# Violin: A Framework for Extensible Block-level Storage

**Michail Flouris**

**Dept. of Computer Science, University of Toronto, Canada**

flouris@cs.toronto.edu

**Angelos Bilas**

**ICS-FORTH & University of Crete, Greece**

bilas@ics.forth.gr

# Large-scale Storage

- Large-scale storage systems in a data center
- Driven by
  - Need for massive amounts of scalable storage
  - Consolidation potential for lower costs
- Challenges in scalable storage
  - Scalability and performance with commodity components
  - Reduction of management cost
  - Wide-area storage sharing

# Reducing Costs With…

- Emerging scalable, low-cost hardware
  - Commodity clusters / grids (x86 with Linux / BSD)
  - Commodity interconnects and standard protocols
    - (PCI-X/Express/AS, SATA, SCSI, iSCSI, GigE)
- Storage virtualization software that
  - Offers diverse storage views for different applications
  - Automates storage management functions
  - Supports monitoring
  - Exhibits scalability and low overhead
- We want to improve virtualization at the block-level

# Block-level Virtualization

- “Virtualization” has two meanings
- Notion 1: Indirection
  - Mapping between physical and logical resources
  - Facilitates resource management
- Notion 2: Sharing
  - Hides system behind abstractions for sharing
- Our goal
  - Provide block-level virtualization mechanisms to improve indirection and resource management
- Why block-level ?
  - Transparency, Performance, Flexibility

# Issue with existing virtualization

- Current software has "configuration flexibility"
  - Use of a small set of predefined modules (RAID levels, volume management)
  - Module combination in arbitrary manner
- But missing "Functional Flexibility"
  - Ability to extend the system with new functionality
  - Extensions implemented by modules loaded on-demand
  - Not compromising configuration flexibility

    (New extension modules are combined with old ones)
  - Add management, performance, reliability-related features (e.g. encryption, versioning, migration, compression)

# Why Functional Flexibility?

- Reduce implementation complexity
  - Combine simple modules to build complex features
- Customizing system to user’s/application’s needs
  - Adaptivity to existing or new applications
- Incremental system evolution
  - Add new functionality before or after deployment
  - Optimize or upgrade components
- Prototyping and evaluation of new ideas
- Not compromising configuration flexibility
- Creating extensions should be easy
  - Developed by vendors, users, or storage administrators

# Our Goals

- Designing a system with automated storage management without extensions, does not work
- We intend to add desirable management, performance, reliability-related features, in an incremental evolution fashion
- Violin provides the mechanisms to achieve this
- Management automation
  - Will evolve over time
  - Will include an initial configuration phase and continuous monitoring and dynamic reconfiguration afterwards
- System will be able adapt to new applications

# Related Work

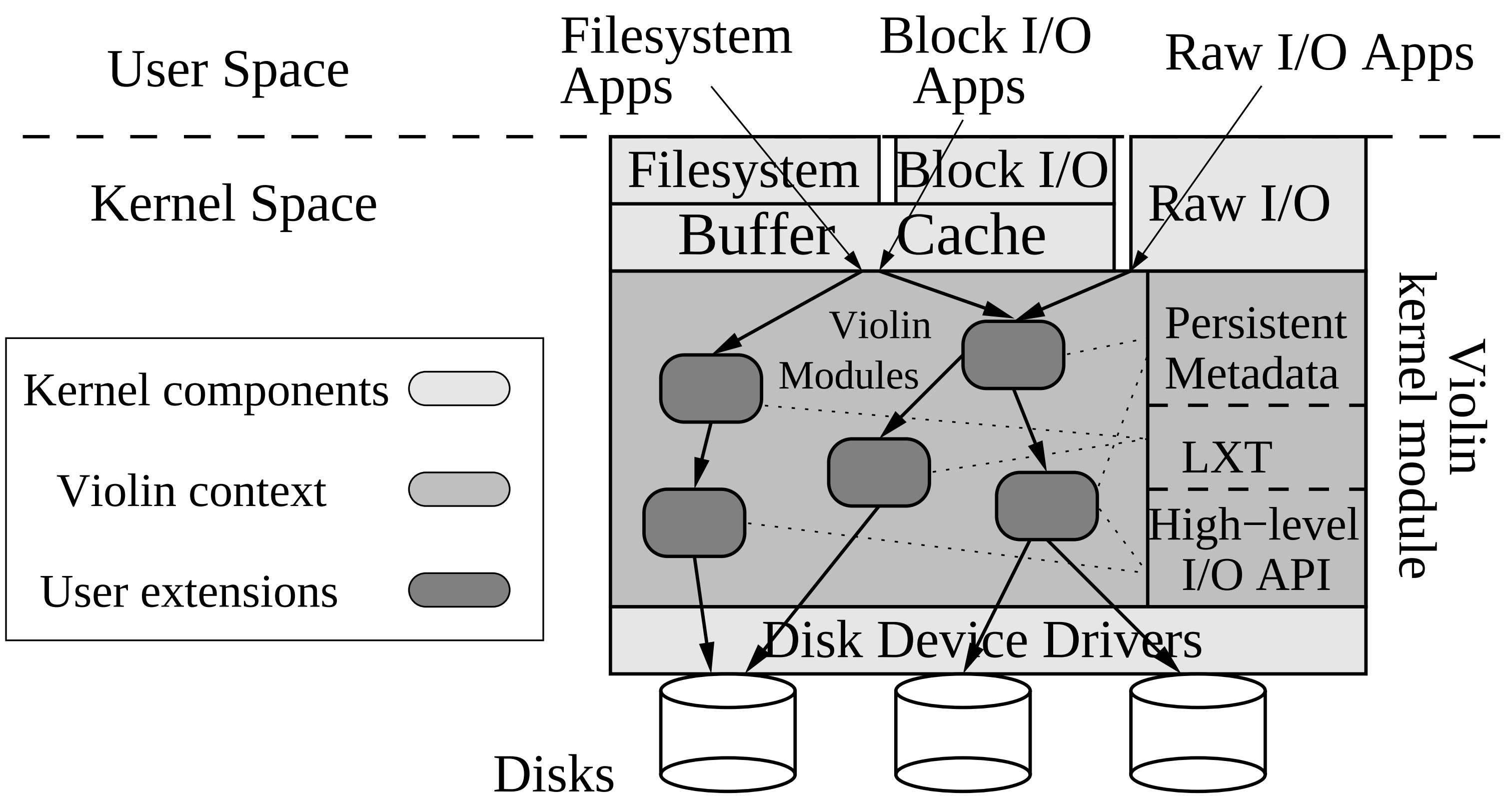
- Extensible Filesystems
  - Ficus, FiST. Not at block-level, complementary approach.
- Extensible Network Protocols
  - Click Router, X-kernel, Horus
  - Similar layer stacking for network protocol extensions
  - But: basic differences between network and storage
- Block-level virtualization software
  - OSS Volume Managers: Linux MD, LVM, EVMS, GEOM
  - Numerous Commercial Solutions
  - Provide only configuration flexibility

