Shared & Flexible Block I/O for Cluster-Based Storage

Michail D. Flouris[†], Renaud Lachaize, and Angelos Bilas[‡]
Institute of Computer Science,
Foundation for Research and Technology,
P.O. Box 1385, Heraklion, GR-71110, Greece
Email: {flouris, rlachaiz, bilas}@ics.forth.gr

Abstract

High-performance storage systems are evolving from centralized architectures and specialized hardware to "storage bricks", i.e. a large set of decentralized commodity components with more processing power and network throughput. These emerging systems offer increased flexibility for tailoring storage to application needs.

In this paper, we present Locus, an extensible framework for cluster storage virtualization and sharing at the block-level. Locus allows to build customized storage systems by composing hierarchies of virtual storage devices on top of distributed physical disks. Locus hierarchies may be distributed almost arbitrarily over storage nodes and application servers, introducing significant freedom in mapping functions to available resources. Furthermore, Locus allows sharing of storage volumes at the block-level by providing block-locking and block-allocation services as modules that may be inserted in virtual hierarchies. To demonstrate the benefits of such an enhanced block interface, we show that it can substantially simplify the design of higher-level storage services, such as distributed (cluster) file systems.

We implement the Locus framework and Locus-fs (a stateless, pass-trough file-system) under Linux and evaluate them over various setups using both single and multiple client and storage nodes. We find that the flexibility offered by Locus introduces little overhead beyond mandatory communication and disk access costs. Furthermore, experiments with a cluster of 16 nodes show that Locus scales well both at the block and file-system level.

1 Introduction

Storage systems have always been one of the key components of computing infrastructures. As the volume of processed data keeps increasing, it becomes more and more important to improve their overall quality.

Over the past decade, several trends have significantly shaped the design of storage systems. First, storage area networks (SANs) [24] that allow many servers to share a set of disk arrays, have emerged as a popular solution to improve efficiency and manageability, yet with limited extensibility and increased costs. Second, much research work has investigated the benefits of moving file-system, database, or even application-level processing closer to the data store, e.g. offloading some tasks within the storage devices themselves. Although not yet widely available, new interfaces such as object-based storage devices (OSD) [21] and active disks [1] are currently being refined and standardized. Third, 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 storage architectures (often referred to as "storage bricks") made of many decentralized commodity components with increased processing and communication capabilities [11].

This new architecture offers potential for flexible configuration of storage systems to better match application needs and thus, reduce system cost and improve efficiency. This is an important concern because distinct application domains have very diverse storage requirements; Systems designed for the needs of scientific computations, data mining, e-mail serving, e-commerce, search engines, operating system (OS) image serving or data archival impose different tradeoffs in terms of dimensions such as speed, reliability, capacity, high-availability, security, data sharing and consistency. Yet,

[†]Also, with the Dept. of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada.
[‡]Also, with the Dept. of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

current systems can not be easily tailored to the needs of a particular application. Furthermore, combining many independent, heterogeneous layers in order to obtain the desired set of features is often not an acceptable solution because it incurs increased burden for administrators, performance penalties, e.g. due to poorer potential for optimizations and extra boundaries to cross, and difficulty to supporting facilities such as global, dynamic reconfiguration.

In this paper, we present Locus, a low-level software framework aimed to take advantage of brick-based architectures. Our main goal is to ease the development, deployment, and adaptation of low-level storage services for various environments and application needs. Locus treats data and control requests in a uniform manner, propagating all I/O requests through the same I/O path from the application to the physical disks. Thus, it essentially eliminates out-of-band services, such as lock servers and block allocation that complicate the design of distributed (clustered) storage systems. We show that providing flexibility does not introduce significant overheads and does not limit scalability. Moreover, our propositions can adapt to high requirements in terms of important concerns such as reliability (fault tolerance, data integrity) and manageability (dynamic reconfiguration, automatic tuning) but these dimensions are outside of the scope of the present paper.

The main contributions of our work can be summarized as follows:

- We provide a novel approach for building cluster storage systems which consolidates system complexity in a single point, the Locus hierarchies. We examine how extensible, block-level I/O paths can be supported over distributed storage systems. Our proposed infrastructure deals with metadata persistence and allows for hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources.
- We examine how sharing can be provided at the block level. We design block-level locking and allocation facilities that can be inserted in distributed I/O hierarchies where required. A distinguishing feature of our approach is that allocation and locking are both provided as in-band mechanisms, i.e. they are part of the I/O stack and are not provided as external services. We believe that our approach simplifies the overall design of a storage system and leaves more flexibility for determining the most adequate hardware/software boundary. Moreover, using in-band control operations takes advantage of available support for scaling regular data requests. Our current prototype provides simple but scalable policies for locking and allocation. More involved policies can be implemented through additional modules.
- We examine how higher system layers can benefit from an enhanced block interface. Locus's support for block locking, allocation, and metadata persistence can significantly simplify file-system design. We design and implement a *stateless, pass-through* file-system (Locus-fs), that takes advantage of Locus's advanced features and distributed virtualization mechanisms. In particular, this file-system provides basic file and directory semantics and I/O operations, while it does not maintain any internal distributed state and does not require explicit communication among its instances.

To demonstrate our approach, we implement Locus as a block device driver under Linux and Locus-fs as a user-level library. We perform experiments with a cluster of 16 nodes (8 storage and 8 application nodes) interconnected with Gigabit Ethernet. We evaluate the effectiveness of our approach by examining the overheads introduced by the block level extensions for locking and block allocation, the overheads associating with providing a stateless file system, and the scalability of the system at both the block and file-system level.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 presents the design of Locus, emphasizing its main contributions, whereas Section 4 discusses our prototype implementation. Section 5 presents our results. Section 6 discusses limitations and future work and Section 7 draws our conclusions.

2 Related Work

In this section we comment on previous and related work on building storage systems for high performance clusters of servers. Next, we discuss the limitations of current, conventional solutions and we review prototypes that share similar goals with our work, and we contrast our approach to other contributions.

2.1 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 they are not able to support transparent sharing without application support.

Efforts in this direction include distributed *cluster file systems* often based on VAXclusters [17] 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 the Global File System (GFS) [31] and the General Parallel File System (GPFS) [28] 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. In our work we examine how such issues can be addressed in future storage systems.

