Orchestra: Extensible Block-level Support for Resource and Data Sharing in Networked Storage Systems

Michail D. Flouris[†], Renaud Lachaize[⋆], Angelos Bilas[‡]
Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece
{flouris, rlachaiz, bilas}@ics.forth.gr

Abstract

High-performance storage systems are evolving towards decentralized commodity clusters that can scale in capacity, processing power, and network throughput. Building such systems requires: (a) Sharing physical resources among applications; (b) Sharing data among applications; (c) Allowing customized views of data for applications.

Current solutions satisfy typically the first two requirements through a distributed file-system, resulting in monolithic, hard-to-manage storage systems. In this paper, we present Orchestra, a novel storage system that addresses all three above requirements below the file-system by extending the block layer. To provide customized views, Orchestra allows applications to create semantically-rich virtual block devices by combining simpler ones. To achieve efficient resource and data sharing it supports block-level allocation and byte-range locking as in-band mechanisms.

We implement Orchestra under Linux and use it to build a shared cluster file-system. We evaluate it on a 16-node cluster, finding that the flexibility offered by Orchestra introduces little overhead beyond mandatory communication and disk access costs.

1 Introduction

Over the past decade, several trends have significantly shaped the design of storage systems. First, storage area networks (SANs) [15] that allow many servers to share a single pool of physical resources, such as disk arrays, have emerged as a popular solution to improve efficiency and manageability, yet at high costs. Second, to achieve cost efficiency, storage systems will be increasingly assembled from commodity components, such as workstations, SATA disks and Ethernet networks. Thus, we are in the middle of an evolution towards new architectures ("storage bricks") made of many decentralized, networked components with increased processing and communication capabilities [7]. Third, much research work has investigated how systems software can provide a single storage image to applications, without compromising the features and performance offered by high-end storage systems [17, 18, 6].

Overall, this transition to "pooled", networked storage systems involves addressing three major challenges: (i) Flexible resource and data sharing, (ii) Reliability and availability, and (iii) Security. These issues are important because on one hand traditional storage systems have provided strong guarantees and on the other, networked storage systems cannot use traditional approaches due to their distributed nature and are more susceptible to related problems. In this paper we address the first of these issues by providing support at the block-level that allows seamless sharing of physical resources among applications and shared access to the stored information.

Sharing in a storage system refers usually to two distinct aspects of the system: (i) physical resources and (ii) access to data. Traditionally both of these functions have been supported by the file-system, which has been responsible for pooling the available resources and coordinating read/write access to stored data. Over time, sharing of physical resources has been increasingly supported by adding management and configuration flexibility at the block-level. Today's storage area networks allow multiple hosts to share physical resources by placing multiple virtual disks over a single pool of physical disks. However, current levels of resource sharing incur significant limitations; For instance, logical volumes in a Fibre-Channel storage system should not span multiple storage controllers. Besides, distinct application domains have very diverse storage requirements; Systems designed for the needs of scientific computations,

[†]Also, with the Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, ON, M5S 3G4, Canada.

[⋆]Currently, with the Department of Computer Science, Universite Joseph Fourier, Grenoble, France.

[‡]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

e-mail serving or data archival impose different trade-offs in terms of dimensions such as speed, availability and consistency. Yet, current block-level systems provide only a limited set of functions, for instance they rarely support versioning or space saving techniques.

In terms of data sharing, e.g. concurrently accessing data in a single logical disk from many application clients, storage systems employ a (distributed) file-system to coordinate data accesses. Such systems require mechanisms for distributed coordination, allocation, and metadata consistency, which have proven extremely challenging to scale at the file-system level.

In this paper, we present Orchestra, a system that supports resource and data sharing at the *block level*. Our contributions are mainly two: First, the main idea for providing highly flexible resource sharing is to treat data and control requests in a uniform manner, propagating all I/O requests through the same path from the application to the physical disks. This allows us to create distributed, virtual I/O hierarchies that provide functional flexibility, over the same pool of resources. Second, we examine how in-band locking and space allocation at the block level can be used to support data sharing. Our experience shows that this approach reduces complexity significantly. We are able to prototype a simple, user-level file-system that supports sharing with limited effort. Our performance results show that our approach does not introduce significant overheads and scales both at the block as well as the file-system layer.

Finally, we should mention that our prototype supports a novel, lightweight transactional approach to also achieving reliability at the block-level [5]. However, this is beyond the scope of this paper, which focuses on sharing mechanisms and scalability. For this reason we perform all experiments with the transactional extensions switched off.

