Violin: A Framework for Extensible Block-level Storage

Michail D. Flouris[†]

Department of Computer Science,
University of Toronto,
Toronto, Ontario M5S 3G4, Canada
flouris@cs.toronto.edu

Angelos Bilas[‡]

Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas
P.O.Box 1385, Heraklion, GR 71110, Greece
bilas@ics.forth.gr

Abstract

In this work we propose Violin, a virtualization framework that allows easy extensions of block-level storage stacks. Violin allows (i) developers to provide new virtualization functions and (ii) storage administrators to combine these functions in storage hierarchies with rich semantics. Violin makes it easy to develop such new functions by providing support for (i) hierarchy awareness and arbitrary mapping of blocks between virtual devices, (ii) explicit control over both the request and completion path of I/O requests, and (iii) persistent metadata management.

To demonstrate the effectiveness of our approach we evaluate Violin in three ways: (i) We loosely compare the complexity of providing new virtual modules in Violin with the traditional approach of writing monolithic drivers. In many cases, adding new modules is a matter of recompiling existing user-level code that provides the required functionality. (ii) We show how simple modules in Violin can be combined in more complex hierarchies. We demonstrate hierarchies with advanced virtualization semantics that are difficult to implement with monolithic drivers. (iii) We use various benchmarks to examine the overheads introduced by Violin in the common I/O path. We find that Violin modules perform within 10% of the corresponding monolithic Linux drivers.

1. Introduction

Storage is becoming an increasingly important issue as more and more data need to be stored either for archival or online processing purposes. As the amount of storage required increases, scalable storage systems provide a means of consolidating all storage in a single system and increasing storage efficiency. However, storage consolidation leads to increased requirements for "flexibility" that will be able to serve multiple applications and their diverse needs. This flexibility refers to both storage management and application access issues and is usually provided through virtualization techniques: Administrators and applications see various types of virtual volumes that are mapped to physical devices but offer higher-level semantics through virtualization mechanisms.

The quality of virtualization mechanisms provided by a storage system affects storage management complexity and storage efficiency, both of which are important problems of modern storage systems. Storage virtualization may occur either at the filesystem or at the block level. Although both approaches are currently being used, our work addresses virtualization issues at the block-level. Storage virtualization at the filesystem level has mainly appeared in the form of extensible filesystems [11, 24, 30, 32].

We argue that the importance of block-level virtualization is increasing for two reasons. First, certain virtualization functions, such as compression or encryption, may be simpler and more efficient to provide on unstructured fixed data blocks rather than variable-size files. Second, block-level storage systems are evolving from simple disks and fixed controllers to powerful storage nodes [1, 9, 10] that offer block-level storage to multiple applications over a storage area network [23, 22]. In such systems, block-level storage extensions can exploit the processing capacity of the storage nodes, where filesystems (running on the servers) cannot. For these reasons and over time, with the evolution of storage technology a number of virtualization features, e.g. volume management functions, RAID, snapshots, moved from higher system layers to the block level.

Today's block-level storage systems provide flexibility in managing and accessing storage through I/O drivers (modules) in the I/O stack or through the filesystem. For instance, Linux-based storage systems use drivers, such as MD [2] and LVM [25] to support RAID and logical volume management functions. However, this flexibility is limited by the fact that current I/O stacks require the use of monolithic I/O drivers that are both complex to develop and hard to combine. As a result, current block-level systems offer predefined virtualization semantics, such as virtual volumes mapped to an aggregation of disks or RAID levels. In this category belong both research prototypes [2, 6, 8, 18, 25] and commercial products [4, 5, 12, 27, 28]. In all these cases the storage administrator can switch on or off various features at the volume level, but cannot extend them.

In this work we address this problem by providing a kernel-level framework for (i) building and (ii) combining virtualization functions. We propose, implement, and evaluate *Violin* (Virtual I/O Layer INtegration), a virtual I/O framework for commodity storage nodes that replaces the current block-level I/O stack with an improved I/O hierarchy that allows for (i) easy extension of the

---

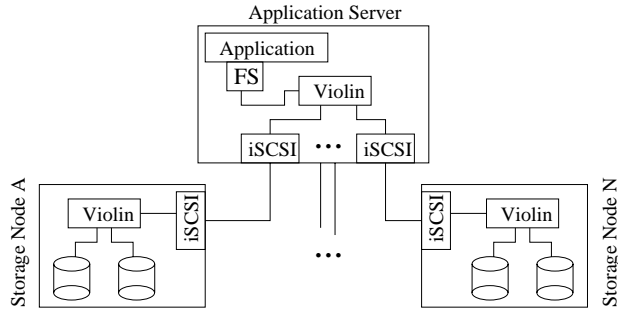**Figure 1. Violin in a distributed environment.**

Figure 2. Violin in the operating system context.

storage hierarchy with new mechanisms and (ii) flexible combining of these mechanisms to create modular hierarchies with rich semantics.

Although our approach shares similarities with work in modular and extensible filesystems [11, 24, 30, 32] and network protocol stacks [17, 19, 20, 26], existing techniques from these areas are not directly applicable to block-level storage virtualization. A fundamental difference from network stacks is that the latter are essentially stateless (except for configuration information) and packets are ephemeral, whereas storage blocks and their associated metadata need to persist. Compared to extensible filesystems, block-level storage systems operate at a different granularity, with no information about the relationships of blocks. Thus, metadata need to be maintained at the block level resulting potentially in large memory overhead. Moreover, block I/O operations cannot be associated precisely with each other, limiting possible optimizations.

The main contributions of *Violin* are: (i) it significantly reduces the effort to introduce new functionality in the block I/O stack of a commodity storage node and (ii) provides the ability to combine simple virtualization functions into hierarchies with semantics that can satisfy diverse application needs. *Violin* provides virtual devices with full access to both the request and completion paths of I/Os allowing for easy implementation of synchronous and asynchronous I/O. Supporting asynchronous I/O is important for performance reasons, but also raises significant challenges when implemented in real systems. Also, *Violin* deals with metadata persistence for the full storage hierarchy, offloading the related complexity from individual virtual devices. To achieve flexibility, *Violin* allows storage administrators to create arbitrary, acyclic graphs of virtual devices, each adding to the functionality of the successor devices in the graph. 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.

Systems such as *Violin* can be combined with standard storage access protocols, such as iSCSI, to build large-scale distributed volumes. Figure 1 shows a system with multiple storage nodes that provide a common view of the physical storage in a cluster. We believe that future, large-scale storage systems will be built in this manner to satisfy application needs at a cost-efficient manner.

We implement *Violin* as a block device driver under Linux. To demonstrate the effectiveness of our approach in extending the I/O hierarchy we i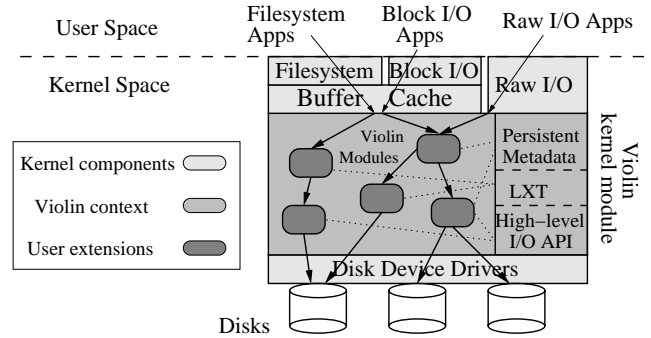mplement various virtual modules as dynamically loadable kernel devices that bind to *Violin*'s API. We also provide simple user level tools that are able to perform on-line fine-grain configuration, control, and monitoring of arbitrary hierarchies of instances of these modules.

We evaluate the effectiveness of our approach in three areas: ease of development, flexibility, and performance. In the first area we are able to quickly prototype modules for RAID levels, versioning, partitioning and aggregation, MD5 hashing, migration and encryption. In many cases, writing a new module is just a matter of recompiling existing user-level library code. Overall, using *Violin* encourages the development of simple virtual modules that can later be combined to more complex hierarchies. Regarding flexibility, we are able to easily configure I/O hierarchies that combine the functionality of multiple layers and provide complex high-level semantics that are difficult to achieve otherwise. Finally, we use Postmark and IOmeter to examine the overhead that *Violin* introduces over traditional block-level I/O hierarchies. We find that overall, internal modules perform within 10% (throughput) of their native Linux block-driver counterparts.

The rest of the paper is organized as follows. Sections 2 and 3 present the design and implementation of *Violin*. Section 4 presents our results, while Section 5 discusses related work. Finally, Section 6 discusses limitations and future work and Section 7 draws our conclusions.

2. System Architecture

Violin is a virtual I/O framework that provides (i) support for easy and incremental extensions to I/O hierarchies and (ii) a highly configurable virtualization stack that combines basic storage layers in rich virtual I/O hierarchies. *Violin*'s location in the kernel context is shown in Figure 2, illustrating the I/O path from the user applications to the disks. There are three basic components in the architecture of *Violin*:

1. High-level virtualization semantics and mappings.

2. Simple control over the I/O request path.

3. Metadata state persistence.

Next we discuss each of these components in detail.