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. The two most advanced open-source volume managers currently are EVMS and GEOM. EVMS [6], is a user-level distributed volume manager for Linux. It uses the MD [4] and device-mapper kernel modules to support user-level plug-ins. Recent versions offer persistent metadata and block re-mapping primitives to these plug-ins. However, EVMS does not support generic extensions (such as versioning or advanced reliability features), unlike Locus. GEOM [10] is a single-node stackable block I/O subsystem under development for FreeBSD. The basic I/O stack concepts of GEOM are similar to Locus, however, GEOM does not support distributed hierarchies, volume sharing or persistent metadata which, combined with dynamic block mapping are necessary for advanced modules such as versioning [8].

Beyond open-source software, there exist numerous commercial virtualization solutions as well, such as HP OpenView Storage Node Manager [14], EMC Enginuity [5] and Veritas Volume Manager [33]. However, in all cases, the offered virtualization functions are predefined and there is no support for extending the I/O stack with new features.

In contrast, Locus has all the configuration and flexibility features of a volume manager coupled with the ability to write extension modules with arbitrary virtualization semantics.

2.2 Flexible support for 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* [29, 18, 20], 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 [18, 32] 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, Locus 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, Locus 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.

Device-served locks have been envisioned in the past, notably in the context of GFS, which led to the specifications of the DMEP/DLOCK SCSI commands [25]. However, DMEP/DLOCK capabilities never really became a reality due to the reluctance of the storage vendors to evolve the interface of their products. Unlike these past efforts, our propositions do not require any hardware modification and enables a broader range of constructs (such as hierarchical locking) so as to allow more scalability.

Recent research has investigated more deeply the modular approach for building cluster storage systems, in particular

with the Swarm [12] and Abacus [2] projects. However, Swarm does not enable data sharing among multiple (client) nodes. Besides, due to its log-based interface to storage devices, Swarm leaves little freedom regarding the placement of most modules: much of the functionality is pushed on the client side. Abacus has mostly focused on optimal, dynamic function placement through automatic migration of components. However, to the best of our knowledge, it has not addressed issues, such as functional extensibility and simplified metadata management.

Furthermore, a number of frameworks for extensible files systems have been proposed [13, 27, 30, 35]. However, these proposals have been limited to schemes relying on a central server, which are not well suited in data-centers, where scalability is a primary concern. While we strive to present a single system image to the clients, our works target large storage systems based on a distributed (and more scalable) topology.

2.3 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 groupings and associations may occur at any layer in a virtual storage hierarchy. For instance, a versioning virtual device [8] 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. In addition, our propositions do not necessarily require to modify the current interface of storage devices.

In both approaches, allocation occurs closer to the block-level. In OSD, objects are allocated by the storage device, whereas in Locus by a virtual module that is inserted in the storage hierarchy. In terms of locking, and to the best of our knowledge, the current OSD specification [34] does not state explicitly how mutual exclusion should be provided, however, object attributes may be used to implement mutual exclusion mechanisms on top of it. In Locus, similarly to allocation, locking is provided by a new virtual module that is inserted to the storage hierarchy on demand. OSD specifies that object devices perform protection checking, whereas in Locus we envision that, beyond traditional permissions in the file-system, protection will also be provided by custom virtual modules at fine granularity.

Overall, our work shares many goals with OSD in enriching the semantics of the block-level. However, our approach allows for more flexibility on how storage devices are "composed" from distributed physical disks and, as we explore in this work, facilitates simpler cluster file-system design.

Ursa Minor [7], 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.

The Federated Array of Bricks (FAB) [26] discusses how storage systems may be built out of commodity storage nodes and interconnects and yet compete (in terms of reliability and performance) with custom, high-end solutions for enterprise environments. Overall, we share similar objectives. However, FAB has mostly investigated optimized protocols for consistent updates to replicas and quick background recovery using a non-extensible, out-of-band approach, while our main focus is to examine how future block-level systems can be tailored to support changing application needs without compromising transparency and scale.

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 [3] and efficient request scheduling [19]. 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.

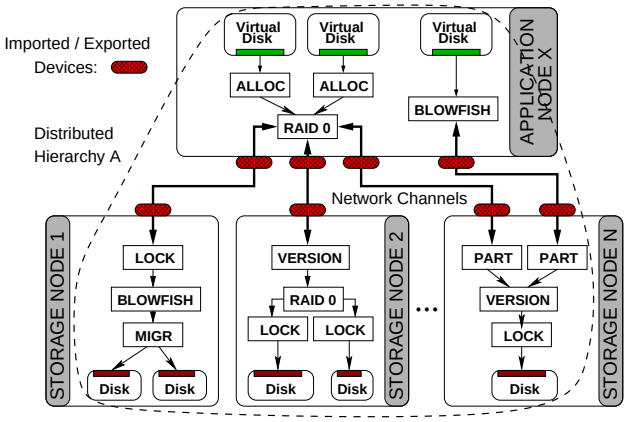


Figure 1: A distributed Locus hierarchy.

2.4 Summary

Overall, current approaches have important limitations. First, block-level virtualization systems tend to provide simple semantics, and rely on higher-level layers to deal with consistent and structured data sharing. Moreover, cluster file systems are complex and monolithic, which makes it very challenging to tune them for specific application needs. Recent work on cluster-based storage has mostly investigated issues related to reliability, security, and performance optimizations. However, to the best of our knowledge, techniques for building highly configurable storage systems from (distributed) block devices have received far less attention.

3 System Design

Locus aims at providing the underlying infrastructure for building scalable and extensible storage systems to support application needs in a cost-effective manner. Locus is designed around three goals:

- **Distributed virtual hierarchies:** Locus uses the concept of *distributed* virtual hierarchies to allow storage systems to extend the functions they support and to provide different views and semantics to applications without compromising scalability. New virtual modules may be used to provide novel storage functions while maintaining the illusion of a single data store to the upper layers (file-system and applications). This requires "splitting" hierarchies over the network in a transparent manner, and in particular, transferring *control requests* among nodes.
- **Distributed block-level sharing:** Locus allows applications to share its virtual devices. To facilitate sharing Locus incorporates block-level locking and allocation mechanisms that are implemented as virtual modules. Essentially, this makes locking and allocation in-band operations, eliminating out-of-band services that are usually used in cluster storage systems (e.g. Petal, FAB). The locking mechanism is integrated in the virtual hierarchies as an optional virtual module and may be used by Locus devices to lock shared metadata or by applications that share data at the block-level. The block allocator performs distributed free-block management and is also built as an optional virtual device that may be inserted at appropriate places in a virtual hierarchy.
- **Distributed file-level sharing:** To support seamless, coherent sharing for distributed, file-based applications, Locus provides a simple user-level library, Locus-fs that implements a conventional file-system API. Since Locus already includes advanced functions, Locus-fs is essentially a *stateless, pass-through* file-system that translates application calls to the underlying block-level operations.