The rest of the paper is organized as follows. Section 2 presents the design of Orchestra, emphasizing its main contributions. Section 3 describes how Orchestra facilitates resource and data sharing. Section 4 presents details of our prototype implementation. Section 5 presents our experimental results. Section 6 discusses related work and finally, Section 7 draws our conclusions.

2 System Design

Orchestra supports *extensible, distributed virtual hierarchies* to allow storage systems to extend their functionality and to provide different views and semantics to applications without compromising scalability. This basic mechanism helps achieve Orchestra's three main goals, discussed next.

2.1 Application-specific customization

Orchestra builds on Violin [4] for creating custom virtual hierarchies by writing new virtual I/O modules or combining existing ones. Examples of three distributed Orchestra hierarchies are shown in Figure 1. Virtual hierarchies allow applications to create storage views that are highly tuned to their needs.

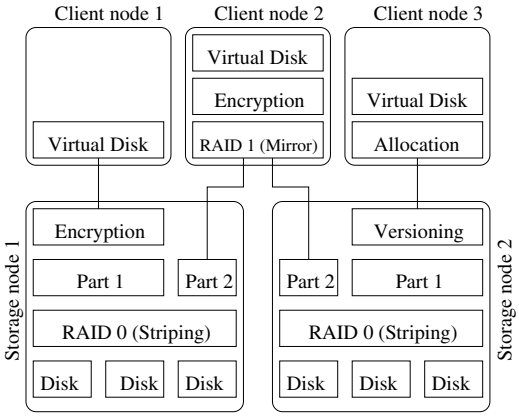


Figure 1. Examples of Orchestra hierarchies.

Violin [4], is an extensible virtual I/O framework for (i) building and (ii) combining block-level virtualization functions in single-node systems. Violin allows storage administrators to create arbitrary, acyclic graphs of virtual devices, each adding to the functionality of the successor device(s) in the graph. Every virtual device's function is provided by independent virtualization modules that are linked to the main Violin framework. A linked device graph is called a hierarchy and represents essentially a virtualization stack of layered modules (also denoted as "layers"). Blocks of each virtual device in a hierarchy can be mapped in arbitrary ways to the successor devices, enabling advanced storage functions, such as dynamic relocation of blocks.

Violin provides virtual devices with full access to both the request and completion paths of I/Os allowing for easy handling of both synchronous and asynchronous I/O. Additionally, it deals with metadata persistence of the full storage hierarchy, offloading the related complexity from individual modules. Overall, Violin facilitates the development of simple virtual modules, which can be layered to hierarchies with rich semantics that are difficult to achieve otherwise, especially in the OS kernel.

2.2 Scalable Resource Sharing

Virtual hierarchies share a common set of distributed physical resources. This requires "splitting" hierarchies over the network in a transparent manner and transferring both data and control requests among nodes. Users may create distributed hierarchies that span application and storage nodes, based on application and system requirements. Most systems traditionally support in-band operations for data requests and out-of-band for control requests, such as configuration and synchronization [20].

Orchestra propagates both data and control requests through virtual hierarchies, making use of a single in-band mechanism. Locking and allocation, for instance, are in-band operations in Orchestra. This approach offers potential for scalability, since the load can be distributed to more resources, while preserving a good degree of flexibility regarding the mapping of control operations to system resources and the structure of the "control plane".

In general, I/O requests may be: (a) Device-level requests that refer to a block device, such as a command to take a new snapshot or to allocate one or more blocks. (b) Block-level requests that refer to one or more blocks, such as read, write, and block-range locking. Device-level requests are straightforward to implement. The main issue with block-level requests, both the traditional data read-write requests and our extended control requests, that traverse virtual hierarchies is related to how we treat block address arguments. Each virtual device is able to "see" and thus, reference only the block addresses of its underlying device(s). Thus, as control requests propagate in the virtual hierarchy, the addresses of the blocks they reference need to be translated, similar to I/O requests.

We have generalized this concept in Orchestra by allowing devices to provide address-mapped control requests (commands) in addition to existing regular control requests. All control requests may be issued by any virtual device and traverse the virtual hierarchy top to bottom. If a device does not understand a specific control request it forwards it to its lower device(s). Eventually, a control command reaches a device that handles it, possibly generating or responding to other control and data requests. When a control request is issued, it is either handled by the current module in the hierarchy or automatically forwarded to the next device(s).