2.1. Virtualization Semantics

A virtual storage hierarchy is generally represented by a *directed acyclic graph* (DAG). In this graph, the vertices or *nodes* repre-
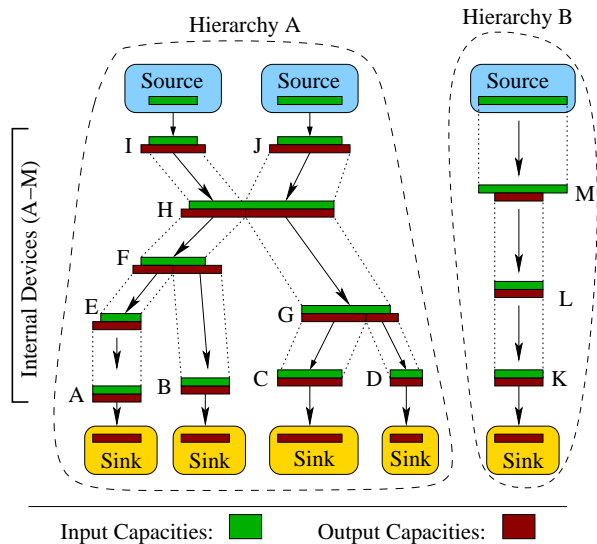
Figure 3. Violin's virtual device graph.

Figure 4. The LXT address translation table.

Violin are objects that are explicitly created before adding virtual devices (nodes).

To manage devices and hierarchies, users may specify the following operations on the device graph:

- Create a new internal, source, or sink node and link it to the appropriate nodes, depending on its type. The DAG is created in a bottom-up fashion to guarantee that an I/O sink will always exist for any I/O path.

- Delete a node from the graph. A node may be deleted only if its input devices have been deleted (top-down).

- Change an edge in the graph, i.e. remap a device.

Violin checks the integrity of a hierarchy at creation time and each time it is reconstructed. Checking for integrity includes various simple rules, such as the presence of cycles in the hierarchy graph and lack of input or output edges in internal nodes. Creating hierarchies and checking dependencies reduces the complexity of each *Violin* module.

A hierarchy in *Violin* is constructed with simple user-level tools implementing the above graph operations and linking the source and sink nodes to external OS block devices. Currently, the storage administrator has to specify the exact order in which virtual devices will be combined. The user-level tools can also be used online, during system operation to modify or extend I/O hierarchies. However, it is the administrator's responsibility to maintain data consistency while performing online structural changes to a hierarchy.

2.1.1. Dynamic Block Mapping and Allocation Nodes in a hierarchy graph do not simply show output dependencies from one device to another but rather map between block address spaces of these devices. As can be seen in Figure 3, a storage device in the system represents a specific storage capacity and a block address space, while the I/O path in the graph represents a series of translations between block address spaces. *Violin* provides transparent and persistent translation between device address spaces in virtual hierarchies.

Devices in a virtual hierarchy may have widely different requirements for mapping semantics. Some devices, such as RAID-0, use simple mapping semantics. In RAID-0 blocks are mapped statically between the input and output devices. There are, however, modules that require more complex mappings. For instance, providing snapshots at the block level, requires arbitrary translations and dynamic block remappings [7]. Similarly, volume managers [25] require arbitrary block mappings to support volume resizing and data migration between devices. Another use of arbitrary mappings is to change the device allocation policy, for instance, to a log-structured policy. The Logical Disk [3] uses dynamic block remapping for this purpose.

sent *virtual devices*. The directed edges between nodes signify *device dependencies* and *control flow* through I/O requests. Control in *Violin* flows from higher to lower layers. This arises from the traditional view of the block-level device as a dumb passive device. Each virtual device in the DAG is operated by a virtualization module that implements desired functionality. Virtual devices that provide the same functionality are handled by different instances of the same module. From now on, we will use the terms module and device interchangeably.

Figure 3 shows an example of such a device graph. Graph nodes are represented with horizontal bars illustrating the mappings of their address spaces and they are connected with directed vertices. There are three kinds of nodes and accordingly three kinds of I/O modules in the architecture:

- *Source nodes* that do not have incoming edges and are top-level devices that *initiate I/O requests* in the storage hierarchy. The requests are initiated by external kernel components such as file systems or other block-level storage applications. Each of the source devices has an external name, e.g. an entry in /dev for Unix systems.

- *Sink* nodes that do not have outgoing edges and correspond to bottom-level virtual devices. Sink nodes sit on top of other kernel block-level drivers, such as hardware disk drivers and, in practice, are the final recipients of I/O requests.

- *Internal* nodes that have both incoming and outgoing edges and provide virtualization functions. These nodes are not visible to external drivers, kernel components, or user applications.

Violin uses the above generic DAG representation to model its hierarchies. A *virtual hierarchy* is defined as a set of connected nodes in the device graph that do not have links to nodes outside the hierarchy. A hierarchy within a device graph is a self-contained sub-graph that can be configured and managed independently of other hierarchies in the same system. Hierarchies in
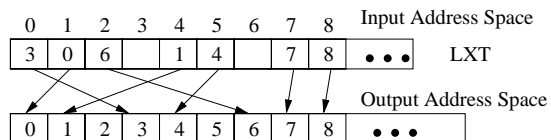
```
err_code vl_alloc_lxt(_device_t *dev);
err_code vl_alloc_pxb(_device_t *dev);
int  get/set_lxt_bool_flag_on/off(int extent,
            VL_FLAGMASK_TYPE flagmask);
int  vl_phys_address_get/set(int extent);
int  vl_phys_extent_alloc(int alloc_policy);
void vl_phys_extent_free(int extent);
```

**Figure 5. Violin API for LXT and PXB data structures.**

Violin supports dynamic block mapping through a logical-to-physical block address translation table (LXT). Figure 4 shows an example of such an LXT mapping between the input and output address spaces of a device. Dynamic block mapping capabilities give to a virtual device the freedom to use its own disk space management mechanisms and policies, without changing the semantics of its input devices, higher in the I/O stack.

Additionally, *Violin* provides free-block allocation and management facilities. It offers a variety of disk allocation schemes, including first-available, log-structured, and closer-to-last-block, all of which use either a free list (*FL*) or a physical block bitmap (*PXB*) to distinguish between occupied and free physical blocks. However, since modules may need their own space allocation algorithm, *Violin* allows module code to directly access the PXB data structure as well as its persistent metadata objects (explained below). In this case, however, code and complexity increase for the module developer.

The API for dynamic block mapping and allocation is shown in Figure 5. Since the LXT and the PXB are addressed in extents (explained below), the library functions take as arguments an extent number, which addresses an LXT or PXB entry. Functions get/set_lxt_bool_flag_on/off() handle binary flag values that can be maintained on the LXT for the extents of a device. This is useful for modules that *tag* extents according to their properties (e.g. clean/dirty). Functions vl_phys_address_get/set() handle the physical addresses stored on the LXT entries. Functions vl_phys_extent_alloc() and vl_phys_extent_free() operate on the PXB bitmap. The first routine locates a free physical block (according to an algorithm) and marks it on the PXB, while the second frees a used data block and clears its PXB bit.

The LXT and PXB data structures are allocated as persistent objects using the persistent metadata calls we describe in Section 2.3 and their persistence is handled automatically by *Violin* similarly to the rest of the hierarchy metadata.

An issue with supporting arbitrary block mappings is the size of LXT and FL or PXB persistent objects, which can become quite large, depending on the capacity and block size of a device. Keeping such objects in memory can increase memory footprint substantially. *Violin* offers two solutions: First, it supports an *internal block size* ("extent size") for its devices, which is independent of the OS device block size. The extent size, in other words, is a fixed logical block for the device and is specified when a device is created. Increasing the extent size can greatly reduce the size of the LXT. For example, the LXT of a 1 TByte device with 32-bit extent addressing and 4 KByte extent size would require
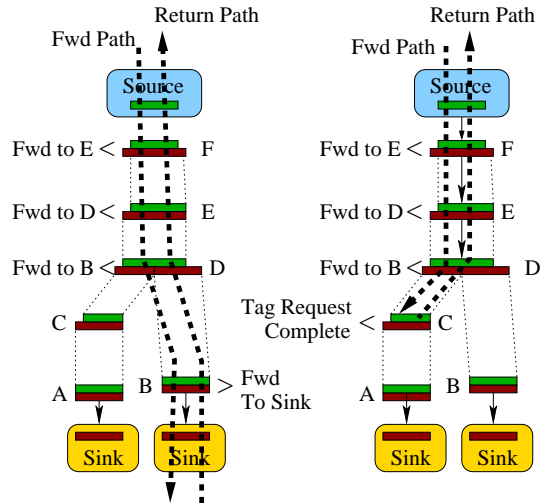
Figure 6. Example of request flows (dashed lines) through devices. Forward paths are directed downwards, while return paths upwards.

about 1 GByte of memory. By increasing the extent size to 32 KBytes, we achieve a reduction of the required metadata to 128 MBytes. Second, *Violin* provides the capability to explicitly load and unload persistent metadata to and from memory, as explained in Section 2.3.

