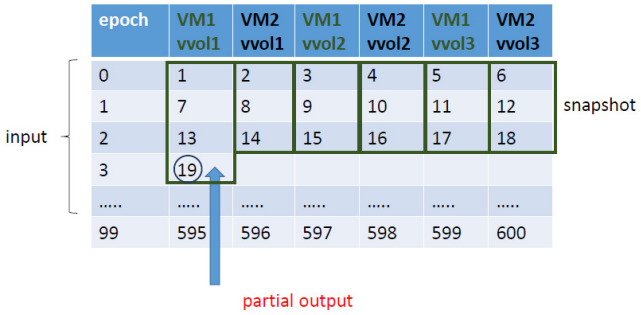

Figure 5: Multi-VM, multi-vVol, and multi-ESXi results. Odd integers go to the vVols of VM1 (local) and even integers to the vVols of VM2 (remote). The highlighted portions represent the individual vVol snapshots that comprise the crash-consistent snapshot of the consistency group containing VM1 and VM2.

4.2.3 *Necessity of Coordinated Pause*

The third question we wish to answer is whether the coordinated Pause is necessary. Using the multi-VM setup (Figure 4) we allow the writer program to run and first take a snapshot of one disk first, and then take snapshots (in parallel) of all the remaining disks. Figure 6 shows the "gaps" (missing writes) in the union of the snapshots of each vVol, i.e., the result does not capture a prefix of dependent writes and thus is *not* a crash-consistent snapshot using our correctness criteria (Section 2). Thus all the disks in the consistency group must be simultaneously paused to ensure crash consistency.

4.2.4 *Overhead of VM Pause*

The fourth question we wish to answer is how long a VM needs to remain paused while we take a crash-consistent snapshot. We configure a VM with multiple VMDKs of different sizes – ranging from 1GB, 10GB, 100GB and measure the time taken to create the snapshot. For this experiment we use VMware Virtual Machine File System (VMFS) [32]. Figure 7 shows that the time to take a snapshot is small (a few tens of milliseconds) and is independent of the size of the VMDK. In the case of VMFS, the time to create snapshots using redo logs depends on the number of changed sectors [14]. Later in Section 4.3.3, we also benchmark snapshot creation times for a NetApp storage array based vVol disk.

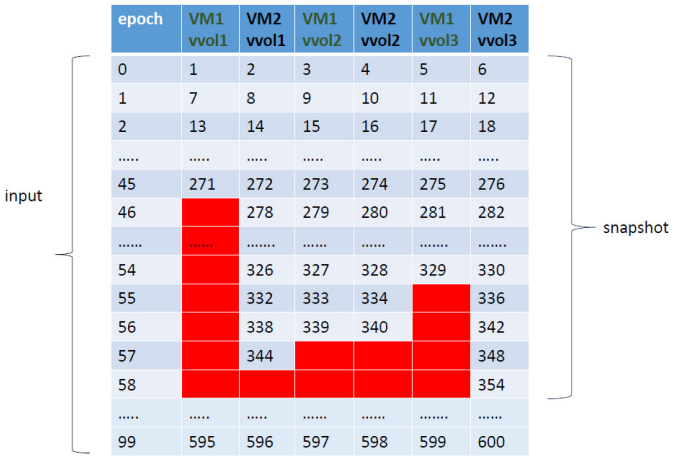

Figure 6: Multi-VM, multi-vVol, and multi-ESXi results. Without pause, the union of snapshot contents contains gaps (missing preceding writes) and as a result the result is *not* a crash-consistent snapshot.

VMDK type	Size (GB)	Snapshot time time (msecs)
OS	20	49
Data1	1	56
Data2	10	40
Data3	100	41

Figure 7: Time to create a snapshot on VMFS for different VMDK sizes.

4.3 Deployment Experiments

In this section we move towards reality in two aspects: (1) we replace the toy writer application with non-trivial applications, and (2) we replace the sample vendor provider with a real storage array

(NetApp) and demonstrate the recovery of two non-trivial database applications: PostgreSQL [7] and Apache Cassandra [1], both of which use a write ahead log [27] for application level recovery.