Locus allows users to create distributed hierarchies that span application and storage nodes, based on application and system requirements. A sample distributed Locus hierarchy is shown in Figure 1. Next, we discuss each of the above issues in more detail.

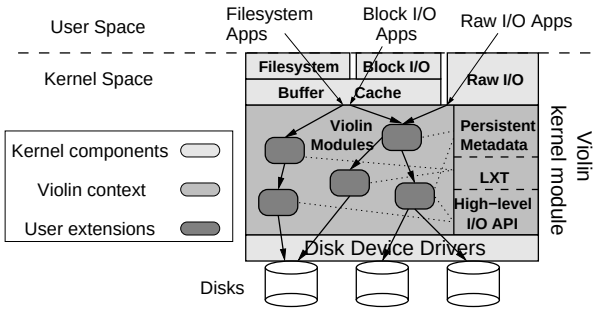


Figure 2: Violin in the OS.

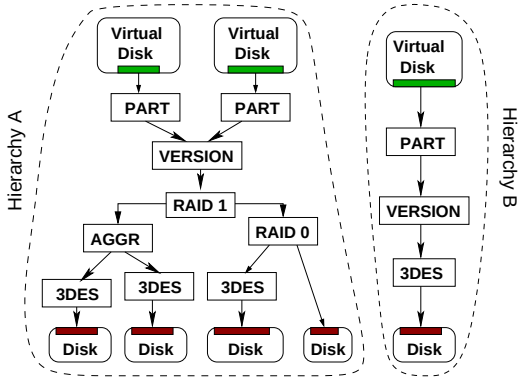


Figure 3: The virtual device graph in Violin.

3.1 Distributed Virtual Hierarchies

Locus builds on Violin [9], a framework developed for single node storage virtualization, and provides all the mechanisms required to support distributed and shared storage systems. For completeness, we first summarize next how Violin works (in 3.1.1) and then discuss the extensions developed in the context of Locus.

3.1.1 Violin

Violin, is a kernel-level framework for (i) building and (ii) combining block-level virtualization functions in a single system. Violin is a virtual I/O framework for commodity storage nodes that replaces the current block-level I/O stack with an extensible I/O hierarchy. Thus, Violin allows for easy extension of the virtual storage hierarchy with new mechanisms and flexible combination of these mechanisms to create modular hierarchies with rich semantics. Figure 2 illustrates a high-level view of Violin in the operating system context.

Violin allows storage administrators to create arbitrary, acyclic graphs of virtual devices, each adding to the functionality of the successor devices in the graph. Figure 3 shows such a device graph, where mappings between higher and lower devices are represented by arrows. 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. In each hierarchy, blocks of each virtual device 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 implementation of both synchronous and asynchronous I/O. Additionally, Violin deals with metadata persistence of the full storage hierarchy, offloading the related complexity from individual virtual devices. It supports persistent objects for storing virtual device metadata. Violin automatically synchronizes persistent objects to stable storage, resulting in much less effort for the module developer.

Overall, Violin facilitates the development of simple virtual modules, which, due to its in-band nature, can be layered to hierarchies with rich, high-level semantics that are difficult to achieve otherwise, especially in the OS kernel. Writing a new module is, in many cases, a matter of recompiling existing user-level library code, adjusted to the Violin APIs. For instance, we are able to quickly prototype modules for RAID levels, versioning, partitioning, aggregation, MD5 hashing, migration and encryption. Further examples of useful functionality that can be implemented as modules include compression, dynamic load balancing over heterogeneous devices, virus scanning and transparent disk layout algorithms (e.g. log-structured allocation or cylinder group placement).

3.1.2 In-band control and data requests

As mentioned earlier, Locus supports control and data I/O requests in the same manner, by using the same in-band mechanism for data and control propagation in virtual hierarchies. An issue with both data and control requests that traverse virtual hierarchies is related to how we treat block address arguments. Each layer is able to "see" and thus,

reference only the block addresses of its underlying layers. 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 Locus by allowing layers to provide address-mapped control requests (commands) in addition to existing regular control requests. All control requests may be issued by any virtual layer and traverse the virtual hierarchy top to bottom. If a layer does not understand a specific control request it forwards it to its lower layers. Eventually, a control command reaches a layer that handles it, possibly generating or responding to other control and data requests. Control requests provided by individual layers in Locus hierarchies are automatically inherited by *all* higher layers in the hierarchy. When a control request is issued, it is either handled by the current layer in the hierarchy or automatically forwarded to the next layer(s).

In addition, address-mapped commands are subject to translation of arguments that represent byte-ranges. This is achieved by augmenting each layer, i.e., extending the layer API, with a address-mapping API call, `address_map()`. This call is written by module developers for every module that is loaded in an Locus 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, Locus needs to transfer I/O requests between virtual modules that execute on different nodes. For this purpose, we currently use a simple network protocol over TCP/IP that implements a network block device that is able to handle both data read-write requests as well as control requests between system nodes.

3.2 Distributed Block-level Sharing

Sharing virtual volumes requires coordinating (i) accesses to data and, as mentioned above, metadata via mutual exclusion and (ii) allocation and deallocation of storage space. In Locus, concurrent accesses to the data are coordinated by means of a locking virtual module, whereas space allocation is managed by a distributed block allocation facility.

3.2.1 Byte-range locking

As mentioned, Locus 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 allows us to distribute the locking layers to any desirable serialization point in a distributed hierarchy. Such points include for example 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 Locus communication mechanism and the associated addresses are mapped according to the distributed hierarchy mappings.

Note that the metadata of a locking layer does not need to 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.

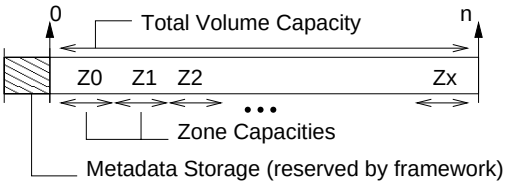


Figure 4: View of the allocator layer's capacity.