In addition, address-mapped commands are subject to translation of arguments that represent byte-ranges. This is achieved by extending the module API, with an address-mapping API call. This call is written by module developers for every module that is loaded in an Orchestra hierarchy and translates an input byte-range to any output byte-range(s) in one or more output devices. Block mapping depends on the functionality of individual modules. Although complex mappings are possible, in many cases, mappings are simple functions. Implementing the address-mapping method is a straightforward task for the module developer, since this is usually identical to the mappings used for read and write I/O calls through the layer to the output devices.

Finally, to allow virtual hierarchies to span multiple nodes, Orchestra needs to transfer requests between virtual modules that execute on different nodes. For this purpose, we currently use a simple network protocol over TCP/IP.

2.3 Data Sharing at the block level

Orchestra allows applications to share its virtual devices at the block level. Sharing requires coordinating (i) accesses to data/metadata via mutual exclusion and (ii) allocation and deallocation of storage space. In Orchestra, concurrent accesses to shared data or metadata are coordinated by means of a locking mechanism integrated in the virtual hierarchies as an optional virtual module. Free-space allocation is managed by a distributed free-block allocation facility, called "allocator", which is also built as an optional virtual device that may be inserted at appropriate places in a virtual hierarchy. To support seamless, coherent sharing for distributed, file-based applications, Orchestra provides a simple user-level library, 0FS that implements a conventional file-system API (described in section 3).

2.3.1 Byte-range locking

As mentioned, Orchestra provides support for byte-range locking over a distributed block volume. The main metadata in the locking layer is a a free-list that contains the *unlocked* ranges of the managed virtual volume. When a lock control request arrives, the locking layer uses its internal metadata to either complete or block the request. At an unlock request the locking layer updates its metadata and possibly unblocks and completes a previous pending lock request. Our locking API currently supports multiple-reader, single-writer locks in both blocking and non-blocking modes.

To achieve mutual exclusion, locks for a specific range of bytes should be serviced by a single locking virtual layer. This is achieved by placing locking layers at specific points in the distributed hierarchy. Multiple layers may be used for servicing different byte ranges. Thus, load balancing lock requests across multiple nodes is simplified.

Lock and unlock requests are address-mapped commands, which allow us to distribute locking layers to any desirable serialization point in a distributed hierarchy. Such points are typically places where a local device is exported from a storage node. A locking device in this case allows locking support for any remote layer using this exported device. The lock and unlock commands are forwarded to the specific node through the Orchestra communication mechanism and the associated addresses are mapped according to the distributed hierarchy mappings.

Note that the metadata of a locking layer does need not be persistent. Instead, a lease-based mechanism to reclaim locks from a failed client is adequate. Finally, lock availability can be achieved through mere replication of storage nodes, in the same way that one would configure a storage hierarchy for increased data availability.

2.3.2 Block allocation

The block allocator handles distributed block management in a consistent manner for clients sharing the same volume. It distributes free blocks to the clients and maintains a consistent view of occupied and free blocks. All such block-liveness information is maintained by the allocator, offloading all the potentially complex free-block handling code from higher system and application layers.

In addition, providing a space allocation mechanism at the block level allows to handle (dynamic) reconfiguration in an easy way: The allocation policy and size of a volume can be modified transparently to higher layers. Besides, the allocator's block-liveness state can be used to make necessary management tasks, such as backup and migration, more time- and space- efficient.

The allocator metadata need to be maintained consistent across allocator instances running in different nodes (Figure 2). This is achieved by using byte-range locking to synchronize access to the shared free-list and bitmaps. Frequent locking at fine granularity will result in high allocation overheads. To address this issue we amortize the overhead associated with locking metadata by dividing the available (block) address space of a shared volume in a sufficiently large number of allocation zones. Each zone is independent and has its own metadata, which can be locked and cached in the node using this particular zone.

Locking works as follows: on the first allocation request for a free block, the allocator identifies an unused (and unlocked) zone in the shared device and locks the zone, loading the zone's metadata in memory. Subsequent allocation requests will use the same zone, as long as there are available free blocks, thus, avoiding locking overheads. When a zone does not have enough free blocks to satisfy a request, a new zone is allocated and locked. This scheme essentially increases the locking granularity to full zones. The metadata of locked zones are automatically synchronized to stable storage, similarly to all other module metadata in Violin, in two occasions: (i) periodically every few seconds and (ii) when a zone is unlocked and its metadata is released.