2.2. *Violin* I/O Request Path

The second significant aspect of *Violin* is how the I/O request path works. *Violin* is not only reentrant but also supports synchronous and asynchronous I/O requests. I/O requests never block in the framework, unless a driver module has explicitly requested it. Moreover, since *Violin* is reentrant, it runs in the issuer's context for each request issued from the kernel. Thus, many requests can proceed concurrently in the framework, each in a different context.

A generic virtual storage framework must support two types of I/O requests:

- *External requests* are initiated by the kernel. They enter the framework through the source devices (nodes), traverse a hierarchy through internal nodes usually until they reach a sink node and then return back up the same path to the top of the hierarchy.

- *Internal requests* are generated from internal devices as a response to an external request. Consider for example a RAID-5 module that needs to write parity blocks. The RAID-5 device generates an internal write request to the parity device. Internal requests are indistinguishable from external ones for all but the generating module and are handled in the same manner.

I/O requests in *Violin* move from source to sink nodes through some path in a virtual hierarchy, as shown in Figure 6. Sink devices are connected to external block devices in the kernel, so after a request reaches a sink device it is forwarded to an external

device. When multiple output nodes exist, routing decisions are taken at every node, according to its mapping semantics. Virtual devices can control requests beyond simple forwarding. When a device receives an I/O request it can make four control decisions and set corresponding control tags on a request:

- *Error Tag* indicates a request error. Erroneous requests are returned through the stack to the issuer with an error code.

- *Forward Tag* indicates that the request should be forwarded to an output device. In this case, the target device and block address must also be indicated. Forwarding occurs to the direction of one of the output graph vertices.

- *Return Control Path Tag* indicates that a device needs return path control over the request. Some devices need to know when an asynchronous I/O request has completed and need control over its return path through the hierarchy (Figure 6). For instance, an encryption module requires access to the return path of a read request, because data needs to be decrypted after it is read from the sink device. However, many modules do not require return path control. If no module in a path through the hierarchy requests return path control, the request is merely redirected to another kernel device outside the framework (e.g. the disk driver) and returned directly to the issuer application, bypassing the framework's return-path control. If, however, some module requests return-path control, the framework must gain control of the request when it completes in another kernel device. In *Violin*'s Linux implementation, the driver sets an asynchronous callback on the request to gain control when it completes. When the callback receives the completed request, it passes it back up through the module stack in the reverse manner, calling the *pop I/O handlers*, described later in the API Section 2.4. Errors in the return path are handled in a similar manner as in the forward (request) path.

- *Complete Tag* indicates that a request is completed by this device. Consider the example of a caching module in the hierarchy. A caching device is able to service requests either from its cache or from the lower-level devices. If a requested data block is found in the cache, the device loads the data in the request buffer and sets the "complete" tag. The completed request is not forwarded deeper in the hierarchy, but returns from this point upwards to the issuer as shown at the right of Figure 6 for device C. If the return control path tag is set, the framework passes control to the devices in the stack that have requested it. Using this control tag, an internal device behaves as an new internal sink device.

A final issue with I/O requests flowing through the framework, is dependencies between requests. For instance, there are cases where a module requires an external request to wait for one or more internal requests to complete. To deal with this, when an internal request X is issued (asynchronously) the issuer module may register one or more dependences of X to another requests (Y, Z, ...) and provide asynchronous callback functions. Requests X, Y, Z are processed concurrently and when each completes the callback handler is called. The callback handler of the module then processes the dependent requests according to the desired ordering (i.e. it may wait for all or a few requests to finish before releasing them). This mechanism supports arbitrary dependencies

```
void *vl_persistent_obj_alloc(int oid,int size);
void *vl_persistent_obj_[un]load(int oid);
void vl_persistent_obj_flush(int oid);
void vl_persistent_obj_free(int oid);
```

Figure 7. Violin API for metadata management.

among multiple requests. Cyclic dependencies that lead to deadlock can occur by erroneous module code which is the responsibility of the module developer. Most such cases occur when a device sends requests to devices higher in the same I/O path.

2.3. State Persistence

One of the most important issues in storage virtualization is metadata management and persistence. *Violin* supports metadata management facilities. The three main issues associated with metadata are: facilitating the use of persistent metadata, reducing memory footprint, and providing consistency guarantees.

2.3.1. Persistent Metadata In *Violin*, modules can allocate and manage persistent metadata objects of varying sizes using the API summarized in Figure 7. To allocate a metadata object, a module calls vl_persistent_alloc() using a unique object ID and its size, as it would allocate memory.

Modules access metadata directly in memory, as any other data structure. However, since device metadata are loaded dynamically when the device in initialized, pointers may not point to the right location. There are two methods for addressing this issue: (i) disallow pointers in metadata accesses and (ii) always force loading of metadata to the same address. Since the latter can be a problem for large metadata objects and in our experience the former is not overly restrictive, in our current design we disallow pointers in metadata objects. However, this issue requires further investigation and is left for future work. When a module needs to destroy a persistent object it calls the vl_persistent_free() function.

By default, when a virtual device is created in *Violin*, the system stores its metadata in the same output address space as the data. Alternatively, the user may specify explicitly a metadata device that will be used for storing persistent metadata for the new virtual device. This can be very useful when metadata devices are required to have stronger semantics compared to regular data devices, e.g. a higher degree of redundancy.

Violin's metadata manager automatically synchronizes dirty metadata from memory to stable storage in an asynchronous manner. The metadata manager uses a separate kernel thread to write to the appropriate device metadata that are loaded in memory. The user can also flush a metadata object explicitly with the vl_persistent_flush() call.

Internally, each metadata object is represented with an object descriptor, which is modified only by allocation/deallocation calls. During these calls metadata object descriptors are stored in a small index in the beginning of the metadata device. A pointer to the metadata header index is stored with each virtual device in the virtual hierarchy. Thus, when the virtual hierarchy is being loaded and recreated, *Violin* reads the device metadata and loads it to memory.

2.3.2. Reducing Memory Footprint This is important for large metadata objects that may consume a large amount of main memory or incur large amounts of I/O during saving to the metadata device. *Violin* provides API calls for explicitly loading and unloading metadata to and from memory within each module (Figure 7). Another possible solution is to transparently swap metadata to stable storage. This solution is necessary for extremely large devices, where the size of the required metadata can sometimes exceed the available memory of a machine. Since it is not clear whether such modules actually need to be implemented, *Violin* does not support this feature yet.

Violin retains a small amount of internal metadata that represent the hierarchy structure and are necessary to reconstruct the device graph. Each graph node and edge requires a pointer, which amounts to a few bytes per virtual device. The hierarchy metadata is saved in all physical devices within a hierarchy. For this purpose, *Violin* reserves a superblock of a few KBytes in every physical device of a hierarchy. A complete hierarchy configuration can thus be loaded from the superblock of any physical device that belongs to the hierarchy.

In summary, each module needs to allocate its own metadata, and in the common case, where metadata can be kept in memory, it does not need to perform any other operation. The framework will maintain metadata persistent and will reload the metadata from disk each time the virtual device is loaded, significantly simplifying metadata management in individual modules.

2.3.3. Metadata Consistency In the event of system failures, where a portion of the in-memory metadata may be lost or partially written, application and/or *Violin* state may be corrupted. We can define the following levels of metadata consistency:

1. *Lazy-update consistency,* that is, metadata are synchronized on disk overwriting the older version every few seconds. This means that if a failure occurs between or during updates of metadata then metadata may be left inconsistent on-disk and *Violin* may not be able to recover. In this case, there is a need for a *Violin*-level recovery procedure (similar to fsck), which however, we do not currently provide. If stronger guarantees are required then one of the next forms of consistency may be used instead.

2. *Shadow-update consistency*, where we use two metadata copies on disk and maintain at least one of the two consistent at all times. If during an update the set that is currently being written becomes inconsistent due to a failure, *Violin* uses the second copy to recover. In this case, it is guaranteed that *Violin* will recover the device hierarchy and all its persistent objects and will be able to service I/O requests. However, application data may be inconsistent with respect to system metadata.

3. *Atomic versioned-metadata consistency,* guarantees that after a failure, the system will be able to see a previous, consistent version of application data and system metadata. Thus, this is equivalent to a rollback to a previous point in time. In *Violin* this can be achieved by using a versioning layer [7] at the leaves of a hierarchy. Although such a layer is available in *Violin*, given its current implementation we need to slightly modify it so that its own metadata are handled differently in this particular case.

```
-> initialize (_device_t *dev, ...);
-> open  (_device_t *dev,void *data);
-> close (_device_t *dev,void *data);
-> read_push  (_device_t *dev, ...);
-> read_pop   (_device_t *dev, ...);
-> write_push (_device_t *dev, ...);
-> write_pop  (_device_t *dev, ...);
-> ioctl  (_device_t *dev, int cmd, ...);
-> resize (_device_t *dev, int new_size);
-> device_info (_device_t *dev, ...);
```

Figure 8. API methods for Violin's I/O modules.

