Policy Based Storage System For Heterogeneous Environment

Dai Qin,[*] Ashvin Goel, Angela Demke Brown,
University of Toronto
{mike,ashvin}@eecg.toronto.edu, demke@cs.toronto.edu

1 Introduction

Storage environments are highly complex and heterogeneous because of different applications generate different workloads and different hardware have different characteristics. Therefore, it is very hard to adapt storage systems to the needs of specific applications and hardware. Most storage systems are not aware of this issue: file systems or other applications talk to the storage system using a simple block layer interface without specifying their needs; storage system treat all data as equal and therefore cannot apply further optimizations to fully take advantage of different hardware.

Current solutions tend to break the encapsulation between storage systems and applications, which results in huge *monolithic* applications. By integrating storage management features, these monolithic applications can fully take advantage of heterogeneous hardware. Complex file systems like ZFS[1] and btrfs[2] integrate some storage management features like metadata replication and software RAID; commercial NFS servers such as Netapp appliance even integrate special hardware along with their software. However, these monolithic applications are very inflexible: features are hardcoded into the file system, and if system administrators want to add new features, they would have to hack the source code.

In contrast with the current solutions which present a complete redesign of the whole storage stack, we believe it's possible to solve this problem outside the application scope. Instead of building special file systems or applications, storage systems can transparently satisfy the needs of generic applications and administrators while fully taking advantages of different hardware capabilities. We believe these current solutions are too monolithic to meet the specific need of applications and system administrators in an extremely heterogeneous environment. It is time to split the monolithic system into a lightweight framework with small modules to solve this problem.

In our framework, we separate the storage management and the applications so that storage system is only responsible for storage management. Features provided by storage system are decomposed into high level storage policies. Different policies can be apply to different applications and different hardware. Composing these policies together, they could meet the demands of applications and system administrators as well as fulling taking advantage of heterogeneous hardware. System administrators can easily configure the storage system by choosing policies from build-in pool. If there is no build-in policy that satisfy their needs, it should be simple for system administrators or developers to write new policies.

2 Approach

Our current system presents a logical block storage device to clients. This model is becoming increasingly popular in virtualization and cloud computing environments because it allows the storage environment to be highly flexible and heterogeneous, while minimizing the requirements on the storage clients. Storage clients can run any applications and file system on the top of the logical block storage devices we presented. In order to provide a common abstraction for policies, we built an abstract mapping layer that can map logical blocks to physical blocks, and our policy specification is implemented by manipulating the mapping layer. For example, for security concern, all unused blocks by the file system need to be mapped to a zeroed out block.

To meet the needs of applications, we often need upper layer application semantics. These semantics can be retrieved either by transparently inferring applications semantics or adding hints to the I/O requests issued by applications. Some previous work has been done in this area: semantically smart disk[3] can infer the semantics of file systems and databases; Differentiated Storage

[*]Student

Services[4] can pass down file systems and applications semantics by adding hints to SCSI protocol. After we receive an I/O request, we automatically tag it using the semantics information we have and pass it down to the policies.

In our system, policies are composable. Different tags on the I/O requests could trigger different policies to handle this request. For example, the caching policies indicate metadata should have a higher cache priority, while at the same time, free blocks need to be mapped to a zeroed out block for security concern. In this example, caching policy are triggered when metadata blocks are requested, while free block policy is triggered specifically when bitmap blocks are updated.

In our experience, the abstract mapping layer is difficult to implement correctly and efficiently. As application metadata could be mapped into different locations, it is necessary to keep the mapping entries consistent with the policy specification as well as the application. From applications' perspective, write requests are usually of 2 types: transactional updates and non-transactional updates. Transactional updates do not happen in-place; data are being written to new locations on the storage, such as file system journal or database log. Application will issue a sync request (disk barrier) to commit a transaction, and the data is guarantee to be consistent at this point. Non-transactional updates are in-place update and issued when application do not care about the consistency of the data; for example, writing data blocks or checkpointing metadata blocks in a journaled file system are non-transactional. Application will issue a sync request only when the data needs to be durable (e.g. `fdatasync()`), and at this point, data might not be consistent. Applications like file system constantly mix these 2 types of updates together and issue to the storage system, and on a sync request it is very hard for storage system to decide whether or not to commit the our mapping table.

3 Current Status

Currently we persist the mapping on every sync request to guarantee data durability, and idempotent policy could work fine because at anytime because it could get recovered by replaying the journal during recovery. We have some example policies like using SSD as a cache, we decompose the caching policy into sub-policies including eviction policy and write-back policy. Different eviction policy and write-back policy can mix and match to satisfy the need of deployment environment. In the near future we will try to come up some more complex policies that could further take advantage of the dynamic properties of the hardware.

References

[1] "Oracle solaris 11 zfs technology," http://www.oracle.com/technetwork/server-storage/solaris11/technologies/zfs-338092.html.

[2] "A checksumming copy on write filesystem," https://oss.oracle.com/projects/btrfs/.

[3] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-smart disk systems." in *FAST*, 2003.

[4] M. P. Mesnier, F. Chen, T. Luo, J. B. Akers, and J. B. Akers, "Differentiated storage services." in *SOSP*, 2011.