Freeing blocks can be more complex than block allocation. If the blocks to be freed are in a locally locked zone, the module will free the blocks in the local metadata maps. If, however, the block belongs to a zone not locally locked, the module will first attempt to lock the corresponding zone in order to free the blocks. If it is successful, it will free the blocks and will keep the zone locked for a short time to avoid the locking penalty of successive free operations. Even though this case may incur high overhead, we expect that block deallocation will be clustered in zones due to the allocation policy and the (expected) large zone sizes, thus, minimizing deallocation overheads.

If, during free operations, the allocator fails to lock a specific zone, it uses small logs for *deferred free operations*, called *defer logs*. The defer log for a zone is a persistent metadata object. When an allocator cannot lock a zone for deallocation purposes, it locks the zone's defer log, appends the pending free operation, and releases the defer log lock in case other allocator modules run into the same situation. If the defer log is full or already locked, the allocator module waits until the log is emptied or unlocked. The responsibility for processing the defer log lies upon the allocator that has locked the corresponding zone. Every allocator module that owns a lock on a zone, periodically checks the defer log for pending deallocation requests. If there are any pending operations in the log, they are performed on the locked metadata and the defer log is emptied.

3 Efficient File-level Data Sharing

Many applications access storage through a file-system interface. To allow such applications to take advantage of the advanced features of Orchestra and to demonstrate the effectiveness of our volume sharing mechanisms, there is a need to provide a file-system interface on top of Orchestra.

One approach to achieve this is to use an existing distributed file-system. Depending on the requirements of the file-system, Orchestra can be used to either provide a number of virtual volumes each residing in a single storage node [16], or a single distributed virtual volume built out of multiple storage nodes [20]. However, existing systems in either case would not take advantage of distributed block allocation and locking primitives provided by Orchestra.

For these reasons, we provide our own file-system on top of Orchestra. ZeroFS (0FS) is a *stateless, pass-through* file system that translates file calls to the underlying Orchestra block device. Our approach demonstrates also that, by extending the block layer, we are able to significantly simplify file-system design in distributed environments and especially when it is necessary to support system extensibility. Thus, the block allocator uses the block volume facilities for free-list allocation and locking. The main feature of 0FS is that, unlike distributed file-systems, it does not require explicit communication between separate instances running on different application nodes. Usually, communication is required for two purposes: (i) mutual exclusion and (ii) metadata consistency. 0FS uses the corresponding block-level mechanisms provided by Orchestra volumes for this purpose:

Mutual exclusion: 0FS uses the multiple-reader, single-writer locking mechanism provided by Orchestra to achieve mutual exclusion between multiple applications accessing a single file-system through a single or multiple application nodes. Currently, 0FS locks and unlocks files during the `open/close` calls, which is coarser than locks on read/write operations, but realistic enough for many applications. Support for locking on reads/writes, is nonethe-

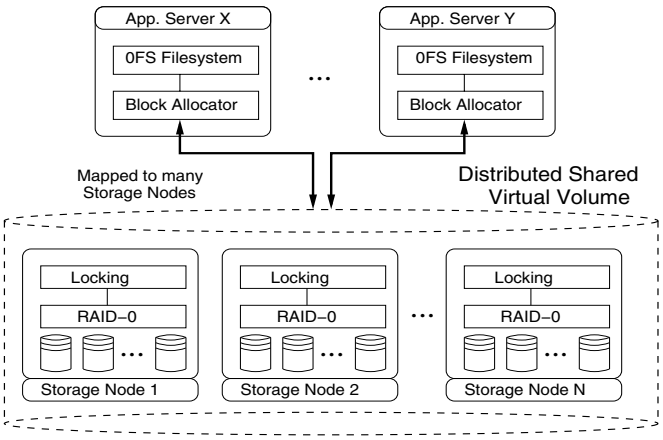


Figure 2. Distributed file-level sharing.

less important for clustered applications that heavily rely on shared files. We expect to introduce these features in a future version without significant effort.

Metadata consistency: The only metadata required by 0FS are i-nodes and directories. To avoid maintaining internal consistent state in memory, 0FS does not perform any caching of metadata or data but uses the underlying block device. Thus, accesses to files or directories in 0FS may result in multiple reads to the underlying block device for the corresponding i-nodes and directory blocks. In the worst case, four read requests may be necessary for very large files. Figure 2 shows how 0FS is combined with Orchestra to provide file-level sharing over distributed volumes.