Violin currently supports the first and second forms of metadata consistency. We expect that all three forms of consistency will be available in the future releases of the framework code.

2.4. Module API

Extending an I/O hierarchy with new functionality is an arduous task in modern kernels. The interface provided by kernels for block I/O is fairly low-level. A block device driver has the role of servicing block I/O read and write requests. Block requests adhere to the simple block I/O API, where every request is denoted as a tuple of (block device,read/write,block number,block size,data). In Linux, this API involves many tedious and error prone tasks, such as I/O request queue management, locking and synchronization of the I/O request queues, buffer management, translating block addresses and interfacing with the buffer cache and the VM subsystem.

Violin provides to its modules high-level API calls, that intuitively support its hierarchy model and hide the complexity of kernel programming. The author of a module must set up a *module object,* which consists of a small set of variables with the attributes of the implemented module and a set of methods or API functions that can be seen in Figure 8. The variables of the module object in our current implementation include various static pieces of information about each module, such as the module name and id, the minimum and maximum number of output nodes supported by the device, and the number of non-read and write (ioctl) operations supported by the module. The role of each method prototype is:

- initialize() is called once, the first time a virtual device is created to allocate and initialize persistent metadata and module data structures for this device. Note that since *Violin* retains state in persistent objects, this method will not be called when a hierarchy is loaded, since device state is also loaded.

- open(), close() are called when a virtual device is loaded or unloaded during the construction of hierarchy. The module writer can use the close() call to perform garbage collection and other necessary tasks for shutting down the device.

- read(), write() handle I/O requests passed to a virtual device. Each has two methods, push and pop, that represent the different I/O paths. Push handlers are called in the forward path, while pop handlers are called in the return path, if requested. These API methods are higher-level compared to
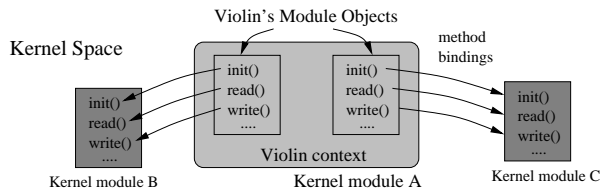
Figure 9. Binding of extension modules to Violin.

their kernel counterparts and do not require complex I/O buffer management and I/O queue management.

- `ioctl()` handles custom commands and events in a module, in case direct access to it is required. This is used in some of our modules, for example to explicitly trigger a snapshot capture event in the versioning module.

- `resize()` is an optional method that specifies a new size for an output virtual device. When this method is called in a virtual device, it means that one of its output devices has changed size as specified and thus, the virtual device has to adjust all internal module metadata appropriately.

- `device_info()` is used to export runtime device information in human or machine readable form.

The main difference of this API from the classic block driver API in Unix systems is the distinct push() and pop() methods for read and write operations, versus a read(), write() in the classic API. This is the key features of *Violin*'s API, that allows asynchronous behavior. Finally the role of the initialize() call is different from the classic API. Since state persistence is maintained, device initialization is necessary only once in its lifetime, at its creation.

3. System Implementation

We have implemented *Violin* as a loadable block device driver in the Linux 2.4 kernel accompanied by a set of simple user-level management tools. Our prototype implements fully the I/O path model described in Section 2.4. *Violin* extension modules are implemented as separate kernel modules that are loaded on demand. However, they are not full Linux device drivers themselves but require the framework's core. Upon loading, each module registers with the core framework module, binding its methods to internal *module objects* as shown in Figure 9.

A central issue with the *Violin* driver is the implementation of its asynchronous API on Linux 2.4. Next, we explain how I/O requests are processed in Linux block device drivers and then describe *Violin*'s implementation.

3.1. I/O Request Processing in Linux

The Linux kernel uses an *I/O request* structure to schedule I/O operations to block devices. Every block device driver has an *I/O request queue*, containing all the I/O operations intended for this block device. Requests are placed in this queue by the kernel function `make_request_fn()`. This function can be overridden by a block driver's own implementation. Requests are removed from the queue and processed with the *strategy* or *request* function of a block driver. When a request is complete, the driver releases the I/O request structure back to the kernel after tagging
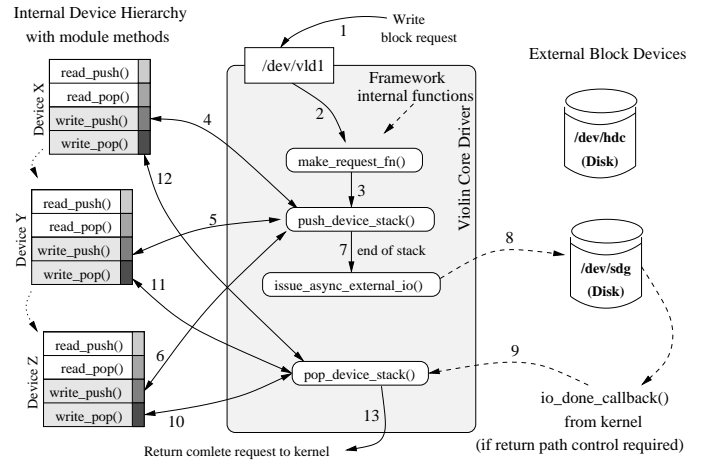

Figure 10. Path of a write request in Violin.

it as successful or erroneous. If a released request is not appropriately tagged or the data is incorrect, the kernel may crash.

Thus, there are two ways of processing I/O requests in a Linux device driver. First, a device driver can use a request queue as above where the kernel places requests and the *strategy* driver function removes and processes them. This approach is used by most hardware block devices. Second, a driver may override the `make_request_fn()` kernel function with its own version and thus, gain control of every I/O request before it is placed in the request queue. This mode of operation is more appropriate for virtual I/O drivers. In this mode, the driver does not use an I/O request queue, but rather *redirects* requests to other block devices. This is the mode of operation used by many virtual I/O drivers, such as LVM and MD. When a new I/O request arrives, the device driver performs simple modifications to the I/O request structure, such as modifying the device ID and/or the block address and returns the request to the kernel that will redirect it to the new device ID.

3.2. *Violin* I/O path

Violin uses the method of replacing the `make_request_fn()` call and operates asynchronously without blocking inside this function. For every external I/O request received through the `make_request_fn()` call, *Violin*'s device driver traverses the I/O stack and calls the necessary module I/O handlers (`read_push()` or `write_push()` depending on the type of request). The I/O path traversal for a write request can be seen in Figure 10.

Each handler processes the incoming request (modifying its address, data, or both) and then returns control to the `push_device_stack()` function, indicating the next device in the path from one of its dependent (or lower) devices, marked with outward arrows. Thus, the `push_device_stack()` function passes the request through all the devices in the path. When the request reaches an exit node, the driver issues an asynchronous request to the kernel to perform the actual physical I/O to another kernel driver (e.g. the disk driver). When return path control is needed, the driver uses callback functions that Linux attaches to I/O buffers. Using these callbacks the driver is able to regain con-

trol of the I/O requests when they complete in another driver's context and perform all necessary post-processing.

A Linux-specific issue that *Violin*'s kernel driver must deal with, is handling buffer cache data that need module processing and modification (e.g. encryption). The Linux kernel I/O request descriptors, contain pointers to memory pages with the data mapped by the buffer cache and other kernel layers, e.g. the filesystem, to disk blocks. While an I/O request is in progress, the memory page remains available for read access in the buffer cache. Data-modifying layers, e.g. encryption, however, need to transform the data on the page before writing it to the disk. Data modification on the original memory page, results in the corruption of the filesystem and buffer cache buffers. To resolve this problem with data-modifying modules, *Violin* creates a copy of the memory page, where modules can modify the data and writes the new page to the disk. The original page remains available in the buffer cache.

4. Evaluation

In this section we evaluate the effectiveness of *Violin* in three areas: ease of development, flexibility, and performance.

4.1. Ease of Development

Measuring development effort is a hard task, since there exist no widely accepted metrics for it. We attempt to quantify the development effort by looking at the code size reduction that we can achieve using *Violin*. Similar to FiST [32], we use code size as a metric to examine relative complexity. We compare code sizes of various block device drivers that exist for Linux with implementations of these modules under *Violin*:

- **RAID:** Implements RAID levels 0, 1, and 5. Failure detection and recovery is also implemented for levels 1 and 5. Using *Violin*'s ability to synthesize complex hierarchies, this module can be used to create composite RAID levels, such as 1+0.

- **Versioning:** Offers on-line snapshot functionality at the block level [7].

- **Partitioning:** Creates a partition on top of another device. It supports partition tables and partition resizing.

