Providing High Availability in Cloud Storage by decreasing Virtual Machine Reboot Time

Shehbaz Jaffer, Mangesh Chitnis, Ameya Usgaonkar
NetApp Inc.

Abstract

A Virtual Storage Architecture (VSA) is a storage controller deployed as a virtual machine on a server with a hypervisor. The advantage of VSA is to leverage shared data storage services without procuring additional storage hardware, which is a cost effective solution. In case of VSA, high availability (HA) is achieved by restarting the failed virtual machine on an event of a software failure. Rebooting the VSA is a slow operation thereby reducing the overall service availability. In this paper, we describe the challenges and approaches taken to decrease reboot time of VSA to achieve High Availability. We have been able to reduce the VSA reboot time by 18% using our optimizations. We also explore the changes required in Journal based File systems for efficient operation in the Cloud.

Keywords- Cloud Storage, Virtualization, High Availability, Data Center, Software Failure

1. Introduction

The QOS offered by a data service provider is largely determined by its capability to provide non-disruptive service to its clients. Enterprise Storage solutions such as NetApp® FAS boxes [1] and Amazon EC2 [11] report 99.999 percent availability, which translates to a downtime of 5 minutes in a year. Achieving these figures in a cost effective manner with minimal changes in the storage software is a big challenge today.

Virtualized Enterprise storage solutions provide high availability (HA) in VSA by rebooting a node (Virtual Machine) on the event of a crash. The total time taken to reboot a VSA has been reported to be of the order of few minutes for setups having large number of volumes. With all enterprise solutions looking at storing most of their data in the cloud, providing high data availability in the cloud has become imperative.

In Filer (storage hardware controllers) setups, High Availability is provided by shipping hardware boxes in an active-active or active-passive HA pair. On the event of a crash, the client requests are redirected to the partner filer while the first filer reboots and restores its state. Once the first filer restores its state, the client requests are redirected to the first filer. This requires two filers or two storage instances. VSA is a more cost effective (but less failure resilient) cloud storage solution offered by enterprise storage companies. Here, the storage solution consists of one node and all client requests are redirected to that node. On the event of a failure, the VSA reboots and client I/O operations get serviced only after the VSA is up and running.

In this paper, we propose quick reboot of VSA to achieve high availability. For this purpose, we cache file system metadata that is read by the Operating System during volume mount inside the host RAMDisk. This cached metadata is then read at the time of reboot to reduce the overall boot time. While doing this, we have found some interesting insights for using Journaling filesystems such as the Ext3 file system [3] in the cloud, and optimizations that can be done in such filesystems to make it suitable for quick reboot in cloud environments. We have used a prototypical system developed in our research facility as a Virtual Storage Architecture (VSA) for all the experiments and architectural changes mentioned in this paper.

2. Related Work

Protecting memory state across an OS reboot has been explored earlier [4][6]. P Chen et.al have also explored caching and retaining file system metadata across OS reboots. [5]. In order to eliminate the corruption of file system metadata cached in the file cache across reboot, the write-permission bit in the page table is enabled only while writing to a page and disabled afterwards. This gives us the same reliability against corruption that is provided by disks.

Linux [7] itself has undergone massive changes over the kernel versions to reduce the boot time by reducing the concurrent processes that get activated during the first few seconds of Linux boot[8]. Most of the Linux distributions now offer Kexec [9] feature to bypass the boot process through BIOS. Recovery-Oriented Computing (ROC) [16] proposes novel techniques based on isolation and redundancy for fine grained recovery of hardware and software components. In contrast, our approach employs a light weight VM based quick reboot solution to increase the overall availability. *Our solution is based on the assumption that the likelihood of hardware and hypervisor failures is much less compared to application failures (i.e. VMs).*

Large data centers [10] also provide fast restarts in their database systems by using shared memory. The Scuba™ [12] in-memory database system decouples memory lifetime from process lifetime by storing a valid memory state across system reboot. The time to restore the memory state from another shared memory is much less than rebuilding the memory state by making read requests to disk.