3.2.2 Block allocation

The role of the Locus's block allocator is to handle distributed block management in a consistent manner for clients sharing the same block volume. The allocator distributes free blocks to the clients and maintains a consistent view of used 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 a 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 for managing free blocks consist of free-lists and bitmaps to handle blocks of various sizes. The allocator uses a free-list for keeping track of large blocks and bitmaps for small blocks. Currently, the allocator is configured to use two block sizes, a large block size (4 KBytes) for data and a small block size (256 Bytes) for file-system i-nodes. However, we envision that future versions of Locus may rely on a single block size using the notion of "stuffed i-nodes" [22].

The allocator metadata need to be maintained consistent across allocator instances running in different nodes. This is achieved by using the persistent metadata locking primitives 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 as shown in Figure 4. Each zone is independent and has its own metadata, which can be locked and cached in the node using this particular zone.

The locking algorithm works as follows. When the first allocation request for a free block arrives, the allocator identifies an unused (and unlocked) zone in the shared device and locks the zone. When a module successfully locks a zone, it loads 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 released from the cache.

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.

3.3 Distributed File-level Sharing

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

One approach to achieve this is to use an existing distributed file system. Depending on the requirements of the distributed file-system, Locus can be used to either provide a number of virtual volumes each residing in a single storage node [31], or a single distributed virtual volume built out of multiple storage nodes [32]. However, existing file-systems in either case would not take advantage of distributed block allocation and locking primitives provided by Locus at the block-level.

For these reasons, we provide our own file-system on top of Locus, using a user-level library that provides the basic file-system functions, such as file and directory naming, as well as standard file I/O operations. Locus-fs is a *stateless*, *pass-through* file system that translates file calls to the underlying Locus 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 Locus-fs 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. Locus-fs uses the corresponding block-level mechanisms provided by Locus volumes for this purpose:

- **Mutual exclusion:** Locus-fs uses the multiple-reader, single-writer locking mechanism provided by Locus to achieve mutual exclusion between multiple applications accessing a single file-system through a single or multiple application nodes. Currently, Locus-fs 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 nonetheless 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 Locus-fs are i-nodes and the directory structure. To avoid maintaining internal consistent state in memory, Locus-fs does not perform any caching of metadata or data but uses the underlying Locus-fs block device. Thus, accesses to files or directories in Locus-fs 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 5 shows how Locus-fs is combined with Locus to provide file-level sharing over distributed Locus volumes.

3.4 Differences of Locus vs. Violin

Violin [9] and Locus presented in this paper are both virtualization frameworks and thus share basic virtualization principles, such as easily extensible hierarchies made of virtual devices. However, there are significant differences between the two systems, the most important being that Violin offers single-node virtualization, while Locus is a distributed virtualization framework for a cluster.

In terms of design/implementation, the major differences of Locus and Violin are:

- The networking layer allowing hierarchy distribution in the cluster.
- Extensions to command API for propagating commands across nodes and manipulating control and configuration information.
- Support of block-level locking and allocation in distributed configurations, which is essential for sharing volumes between many clients.
- Design, implementation, and evaluation of a stateless filesystem (Locus-fs) built on top of distributed Locus hierarchies using block-level locking and allocation facilities. To our knowledge, Locus is the only existing system that decouples

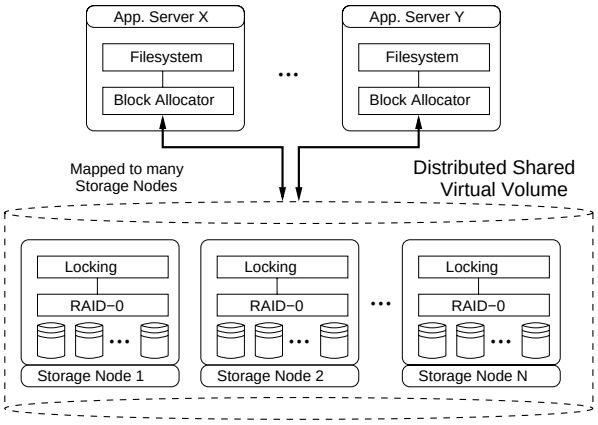


Figure 5: The allocator and locking layers allow many FS clients to share a distributed volume mapped on many nodes.

Component	# code lines
Locus Core Framework	19558
Various Modules (RAID,etc.)	6114
Locking	1137
Block Allocator	2479
Locus-fs	7600
Locus Total (approx.)	36900

Figure 6: Locus modules in kernel code lines.

free-block allocation and locking from the distributed filesystem and integrates them as independent components in the block-level I/O stack for improving scalability.

Finally, there are important differences between the content and focus of the two papers (this paper and [9]) in terms of evaluation and results.

4 System Implementation

Locus is implemented as a block device driver module in the Linux 2.6 kernel and is loadable at runtime. User-level tools can be used online, during system operation to create new or modify existing I/O hierarchies. However, the user is responsible for maintaining data consistency while modifying the structure of virtual hierarchies. The user-level tools communicate with the framework device driver through the kernel's `ioctl()` interface.

Virtual I/O layers are implemented as separate kernel modules that are loaded on demand. Upon loading, Locus layers register their presence to the Locus framework driver. These modules are not "classical" device drivers themselves, but rather use the framework's programming API.

We have implemented various virtual modules, notably RAID 0 and RAID 1 (including recovery), versioning, device partitioning, device aggregation, DES and triple-DES encryption, and data migration between devices. This set of I/O layers has been used to demonstrate the flexibility of the distributed hierarchies we propose.

The networking channels for exporting Locus devices to other storage nodes or clients are currently implemented over TCP/IP in the core Locus framework. The networking component implements the client and server kernel threads and the request queues needed for the networking channels, as well as the Locus commands to connect and disconnect remote devices from the local Locus stack.

When a virtual I/O hierarchy is configured in the framework and linked with a `/dev` device, it can be manipulated as a normal block device with existing tools. For instance, the user can build a file system on top of it with `mkfs` and then `mount` it as a regular single-node file-system or share it through Locus-fs.

Locus-fs is currently implemented as a user-level library that may be linked with any application during execution. We choose to implement Locus-fs at user-level to reduce the development effort as well as to allow for easy customization. Locus-fs currently supports most of the conventional file-system operations, such as open, close, remove, creat, read, write, stat, rename, mkdir, rmdir, opendir, readdir, chdir, getcwd. We were able to compile and run standard FS benchmarks, such as IOzone and Postmark without any code modifications.

Table 6 shows the size of the code for Locus. The current version, as used in our experiments, is about 36800 lines of code.