- **Aggregation:** Functions as a volume group manager. It aggregates many devices into one, either appending each after the other or using striping and allows resizing of the individual or aggregate device. Also, devices can be added or removed and data can be migrated from one device to another.

- **MD5 hashing:** Computes the MD5 fingerprint for each block. Fingerprints are inserted in a hash table and can be queried. Its purpose is to offer content-based search capabilities and facilitate differential compression.

- **Encryption:** Encrypts/decrypts data blocks, using a user-specified key. We have currently implemented the Blowfish, DES and 3DES encryption algorithms.

- **Online Migration:** Transparently migrates data from one device to another, at a user-controlled speed.

Table 4.1 shows the code size for each functional layer implemented as a driver under Linux and as a module under *Violin*. The features implemented in *Violin* modules are close to the features

Table 1. Linux drivers and Violin modules in kernel code lines.

Virtualization Layers / Functions	Number of code lines	
	Linux Driver	*Violin* Module
RAID	11223 (MD)	2509
Partition & Aggregation	5141 (LVM)	1521
Versioning	4770 (Clotho)	809
MD5 Hashing	–	930
Blowfish Encryption	–	804
DES & 3DES Encryption	–	1526
Migration	–	422
Core *Violin* Framework	14162	–

provided by the block drivers under Linux, including LVM. The only exception exists in the MD Linux [2] driver. Our implementation, although it includes code for failure detection and recovery in RAID, does not support the feature of spare disks in a RAID volume. The main reason for not adding this functionality is that we expect it to be handled by remapping failed disks to spare ones with *Violin*.

We see that the size of the *Violin* framework core is about 14200 lines of code. Looking at individual modules, in the case of LVM, code length is reduced by a factor of three, while in MD and Versioning the reduction is about a factor of four and six respectively. In MD, more than half of the module code (1300 lines) is copied from MD and is used for fast XOR computation. This piece of code is written mainly in assembly and tests various XOR implementations for speed in order to select the fastest one for each system. If we compute code-length differences without the user-level copied code, i.e. the XOR code, the code difference for the RAID module reaches an order of magnitude.

Clearly code size cannot be used as a precise measure of complexity. However, it is an indication of the reduction in effort when implementing new functionality under *Violin*. For the modules we implement, we can attribute the largest code reductions to the high-level module API and to the persistent metadata management support.

To examine how new functionality can be provided under *Violin*, we implement two new modules for which we could not find corresponding Linux drivers: MD5 hashing and DES-type encryption. For each module we use publicly available user-level code. Table 4.1 shows that MD5 hashing is about 900 lines of code in *Violin* out of which 416 lines are copied from the user-level code. Similarly, our DES encryption module uses about 1180 lines of user-level code, out of a total of 1526 lines. Creating each module was an effort of just a few hours.

4.2. Flexibility

To demonstrate the flexibility of the proposed framework we show how one can create more complex multi-layered hierarchies by combining simple modules. Providing the same functionality with standard OS mechanisms is much more difficult. In this demonstration, we construct a complex hierarchy of virtual devices over a set of physical devices, adding one layer at a time.
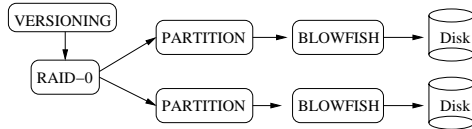
```
#SCRIPT TO CREATE THE HIERARCHY:
#Initialize a hierarchy named "voila"
hrc_init /dev/vld voila 32768 1
#Import two IDE disks: /dev/hdb and /dev/hdc
dev_import /dev/vld /dev/hdb voila hdb
dev_import /dev/vld /dev/hdc voila hdc
#Create a Blowfish Encryption Layer on each disk
dev_create /dev/vld Blowfish voila BF_Dev_1 1 \
          hdb BF_Dev_1 0
dev_create /dev/vld Blowfish voila BF_Dev_2 1 \
          hdc BF_Dev_2 0
#Create two Partitions on top of these
dev_create /dev/vld Partition voila Part_Dev_1 \
          1 BF_Dev_1 -1 900000 Part_Dev_1 0
dev_create /dev/vld Partition voila Part_Dev_2 \
          1 BF_Dev_2 -1 900000 Part_Dev_2 0
#Create a RAID-0 device on the 2 partitions
dev_create /dev/vld RAID voila RAID0_Dev 2 \
          Part_Dev_1 Part_Dev_2 0 32 RAID0_Dev 0
#Create a Versioned device on top
dev_create /dev/vld Versioning voila Version_Dev \
          1 RAID0_Dev 40 Version_Dev 200000
#Link hierarchy "voila" to /dev/vld1
dev_linkpart /dev/vld voila Version_Dev 1
#Make a filesystem on /dev/vld1 and mount it
mkfs.ext2 /dev/vld1
mount -t ext2 /dev/vld1 /vldisk
```

Figure 11. A hierarchy with advanced semantics and the script that creates it.

Figure 11 depicts how such a hierarchy is built in a bottom-up fashion using simple user-level tools. Initially two disks are inserted in the hierarchy as sink devices. Next, a partitioning layer is added, which creates two partitions, one on each disk. Then, a RAID-0 layer creates a striped volume on top of the two partitions. Next, a versioning layer is added on top of the striped volume to provide online snapshots. Finally, a Blowfish encryption layer is added in the hierarchy to encrypt data. While this module can be placed anywhere in the hierarchy, we chose to place it directly above the disks.

Overall, we find that *Violin* simplifies the creation of complex hierarchies that provide advanced functionality, once the corresponding modules are available.

4.3. Performance

To evaluate the performance of *Violin* we use two well-known I/O benchmarks, Iometer [13] and PostMark [15]. Iometer is a benchmark that generates and measures I/O workloads with various parameters. PostMark is a synthetic benchmark that emulates the operation of a mail server. Postmark runs on top of a filesystem, while Iometer runs on raw block devices.

The system we use in our evaluation is a commodity x86-based Linux machine, with two AMD Athlon MP 2200+ CPUs,

512 MB RAM, three Western Digital WDC WD800BB-00CAA1 ATA Hard Disks with 80 GB capacity, 2MB cache, and UDMA-100 support. The OS is Red Hat Linux 9.0, with RedHat's latest 2.4.20-31.9smp kernel.

First, we examine the performance of single *Violin* modules compared to their Linux driver counterparts. We use three different configurations: raw disk, RAID, and logical volume management. In each configuration we use layers implemented as a block driver under Linux and as modules under *Violin*:

A. The raw Linux disk driver (Disk) vs. a *pass-through* module in *Violin* (Disk Module). The same physical disk partition is used in both cases.

B. The Linux Volume Manager driver (LVM) vs. a dual module setup in *Violin* using the partitioning and aggregation modules (Aggr.+Part. Modules). In the *Violin* setup we use two disks that are initially aggregated into a single resizable striped volume. On top of this large volume we create a partition where the filesystem is created. In LVM we use the same setup. We initialize the same disks into physical volumes, create a volume group consisting of the two disks, and finally create a striped logical volume on top. The size of the stripe block is 32 KBytes in both setups.

C. The Linux MD driver (MD RAID-0 & 1) vs. the *Violin* RAID module (RAID-0 & 1 Module). Both modules support RAID levels 0, 1, and 5. We compare performance of RAID levels 0 (striping) and 1 (mirroring) for MD and *Violin*. In both setups we use the same two disks and a stripe block of 32 KBytes for RAID-0. Note that the system setup in the RAID-0 case is identical to the LVM experiments, with the exception of different software layers being used.

4.3.1. Iometer Iometer [13] generates I/O workloads based on specified parameters (e.g. block size, randomness) and measures them on raw block devices. We vary three parameters in our workloads: (i) *Access pattern*: we use workloads that are either sequential or random. (ii) *Read-to-write ratio*: we explore three ratios, 100% reads, 100% writes and a mix of 70% reads - 30% writes. (iii) *Block size*: we use 512 byte to 64 KByte block sizes.

Figures 12-15 summarize our results with Iometer version 2004.07.30. Figure 12 compares the performance of a pass-through to disk *Violin* module to the plain system disk. Figures 13, 14 and 15 compare the performance of *Violin* modules to their kernel counterparts, LVM and MD RAID levels 0 and 1. The left column graphs represent sequential I/O performance, while the right graphs show random I/O performance. In most cases we observe that the performance difference between similar functionality modules is less than 10%. There are cases when *Violin* has a bit better performance than the kernel drivers and cases when it performs a little worse.