3. Architecture

3.1 Problem Description

Users generally deploy VSA in order to leverage storage data services while using their commodity disks in order to

save on cost. This cost advantage comes at the expense of High Availability and performance. High Availability is measured in terms of downtime with respect to total time. The downtime in case of VSAs is primarily due to the virtual machine shutting down because of hardware or software failures. This involves the time required to perform a clean shutdown of the corrupted virtual machine and rebooting it to a point where it can start accepting client requests. This downtime can be further broken down in to the following stages: a. core dump before shutdown; b. POST, kernel, module load; The kernel and module blocks have to be read from disk (vis-à-vis flash card used in Filers) ; c. core save on boot; d. V-NVRAM (Virtual Non Volatile RAM) load, NVRAM is used to store the journal data - In case of VSA, NVRAM is emulated by mapping a portion of OS memory backed by a disk file which could be synchronous or asynchronous depending on the required performance and reliability. The V-NVRAM load time involves reading this file and mapping its contents on the mapped memory region; and e. File System (FS) mount and File System (FS) replay.

The mount and replay time involves loading volume metadata blocks (superblock, inode map, etc.) and user blocks affected during the shutdown. This time has significant impact on the overall downtime and depends on the number of active volumes associated with the VSA. The following diagram shows the boot timeline.

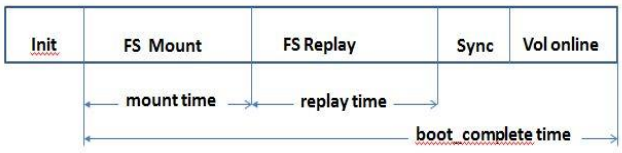


Figure 1: Boot time line

3.2 Host RAMDisk Architecture

The high-level functional block diagram is depicted in Figure 2. VSA runs as a Virtual Machine (VM) on a Host OS (Type 2 hypervisor) or on a bare-metal hypervisor (Type 1 hypervisor) which we call as Host Machine (HM). In either case, VSA uses a portion of HM's RAM (fast storage medium) as persistent storage to store File System meta-data blocks used during mount operation. The main intuition behind this setup is that, as accessing RAM is much faster than disk, the reboot mount time will be reduced significantly. The primary assumption here is that, whenever VSA has to undergo reboot due to a software panic, the fault causing this panic is isolated within the VSA's VM and HM is unaffected by this fault. Hence, although VSA is rebooted, HM is still in a running state and is able to maintain its memory state. Major cause of a VSA reboot is a software failure. We propose that for such software failures, a persistent media with respect to VSA reboot could help decrease the total VSA downtime.

The RAM portion of HM is used as RAMDisk to store meta-data disk blocks and V-NVRAM blocks. This RAMDisk is then exposed as a SCSI/IDE disk to VSA. The disk files for V-VNRAM and meta-blocks in VSA are now located on HM's RAM (i.e. low latency medium). This involves minimal change in VSA's existing implementation.

The mount and replay time is minimized by synchronizing the meta-data buffers (i.e. file system buffers involved during volume mount operation) between VSA memory buffer cache and HM's memory buffer cache in RAMDisk. User metadata buffers (i.e. Inode buffers) involved in file operations during CP are also kept in this metadata cache. V-NVRAM log is synchronized between VSA and Host RAMDisk. The V-NVRAM which is typically backed by a disk file is now also backed by an in-memory V-NVRAM file on HM as shown in Figure 2. This eliminates the delay in reading the V-NVRAM file from disk on reboot.

Since, the metadata buffers and V-NVRAM logs are present in memory the disk access time which amounts to the bulk of downtime during mount and replay is reduced. The VSA also saves on the time required to dump and save core. On reboot, the VSA starts loading V-NVRAM and metadata buffers from HM RAMDisk files instead of loading these buffers from disk. It is possible to place Host RAMDisk buffers on different machines in order to improve reliability. It is also possible to extend the functionality of HOST RAMDisk as an external caching device.