Finally, the current Locus implementation uses a number of kernel threads mainly for network channels between de-

vices: (a) One thread that listens, authenticates and accepts new connections from remote Locus hierarchies. This thread also creates the threads needed for a new remote device connection. (b) A client request sender thread that removes I/O requests from a queue and sends them to the server. (c) A server request receiver (per exported device) that receives requests from the socket. (d) A server response sender thread that sends responses when the disk I/O is done. (e) A client request receiver thread (per connected device) that receives server responses and sends call-backs to the waiting application.

5 Experimental Results

In this section we present preliminary results on the overhead of basic operations in Locus and we examine the scalability of Locus and Locus-fs on a setup with multiple storage and application nodes.

Our evaluation platform is a 16-node cluster of commodity x86-based Linux systems. All the cluster nodes are equipped with dual AMD Opteron 242 CPUs and 1 GByte of RAM, while storage nodes have four 80GB Western Digital SATA Disks. All nodes are connected with a 1 Gbit/s (Broadcom Tigon3 NIC) Ethernet network through a single 48-port GigE switch (D-Link DGS-1024T). All systems run Fedora Core 3 Linux with the 2.6.12 kernel.

We use three I/O benchmarks: xdd [15], IOzone [23] and PostMark [16]. We use Postmark and IOzone to examine the basic overheads in Locus-fs, and xdd on raw block devices to examine block I/O overheads.

xdd [15] generates I/O workloads based on specified parameters, such as read/write mix, request size, access pattern, number of outstanding I/Os. We vary three parameters in our workloads: (i) *Number of outstanding I/Os*: This is equivalent to specifying the maximum queue depth in the I/O path. In our experiments, we vary the queue depth from 1 to 32 I/Os. (ii) *Read-to-write ratio*: We use 100% reads and 100% writes, and a mix of 70% reads–30% writes. (iii) *Block size*: we use block sizes ranging from 4 KBytes to 1 MByte. In all cases, we run xdd on a 4 GByte file (4 times the RAM of the nodes) and we report numbers averaged over multiple runs for each block size. All runs for a specific device are run in sequence and the total running time for all block sizes and workloads is approximately 4 hours.

IOzone is a file-system benchmark tool that generates and measures a variety of file operations. We use IOzone version 3.233 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.

PostMark [16] is a synthetic file-system benchmark that creates a pool of continually changing files on a file-system and measures the transaction rates for a workload simulating a large Internet electronic mail server. Initially, it generates a pool of random text files. Then it issues a specified number of transactions. Each transaction consists of (i) a create file or delete file operation and (ii) a read file or append file operation. Each transaction type and its affected files are chosen randomly. When all transactions are complete, the remaining files are deleted.

In our evaluation we examine basic overheads and scalability of Locus and Locus-fs. 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 (Figure 9).

5.1 Locus

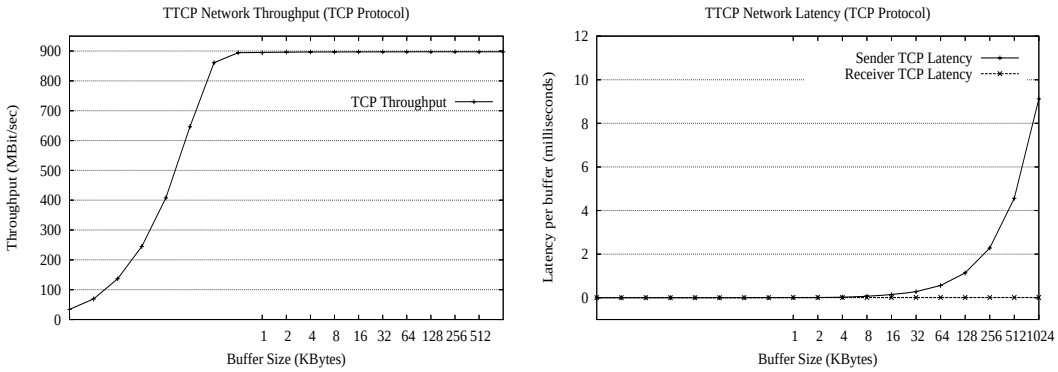
In this subsection we examine the overheads associated with Locus in single- and multi-node setups.

5.1.1 Basic costs

We measure the cost of individual operations in a local and a distributed virtual hierarchy (volume). Table 1 summarizes our results. Values are averaged over one thousand calls. We see that a local null ioctl costs about 4 μs. Allocating and freeing blocks through a local allocator virtual module costs about 12 and 13 μs respectively. This cost does not include saving the allocator metadata to disk, which is performed periodically (every 10 seconds) in the background. In the distributed volume case, there is an additional 310 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 PCI Express/AS and 10 Gigabit Ethernet. Finally, using `ttcp` in our setup results in a maximum throughput of about 112 MBytes/s (900 Mbits/s).

	NULL	Allocate	Free	Lock	Unlock
Local Ioctls	4	12	13	14	15
Remote Ioctls	295	-	-	310	317

Table 1: Measurements of individual control operations. Numbers are in μsec.



(a) Throughput (b) Latency

Figure 7: TTCP throughput and latency.

Figure 7 shows the throughput and latency between two system nodes as reported by the TTCP benchmark. We see that the maximum throughput achieved between two nodes in our system is about 900 MBits/s.

Figure 8 compares the throughput and latency of a simple Locus hierarchy of a single RAID0 virtual device on four SATA disks, with a similar hierarchy built with existing Linux drivers (MD) in a single node. We use xdd with a 4GByte workload for the measurements. The throughput obtained for all workloads (sequential and random) over Locus is similar to the throughput of MD, as shown in Figure 8(a). In terms of latency (shown in Figure 8(b)), Locus also exhibits similar performance to MD.

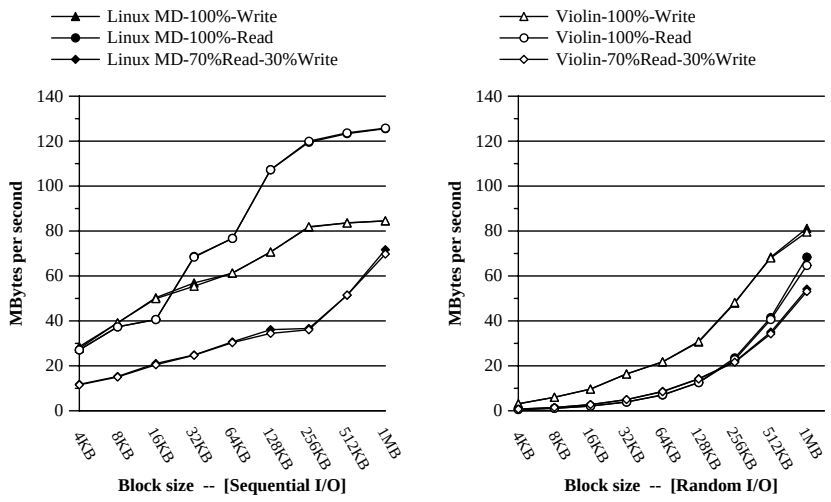
5.1.2 Scalability

Now we examine the scalability of Locus at the block-level using xdd. Figure 9 shows the setup we use for these experiments. Each xdd process runs at different block offsets and thus accesses separate block ranges than other xdd processes, thus minimizing caching effects on the storage nodes.