The larger performance differences observed are: (i) a low sequential performance of LVM and *Violin* modules for block sizes less than 4KB. This is due to their default block size for their device drivers, which is set to 4KB, compared to 1KB for raw disk and the MD driver. This explains the steep performance increase for *Violin* and LVM that is consistently observed between block sizes of 2KB and 4KB. Using a smaller system block size on *Violin* is possible, but requires changes to the current *Violin*
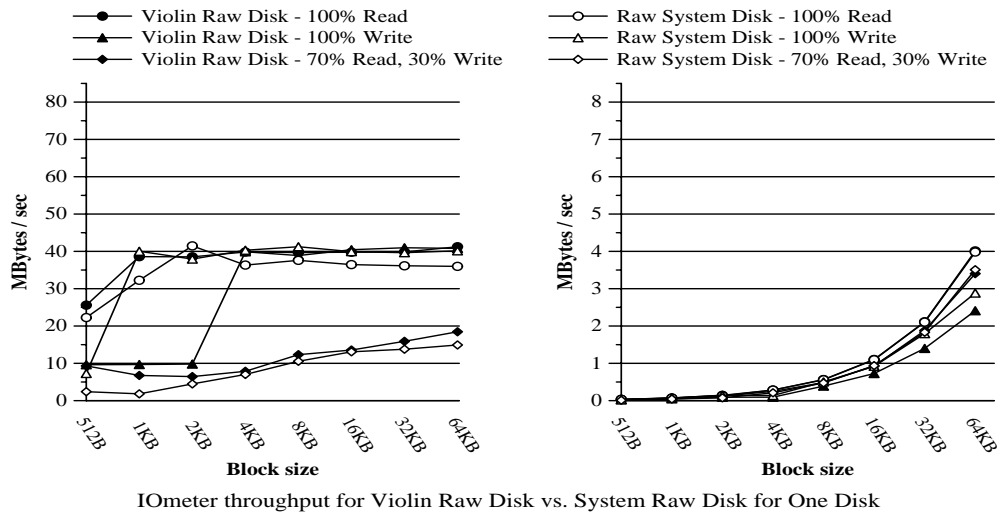
IOmeter throughput for Violin Raw Disk vs. System Raw Disk for One Disk

Figure 12. Raw disk throughput (MBytes/sec) for sequential (left) and random (right) workloads.

IOmeter throughput for Violin Aggregation+Partition vs. LVM for 2 Striped Disks (32K stripe)
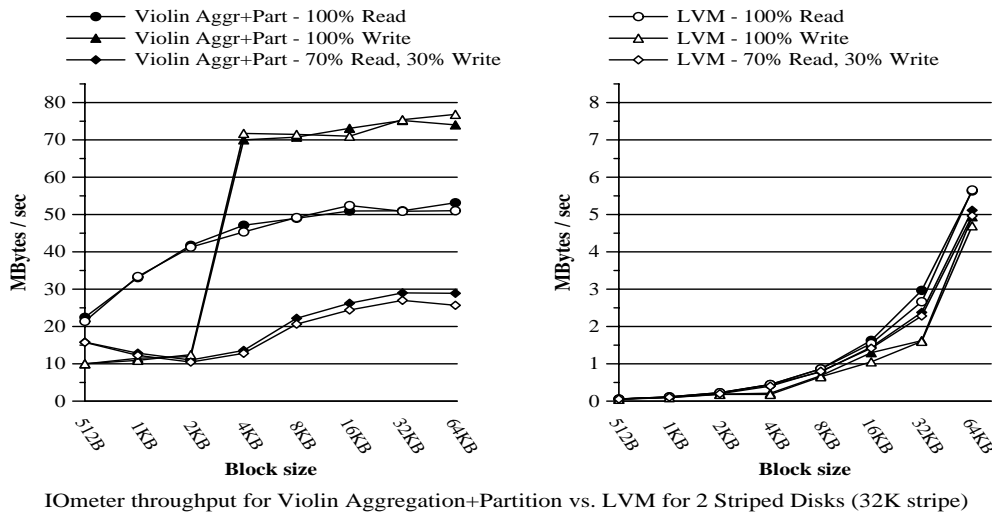
Figure 13. LVM throughput (MBytes/sec) for sequential (left) and random (right) workloads.

code. (ii) MD has higher performance for random write operations, both in RAID-0 and RAID-1 modes. This is due to MD's optimized buffer cache management, which caches writes and can more effectively cluster write requests. This is mainly a Linux implementation issue and we are currently working on improving buffer cache management in *Violin*'s driver.

4.3.2. Postmark results PostMark [15] creates a pool of continually changing files on a filesystem and measures the transaction rates for a workload approximating a large Internet electronic mail server. Initially a pool of random text files is generated, ranging in size from a configurable low bound to a configurable high bound. Once the pool has been created, a specified number of transactions occurs. Each transaction can be one of two pairs of I/O operations: (i) create file or delete file and (ii) read file or append file. Each transaction type and its affected files are chosen randomly. When all of the transactions have completed, the remaining active files are deleted.

To evaluate the performance of each configuration, we use a workload consisting of: (i) files ranging from 100KBytes to 10MBytes in size, (ii) an initial pool of 1000 files, and (iii) 3000 file transactions with equal biases for read/append and create/delete. Each run is repeated five times on the same filesystem to account for filesystem staleness and the results are averaged over the five runs. During each run there are over 2500 files created, about 8.5GBytes of data read and more than 15GBytes of data written.

Figure 16 shows our PostMark results. In each graph and configuration, the light bars depict the kernel device driver measurements, while the darker bars show the numbers for *Violin* modules. The bars have been grouped according to each system configuration and PostMark experiment. In all cases, except RAID-1, there is a small difference, less than 10% between the two systems, Linux driver vs. *Violin* module. In the RAID-1 case there is a large difference of about 30%, due again to the better buffer-
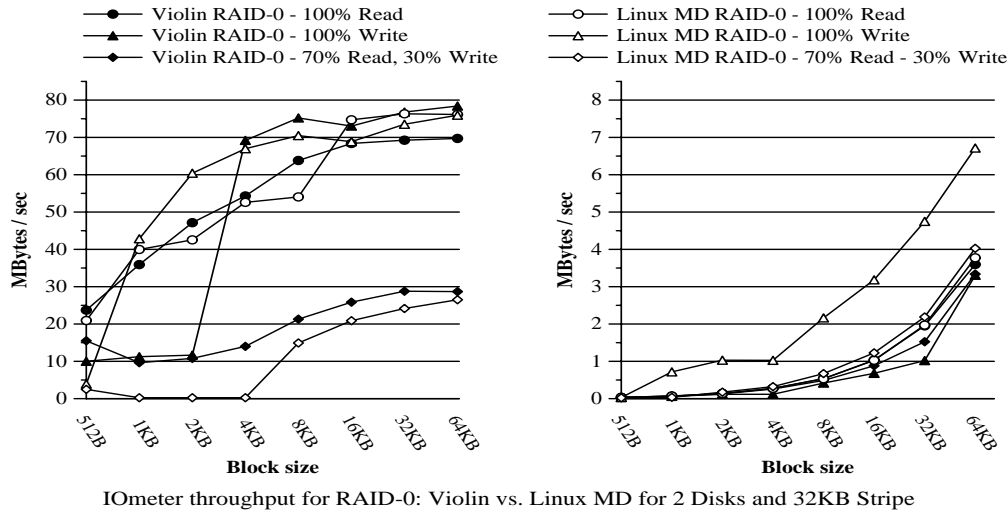
IOmeter throughput for RAID-0: Violin vs. Linux MD for 2 Disks and 32KB Stripe

Figure 14. RAID-0 throughput (MBytes/sec) for sequential (left) and random (right) workloads.

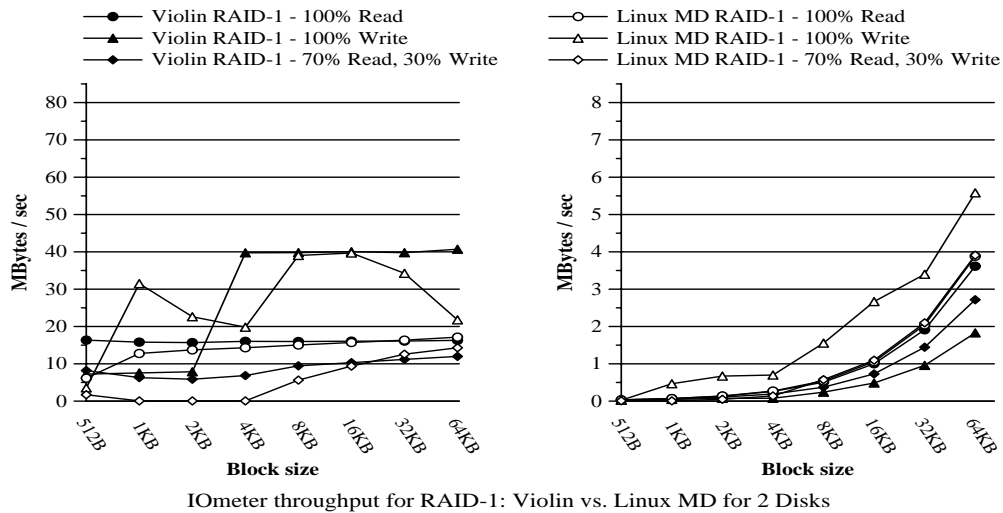IOmeter throughput for RAID-1: Violin vs. Linux MD for 2 Disks

Figure 15. RAID-1 throughput (MBytes/sec) for sequential (left) and random (right) workloads.

cache management that MD has. As mentioned before, we are currently improving this aspect of *Violin*'s driver.

