Virtual Allocation: A Scheme for Flexible Storage Allocation

Sukwoo Kang, and A. L. Narasimha Reddy
Dept. of Electrical Engineering
Texas A & M University
College Station, Texas, 77843
{*swkang, reddy*}*@ee.tamu.edu*

Abstract

Traditional file systems allocate and tie up entire storage space at the time the file system is created. This creates a situation where one file system could be running out of space, while another file system has ample unused storage space. In such environment, storage management flexibility is seriously hampered. This paper presents *virtual allocation*, a scheme for flexible storage allocation. It separates storage allocation from file system. It employs dynamic allocation strategy based on time of write, which lets applications fit into the actual usage of storage space without regard to configured file system size. Unused storage space can be provided to any application or file system as the needs arise. This improves flexibility by allowing us to share storage space across different file systems. It also makes it possible for a storage device to be expanded easily to other available storage resources because storage allocation is not tied to file systems. This paper presents the design of virtual allocation and an evaluation of it through benchmarks. To illustrate our approach, we implemented a prototype system on PCs running Linux. We present the results from the prototype implementation and its evaluation.

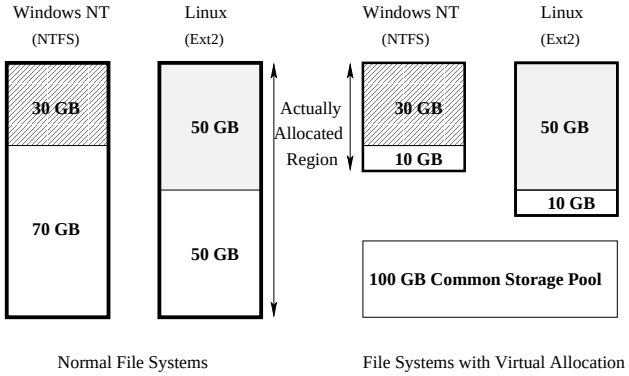

Figure 1: *Example illustrating virtual allocation. Virtual allocation lets storage space be allocated based on actual usage. Highlighted region indicates current usage and bold line represents actually allocated region. In this case, with virtual allocation, we can get 100 GB (60 GB + 40 GB) unallocated storage space across different operating systems and different file systems.*

1 Introduction

Traditional file systems are closely tied to the underlying storage. Storage space is allocated and owned at the time the file system is created. This directly conflicts with the notion of storage as a discoverable resource since unused space in one file system cannot be used easily by another file system running out of space. The file systems currently employ a static allocation approach where entire storage space is claimed at the time of file system creation. This is somewhat akin to running with only physical memory in a memory system. Memory systems employ virtual memory for many reasons: to allow flexible allocation of resources, safe sharing, to allow larger programs to run etc. Traditional file systems lack such flexibility. To address this problem and to enable storage as a discoverable resource, we have developed a *virtual allocation* technique at the block level.

A file system can be created with X GBs. Instead of allocating the entire X GBs of space, virtual allocation allows the file system to be created with only Y GBs, where Y could be much smaller than X. The remaining storage space $(Y - X)$ GBs will be unused and available as a discoverable resource. If the created file system does not exceed the actual allocation of Y GBs, the remaining space $(Y - X)$ GBs can be used by another file system, or application. As the file system grows beyond Y GBs, the storage space can be allocated on demand or when certain usage thresholds are exceeded.

Such an approach separates the storage space allocation from file system size and allows us to create virtual storage systems where unused storage space can be provided to any