# Outline

- ✓ Motivation
- Violin Design
- Implementation
- Evaluation
- Conclusions

# Violin

- An extensible block-level hierarchy over physical devices

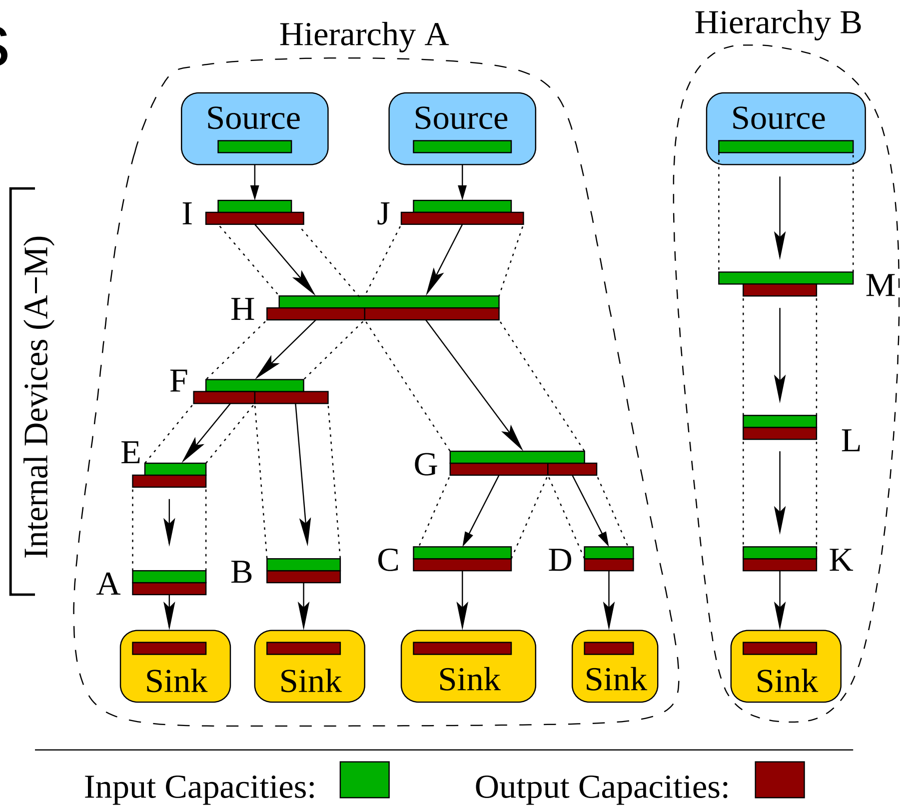


# Violin Design

- Goals
  - Easy to develop new extensions
  - Easy to combine them in I/O hierarchy
  - Low overhead
- Violin achieves this by providing
  1. Convenient semantics and mappings
  2. Simple control of the I/O request path
  3. Persistent metadata support

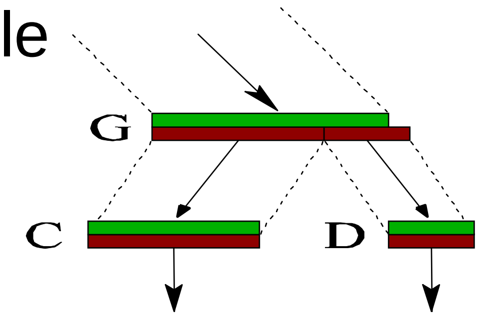
# Virtualization Hierarchies

- Device graph maps I/O sources to I/O sinks
  - I/O requests pass through devices (layers) in graph
- Nodes are virtual devices
- Edges are mappings
- Hierarchies: connected device sub-graphs, or independent I/O stacks
- Graph is DAG
  - Directed acyclic graph

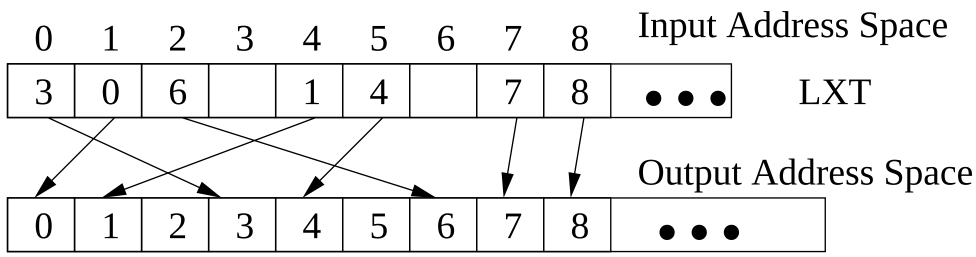


# Virtual Devices

- A virtual device is driven by an extension module
  - Device/Layer is runtime instance of module
  - Sees input, output address spaces and one or more output devices
  - Maps arbitrarily blocks between devices
  - Transforms data between input and output, vice versa