4.4. Hierarchy Performance

Finally, for the hierarchy of Figure 11, we evaluate with Iometer the performance of the hierarchy as each module is introduced. Our goal is to provide some intuition on expected system performance with complex configurations. Figure 17 shows the results for sequential (top) and random (bottom) workloads. Each curve corresponds to a different configuration as each module is introduced to the hierarchy. In the sequential workloads, we note that variations in performance occur depending on the functionality of the layers being introduced. We see a large performance reduction in the versioning module, which is due to the disk layout that this module alters [7]. Encryption overhead on the other hand, appears to be about 10-15%. In random workloads we observe that the hierarchy performance is influenced only by the versioning

layer functionality, which can increase performance substantially. This happens because of the change in the disk layout that versioning incurs, through write request logging [7]. Encryption or RAID-0, on the other hand, do not seem to influence performance for random I/O.

5. Related Work

In this section we comment on previous and related work on (a) extensible filesystems, (b) extensible network protocols, and (c) block-level storage virtualization.

5.0.1. Extensible filesystems Storage extensions can be implemented at various levels in a storage system. The highest level is within the filesystem, where combined software layers implement the desired functionality. The concept of layered filesystems has been explored in previous research.

Ficus [11], one of the earliest approaches, proposes a filesys-

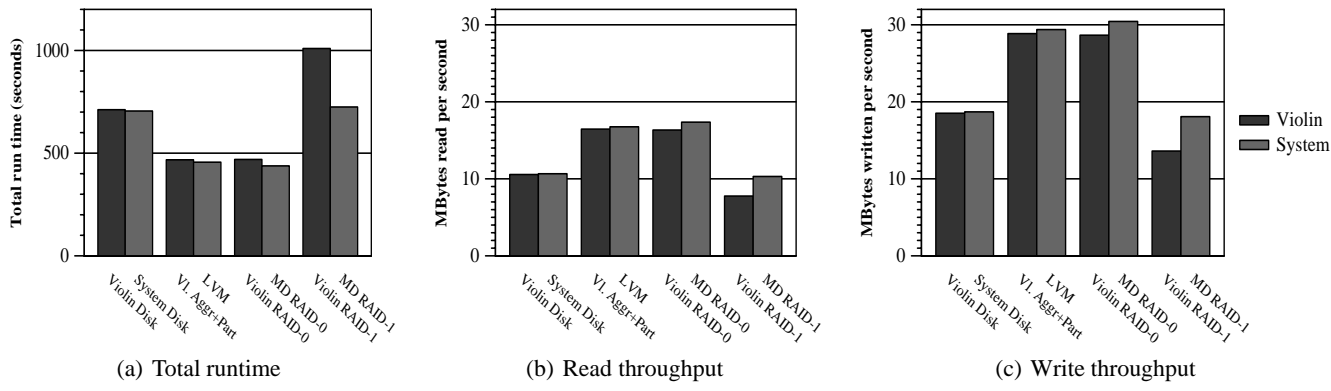111

(a) Total runtime   (b) Read throughput   (c) Write throughput

Figure 16. Postmark results.

Workload: 100% Sequential Read   Workload: 100% Sequential Write   Workload: 70% Read - 30% Write Sequential

Workload: 100% Random Read   Workload: 100% Random Write   Workload: 70% Read - 30% Write Random
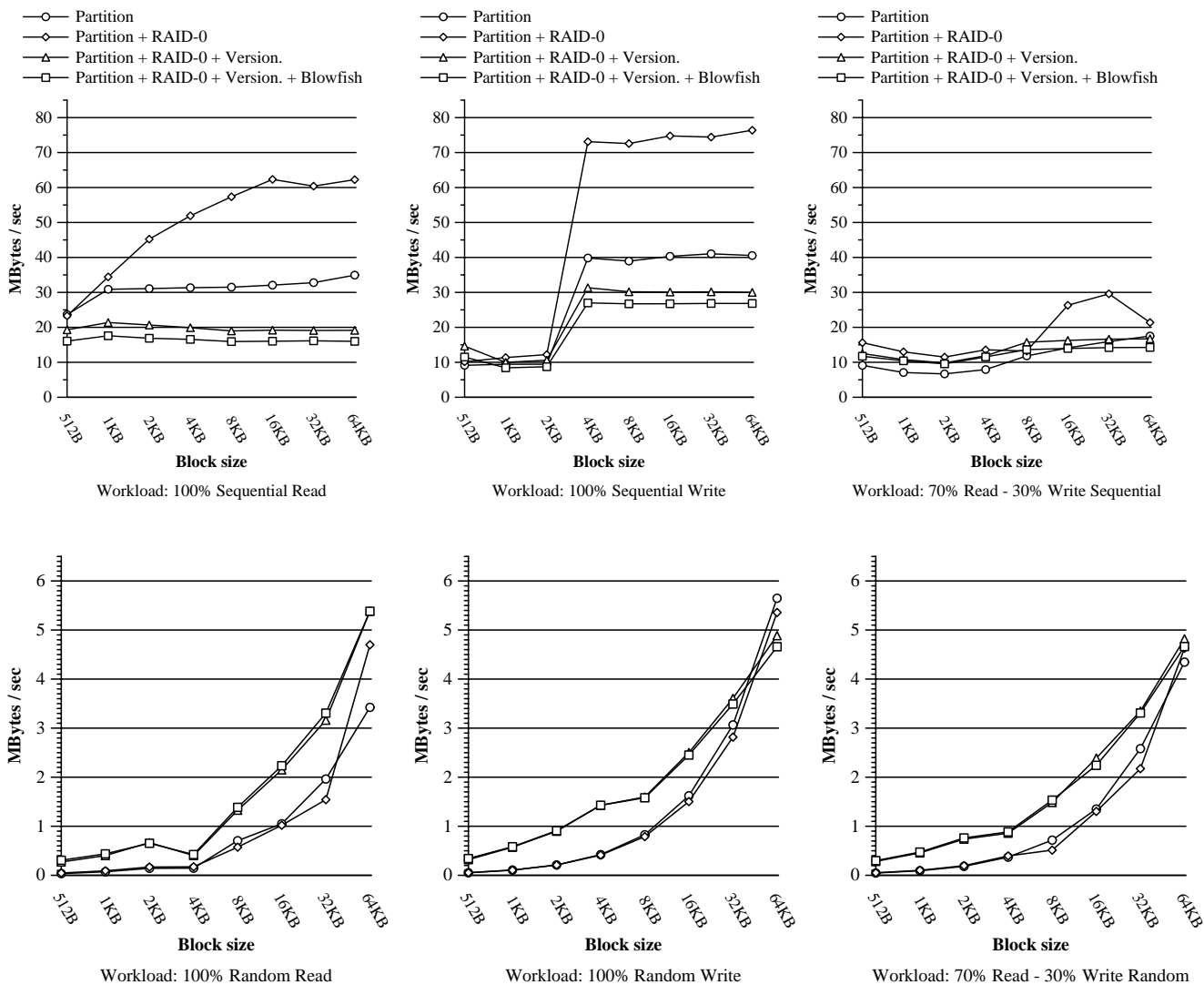
Figure 17. Throughput for hierarchy configuration as layers are added, using Iometer.

tem with a stackable layer architecture, where desired functionality is achieved by loading appropriate modules. A later approach [30] proposes layered filesystem implementation by extending the vnode/VFS interface to allow "stacking" vnodes. A similar concept at the user-level, was proposed in the recent Mona FS [24], which allows application extensions to files through streaming abstractions. Another recent approach, FiST [32], uses a high-level specification language and a compiler to produce a filesystem with the desired features. FiST filesystems are built as extensions on top of *baseFS*, a native low-level filesystem. A similar concept was used in the Exokernel [14], an extensible OS that comes with XN, a low-level in-kernel storage system. XN allows users to build library filesystems with their desired features. However, developing library filesystems for the Exokernel requires significant effort and can be as difficult as developing a native monolithic FS for an operating system.

Our work shares similarity with these approaches to the extent that we are also dealing with extensible storage stacks. However, the fact that we work at the block-level in the storage system requires that our framework, provides different APIs to modules, uses different I/O abstractions for its services, and different kernel facilities in its implementation. Thus, although the high level concept is similar to extensible filesystems, the challenges faced and solutions provided are different.

Implementing storage extensions at either the filesystem level or the storage system level is desirable, each for different reasons and each approach is not exclusive of the other. The basic pros and cons of each approach stem from the API and metadata each level is aware of. One fundamental characteristic of filesystems, for example, is that they have file metadata. Thus, they can associate blocks that belong to the same file and are able to provide policies at the file level. On the other hand, storage systems operate at the block-level and they have no information about the relationships of blocks. Thus, metadata need to be maintained at the block level resulting potentially in large memory overhead. Moreover, block I/O operations cannot be associated precisely with each other, limiting possible optimizations. On the positive side, block-level extensions are transparent to any filesystem and volume-level policies can be provided. Moreover, many block-level functions, e.g. encryption or RAID-levels, can operate faster than at the file system level, since they operate on raw fixed-size blocks and not on the structured variable-sized data of the filesystem.