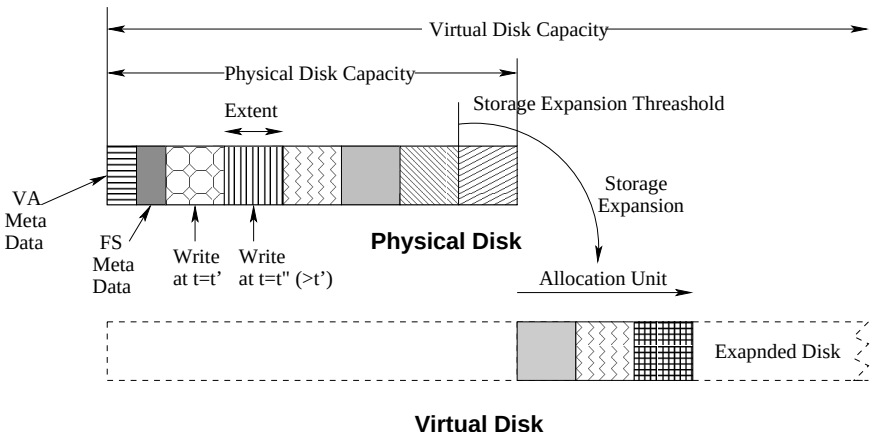

Figure 2: *Allocation strategy of virtual allocation. It employs dynamic allocation strategy based on time of write. All writes is done by the unit of the extent.*

application or file system as the needs arise. This approach allows us to share the storage space across multiple (and possibly different) operating systems. The file systems can function transparently on top of such virtual storage systems as if they have access to a large storage device even though only a small fraction of that device is actually allocated and used at this time. Figure 1 shows two example file systems with and without virtual allocation. In Figure 1, highlighted regions illustrate the current usage of disks and bold rectangles indicate actually allocated region. The figure shows that unused storage space can be pooled across different operating systems and different file systems with virtual allocation.

Many of the ideas presented in this paper are pursued in different forms earlier. IBM's Storage Tank [1] separates meta data operations from data operations to detach storage allocation from file systems. IBM's Storage Tank allows a common pool of storage space to be shared across heterogeneous environments. Object based storage proposed by CMU [2] allows storage allocation to be done at the storage systems as opposed to a file system manager. Log-Structured file systems (for example, LFS [3] and WAFL [4]) allow that storage allocation to be detached from file systems because all new data is written at the end of a log. These approaches are either at the file system level or they require new storage paradigm. In contrast, virtual allocation is a device level approach, so it can provide flexible storage allocation scheme transparent to the file systems. It also works well with existing storage environments with minimal changes. Veritas Volume Manager [5] and other existing LVM products provide considerable flexibility in storage management, but allocate storage space based on file system size.

Virtual allocation has the following combination of characteristics:

- It uses the generic block interface widely used in today's systems. This allows it to support a wide range of heterogeneous platforms, and allows the simplest reuse of existing file system and operating system technology.
- It provides a mechanism to create a virtual storage systems where unused storage space can be provided to any application or file system as a discoverable resource.
- It can be built on existing systems with little changes to operating systems.

The rest of the paper is organized as follows: Section 2 presents the design rationale for virtual allocation and Section 3 describes some details about our prototype implementation and evaluation. Section 4 points to the discussions and future work and Section 5 concludes the paper.

2 Design

In this section, we present the design rationale for virtual allocation and its component architectures. Virtual allocation employs *allocate-on-write* policy i.e., storage space is allocated when data is written. Figure 2 illustrates storage allocation strategy of virtual allocation. First, virtual allocation writes all data to disk sequentially in a log-like structure based on time of write. The figure shows an example that new data is written at time $t = t'$ and at time $t = t''$ where $t'' > t'$. This approach is similar to Log-Structured file systems [3, 4] where every write is written at the end of a log. However, in contrast, in virtual allocation, only storage allocation is done at the end of a log. Once data is written to disk, data can be accessed from the same location i.e., data is updated in-place. Virtual allocation maintains block address mapping information (*block map*) for

Extent Number	Device Number	Start Block Address
0	Dev0	1000
1	Dev0	5000
2	Dev1	2000
3	Dev2	3000

Table 1: *Example of block map in virtual allocation. It contains information where data of each extent really exists.*

this purpose. Virtual allocation's block map is similar to a logical volume manager's (LVM's) block map i.e., it converts file system block addresses to actual physical device block addresses. However, virtual allocation's block map is dynamically constructed as data is written and not at the time of file system creation. Virtual allocation can be seen as a generalization of LVM's functionality.