4 System Implementation

Orchestra is implemented as a loadable block device driver module in the Linux 2.6 kernel. User-level tools are used to create new or modify existing I/O hierarchies. Virtual I/O layers are implemented as separate loadable kernel modules, which are not "classical" device drivers themselves, but rather use Orchestra's programming API. We have implemented various virtual modules, notably RAID 0 and 1, versioning, partitioning, device aggregation, encryption, and data migration between devices. The networking layer for exporting Orchestra devices to other storage nodes or clients, is current TCP/IP. Our latest prototype including the modules consists of about 40,000 lines of kernel C code.

0FS is currently implemented as a user-level library (about 10,000 lines of C) that may be linked with any application during execution. We choose to implement 0FS at user-level to reduce the development effort as well as to allow for easy customization. 0FS currently supports most of the conventional file-system operations, such as *open, remove, creat, read, mkdir*. We are able to compile and run standard benchmarks without code modifications.

5 Experimental Results

In this section we examine the scalability of Orchestra and 0FS on a setup with multiple storage and application nodes. Our evaluation platform is a 16-node cluster. All the cluster nodes are equipped with dual AMD Opteron 242 CPUs and 1 GB of RAM, while storage nodes have four 80GB SATA Disks. All nodes are connected with a 1 Gbit/s Ethernet network through a single switch All systems use the 2.6.12 Linux kernel.

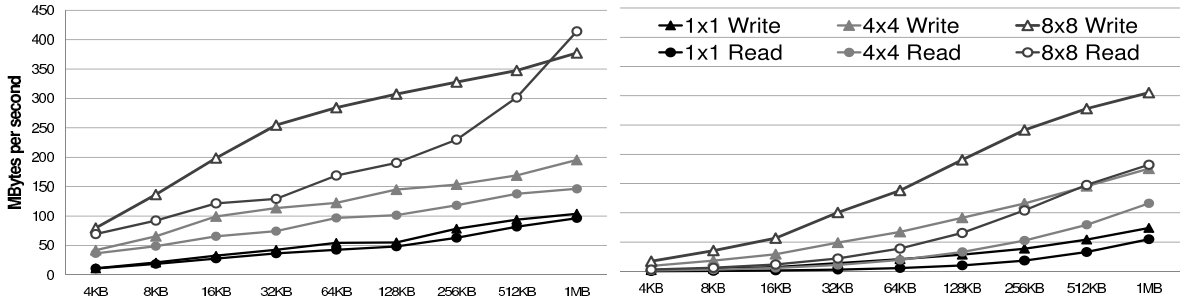
We use three I/O benchmarks: xdd [8], IOzone [14] and PostMark [9]. We use Postmark and IOzone to examine the basic overheads in 0FS and xdd on raw block devices to examine block I/O overheads. With Xdd we use several workloads, varying the number of outstanding I/Os, the read-to-write ratio and the block size. Due to lack of space we only show results with 100% read and 100% write workloads, one outstanding I/O, and block sizes ranging from 4 KBytes to 1 MByte. In all cases, we run xdd concurrently for three minutes and we report numbers averaged over three consecutive runs for each block size.

We use IOzone to study file I/O performance for the following workloads: Read, write, re-read, and re-write. We vary block size between 64 KBytes and 8 Mbytes and we use a file size of 2 GBytes for each client. In PostMark [9], each client uses a workload of 2000 initial files, 5000 transactions and file sizes ranging from 128KB to 1MB.

In our evaluation we examine basic overheads and scalability of Orchestra and 0FS. We present three setups, where we vary the total number of nodes between 2, 8, and 16. To facilitate interpretation of results, we use the same number of storage and application nodes, resulting in three configurations: 1x1, 4x4, and 8x8. Each server node hosts a local RAID-0 module over four disks and each client nodes hosts a RAID-0 module over all the server nodes (1, 4 or 8).

5.1 Orchestra

Basic costs: We measure the cost of individual operations in a local and a distributed virtual hierarchy (volume), averaging values over 1000 calls. A local null ioctl costs about 4 μs, while allocating and freeing blocks through a local allocator module costs about 12 and 13 μs, respectively. In the distributed volume case, there is an additional 300 μs overhead, that includes mainly the network delay and remote CPU interrupt and thread scheduling costs. Overall, we notice that the main overhead in networked volumes comes from the communication subsystem, which is expected to improve dramatically in storage systems with current developments in system area interconnects, such as 10 Gigabit Ethernet and Infiniband. Finally, using `ttcp` in our setup results in a maximum network throughput of about 112 MBytes/s (900 Mbits/s) per node.



