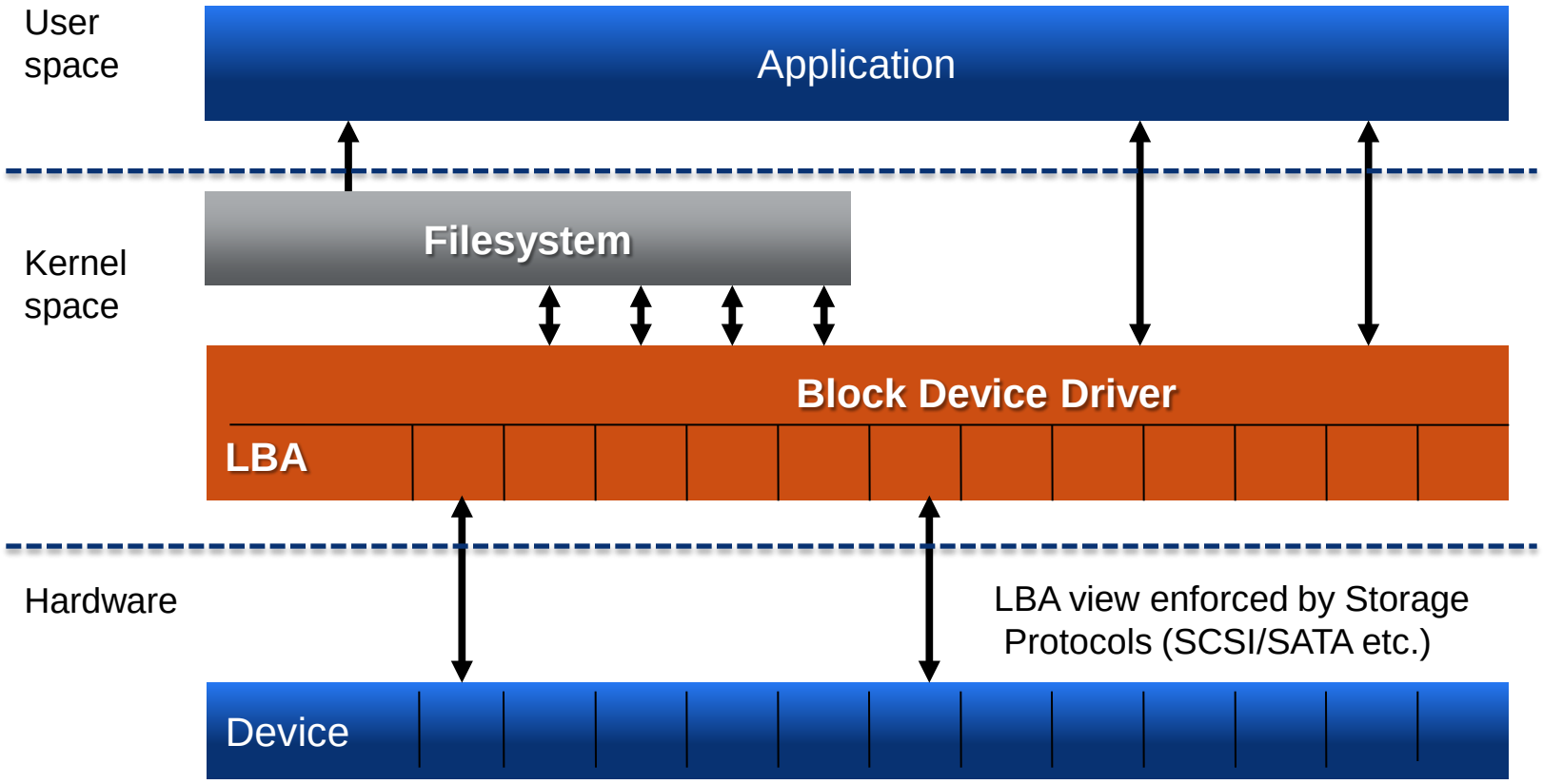
Leveraging the Flash Translation Layer for Application Acceleration