- Some modules need logical translation table (LXT)
  - A type of logical device metadata



# Control of I/O Requests

- Violin API allows layers simple control on requests passing though them
- Layer can initiate, forward, complete or terminate I/O requests using simple tags or calls
- Initiating I/O: Useful for multiplying I/O flows
  - Layers initiate asynchronous I/O requests using callback handlers, executed on completion
- Forward I/O: Send I/O request to a lower layer
- Complete I/O: Useful for caching layers
- Terminate I/O: Error-handling

# Tagged Request Control

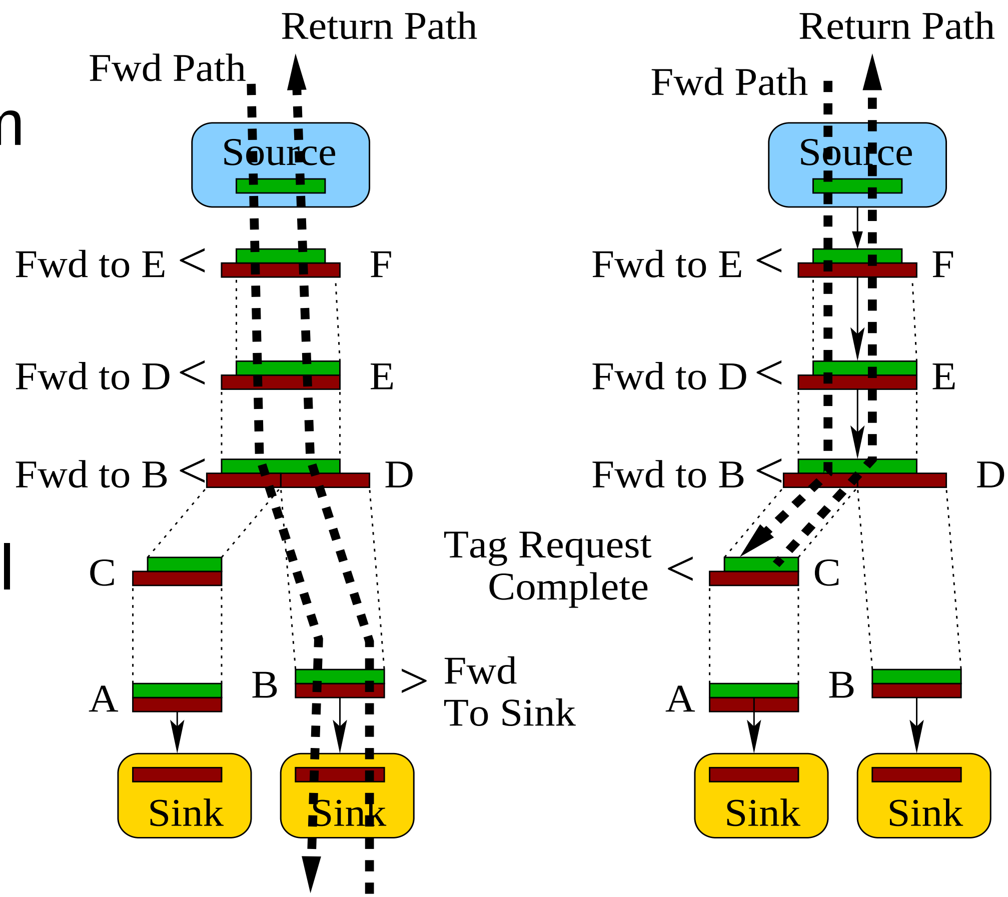
- Left Example:

  Request is forwarded through the hierarchy from source to sink

  (Layer order: F, E, D, B)

  and then back up
- Right Example:

  Request is forwarded until layer C, where it is tagged complete and returns upwards without reaching the sink



# Persistent Metadata

- Storage layers need persistent state
  - For superblocks, partition tables, block maps, etc.
- Violin offers *persistent objects* for layer metadata
  - Persistent Objects are memory-mapped storage blocks, accessed as generic memory objects
  - Automatically synchronized to stable storage periodically
  - Automatically loaded / unloaded during startup / shutdown
  - Layers need only allocate objects once
- Violin internal metadata are also persistent objects
  - Device graph and hierarchy info stored at superblocks

# Metadata Consistency

- Three levels of metadata consistency (weaker to stronger)

1. Lazy-updates
   - Synchronized overwriting older metadata periodically
   - Similar to non-journaling filesystems
2. Shadow-updates
   - Using two copies of all metadata (normal & shadow)
   - Synchronization overwrites first normal metadata, then shadow
   - Guarantees module metadata consistency
3. Atomic versioned-metadata consistency
   - Module metadata and application data are versioned
   - On failure the system rolls back to a consistent snapshot

- Violin currently supports levels 1 and 2

# Block Size and Memory Overhead

- Small block sizes can increase memory footprint of module metadata
  - When metadata proportional to total number of blocks
- Many OSes have small block sizes
  - Linux 2.4.x: 4KB block device size
- Modules need own independent block size to manage metadata in larger chunks
- Violin supports larger “internal” block size
  - Size set by modules
  - Independent from OS block size
  - Reduces memory overhead effectively

# Outline

- ✓ Motivation
- ✓ Violin Design
- Implementation
- Evaluation
- Conclusions

# Implementation

- Violin Core
  - Linux 2.4 loadable block device driver
  - Registers extension modules
  - Provides API & services to extension modules
- Violin Extension Modules
  - Loadable Linux kernel modules that bind to Core
  - Not device drivers themselves (much simpler)