3.3 Metadata Buffer Interception and Indexing

The process of metabuffer caching and indexing is shown in Figure 3. When the VSA starts for the first time, the metabuffer index is empty. Subsequently, as metablocks are read from and written to disk, the read /write calls are intercepted to either fill in the metabuffer cache or read buffers from this cache.

Metabuffer insert is called from two different places during the first boot.

1. Early boot metabuffer load: During the boot phase all volume mount and meta file read operations are intercepted to push the meta blocks in the metacache
2. CP write: On every checkpoint, when 4K buffers are written back to disk the write call is intercepted and these buffers are replaced in the metacache

The modified metacache is flushed back to host RAMDisk on every CP in order to maintain a persistent copy of all modified metablocks. This makes sure that in an event of crash we can recover consistent 4K blocks from metabuffer residing on host RAMDisk.

Metabuffer cache includes two sections:

1. Metabuffer Index: This portion contains information about the buffer such as Volume Id (Aggregate ID), File Id, Aggregate Block Number (this is unique per Aggregate), Meta Block Index (this is index of 4K block)
2. The actual 4K metablock along with metabuffer index values for sanity check

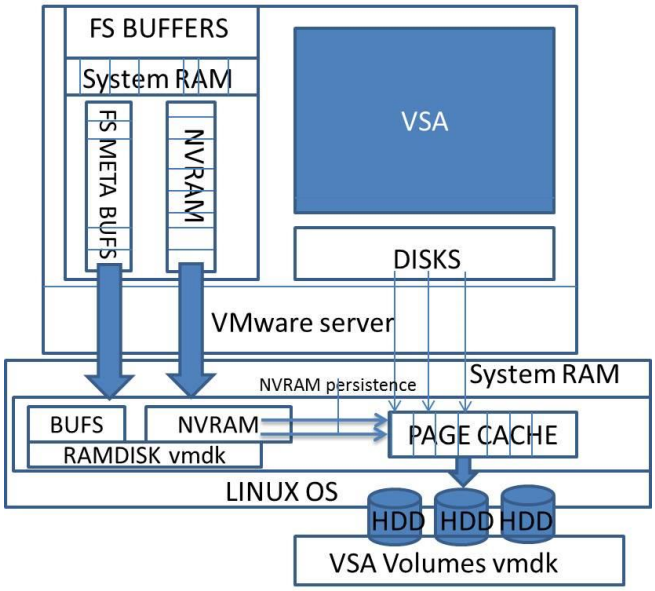


Figure 2: Functional Blocks

The metabuffer index is unsorted. On reboot, when metabuffer cache is loaded from host RAMDisk, the metabuffer index is sorted based on Volume Id and Block Number. This is required to reduce the block search operation on every read call. The block search is essentially a binary search on metabuffer index. Whenever the index is found, the metabuffer Index is used to retrieve the 4K block. This is required because although metabuffer index is sorted, the 4K blocks are left unsorted to reduce the overhead of moving 4K blocks.

Thereafter, during volume mount and file read, all read operations are intercepted to read the 4K block from metacache. The volume Id, file Id and block number tuple is used to search for the 4K block in the sorted metabuffer index. In case of a hit, the 4K block is returned back to the caller; else the read call is forwarded further to read the 4K block from disk.

4. Experiments

Our experiments focus on comparing the re-boot time of an unmodified VSA (Vanilla VSA) with respect to VSA with our metabuffer cache modifications.

We first check the percentage of metadata that has been cached into Host RAMDisk, which we call the hit rate. To count the amount of blocks cached, we make relevant changes (add counters) in the inode blocks to keep statistical information regarding count, block number, per block mount time, block misses and system events.

In order to measure the amount of time taken to complete various phases of boot process, (mount_volume and boot_complete time) we use customized macros which call the FreeBSD gettimeofday() function to evaluate the amount of time taken for a reboot. For measuring the CPU statistics, we use customized macros system_getstats () which internally reads data from FreeBSD emulated processor structures (freebsd_sched_processor_getstats ()).