Ashish Batwara

Fusion-io

Traditional Storage Stack



Flash is Different From Disk

Area	Hard Disk Drives	Flash Devices
Logical to Physical Blocks	Nearly 1:1 Mapping	Remapped at every write
Read/Write Performance	Largely symmetrical	Heavily asymmetrical.
Sequential vs Random Performance	An order of magnitude difference	Minimal difference
Background operations	Rarely impact foreground	Regular occurrence. If unmanaged - can impact foreground
Wear out	Largely unlimited writes	Limited writes
IOPS	100s to 1000s	100Ks to Millions
Latency	10s ms	10s - 100s us
TRIM	Do not benefit	Improves performance

Flash Translation Layer 101

Input

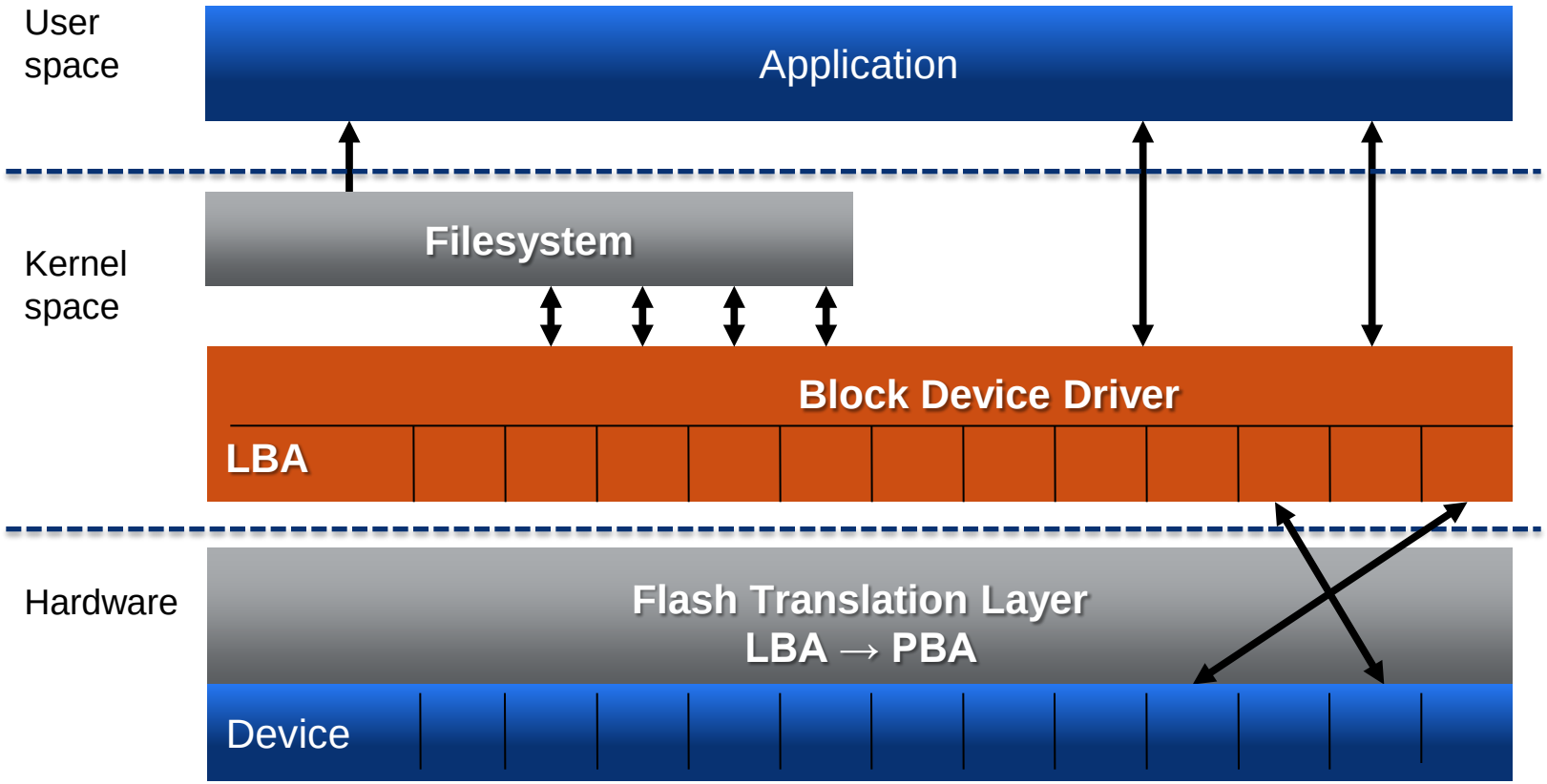
Logical Block Address (LBA)