# Example Modules

- RAID
  - RAID Levels 0, 1 and 5 with recovery.
- Aggregation (plain or striped volumes)
  - Volume Remapping (add, remove, move Volumes)
- Partitioning
  - Managing Partitions (create, delete, resize partitions)
- Versioning (Online Snapshots)
- Online Migration
- Data Fingerprinting (MD5)
- Encryption
  - Currently DES, 3DES and Blowfish algorithms

# Evaluation

- Ease of module development
- Configuration Flexibility
- Performance

# Evaluation: Ease Of Development

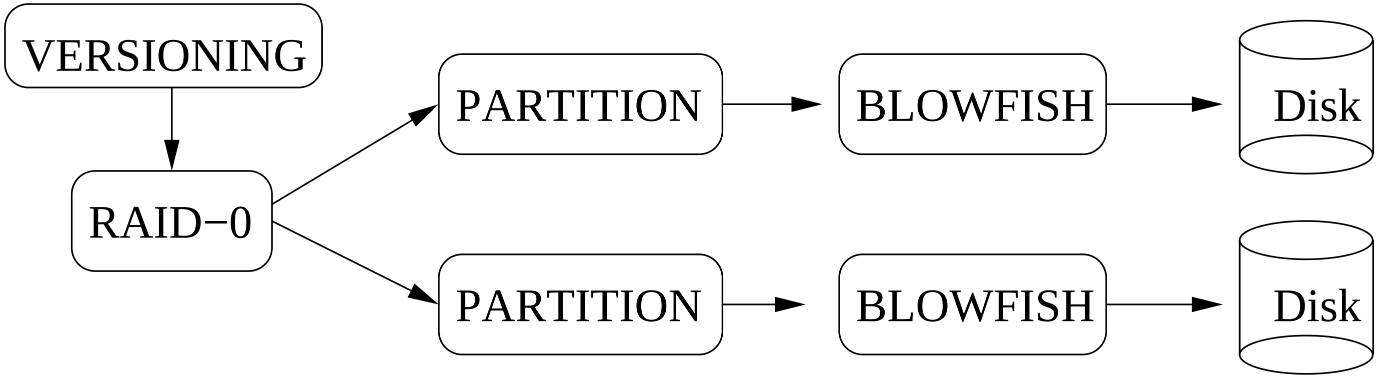
- Loose comparison of number of code lines
- Code lines reduced 2-6 times for similar functionality
- Little to reasonable effort for module development

**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 | – |

# Evaluation: Configuration Flexibility

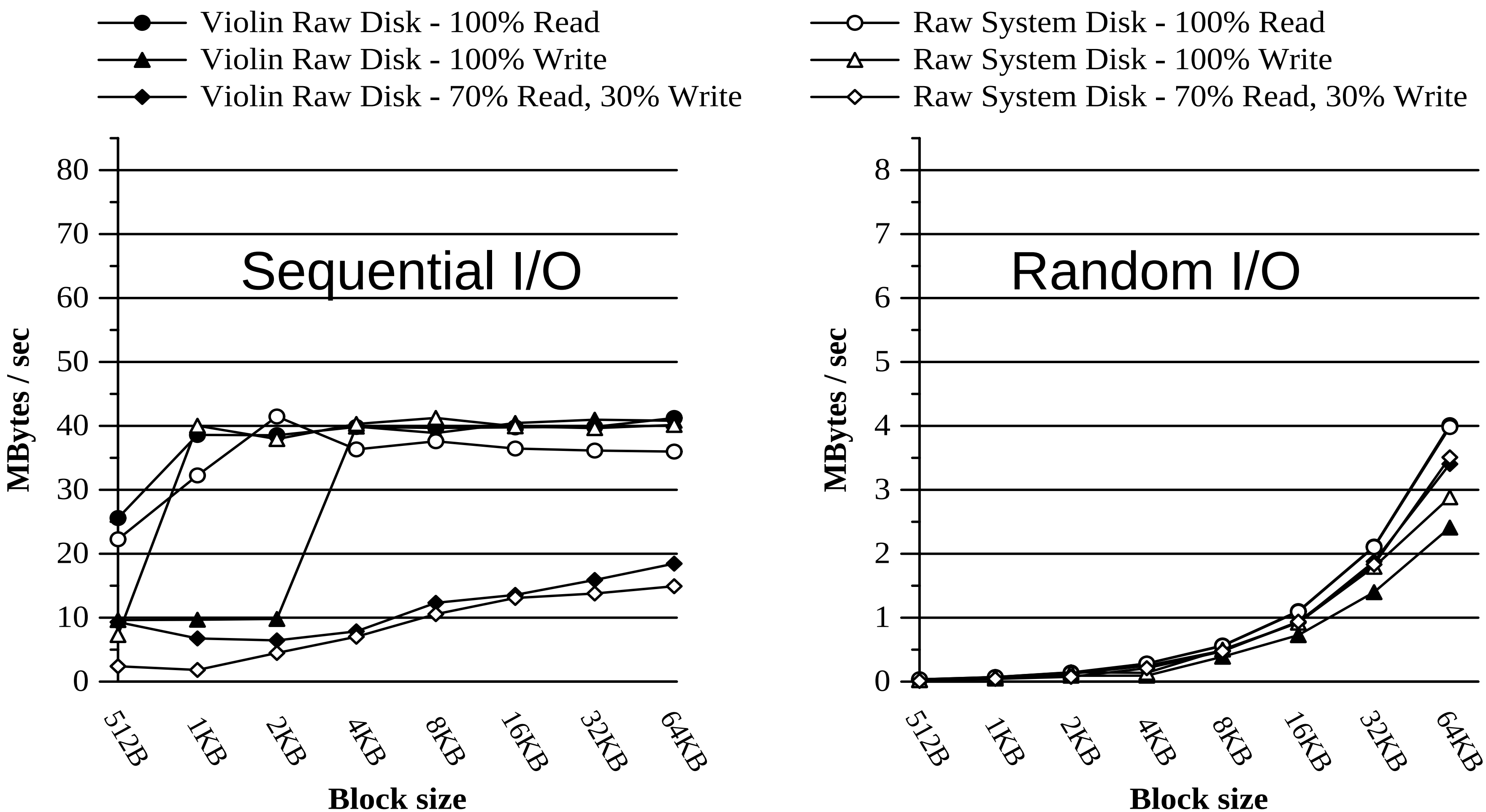
- Easily creating a hierarchy with complex functionality from implemented modules
- Violin allows arbitrary combinations of extension modules