This block map data structure is maintained in memory and regularly written to disk for hardening against system failures. The on-disk block map, called *VA (Virtual Allocation) meta data*, is stored at the front of the allocation log, as shown in Figure 2. Virtual allocation uses an *extent*, which is a group of (file system) blocks, to reduce the VA metadata information that must be maintained. The block map data structure contains information such as extent number, (mapped) device number, and (mapped) block address. Table 1 shows an example of block map data structure. For example, the first row of Table 1 means that data corresponding to *extent 0* is located at *block address 1000* of block device *Dev0*.

Each entry of the block map corresponds to one extent and whenever a new extent is written, meta data is hardened for data consistency in case of a system failure. If we use a 128 KB extent, the size of VA meta data is less than 96 KB per 1 GB disk space. Large extents can retain spatial locality and reduce the block map size that must be maintained. However, larger extents may cause data fragmentation on disk. For example, since even small requests will occupy large allocated space size of an extent, storage space will be fragmented if following requests are not sequential. The tradeoffs with the size of an extent are similar to the tradeoffs in page size in virtual memory or block size in cache memory systems. We plan to explore this space thoroughly in the future.

File systems tend to create metadata at the time of file system creation. Due to our allocation policy, file system meta data is clustered in front of the allocation log. This *meta data clustering* is denoted by a single *FS (file system) meta data* region in the figure 2. IBM Storage Tank takes an approach of separating meta data and normal data at the file system level. Our allocation policy tends to result in a similar separation of file system metadata from the file system data.

Figure 2 also shows that when the capacity of physical disk is exhausted or reaches a certain limit (*storage expansion threshold*), a physical disk can be expanded to other available storage resources. This process of *storage expansion* allows file systems or applications to potentially span multiple devices. Storage expansion is done in terms of *allocation units*. For example, if the allocation unit is 1 GB, whenever storage expansion is done, virtual allocation adds 1 GB of disk space to an existing disk by finding other available storage resources.

We define a *virtual disk* as a storage device seen by file systems or applications. File systems or applications can function transparently on top of such virtual disk as if they have access to a large storage device even though only a small fraction of that device is actually allocated and used at this time. In virtual allocation, storage expansion can be easily done due to allocate-on-write policy, which makes storage devices not tied to file systems.

Virtual allocation is implemented as a loadable kernel module that can be inserted below any file system. Figure 3 shows that virtual allocation is a thin layer between the file system and the disk device driver. Virtual allocation can be integrated into a LVM layer when it is present. When virtual allocation is stacked on top of the disk device driver, all I/O requests are intercepted by virtual allocation before being passed to the underlying disk. For a write request to an allocated extent, virtual allocation dynamically allocates space of an extent and routes the request to that location. It adds this allocation information to the block map so that data can be accessed later from the same location. For a read request, the block map is consulted for mapping information. If it exists, it is routed to that location. Otherwise, it will be an illegal access because an accessed extent is not written yet. Such a read miss will not occur unless file system metadata is corrupted. We plan to address this exception in the future for applications that access the raw device.

Figure 3 shows component architectures of virtual allocation. Disk layout manager in Figure 3 routes incoming read and write requests to appropriate locations and disk block map manager maintains this information as a block map. Disk capacity manager continuously monitors disk usage and it generates a service message to a storage resource manager once the disk usage reaches a storage expansion threshold. If a storage resource manager is invoked, it tries to find available storage resources in local computer or on the network. Once storage is discovered and obtained, the existing device will be expanded to incorporate the new storage.