Flash Translation Layer

Output

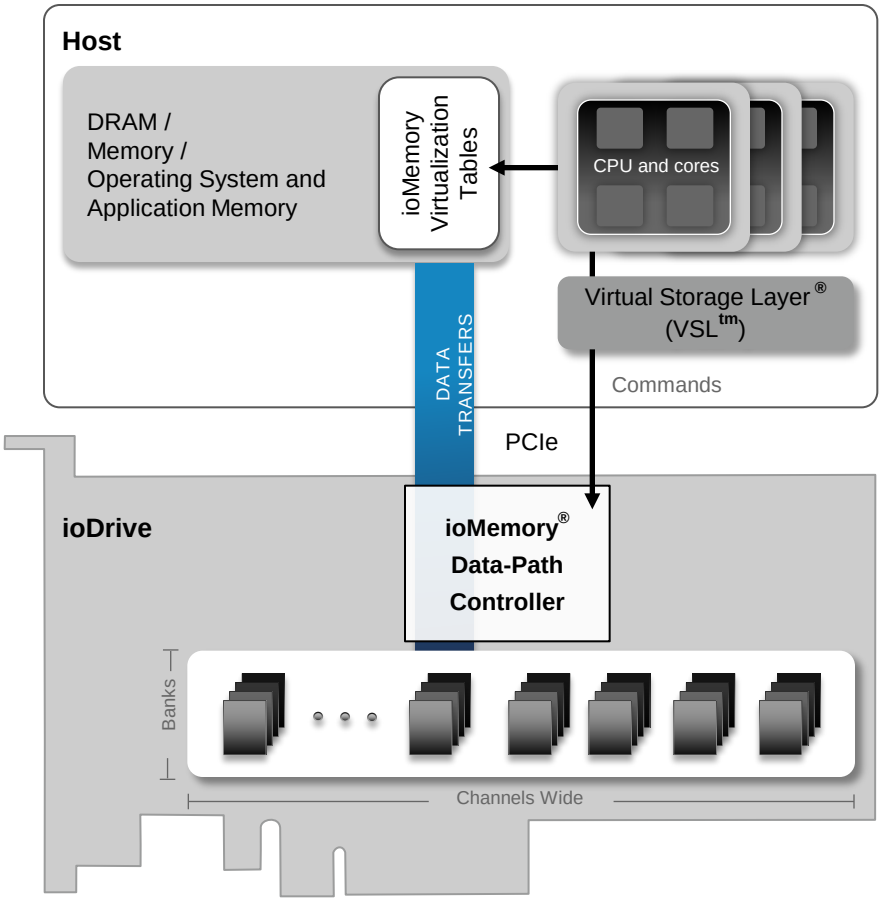
Commands to NAND flash

Flash in Traditional Storage Stacks



Virtual Storage Layer

- Fusion-io's host based FTL
  - Virtual Storage Layer (VSL)
- Cut-thru architecture – avoids traditional storage protocols
- Scales with multi-core
- Provide a large virtual address space
- HW/SW functional boundary defined as optimal for flash
- Traditional block access methods for compatibility
- New access methods, functionality and primitives natively supported by flash