First, we repeated our microbenchmarks on the NetApp storage array and verified that we can achieve crash consistency for the simple writer application described earlier. Next, we move on to the PostgreSQL and Apache Cassandra experiments.

Cassandra cluster can recover from the group snapshot

```
INFO [main] 2015-05-19 20:59:31,470 CommitLog.java (line 127) Log
replay complete, 10 replayed mutations
 INFO [main] 2015-05-19 22:36:51,867 CommitLog.java (line 125)
Replaying /mnt/vvold1/cassandra/commitlog/CommitLog-2-
1432094370899.log, /mnt/vvold1/cassandra/commitlog/CommitLog-2-
1432094370900.log, /mnt/vvold1/cassandra/commitlog/CommitLog-2-
1432094370901.log
```

Figure 8: We show the 2 server Cassandra cluster log from a successful start from a crash-consistent snapshot (subset of log from one replica).

4.3.1 *PostgreSQL Recovery*

PostgreSQL [7] is a popular ACID compliant transactional database system that uses write-ahead logs for recovery [27]. We use the PostgreSQL database v9 (without replication) as the application writing data and pgbench [8] as a workload generator driving writes to the database. pgbench is similar to the TPC-B [12] workload that runs five select, update and insert commands per transaction. We configure PostgreSQL such that the directories concerned with the transaction log are all stored on one vVol while the directories associated with the actual database tables are stored on another [9] (Figure 9).

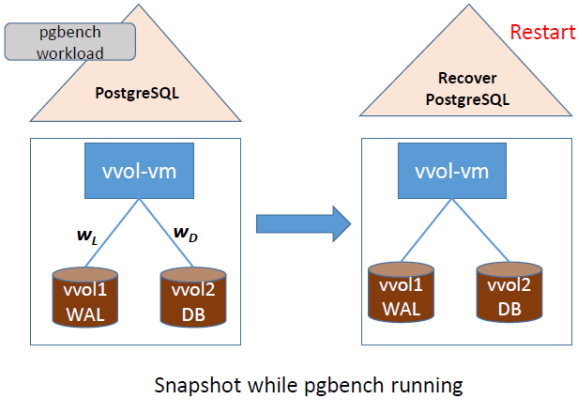

Figure 9: We configure PostgreSQL 9 such that the directories associated with its transaction log are stored on one vVol and the directories associated with actual database tables are stored on another vVol.

Figure 10 shows a successful restart from a crash-consistent snapshot across the PostgreSQL vVols. PostgreSQL performs various integrity checks before starting the database, and the log indicates that the integrity checks of the snapshots were successful.

4.3.2 *Apache Cassandra Recovery*

Next we show we can recover the state of a NoSQL database Apache Cassandra[1], running both in centralized and distributed modes, using our mechanism.

Postgresql can recover from the group snapshot

```
LOG:  database system was interrupted; last known up at 2014-07-24 09:25:41 PDT
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/1782900
LOG:  invalid magic number 0000 in log segment 000000010000000000000001,
offset 7979008
LOG:  redo done at 0/179AFD8
LOG:  last completed transaction was at log time 2014-07-25 00:23:07.301403-07
```

Figure 10: We show the PostgreSQL log from a successful start from a crash-consistent snapshot.

We use YCSB v 0.1.4 [23, 18] to send operations to Apache Cassandra. Each YCSB experiment consisted of a load phase, followed by a work phase. Unless otherwise specified, we use the following YCSB parameters: 1 YCSB client, 1GB data-set (1024 keys, 1 MB size values), and a write-heavy distribution (50% writes). The default key size was 10 B for Cassandra. The default throughput was 1000 ops/s. All operations use a consistency level of `ALL`.

We ran experiments in two modes: first we deploy Cassandra on a single VM, this setup is similar to the PostgreSQL setup shown in Figure 9. Second we deploy Cassandra over a cluster of 2 VMs, each VM located on a separate ESXi host. The experiment setup is shown in Figure 11. For cluster experiments, we setup 2 replicas for each key. We use YCSB to load Cassandra with 1024 keys during the load phase, and inject read/write operations during the work phase. Each YCSB run lasts for 60 seconds.