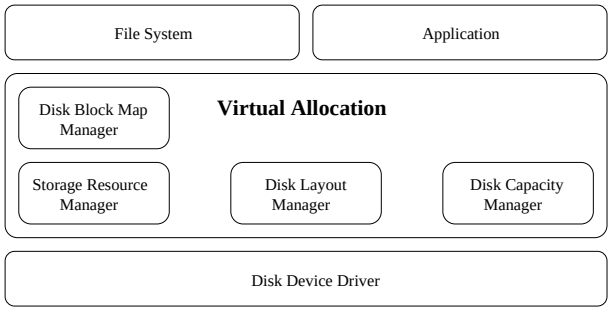

Figure 3: *Virtual allocation architecture. It is located between O/S storage system and disk device driver layer. It intercepts every incoming I/O request and routes it to an appropriate location based on block map.*

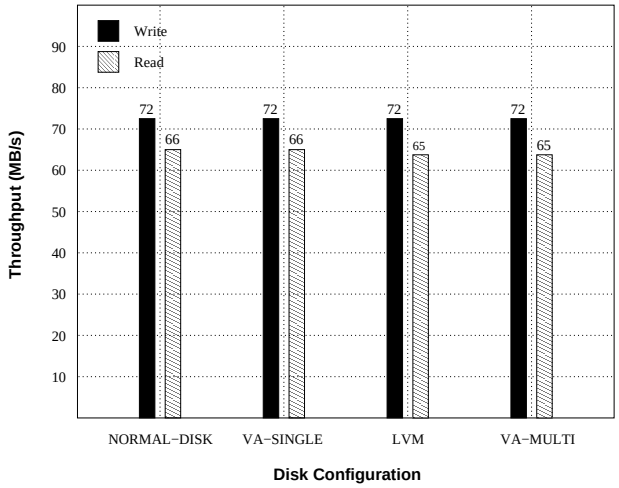

Figure 4: *Throughput for Bonnie++ benchmark. Each group of bars represents a throughput. The leftmost group shows the write and read throughput of a normal disk.*

3 Implementation and Evaluation

We developed a prototype of virtual allocation as a kernel module for Linux 2.4.22 based on layered driver architecture. Virtual allocation layered driver resides between O/S storage system and the disk device driver interface. It intercepts every incoming I/O requests and routes them dynamically based on block map. In-memory block map was implemented as a hash table.

We evaluated the performance of virtual allocation on a 3.0 GHz Pentium 4 machine with 900 MB of RAM. All experiments were conducted on two 36.7 GB 10,000 RPM Seagate ST336607LW SCSI disks formatted with Ext2. The partition size was just large enough to accommodate the test data on each experiment. The machine ran Red Hat Linux 9 with a 2.4.22 kernel. Before doing each test, we unmounted the file system on which the experiments took place to ensure cold cache. We ran all tests at least ten times, and computed 95 % confidence intervals for the mean throughput.

Virtual allocation can be used with multiple different configurations. In our benchmarks, we chose indicative configurations to evaluate performance over available features. We selected four configurations to compare performance:

NORMAL-DISK: It serves as a baseline for performance of other configurations.
VA-SINGLE: Virtual allocation configured to operate on a single disk. 128 KB extent is used.
LVM: Logical volume composed of two local disks. We configured LVM to have 1 GB partition of one disk and 2 GB partition of the other disk.
VA-MULTI: Virtual allocation configured to span two disks. 128 KB extent is used. Its allocation unit is 1 GB and storage expansion threshold is 80 %. Its' disk configuration is same as LVM. If disk usage becomes more than 80 % of one disk, it is expanded to other disk for adding 1 GB (allocation unit) disk space.

We tested our configurations using two workloads: one with large files and the other with typical distribution of file sizes in a file system. We chose these benchmarks so that we could evaluate the performance under different system activity levels. The first workload was sequential reads and writes of large files of Bonnie++ benchmark [6]. We used a test file of 2 GB size and a 4 KB chunk size. It sequentially writes a test file to disk and then sequentially reads it by the unit of the chunk size. In the above configurations of **LVM** and **VA-MULTI**, 1 GB of the test file is written to one disk and the other 1 GB of test file is written to the second disk in both the cases.