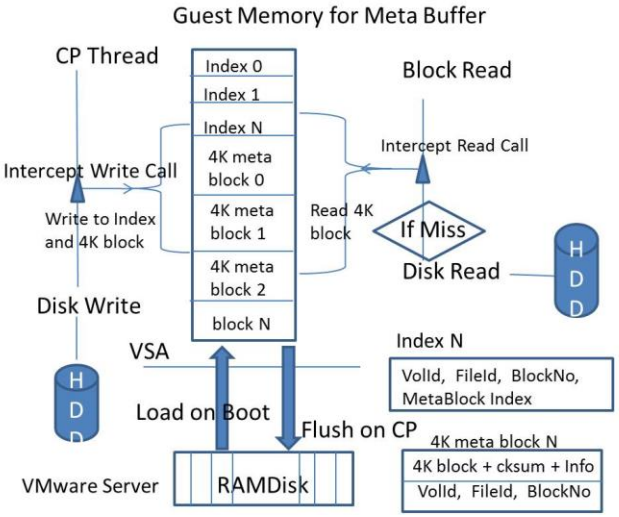


Figure 3: metablock Interception and Indexing

4.1 Hit Rate

Hit rate is the ratio of the total number of metadata blocks cached in the Host RAMDisk to the actual number of metadata blocks that are there for a particular set of Volumes. Figure 4 denotes the total number of metadata blocks captured by our code and cached inside metabuffer cache. Each block cached is of 4K size. As evident in the graph, with the increase in the number of volumes, the number of cached blocks also increases proportionately. The last bar for prefetch X denotes the number of blocks retrieved from metacache buffer when prefetch logic is disabled. Essentially, most of the file systems use a prefetch logic to retrieve read and buffer certain blocks before the mount process begins. This is to reduce the random disk I/O to fetch Metablocks during the actual mount process. boot_complete block reads are the number of metablocks read by VSA until the boot process is complete. We were able to achieve a high hit rate of 89 % for 400 volumes with prefetch disabled.

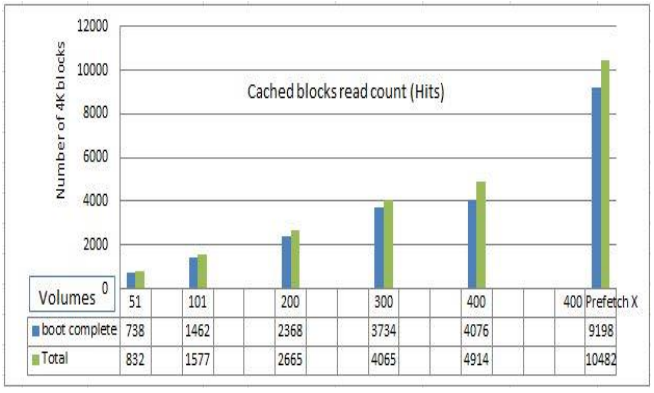
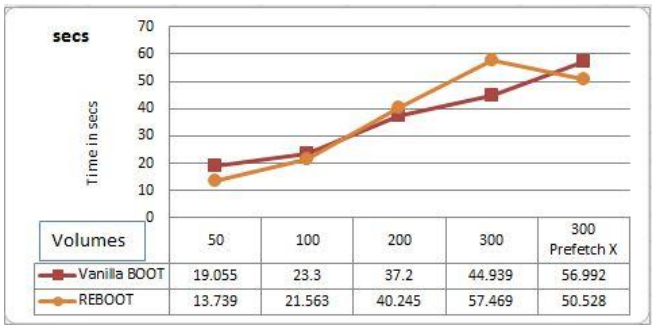


Figure 4: MetaBuffer Hit Ratio vs Volumes

4.2 Effect of Caching on Boot Time



Volumes	50	100	200	300	300 Prefetch X
Vanilla BOOT	19.055	23.3	37.2	44.939	56.992
REBOOT	13.739	21.563	40.245	57.469	50.528

Figure 5: Boot Time vs Volumes

Figure 5 shows the effect of caching on the boot time. "Vanilla Boot complete" is the time required to boot VSA with different number of volumes without metabuffer cache support, i.e. all metablocks are retrieved from disk. Similarly, "boot complete reboot" shows the time required to reboot VSA when metabuffer blocks are retrieved from metabuffer cache. This depends on the hit rate (refer Figure 4). As evident from the graph, even with caching metablocks, we do not see any significant improvement in boot time. On the contrary, up to 300 blocks with prefetch enabled, the reboot time with metabuffer cache is higher than without metacache buffer. Only after 300 volumes with prefetch disabled when the amount of metabuffer blocks grows significantly, we start noticing the effect of metabuffer cache with slight improvement in reboot time.