Figure 10(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. However, scalability is limited because with increasing number of nodes, the efficiency of individual disks drops, as they receive a smaller number of *sequential* requests and thus, incur higher seek times. For instance, in the 1x1 setup, maximum throughput is about 80 MBytes/s while it is 150 MBytes/s in the 4x4 case.

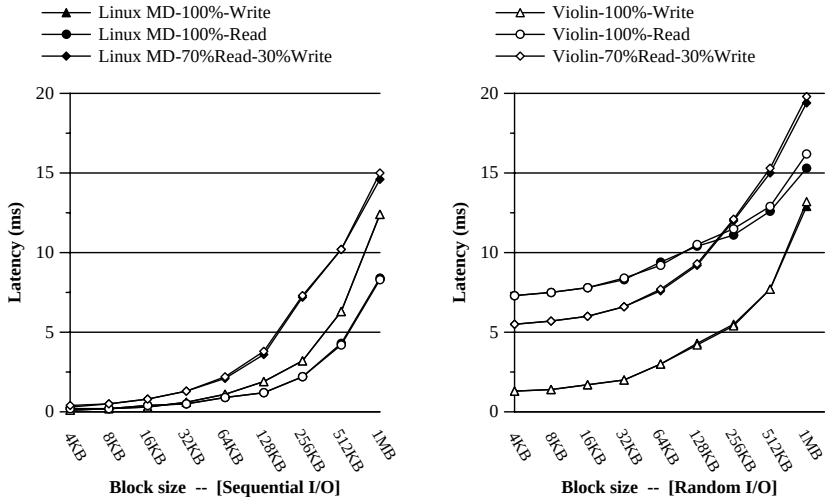
To examine scalability when individual disk utilization is not the bottleneck we increase the number of outstanding I/Os per client, when keeping the request size at 1 MByte. The results in Figure 10(c) show that increasing the effectiveness of individual disks results in improved performance. In the 1x1 setup, throughput of sequential reads reaches a maximum of about 110 MBytes/s for sufficiently large request sizes. This is close to the throughput limit of the network. We also observe similar behavior for the 100% sequential writes workload. When scaling the number of nodes in the 4x4 and 8x8 setups, read throughput scales from 110 to 250 to 450 MBytes/s and write throughput scales from 90 to 300 to 500 MBytes/s.

Finally, if we compare sequential to random I/O we note that for large block sizes, random reads and writes have higher throughput. We believe this is due to two main factors: (a) better caching behavior in the storage nodes because of repeated block accesses and (b) smaller average seek times for individual disks.



XDD Direct Attached: Violin vs. MD

(a) Throughput



XDD Direct Attached: Violin vs. MD

(b) Latency

Figure 8: Throughput and latency for Locus vs. Linux MD over direct-attached disks for sequential (left) and random (right) workloads.

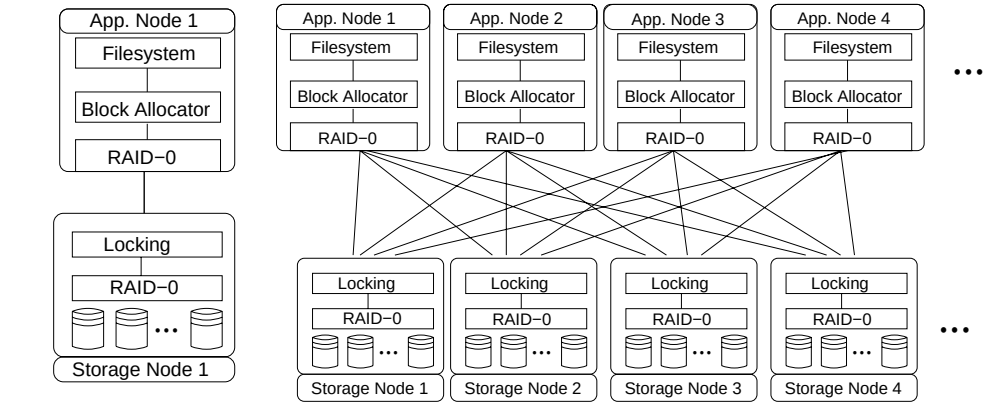
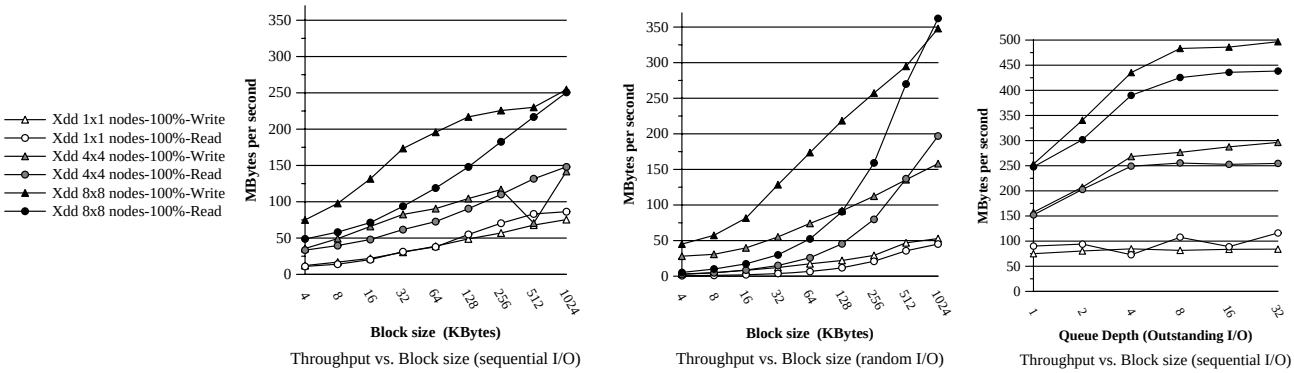
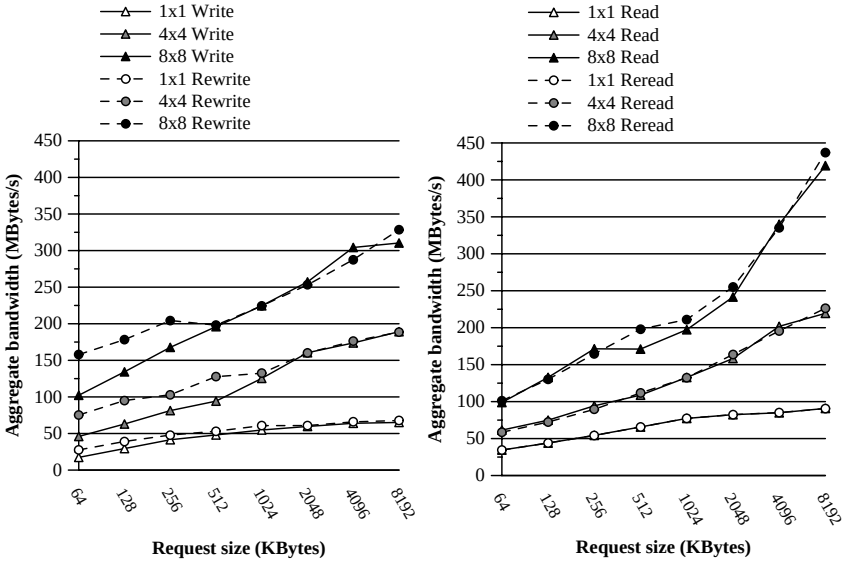