(a) Sequential I/O (b) Random I/O

Figure 3. Throughput scaling with xdd for the 1x1, 4x4, and 8x8 setups.

On a single node we compare the performance of an Orchestra hierarchy using a RAID0 module on four SATA disks, with a RAID0 volume built with existing drivers (Linux MD). The performance (throughput and latency) obtained for all workloads (sequential and random) over Orchestra is similar to the performance of the MD driver.

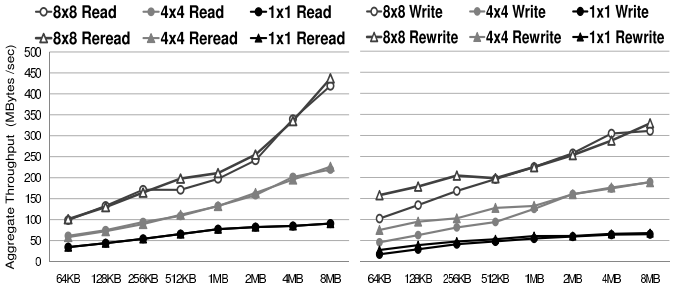
Scalability: We now examine the scalability of Orchestra at the block-level using xdd, with the three previously described setups. Each xdd process runs at different block offsets and thus accesses separate block ranges, minimizing caching effects on the storage nodes.

Figure 3(a,b) shows sequential and random read and write xdd throughput for each configuration: 1x1, 4x4, and 8x8. Overall, we notice that as we increase the number of nodes, both read and write performance scale. For instance, in the 1x1 setup, maximum throughput is about 100 MBytes/s, while in the 8x8 setup it exceeds 400 MBytes/s. However, scalability is limited by the disks, because as the number of nodes (and in turn xdd processes) is increased, the efficiency of individual disks drops, as they perform more seeks between additional separate block ranges.

5.2 ZeroFS (0FS)

Single node performance First, we look into the base performance of 0FS on a single node. We contrast its behavior to the standard Linux ext2 using both IOzone and Postmark (results not shown here due to lack of space). IOzone results show similar performance over reads, while for writes, ext2 is about 20% faster. In the Postmark experiments and for small workloads that fit in the buffer cache, 0FS is about 25% slower. We attribute this performance difference to the increased number of system calls of the user-level 0FS compared to the in-kernel ext2 file-system and the metadata caching that ext2 uses. We see that 0FS performs, in the local case, close to current file-systems.

Performance of distributed setups Next, we examine the scalability of 0FS in distributed setups. In the base distributed configuration (1x1) we create a single volume

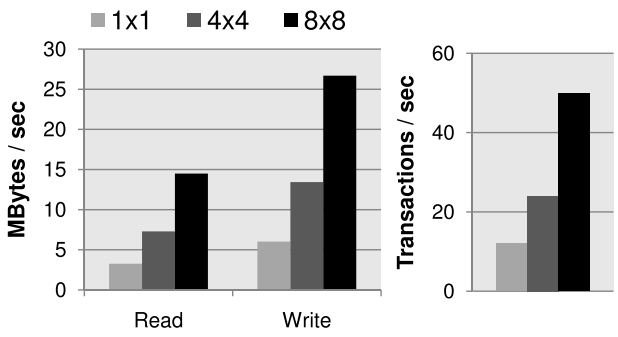


(a) Read / Reread (b) Write / Rewrite

Figure 4. 0FS throughput with IOzone.

over all available system storage and then create different directories for every application node that will access this volume. For each setup (1x1, 4x4, 8x8), each server-side hierarchy hosts a locking module stacked on top of its local RAID-0 module, while each client hierarchy includes an allocator module stacked on top of its RAID-0 module. Besides, each client runs its own instance of 0FS. Each application node uses a separate directory on the distributed 0FS to perform its file I/O workload either with IOzone or Postmark. This is the most realistic scenario for evaluating the scalability of a distributed FS, since no system should be expected to scale well when there is heavy sharing of files (or byte ranges) between clients. Finally, note that since 0FS is stateless, it does not perform any client-side caching, and thus, all I/O requests generate network traffic. Figures 4 and 5 show our multi-node results with IOzone and Postmark, respectively.