# Evaluation: Performance

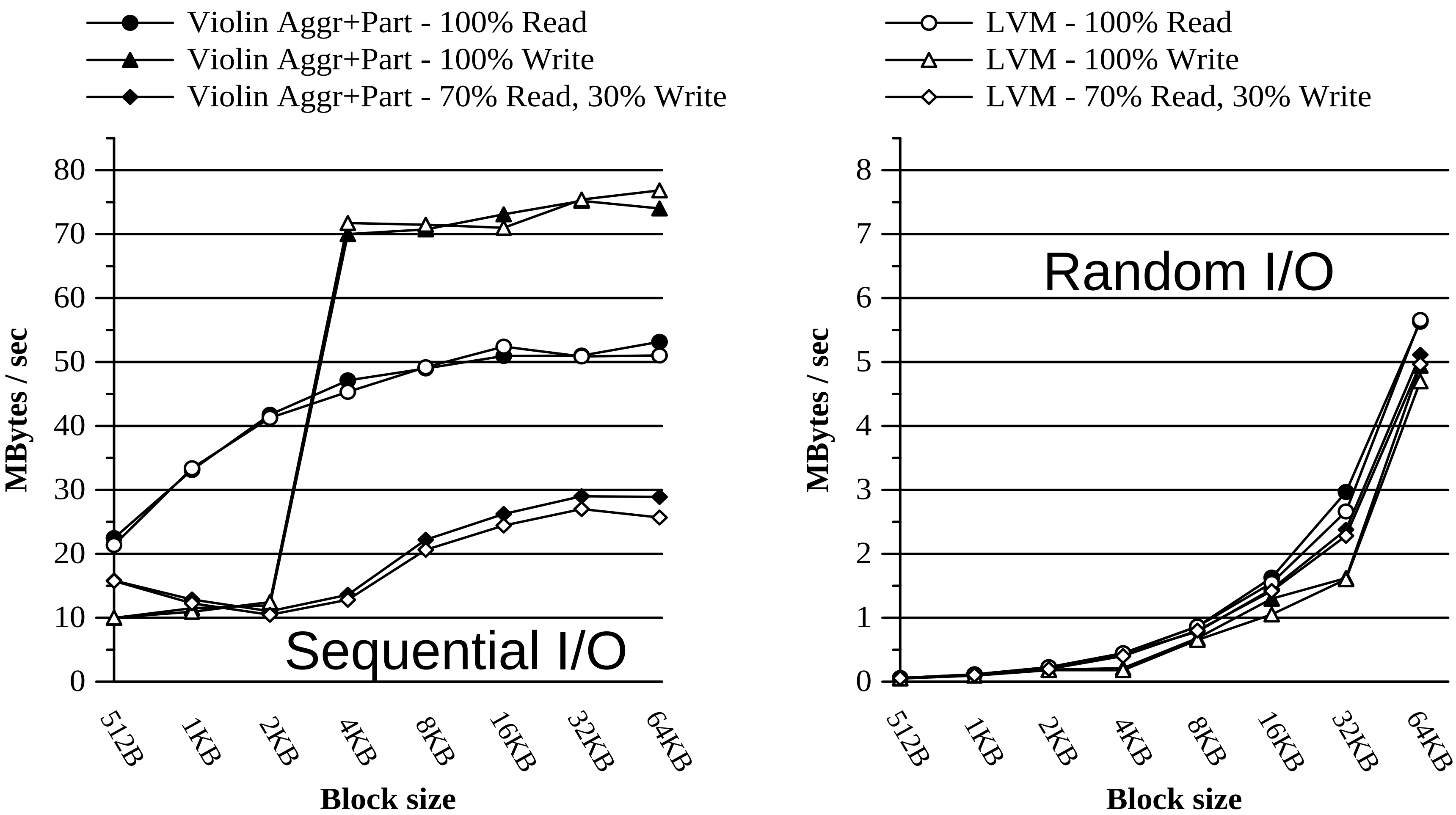
- Platform:
  - Dual Athlon-MP 2200+ PCs, 512MB RAM, GigE NIC, Western Digital 80GB IDE Disks
  - RedHat Linux 9.0 (Kernel 2.4.20-smp)
- Benchmarks
  - IOmeter (2004.07.30) for raw block I/O
  - Postmark for filesystem measurements
- Experiment Cases
  1. Pass-through: System Disk vs. Violin Pass-through Layer
  2. Vol. Manager: LVM vs. Violin Aggregate+Partition Layers
  3. RAID-0: Linux MD vs. Violin RAID
  4. RAID-1: Linux MD vs. Violin RAID