In the following subsection, we will see the reasons for the lack of improvement in reboot time with metabuffer caching and some remedies to tackle these anomalies.

4.3 Effect of Host Page Cache

Apart from our own caching mechanisms, the Host Machine also caches few blocks that were fetched by the Guest Operating System during its execution. This can be seen in Figure 1 where Linux Host uses page cache to buffer Guest OS vmdk disk buffers. To explore the effect of page cache, we calculated the amount of time taken for a reboot after clearing the host page cache against the normal reboot. The results (Table 1) showed that VSA *clean_host_buffer* took more time than a normal VSA run without clearing the host buffer. The effect of host buffer, although not very significant, indicated why in production environment, we were getting variable reboot times.

	mount complete 300 Volumes	boot complete 300 Volumes
Host cache	22.097 secs	39.192 secs
Host clean-cache	28.714 secs	45.498 secs

Table 1: Host Page Cache

In production environment, the host page cache was variably caching the blocks read by the Guest VM. Also, with other guests concurrently running on the same host, the effect of page cache was not very predictable.

4.4 Message passing overhead with low latency device

In order to deal with high latency devices like disk most of the software architectures make use of threads and message passing architecture. This kind of architecture allows threads with high CPU activity to run without blocking for disk I/O. Also, the message queues help to increase the overall throughput by handling more requests per unit time. However, this architecture suffers from high latency when a low latency medium like RAMDisk is used instead. The overhead of message and thread scheduling creates a software bottleneck thereby increasing per block latency and overall boot time. Figure 6 shows time consumed by SCSI read i.e. actual disk block read and raid_send_async i.e. time spent between an actual request of buffer until the buffer is serviced.

Modified sequence diagram in Figure: 7 shows synchronous call mechanism which eliminates message passing overhead for low latency devices like RAMDisk. Most of the asynchronous calls for RAID IO, SCSI and drivers are eliminated whenever a cached buffer is found in the metabuffer. Also, instead of returning the cached buffer as a message back to the calling thread, the modified mechanism calls the read_handler directly thereby avoiding the message and thread scheduling overhead. This reduction in message and thread scheduling overhead is shown in Figure 8.

4.5 Per Block Fetch Breakdown Analysis

We mapped the time taken per block while fetching each of the metadata blocks from the host RAMDisk to memory. Interestingly, we found that there were two sections as shown in Figure 9 where the time to fetch the blocks were much higher than the rest of the blocks. The first peak (we call it the *boot_time_peak*) occurred during early boot when all the buffers were retrieved from metabuffer cache with no cache miss. Similarly, for the second peak (*CP_peak*) most of the buffers were read from metabuffer cache; however there was a significantly high CPU activity.