We configure Apache Cassandra such that the commit log directories (write-ahead logs) and data directories were located on separate vVols. Our Cassandra experiments involve vVols carved out of a 300GB NetApp storage array.

To verify that the snapshot of write operations actually is a prefix of all write operations, we instrument YCSB to log all operations. The *operation log* file is stored in the same disk as the commit log. Each entry in the operation log stores the key and the corresponding value for each write (put) operation. Thus when we take a group snapshot of all VMs, we also save as part of the snapshot, the operation log. For correctness, the snapshot operation log file should be a proper prefix of the full operation log file obtained at the end of the YCSB experiment.

During recovery, Apache Cassandra servers use the commit log to recover any missing data [2]. Similar to PostgreSQL, Cassandra writes follow the dependent write principle (writes to log precede writes to database tables) and we are able to correctly recover a single-node Cassandra instance. Due to space constraints we focus on the multi-node Cassandra setup.

In Figure 8, we show entries from the log file of one of the two replicas in the Cassandra cluster recovery experiment. Here the log entries indicate successful recovery.

The YCSB experiment for the Cassandra cluster was scheduled for 60 seconds. With one group snapshot across all vVols, the experiment completed in 61 seconds. This indicates a negligible overhead of 1.6% due to the coordinated VM pause/unpause phases in our mechanism.

We also verify crash consistency via a byte by byte comparison of the snapshot operation log file and the full operation log file. Concretely, we use the Unix `cmp` utility to confirm that the log as per the snapshot is a prefix of the full log.

4.3.3 *Scalability Experiments*

The intuition behind the scalability of our approach is that we snapshot each disk in a group in parallel. Thus the total snapshot

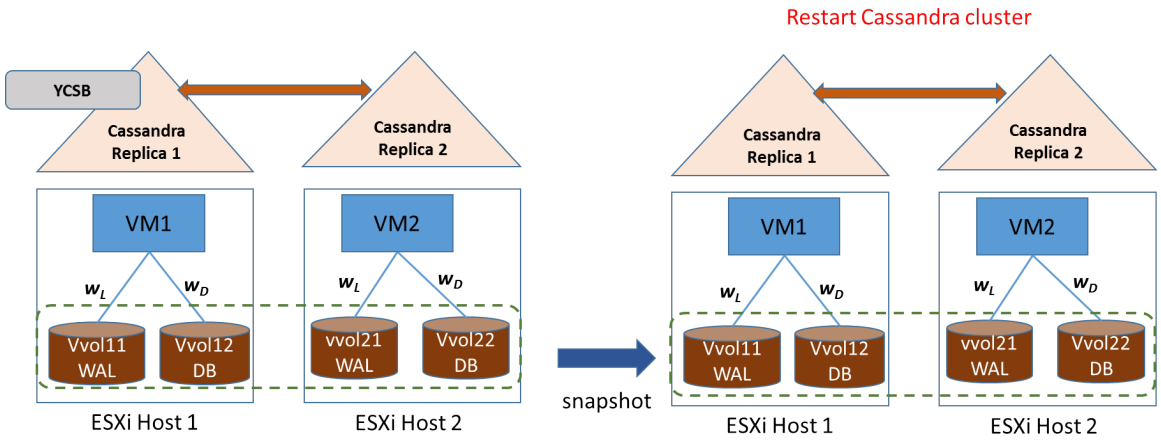

Figure 11: Apache Cassandra cluster setup and group snapshot experiment.

time is only bounded by the worst-case snapshot time of any disk. We evaluate scalability in two dimensions: (1) disk size, and (2) number of VMs.

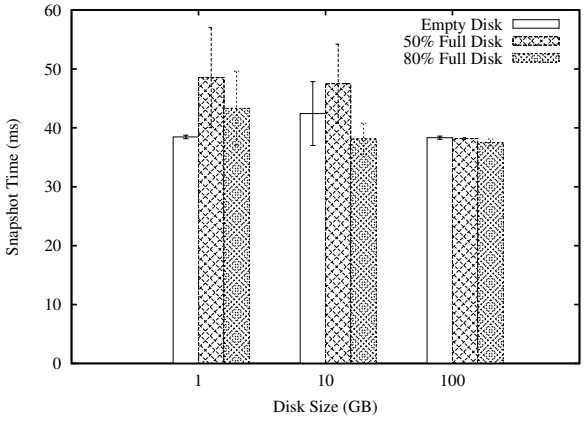

Figure 12: The average snapshot creation time on a NetApp storage array.