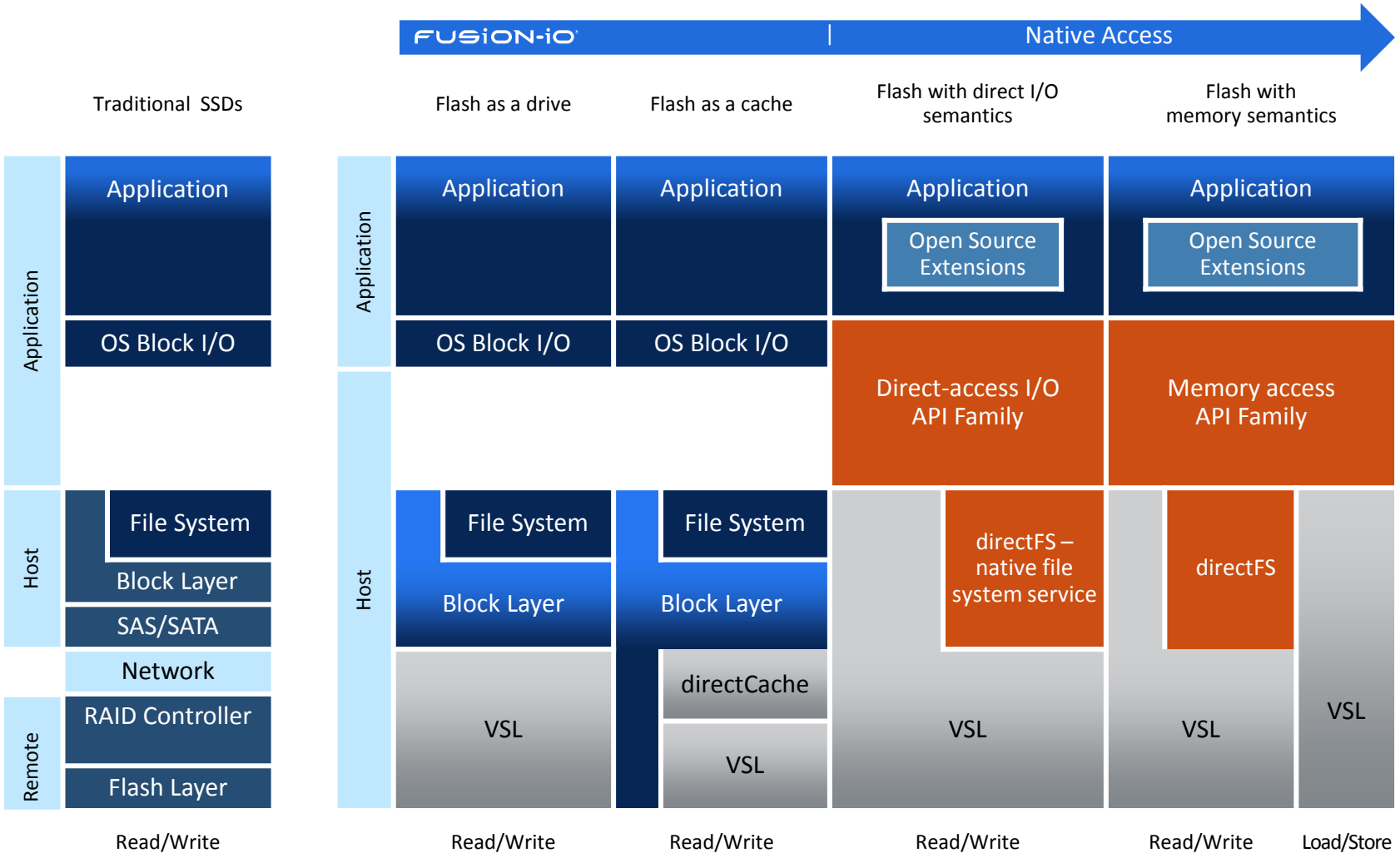


Call to Action

Flash Memory summit 2011

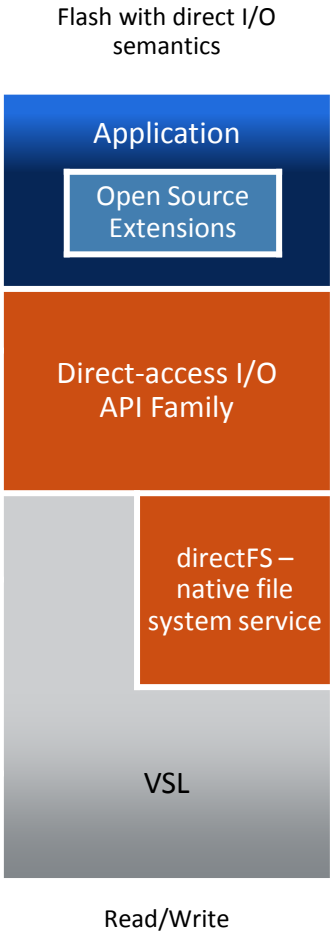
- Host-based FTLs integrate and scale with applications, examples include
  - File Systems
  - Caching
  - Databases
- Power of FTL no longer restricted by traditional block interfaces
- Opportunity for performance, simplicity and reliability improvements