Figure 4 shows that IOzone (over 0FS) scales as the number of nodes increases for both read and write, reaching a maximum throughput of about 430 MBytes/s in the 8x8 configuration. We also see that IOzone performance and scaling follows the behavior of block-level I/O (xdd) in Figure 3. Similarly to the block-level I/O, scaling from 1x1 to 4x4 to 8x8 nodes is not linear because of reduced disk efficiency as the number of nodes (and disks increases). Thus, given that disks are not the bottleneck, 0FS is able to scale



(a) Throughput (MBytes/sec) (b) Transaction rate

Figure 5. Postmark results over 0FS.

well in cases where there is limited contention. Similarly, our Postmark results (Figure 5) show scaling behavior similar to IOzone. We see that quadrupling the number of storage nodes between 1x1 and 4x4 results in about doubling all metrics. Similarly further increasing the number of storage nodes from 4 to 8 results in doubling performance.

5.3 Summary

Overall, Orchestra is able to scale well when throughput is not limited by increased seek overheads in physical disks. Moreover, 0FS follows the same scaling pattern as Orchestra in cases where there is little sharing contention. Finally, these results suggest that our approach for providing increased functionality and higher-level semantics at the block-level has potential for scaling to larger system sizes.

6 Related Work

Next, we review existing systems and then we present related work on flexible software frameworks for distributed storage.

Conventional cluster storage systems: Currently, building scalable storage systems that provide storage sharing for multiple applications relies on layering a distributed file-system on top of a pool of block-level storage. This approach is dictated by the fact that block-level storage has limited semantics that do not allow for performing advanced storage functions and especially that are not able to support transparent sharing without application support. Efforts in this direction include distributed *cluster file systems* often based on VAXclusters [10] concepts that allow for efficient sharing of data among a set of storage servers with strong consistency semantics and fail-over capabilities. Such systems typically operate on top of a pool of physically shared devices through a SAN. However, they do not provide much control over the system's operation.

Modern cluster file-systems such as GFS [16] and GPFS [18] are used extensively today in medium and large scale storage systems for clusters. However, their complexity makes them hard to develop and maintain, prohibits any practical extension to the underlying storage system, and forces all applications to use a single, almost fixed, view of the available data. The complexity of cluster file-systems is often mitigated by means of a logical volume manager which virtualizes the storage space and allow transparent changes to the mappings between data and devices while the system is on-line. To the best of our knowledge, all existing volume managers either lack functional extensibility or combined support for volume sharing, distributed hierarchies and persistent metadata.

Flexible distributed storage: To overcome the aforementioned limitations of the traditional tools, a number of research projects have aimed at building scalable storage systems by defining more modular architectures and pushing functionality towards the block-level. A popular approach is based on the concept of a *shared virtual disk* [19, 11, 13], which handles most of the critical concerns in distributed storage systems, including fault-tolerance, dynamic addition or removal of physical resources, and sometimes (consistent) caching. In this way, the design of the file-system can be significantly simplified. However, most of the complexity is pushed to the level of the virtual disk (assisted sometimes by a set of external services, e.g. for locking or consensus), whose design remains monolithic.

Our work bears similarity with this approach, and in particular with Petal-Frangipani [11, 20] in that all file-system communication happens through a distributed volume layer, simplifying file-system design and implementation. However, contrary to Frangipani which uses an out-of-band lock server and allocates blocks through the FS, Orchestra performs locking as well as block allocation through the block layer. We believe that this in-band management of all the core functions is an important feature, which leaves more freedom to system designers regarding the hardware/software boundary of a storage system. Thus, different trade-offs can be explored more quickly because all the functionalities exported by the virtual volume are built from a stack of modules. Moreover, Orchestra allows storage systems to provide varying functionality through virtual hierarchies and also increases flexibility and control in distributing many storage layers to a number of (possibly cascaded) storage nodes.

Support for cluster-based storage: A number of research projects have explored the potential of decentralized storage systems made of a large number of nodes with increased processing capabilities. One of the pioneering efforts in this regard is based on Object-based Storage Devices (OSD). The OSD approach defines an object structure

that is understood by the storage devices, and which may be implemented at various systems components, e.g. the storage controller or the disk itself. Our approach does not specify fixed groupings of blocks, i.e. objects. Instead, it allows virtual modules to use metadata and define block groupings dynamically, based on the module semantics. These associations may occur at any layer in a virtual storage hierarchy. For instance, a versioning virtual device [3] may be inserted either at the application server or storage node side and specifies through its metadata which blocks form each version of a specific device.