The second workload we chose was Postmark [8]. We configured Postmark to create 30,000 files (between 8KB and 64KB) and perform 100,000 transactions in 200 directories. This file size range matches file size distributions reported in file system studies [7]. Total accessed data of this test was 5 GB (Read: 2 GB, Write: 3 GB), which was much larger than the system RAM (900 MB). Postmark focuses on stressing the file system by performing a series of file system operations such as file creations, deletions, reads, and appends.

Figure 4 shows the results of the Bonnie++ benchmark. The figure depicts write and read throughputs under different configurations. The height of the bar depicts the throughput. Each group of bars shows the throughput for a particular configuration. The leftmost group shows the throughput of normal disk for reference.

Virtual allocation configured on a single disk with 128 KB extent induced little overhead compared to normal disk.

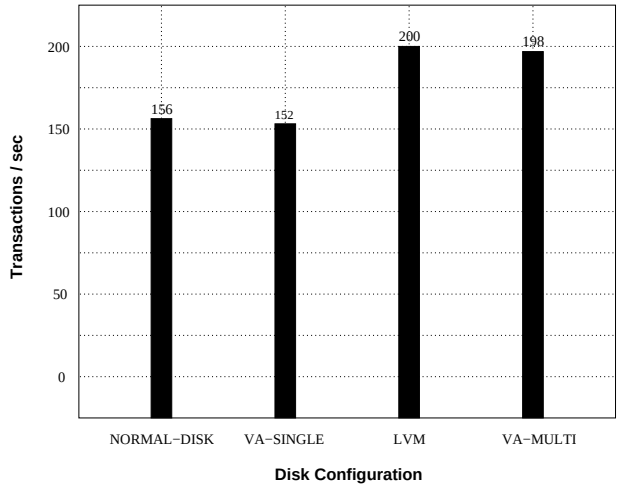

Figure 5: *Transactions for Postmark. Each group of bars represents a transaction throughput. The leftmost group shows the transaction throughput of a normal disk.*

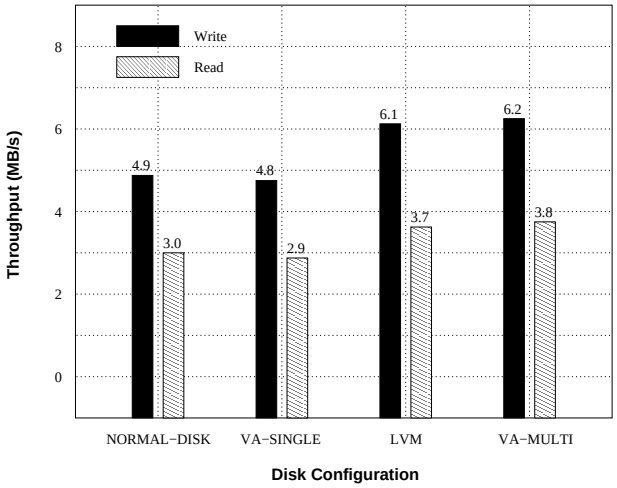

Figure 6: *Write and read throughput for Postmark. Each group of bars represents a throughput. The leftmost group shows the write and read throughput of a normal disk.*

Similarly, virtual allocation configured on multiple disks with 128 KB extent performs comparable to the LVM configuration. Above results show that virtual allocation works comparable to a normal disk in a workload with large files.

Figure 5 shows the transaction throughput for Postmark. This figure has the same structure as Figure 4 and shows results for the same configurations for Postmark workload. The figure shows that virtual allocation on a single disk with 128 KB extent incurs 2.6 % overhead. Virtual allocation on multiple disks with 128 KB extent incurs 0.9 % overhead. Figure 6 shows write and read throughput for Postmark. The figure shows that the read and write performance of virtual allocation is comparable to a normal disk or LVM. These results show that virtual allocation also can be used well in a workload with small files.