For our first experiment, we configure a VM with multiple vVol VMDKs of different sizes – ranging from 1GB, 10GB, 100GB, all carved out of a 300GB NetApp storage array. We measure the time taken to create the snapshot when the disk is empty, 50% full (half full), and 80% full (almost full). Figure 12 shows the average (with 95th percentile confidence intervals) snapshot times. We observe that on NetApp, the average vVol disk snapshot time is below 50 msecs irrespective of disk size and disk space, and thus the storage array snapshot technology only incurs minor overhead. For the 1GB disk, the snapshot time slightly increases as we move from an empty disk to a 50/80% full disk, due to more data to snapshot. For 100GB disks, we observe constant snapshot times irrespective of disk usage, thus is due to the efficient redirect-on-write (ROW) mechanism used by NetApp snapshots [26]. For 10GB disks, we observe a slight decrease in snapshot time as we move from 50% full to 80% disks. We attribute this to measurement noise.

Overall we have observed that snapshot creation times are small (about 50 msecs) and invariant with respect to the amount of data in a virtual disk (empty, half full, 80% full). This indicates we should be able to define the scalability constraint of a software-defined consistency group purely in terms of the number of VMs and not in units of the amount of data in the attached virtual disks.

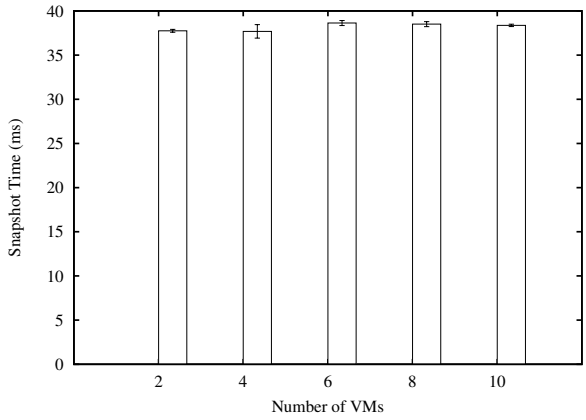

Figure 13: Snapshot time vs. number of VMs.

In the next experiment (Figure 13), we vary the number of VMs from 2 to 10, with increments of 2. Each VM had 3 disks attached to it, thus we run experiments with a maximum consistency group size of 30. With a distributed writer program alternatively writing consecutive integers to each disk in the consistency group, we take group snapshots using our mechanism. For each VM count, we measure the mean and 95th percentile confidence interval of snapshot time for each VM in the group. Since snapshots are done in parallel, and VM pause/unpause is instantaneous, these numbers quantify the overhead of taking group snapshots.

We observe that the worst case snapshot time for VMs in a consistency group stays below 40 ms as the number of VMs in a consistency group increases. This is mainly due to efficient redirect-on-write (ROW) snapshot mechanisms employed by the NetApp storage back-end. NetApp snapshots are based on the WAFL (Write Anywhere File Layout) [26]. The key idea in WAFL is to keep a set of pointers to blocks of data which enables the filesystem to make copies by just copying the pointers. Thus ROW snapshots redirect changes to new blocks, and creating a new snapshot only requires copying volume metadata (pointers) [5]. This indicates that our proposed crash-consistency mechanism scales well with increasing number of VMs in a consistency group. Overall, the two scalabil-

ity experiments demonstrate that for our proposed mechanism, (1) disk size and disk space do not constrain scalability; (2) scalability can be defined purely in terms of number of VMs in a consistency group; and (3) worst case snapshot overhead remains bounded with increasing number of VMs in a consistency group. Thus we conclude that our proposed mechanism scales well.

5. RELATED WORK

There has been much work on enabling consistent replication across multiple VMs. In host-based Replication (HBR) [16], the host itself is responsible for replication, whereas our mechanism lets hardware storage arrays take care of replication. We extend on the HBR approach by adding coordination across multiple ESX hosts. We coordinate group crash consistency at a higher layer compared to HBR.