# Violin Pass-through vs. System Disk



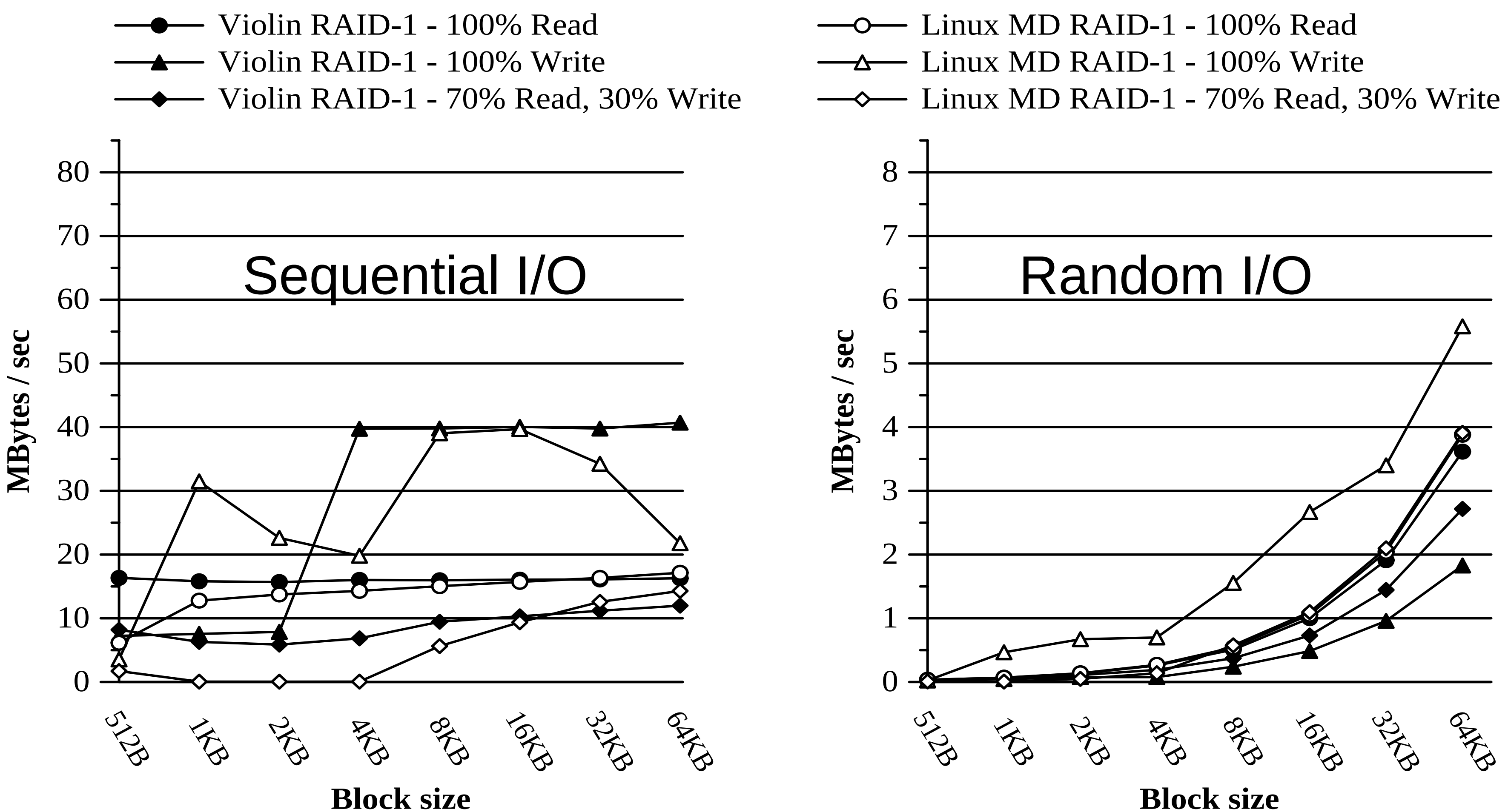

IOmeter throughput for Violin Raw Disk vs. System Raw Disk for One Disk

# Violin vs. LVM (2 Striped Disks)



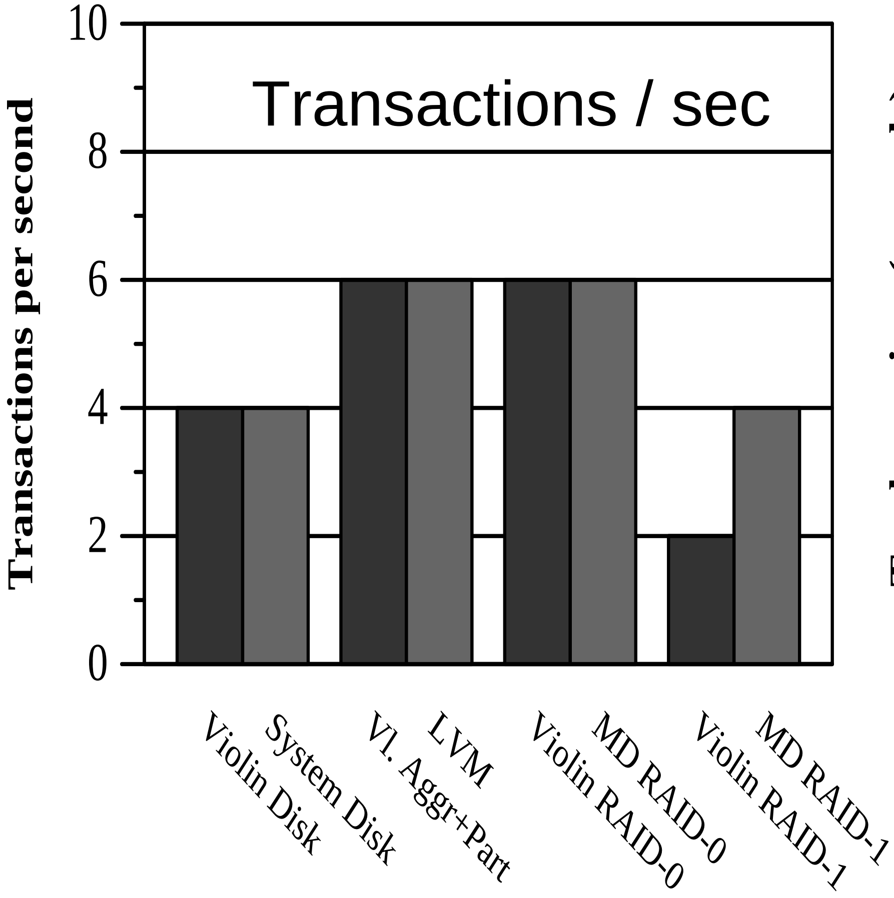