Figure 9: Configuration for multiple node experiments.



Throughput vs. Block size (sequential I/O) | Throughput vs. Block size (random I/O) | Throughput vs. Block size (sequential I/O)

Figure 10: Block I/O throughput for the 1x1, 4x4, and 8x8 setups.



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

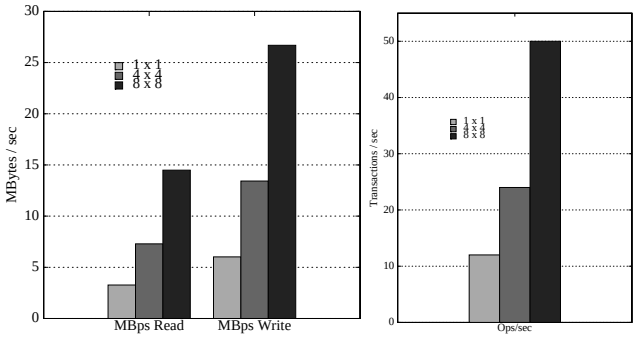
Figure 11: Aggregate throughput of Locus-fs with IOzone.

5.2 Locus-fs

In this subsection we examine the overheads associated with Locus-fs using IOzone and Postmark.

First, we look into the base performance of Locus-fs on a single node. We contrast its behavior to the standard Linux Ext2FS using both IOzone and Postmark. IOzone results, show similar performance over reads, while for writes, ext2FS is about 20% faster. In the Postmark experiments and for small workloads that fit in the buffer cache, Locus-fs is about 25% slower. We attribute this performance difference to the increased number of system calls of the user-level Locus-fs compared to the in-kernel Ext2FS and the metadata caching that Ext2FS uses. Thus, we see that Locus-fs performs, in the local case, close to current file-systems.

Next, we examine the scalability of Locus-fs in distributed setups. In the base distributed configuration (1x1) we create a single volume over all available system storage and then create different directories for every application node that will access this volume. The separate instances of Locus-fs use the locking, allocation modules, and RAID0 modules, as shown in Figure 9. Each application node uses a separate directory on the distributed Locus-fs to perform its file I/O workload either with IOzone or Postmark. In our opinion 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 between clients. Finally, note that since Locus-fs is stateless, it does not perform any client-side caching, and thus all I/O requests generate network traffic. Figures 11 and 12 show our multi-node results with IOzone and Postmark, respectively.



(a) Throughput (MBytes/sec) (b) Transactions / sec

Figure 12: Postmark results over Locus-fs in multiple node setups (1x1, 4x4, 8x8). The workload consists of: (i) files ranging from 128 KB to 1 MByte in size, (ii) an initial pool of 2000 files, and (iii) 5000 file transactions with equal biases for read/append and create/delete. During each run there are over 4500 files created, about 1.5 GBytes of data read and more than 3 GBytes of data written by each client.

Figure 11 shows that IOzone (over Locus-fs) scales as the number of nodes increases for both read and write, reaching a maximum throughput of about 330 MBytes/s in the 8x8 configuration. We also see that IOzone performance and scaling follows the behavior of block-level I/O (xdd) in Figures 10. 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, Locus-fs is able to scale well in cases where these is limited contention. Similarly, our Postmark results (Figure 12) show scaling behavior similar to IOzone. We see that quadrupling the number of storage nodes from 1x1 to 4x4 results in about doubling all metrics, whereas further increasing the number of storage nodes from 4 to 8 results in doubling performance.

5.2.1 Comparison to other distributed filesystems.

Considering how different our approach is from current distributed storage systems, we believe it is difficult to make an apple-to-apple comparison with existing, freely-available cluster filesystems such as GFS (note that GPFS is a commercial closed-source system), since: (i) Cluster filesystems usually rely on hard-wired reliability mechanisms (e.g. journaling) which incur a significant performance penalty. Since currently Locus-fs provides only weak reliability guarantees, a comparison with a file-system like GFS would be neither meaningful nor very fair. Note also that it's very hard to extend or remove features from GFS because of its monolithic structure. We are currently working on providing strong reliability guarantees in our prototype, and although related, we consider them out of the scope of this paper. (ii) Our FS prototype is currently implemented at the user level and thus has more overheads than a filesystem implemented at the kernel level.

Our main objective in this paper is to show that the proposed architecture does not introduce major overheads or scalability bottlenecks and that our novel approach (i.e. decoupling block allocation and locking from the filesystem) simplifies the distributed filesystem.

5.3 Summary

Overall, Locus is able to scale well when throughput is not limited by increased seek overheads in physical disks. Moreover, Locus-fs follows the same scaling pattern as Locus 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 Limitations and Future work

Although Locus provides support for both easily extending storage systems as well as pushing new functions closer to the storage devices, it is currently limited in three ways: (i) continuous operation in the presence of faults, (ii) data protection, and (iii) client-side consistent caching.