Ursa Minor [1], a system for object-based storage bricks coupled with a central manager, provides flexibility with respect to the data layout and the fault-model (both for client and storage nodes). These parameters can be adjusted dynamically on a per data item basis, according to the needs of a given environment. Such fine grain customization yields noticeable performance improvements. However, the system imposes a fixed architecture based on a central metadata server that could limit scalability and robustness, and is not extensible, i.e. the set of available functions and reconfigurations is pre-determined.

Finally, previous work has investigated a number of issues raised by the lack of a central controller and the distributed nature of cluster-based storage systems, e.g. consistency for erasure-coded redundancy schemes [2] and efficient request scheduling [12]. We consider these concerns to be orthogonal to our work but we note that the existing solutions in these domains could be implemented as modules within our framework.

7 Conclusions

In this paper we present Orchestra, a low-level software framework aimed to take advantage of brick-based architectures. Orchestra facilitates the development, deployment, and adaptation of low-level storage services for various environments and application needs, by providing distributed virtual hierarchies that can be distributed almost arbitrarily over storage clusters built with commodity components. Also, Orchestra provides support for storage sharing at the block level through locking and block allocation modules.

To provide access to Orchestra volumes through a file-system API we also design 0FS, a *stateless, pass-through* file system relying on the block-level sharing facilities. Results on a cluster with 16 nodes show that Orchestra scales well both at the block as well as the file-system level and that its modularity introduces little overhead.

Our approach of providing extensible, distributed, virtual, block-level hierarchies for clustered storage systems takes advantage of current technology trends and enables balancing storage functionality among various system components, based on performance and semantic trade-offs.

8 Acknowledgments

We thankfully acknowledge the support of the European Commission under the 6^{th} Framework Program through the the Marie Curie Excellent Teams Award UNISIX (Contract number MEXT-CT-2003-509595). We also thank Dimitrios Xinidis for his work on the first prototype of 0FS.

References

[1] M. Abd-El-Malek et al. Ursa Minor: Versatile Cluster-Based Storage. In *Proc.of the 4th USENIX FAST Conference*, 2005.

[2] K. A. Amiri et al. Highly Concurent Shared Storage. In *Proceedings of ICDCS '00 Conference*, 2000.

[3] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proc. of the 2004 Conf. on Mass Storage Systems and Technologies (MSST2004)*.

[4] M. D. Flouris and A. Bilas. Violin: A Framework for Extensible Block-level Storage. In *Proc. of 2005 Conf. on Mass Storage Systems and Technologies (MSST2005)*.

[5] M. D. Flouris et al. Using Lightweight Transactions and Snapshots for Fault-Tolerant Services Based on Shared Storage Bricks. In *Proc. of the HiperIO '06 Workshop*, 2006.

[6] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th ASPLOS Conf.*, 1998.

[7] J. Gray. Storage Bricks Have Arrived. Invited Talk at the 1st USENIX Conf. on File And Storage Tech. (FAST), 2002.

[8] I/O Performance Inc. XDD v.6.3. www.ioperformance.com.

[9] J. Katcher. PostMark: A New File System Benchmark. http://www.netapp.com/tech_library/ 3022.html.

[10] N. P. Kronenberg et al. VAXcluster: a closely-coupled distributed system. *ACM Trans. on Comp.Sys*, 4(2), 1986.

[11] E. K. Lee et al. Petal: Distributed virtual disks. In *Proceedings of the 7th ACM ASPLOS Conference*, 1996.

[12] C. R. Lumb et al. D-SPTF: Decentralized Request Distribution in Brick-Based Storage Systems. In *Proceedings of the 11th ACM ASPLOS Conference*, 2004.

[13] J. MacCormick et al. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Impl.(OSDI)*, 2004.

[14] W. D. Norcott and D. Capps. IOzone Filesystem Benchmark. http://www.iozone.org.

[15] B. Phillips. Industry Trends: Have Storage Area Networks Come of Age? *Computer*, 31(7):10–12, July 1998.

[16] K. W. Preslan et al. A 64-bit, shared disk file system for Linux. In *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST)*, 1999.

[17] Y. Saito et al. FAB: Building Distributed Enterprise Arrays from Commodity Components. In *Proc. of the 11th ACM ASPLOS Conference*, 2004.

[18] F. Schmuck et al. GPFS: A Shared-disk File System for Large Computing Centers. In *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[19] R. A. Shillner et al. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University, 1996.

[20] C. A. Thekkath et al. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th Symposium on Operating Systems Principles (SOSP)*, 1997.