IOmeter throughput for Violin Aggregation+Partition vs. LVM for 2 Striped Disks (32K stripe)

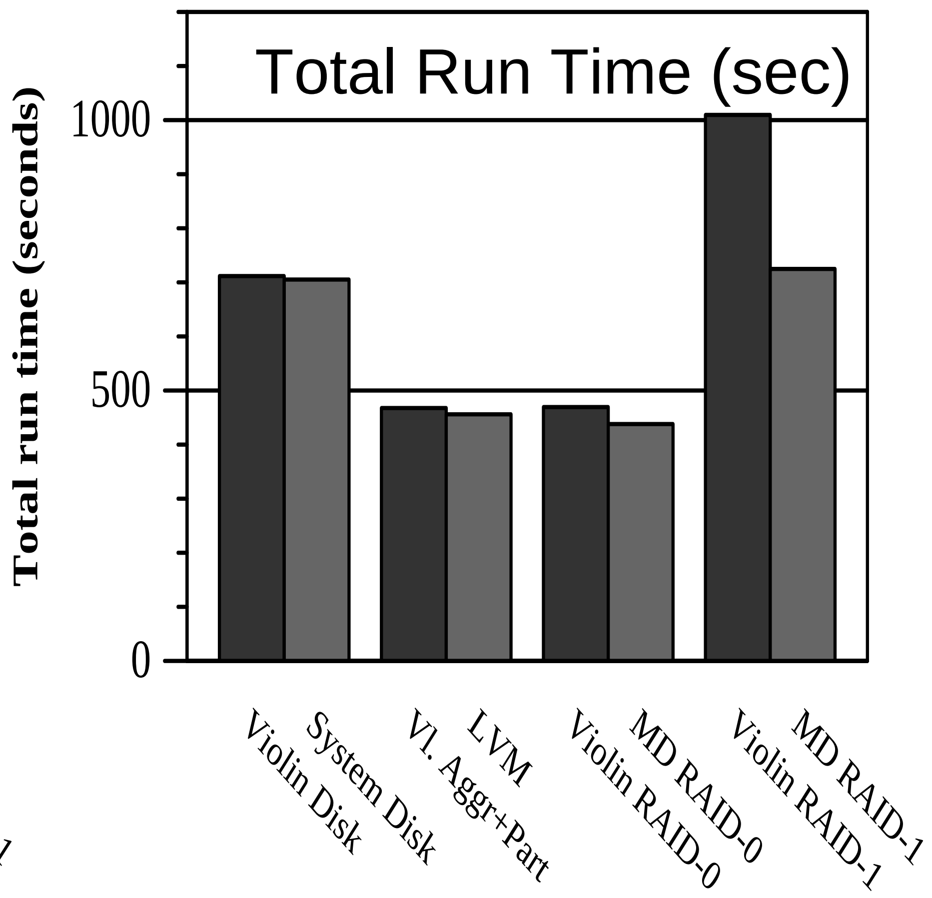
# Violin vs. MD (RAID-1 Mirroring, 2 Disks)