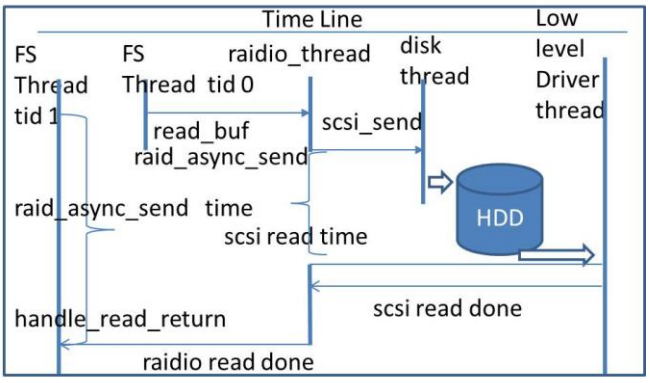


Figure 6: Disk Message Passing

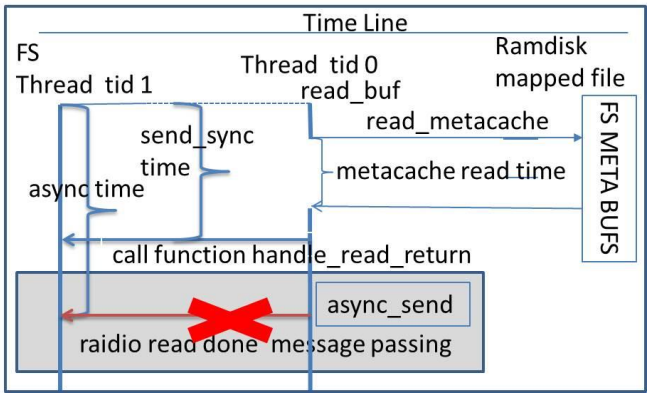


Figure 7: Direct cached buffer passing

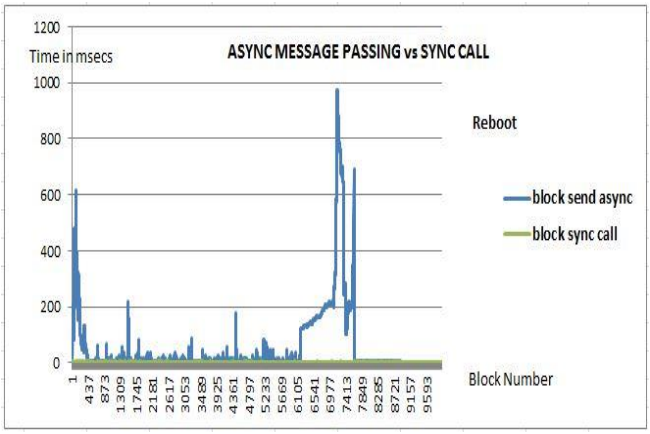


Figure 8: Message passing overhead

4.6 Effect of Read Ahead Parameter

To solve the issue of boot time peak, we investigated which blocks are being read during the early stage of boot time. Many buffers that were being read during the boot cache were dummy reads. The file system was reading them in advance to use them later in the future. These dummy blocks were being fetched from the disk even though their- use was limited during boot time. This disk I/O activity was delaying the mount time of all the metablocks present in metabuffer cache. Disabling read ahead lead to decrease in time for block fetches during boot time peak as shown in Figure 10.