We have designed and are currently implementing the mechanisms required to enforce consistency of distributed metadata (data) in the presence of failures. Our approach relies on using lightweight transactions for providing atomicity with respect to failures and locks for serializability. Our approach is lightweight in the sense that application and storage nodes operate independently in the common case, and only infrequently synchronize at consistency points. Moreover, in our approach recovery is straight-forward and does not require extensive processing. A more detailed description of our approach on achieving fault tolerance is beyond the scope of this work.

Furthermore, Locus currently relies on traditional protection mechanisms, i.e. checking for file attributes in kernel-level file systems. However, the ability to associate blocks with metadata dynamically and on-demand provides the potential for (i) performing finer-grain protection checking, (ii) dynamically adjusting protection levels for blocks based on high-level features, e.g. time-based protection mechanisms.

Finally, Locus may use the OS buffer cache in storage nodes for caching purposes, which results in at least one network round-trip time for each I/O operation. In future clustered storage systems, it may be important to support client-side caching. However, this requires support for consistency since metadata for a single physical block may be cached in multiple application nodes. Such mechanisms can be implemented in Locus as virtual modules and are also an important topic for future work.

7 Conclusions

In this paper we present Locus, a low-level software framework aimed to take advantage of brick-based architectures. Locus facilitates the development, deployment, and adaptation of low-level storage services for various environments and application needs, by providing distributed virtual hierarchies over storage systems built out of commodity compute nodes, storage devices, and interconnects. Locus virtual hierarchies may be distributed almost arbitrarily over application and storage nodes. Finally, Locus provides support for storage sharing at the block level through locking and block allocation. These functions are provided as modules that may be inserted in Locus's virtual hierarchies.

To provide access to Locus volumes through a file-system API we also design and build Locus-fs a *stateless, pass-through* file system that is currently implemented as a dynamically linked user-level library. Locus-fs does not maintain internal state, uses Locus facilities for locking and block-allocation, and does not require explicit communication among its instances.

We implement Locus and Locus-fs under Linux and evaluate them using various setups with single and multiple storage and application nodes. We find that the modularity of Locus introduces little overhead beyond TCP/IP communication and existing kernel overheads in the I/O protocol stack. Results on a cluster with 16 nodes show that Locus scales well both at the block as well as the file-system level.

Overall, we show that providing flexibility and extensibility in a distributed storage stack does not introduce significant overheads and does not limit scalability. More importantly, our approach of providing extensible, virtual, block-level hierarchies over clustered storage systems takes advantage of current technology trends and provides the ability to repartition storage functionality among various system components, based on performance and semantic tradeoffs.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, San Jose, California, Oct. 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000. USENIX Association.

[3] K. A. Amiri, G. A. Gibson, and R. Golding. Highly Concurent Shared Storage. In IEEE, editor, *Proceedings of* 20^{th} *International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 298–307, Taipe, Taiwan, R.O.C, Apr. 2000. IEEE Computer.

[4] M. de Icaza, I. Molnar, and G. Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, Apr. 1997.

[5] EMC. Enginuity(TM): The Storage Platform Operating Environment (White Paper). http://www.emc.com/pdf/techlib/c1033.pdf.

[6] Enterprise Volume Management System. evms.sourceforge.net.

[7] M. A.-E.-M. et al. Ursa Minor: Versatile Cluster-Based Storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technology*, San Francisco, CA, USA, December 2005.

[8] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, Apr. 2004.

[9] M. D. Flouris and A. Bilas. Violin: A Framework for Extensible Block-level Storage. In *Proceedings of 13th IEEE/NASA Goddard (MSST2005) Conference on Mass Storage Systems and Technologies*, Monterey, CA, Apr. 11–14 2005.

[10] FreeBSD: GEOM Modular Disk I/O Request Transformation Framework. http://kerneltrap.org/node/view/454.

[11] J. Gray. Storage Bricks Have Arrived. Invited Talk at the First USENIX Conference on File And Storage Technologies (FAST '02), 2002.

[12] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*. IEEE Computer Society, June 1999.

[13] J. Heidemann and G. Popek. File System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.

[14] HP. OpenView Storage Area Manager. http://h18006. www1.hp.com/ products/ storage/ software/ sam/ index.html.

[15] I/O Performance Inc. Xdd version 6.3. http://www.ioperformance.com.

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

[17] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxcluster: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2), 1986.

[18] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of The 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS7)*, pages 84–92, Cambridge, MA, Oct. 1996.

[19] C. R. Lumb, R. Golding, and G. R. Ganger. D-SPTF: Decentralized Request Distribution in Brick-Based Storage Systems. In *Proceedings of the 11th ACM ASPLOS Conference*, Boston, MA, USA, October 2004.

[20] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI-04)*. USENIX, Dec. 6–8 2004.

[21] M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003.

[22] S. J. Mullender and A. S. Tanenbaum. Immediate files. *Software Practice and Experience*, 14(4):365–368, 1984.

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

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

[25] K. W. Preslan, A. Barry, J. Brassow, M. Declerk, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe. Scalability and Recovery in a Linux Cluster File System. In *Proceedings of the 4th Annual Linux Showcase and Conference*, College Park, Maryland, USA, October 2000.

[26] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: Enterprise storage systems on a shoestring. In *Proc. of the ASPLOS 2004*, Oct. 2004.

[27] P. W. Schermerhorn, R. J. Minerick, P. W. Rijks, and V. W. Freeh. User-level Extensibility in the Mona File System. In *Proc. of Freenix 2001*, pages 173–184, June 2001.

[28] F. Schmuck and R. Haskin. GPFS: A Shared-disk File System for Large Computing Centers. In *USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, Jan. 2002.

[29] R. A. Shillner and E. W. Felten. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University, Princeton, NJ, USA, October 1996.

[30] G. C. Skinner and T. K. Wong. Stacking/ vnodes: A progress report. In *Proc. of the USENIX Summer 1993 Technical Conference*, pages 161–174, Berkeley, CA, USA, June 1993. USENIX Association.

[31] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The Global File System: A File System for Shared Disk Storage, Oct. 1997.

[32] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 224–237, Oct. 5–8 1997.

[33] Veritas. Volume Manager(TM). http://www.veritas.com/vmguided.

[34] R. O. Weber (Editor). Information technology – scsi object-based storage device commands (OSD), revision 10. Technical Council Proposal Document T10/1355-D, Technical Committee T10, July 2004.

[35] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 55–70, Berkeley, CA, June 18–23 2000. USENIX Association.