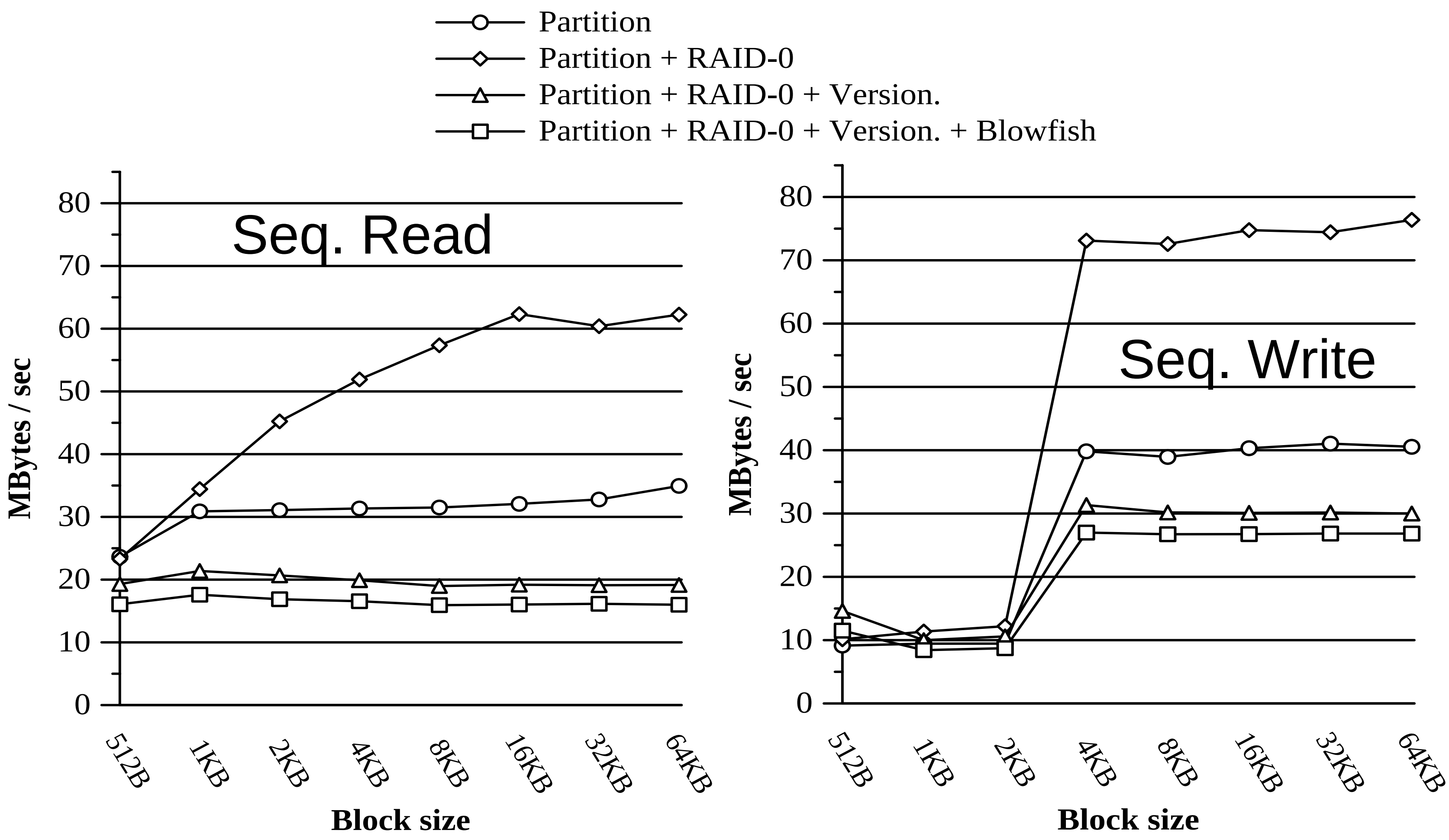

IOmeter throughput for RAID-1: Violin vs. Linux MD for 2 Disks

# Postmark Results over Ext2 Filesystem



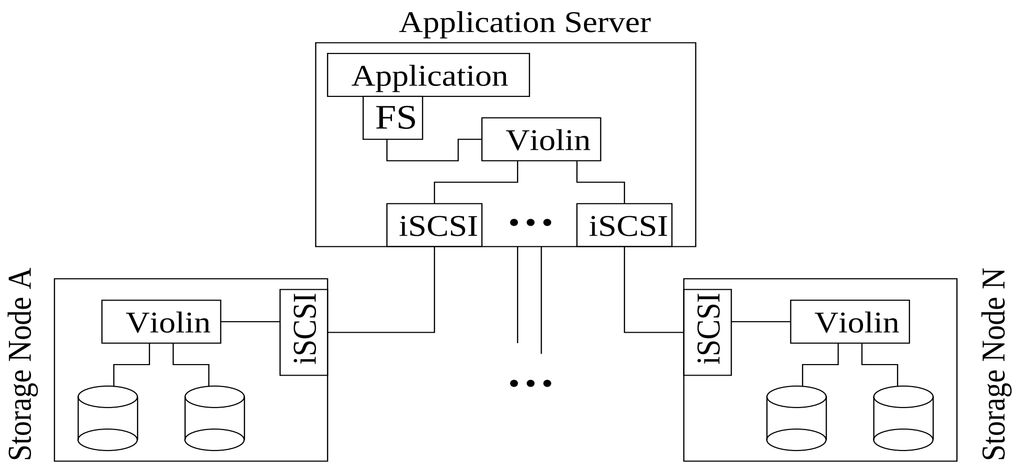
# Multiple Layer Performance

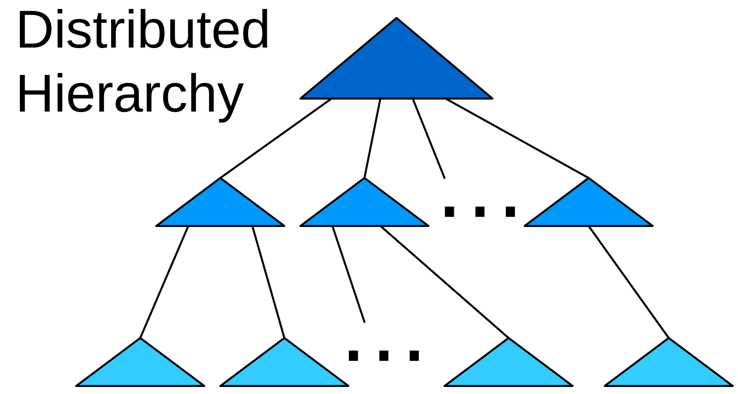



Workload: 100% Sequential Read

Workload: 100% Sequential Write

# Limitations and Future Work




- Violin does not fully support distributed hierarchies
  - No consistency for dynamically updated metadata
- Future work:
  - Supporting distributed hierarchies in a cluster
  - Automated hierarchy configuration and management, according to specified requirements and policies

# Conclusions

- Goal is to improve virtualization in storage cluster
- Propose Violin, an extensible I/O layer stack
- Violin's contributions are mechanisms for
  - Convenient virtualization semantics and mappings
  - Simple control of I/O requests from layers
  - Persistent metadata
- These mechanisms
  - Make it easy to write extensions
  - Make it easy to combine them
  - Exhibit low overhead (< 10% in our implementation)
- We believe that Violin's mechanisms are a step towards automated storage management

# Thank You.

# Questions ?

**“Violin: A Framework for Extensible Block-level Storage”,**

Michail Flouris and Angelos Bilas
flouris@cs.toronto.edu, bilas@ics.forth.gr

http://www.ics.forth.gr/carv/scalable