5.0.2. Extensible network protocols Our work on storage virtualization shares similarities with frameworks for layered network stacks. The main efforts in this direction are the Click Router [17], X-kernel [20], and Horus [26]. All three approaches aim at building a framework for synthesizing network protocols from simpler elements. They use a graph representation for layering protocols, they envision simple elements in each protocol, and provide mechanisms for combining these simple elements in hierarchies with rich semantics and low performance overhead. Scout [19] is a communication-centric OS, also supporting stackable protocols. The main abstraction of Scout is the I/O paths between data sources and sinks, an idea applicable both to network and storage stacks.

In our work we are interested in providing a similar framework for block-level storage systems. We share the same observation that there is an increased need to extend the functionality

of block-level storage (network protocol stacks) and that doing so is a challenging task with many implications on storage (network) infrastructure design. However, the underlying issues in network and storage stacks are different:

- Network stacks distinguish flows of self-contained packets, while storage stacks cannot distinguish flows, but map data blocks from one device to another.

- Network and storage stacks exhibit fundamentally different requirements for *state persistence*. Network stacks do not need to remember where they scheduled every packet in order to recover it at a later time, and thus, do not require extensive metadata. On the other hand, storage stacks must be able to reconstruct the data path for each I/O request passing through the stack, requiring often times large amounts of persistent metadata.

- Send and receive paths in network protocol stacks are fairly independent, whereas in a storage hierarchy there is strong coupling between the request and completion paths for each read and write request. Moreover, an issue not present in network protocol stacks is the the need for asynchronous handling of I/O requests and completions, which introduces additional complexity in the system design and implementation.

5.0.3. Block-level storage virtualization The most popular virtualization software is volume managers. 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 [2] and device-mapper kernel modules to support user-level plugins called *features*. However, it does not offer persistent metadata or block remapping primitives to these plugins. Moreover, EVMS focuses on configuration flexibility with predefined storage semantics (e.g. RAID levels) and does not easily allow generic extensions (e.g. versioning). GEOM [8] is a stackable BIO subsystem under development for FreeBSD. The concepts behind it GEOM are, to our knowledge, the closest to *Violin*. However, GEOM does not support persistent metadata which, combined with dynamic block mapping are necessary for advanced modules such as versioning [7]. LVM [25] and Vinum [18] are simpler versions of EVMS and GEOM. *Violin* has all the configuration and flexibility features of a volume manager coupled with the ability to write extension modules with arbitrary virtualization semantics.

Besides open-source software, there exist numerous virtualization solutions in the industry. HP OpenView Storage Node Manager [12] helps administrators control, plan, and manage direct-attached and networked storage, acting as a central management console. EMC Enginuity [4], a storage operating environment for high-end storage clusters, employs various techniques to deliver optimized performance, availability and data integrity. Veritas Volume Manager [28] and Veritas File System aim at assisting with online storage management. Similar to other volume managers, physical disks can be grouped into logical volumes to improve disk utilization and eliminate storage-related downtime. Moreover, administrators have the ability to move data between different storage arrays, balance I/O across multiple paths to improve performance, and replicate data to remote sites for higher availability. However, in all cases, the offered virtualization functions are predefined and they do not seem to support extensibility

of the I/O stack with new features. For instance, EMC Enginuity currently supports only the following predefined data protection options: RAID-1, Parity RAID (1 parity disk per 3 or 7 data disks), and RAID-5 [5].

RAIDframe [31] enables rapid prototyping and evaluation of RAID architectures, which is similar but narrower than our goals. The authors use a DAG representation for specifying RAID architectures, similar to *Violin*. However, their goal is to evaluate certain architectural parameters (encoding, mapping, caching) and not to provide extensible I/O hierarchies with arbitrary virtualization functions.

The latest versions of the Windows OS support an integrated model for building device drivers, the Windows Driver Model (WDM) [21]. This model specifies the runtime support available to writers of device drivers. However, unlike *Violin* for block-level devices, it only provides generic kernel support and does not include functionality specific to storage.

6. Limitations and Future work

Overall, *Violin* offers a highly configurable environment to easily experiment with advanced storage functionality. The main limitation of our work is that we have evaluated *Violin* with a specific set of modules. However, we believe that this set is broad enough to demonstrate the benefits of our approach. Further extensions to *Violin* are possible as we gain more experience with its strengths and limitations.

There are two important directions for future work. First, given that we can extend the I/O hierarchy with a rich set of mechanisms, it is important to examine how user requirements can be mapped to these mechanisms automatically [16]. An interesting aspect on this research direction is to answer the question of how much a system can do in terms of optimizations in this translation process both statically when the virtual I/O hierarchy is built [29], but mostly dynamically during system operation.

Second, it is interesting to examine how *Violin* can be extended to include the network path in virtual hierarchies. Networked storage systems offer the opportunity to distribute the virtual I/O hierarchy throughout the path from the application to the disk, as shown in Figure 1. Exploring the various possibilities and tradeoffs may provide insight on how virtualization functionality should be split among components in storage systems.

7. Conclusions

In this work we design, implement, and evaluate *Violin*, a virtualization framework for block-level disk storage. *Violin* allows easy extensions to the block I/O hierarchy with new mechanisms and flexible combining of these mechanisms to create modular hierarchies with rich semantics.

To demonstrate its effectiveness we implement *Violin* within the Linux operating system and provide several I/O modules. We find that *Violin* significantly reduces implementation effort. For instance, in cases where user-level library code is available, new *Violin* modules can be implemented within a few hours. Using a simple user-level tool we create I/O hierarchies that combine the functionality of various modules and provide a set of features difficult to offer with monolithic block-level drivers. Finally, we use two benchmarks to examine the performance overhead of *Violin* over traditional, monolithic drivers and driver-based hierarchies,

and find that *Violin* modules and hierarchies perform within 10% of their counterparts.

Overall, we find that our approach provides adequate support for embedding powerful mechanisms in the storage I/O stack with manageable effort and small performance overhead. We believe that *Violin* is a concrete step towards supporting advanced storage virtualization, reducing storage management overheads and complexity, and building self-managed storage systems.

8. Acknowledgments

REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. of the 8th ASPLOS*, pages 81–91, San Jose, California, Oct. 3–7, 1998.

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

[3] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of 14th SOSP*, pages 15–28, 1993.

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

[5] EMC. Introducing RAID 5 on Symmetrix DMX. http://www.emc.com/ products/ systems/ enginuity/ pdf/ H1114_Intro_ raid5_DMX_ldv.pdf.

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

[7] 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.

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

[9] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th ASPLOS*, pages 92–103, San Jose, California, Oct. 3–7, 1998.

[10] J. Gray. Storage Bricks Have Arrived. Invited Talk at the 1st USENIX Conf. on File And Storage Technologies (FAST '02), 2002.

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

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

[13] Iometer team. Iometer: The I/O Performance Analysis Tool. http://www.iometer.org.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Symposium on Operating Systems Principles*, pages 52–65, 1997.

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

[16] K. Keeton and J. Wilkes. Automatic design of dependable data storage systems. In *Proc. of Workshop on Algorithms and Architectures for Self-managing Systems*, pages 7–12, San Diego, CA, June 2003.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[18] G. Lehey. The Vinum Volume Manager. In *Proc. of the FREENIX Track (FREENIX-99)*, pages 57–68, Berkeley, CA, June 6–11 1999. USENIX Association.

[19] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Impl. (OSDI96)*, Oct. 28–31 1996.

[20] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[21] W. Oney. Programming the Microsoft Windows Driver Model, Second Edition. http://www.microsoft.com/mspress/books/6262.asp.

[22] D. Patterson. The UC Berkeley ISTORE Project: bringing availability, maintainability, and evolutionary growth to storage-based clusters. http://roc.cs.berkeley.edu, January 2000.

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

[24] 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 Conference*, pages 173–184, June 2001.

[25] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proc. of USENIX 2001 Technical Conference*, June 2001.

[26] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A Framework for Protocol Composition in Horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[27] Veritas. Storage Foundation(TM). http://www.veritas.com/ Products/ www?c=product&refId=203.

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

[29] J. Wilkes. Traveling to rome: Qos specifications for automated storage system m anagement. In *Proc. of the Int. Workshop on QoS (IWQoS'2001). Karlsruhe, Germany*, June 2001.

[30] G. C. S. 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] W.V. Courtright II and G.A. Gibson and M. Holland andJ. Zelenka. RAIDframe: Rapid Prototyping for Disk Arrays. In *Proc. of the 1996 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, volume 1:24, pages 268–269, May 1996.

[32] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the 2000 USENIX Annual Technical Conference*, pages 55–70. USENIX Association, June 18–23 2000.