Our preliminary results show that virtual allocation may incur little or no significant overhead in both the workloads. As explained earlier, our allocation policy results in the separation of file system's metadata from its data. We plan to conduct more tests in the future to study the impact of such separation. The file systems put significant effort into keeping the metadata in close proximity to the data. Earlier studies [9, 10, 11] have shown the importance of such strategies and it is necessary to closely evaluate our approach in light of these studies.

4 Discussions & Future Work

The IP connectivity of I/O devices [12] makes it possible for storage devices to be added to the network and enable storage as a discoverable resource. Virtual allocation, through its separation of storage allocation from file system creation, potentially allows file systems to span multiple devices across the network. If file systems are designed to span multiple devices across a network, locality and the resulting performance issues will be of immediate concern. While flexible deployment may dictate that we do not worry about the actual location of storage, performance issues may force us to revisit data allocation and organization issues in the future.

When storage resources need to be shared over a network, virtual allocation's block map could be extended to enable local disk caching in unallocated disk space to offset large data access latencies. We plan to pursue this in the future. We plan to investigate further the effects of meta data and data separation. We plan to study the tradeoffs in virtual allocation, such as the impact of the extent size, the impact of the allocation unit and the thresholds for triggering allocation more thoroughly in the future.

We will also investigate the interaction between virtual allocation, and RAID, mirroring and striping typically employed within storage systems [13]. Our allocate-on-write policy would need to be suitably modified to fit into the logical device characteristics advertised by the LVM.

5 Conclusion

We have proposed virtual allocation employing an allocate-on-write policy for improving the flexibility of managing storage across multiple file systems/platforms. By separating storage allocation from file system creation, common storage space can be pooled across different file systems and flexibly managed to meet the needs of different file systems. Virtual allocation also makes it possible for storage

devices to expand easily so that existing devices incorporate other available resources. We have shown that this scheme can be implemented with existing file systems without significant changes to operating systems. Our experimental results from a Linux PC-based prototype system demonstrated the effectiveness of our approach.

6 Acknowledgments

We would like to thank Uday Gupta and Jerry Cotter at EMC. Discussions with them motivated this work. We also thank Pat Shuff for his valuable comments.

References

[1] IBM. IBM Storage Tank – A Distributed Storage System. IBM White Paper, http://www.almaden.ibm.com/storagesystems/file_systems/storage_tank/papers.shtml, 2002.

[2] Mike Mesnier and et al. Object-Based Storage. *IEEE Communications Magazine, v.41 n.8 pp 84-90*, August 2003.

[3] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 1991.

[4] Dave Hitz and et al. File System Design for an NFS File Server Appliance. *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[5] VERITAS. Volume Manager for Windows with MSCS. VERITAS White Paper, http://www.veritas.com/van/products/volumemanagerwin.html, 2002.

[6] Bonnie++. A benchmark suite aimed at performing a number of tests of hard drive and file system performance. *http://www.coker.com.au/bonnie++*, 2001.

[7] W. Vogels. File System Usage in Windows NT 4.0. *In Proceedings of the 17th ACM Symposium on Operating Systems Principles, pages 93-109*, Dec. 1999.

[8] J. Katcher. PostMark: a New Filesystem Benchmark. *Technical Report TR3022, Network Appliance*, 1997. *www.netapp.com/tech_library/3022.html.*

[9] M. McKusick, W. Joy, S. Leffler and R. Fabry. A fast file system for UNIX. *ACM Trans. on Computer systems*, Aug. 1984.

[10] G. Ganger and Y. Patt. Metadata update performance in file systems. *Proc. of OSDI*, 1994.

[11] R. Wang, T. Anderson and D. Patterson. Virtual log based file systems for a programmable disk. *Proc. of OSDI*, Feb. 1999.

[12] M. Krueger, R. Haagens, C. Sapuntzakis, and M. Bakke. RFC 3347:Small Computer Systems Interface protocol over the Internet (iSCSI) Requirements and Design Considerations, 2002.

[13] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Array of Inexpensive Disks (RAID) . *In Proceedings of ACM SIGMOD*, 1988.