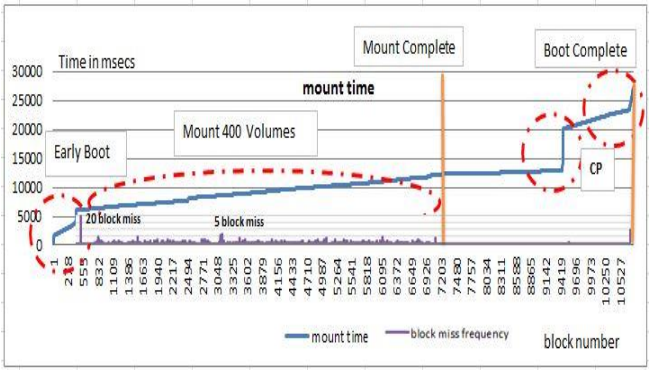


Figure 9: Per block boot time analysis

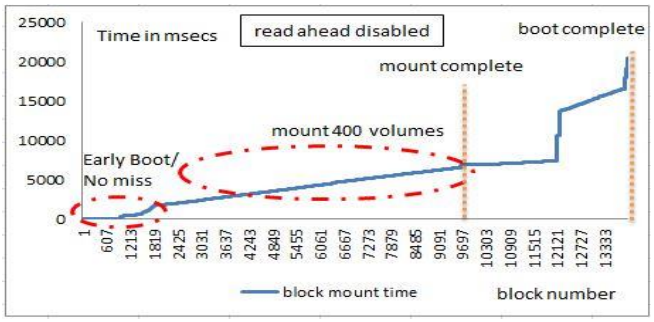


Figure 10: Disabling read ahead optimization

4.7 Consistency Points – Checkpointing

A Journal Based File System [22] replays the journal log periodically or when the journal is full (whichever is earlier). Each such instance of replaying the journal log is called a consistency point (CP). We observed that during replay of the journal log, the time taken to fetch the metadata from buffer cache was hit significantly. During CP which is a CPU intensive activity, the CP thread keeps the vCPU occupied. This job prevents other I/O threads from retrieving the metabuffers from the RAMDisk. Thus during the whole CP cycle, there is a visible increase in the amount of time taken to fetch the metacache blocks from memory. The two peaks shown in Figure 9 during CP are two instances of CP (dirty block flush events) corresponding to two CP sections within the Journal log. This CP is required during boot to bring the file system in to a consistent state prior to accepting and servicing new user I/O requests. We plan to address this problem in our future work by reducing the amount of data required to be flushed during CP.

5. Conclusion

In this paper, we proposed a technique for quick reboot of virtual storage appliance using host hypervisor RAMDisk. Early results were contrary to our intuition that having a faster storage medium would seamlessly help reduce read latency and reboot time. The complete reboot cycle profiling and analysis helped identify reasons for these bottlenecks. We showed the impact that asynchronous message passing and thread scheduling could have on read latency with low latency device. Further, we also identified that read ahead optimization could increase boot time latency, when all the buffers required during boot are already cached. Finally, any journal replay activity further delays read path thereby slowing the boot process. After making appropriate changes to mitigate the effect of the above mentioned bottlenecks we were able to reduce the boot time by approx. 18 %.

VSA Configuration	Mount (%)	boot_complete (%)
Caching Metadata	4.1	5.5
+Disabling Async Call(s)	14.5	15.2
+Disabling Read Ahead	22.1	18.1

Table 2: Percentage Improvement in Reboot time after every optimization

6. References

[1] NetApp High Availability - http://www.netapp.com/in/solutions/data-protection/high-availability.aspx

[2] D. Hitz, J. Lau, M. Malcom, "WAFL – Write Anywhere File Layout" Proc. USENIX Winter 1994 Technical Conference.

[3] S. Tweedie, "The Extended 3 Filesystem" Proc. Linux Symposium 2000, Ottawa.

[4] M. Sullivan and M.Stonebraker, "Using write protected data structures to improve software fault tolerance in highly available database management systems". Proc. 1991 International Conference on Very Large Data Bases (VLDB), pages 171-180. September 1991.

[5] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, D. Lowell, "The Rio File Cache: Surviving Operating System Crashes" 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[6] R. Wahbe, S. Lucco, T. Anderson, S. Graham, "Efficient Software based Fault Isolation" – Proc. 14th ACM Symposium on Operating Systems Principles, pages 203-216, December 1993.

[7] The Linux Kernel – https://www.kernel.org

[8] The Linux Boot Chart http://www.bootchart.org/

[9] Kexec - https://wiki.archlinux.org/index.php/kexec

[10] A. Goel, B. Chopra, C. Gerea, D. Matani, J. Metzler, F. Haq, J. Wiener, "Fast Database Restarts at Facebook" – ACM SIGMOD 2014

[11] Amzon EC2- http://aws.amazon.com/ec2/

[12] O. Barkyn, B. Chopra, C. Gerea, J. Metzler, S. Subhramanian, J. Weiner, D. Reiss, D. Merl "Scuba – Diving into Data at Facebook" Proc. VLDB 2013 .

[13] VMware Server - https://my.vmware.com/web/vmware/free

[14] VMware ESX - https://my.vmware.com/web/vmware/free

[15] Cockcroft, Adrian. AWS RE:INVENT - HIGH AVAILABILITY ARCHITECTURE AT NETFLIX, 2012

[16] Recover- Oriented Computing: http://roc.cs.berkeley.edu/