Compared to the vast literature on checkpointing of distributed programs [24], here we look at checkpointing the state of a single application, which interacts with multiple virtual machines. Also these techniques checkpoint in-flight messages, whereas our mechanism ignores in-flight unacknowledged messages. Compared to hardware consistency group abstractions proposed by NetApp and EMC [3], we move the consistency group abstraction from hardware to software. Distributed process group abstractions were initially proposed in [22], and later incorporated in the ISIS virtual synchrony model [19]. In both cases, the objective is to support group multicast, whereas our goal is to checkpoint and recover a group of VMs for disaster recovery.

Many techniques have been proposed for checkpointing distributed applications. These techniques can be roughly categorized into application level (e. g. , ARIES write ahead log mechanism [27]), library level [21, 31], and OS level [29]. Our VM level group snapshot mechanism is closer to OS-level checkpointing mechanisms.

6. CONCLUSION

In this paper we proposed a practical scalable mechanism for taking crash-consistent snapshots of a group of virtual machines. This enabled us to move the consistency group abstraction from hardware to software with very low overhead ($\sim$ 50 msecs VM freeze time). This low overhead facilitated taking crash-consistent snapshots of large consistency groups at a reasonable frequency. Thus our mechanism can provide low RPO for disaster recovery. Experiments confirmed that in our mechanism, snapshot times are invariant of disk size and disk space, and that it scales well with increasing number of VMs.

7. ACKNOWLEDGMENTS

This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, VMware Graduate Fellowship, Yahoo!, and a generous gift from Microsoft.

8. REFERENCES

[1] Cassandra. http://cassandra.apache.org/.
[2] Cassandra commit log architecture. http://goo.gl/RC1qeT.
[3] EMC SRDF consistency groups. http://goo.gl/wBlgZQ.
[4] Hyper-V vs. VSS snapshots. http://goo.gl/5ia37H.
[5] Netapp snapshots. http://goo.gl/9xOA84.
[6] Openstack Swift. http://docs.openstack.org/developer/swift/.
[7] PostgreSQL. http://www.postgresql.org/.
[8] PostgreSQL benchmarking tool. https://goo.gl/v2ykE3.
[9] PostgreSQL physical storage. http://goo.gl/oYKWRb.
[10] QEMU features/snapshots. http://wiki.qemu.org/Features/Snapshots.
[11] The software-defined data center (sddc). http://www.vmware.com/software-defined-datacenter/.
[12] Transaction processing performance council, TPC-B. http://www.tpc.org/tpcb/.
[13] Vmware , virtual volumes. http://www.vmware.com/products/virtual-volumes.
[14] VMware Inc., disk chaining and redo logs. https://goo.gl/6dxM16.
[15] VMware VASA. http://goo.gl/0yTWja.
[16] VMware vSphere Replication 6.0. https://www.vmware.com/files/pdf/vsphere/VMware-vSphere-Replication-Overview.pdf.
[17] Xen snapshots halting I/O. http://discussions.citrix.com/topic/263681-xen-snapshots-halting-io/.
[18] Yahoo! cloud serving benchmark (YCSB). http://goo.gl/GiA5c.
[19] K. P. Birman. The process group approach to reliable distributed computing. *CACM*, 36(12):37–53, 1993.
[20] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems, 2003.
[21] Y. Chen, K. Li, and J. Plank. Clip: a checkpointing tool for message passing parallel programs. In *Proc. ACM/IEEE SC*, pages 33–33, 1997.
[22] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM TOCS*, 3(2):77–107, 1985.
[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC*, pages 143–154, 2010.
[24] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.
[25] EMC Inc. Emc srdf/a multi-session consistency on z/os. http://goo.gl/81HuZe.
[26] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proc. Usenix WTEC*, pages 19–19, 1994.
[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.
[28] B. Moore. Zfs the last word in file systems. http://goo.gl/a3p8hn.
[29] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS OSR*, 36:361–376, 2002.
[30] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.
[31] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. In *Proc. LACSI*, pages 479–493, 2003.
[32] S. B. Vaghani. Virtual machine file system. *ACM SIGOPS OSR*, 44(4):57–70, 2010.