Flash Memory Evolution

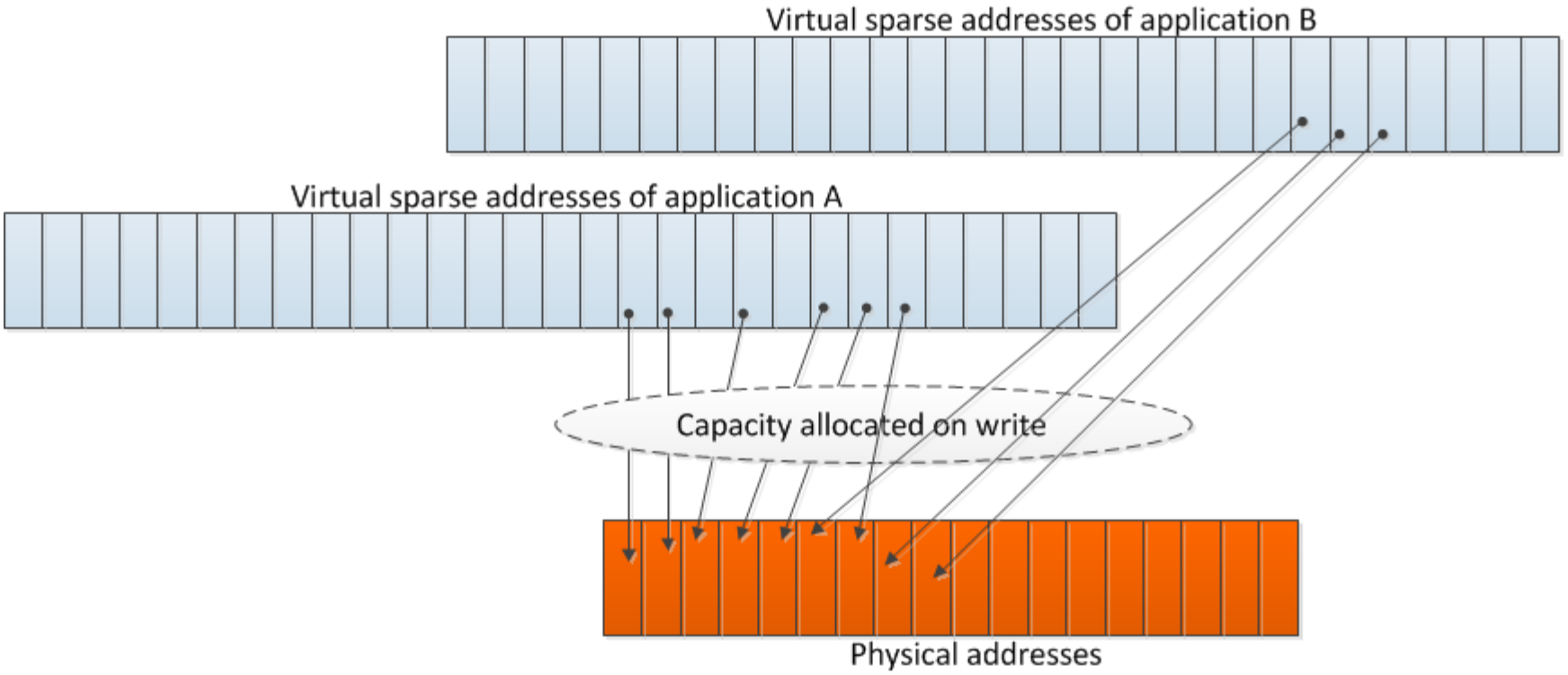


Direct-access I/O API family

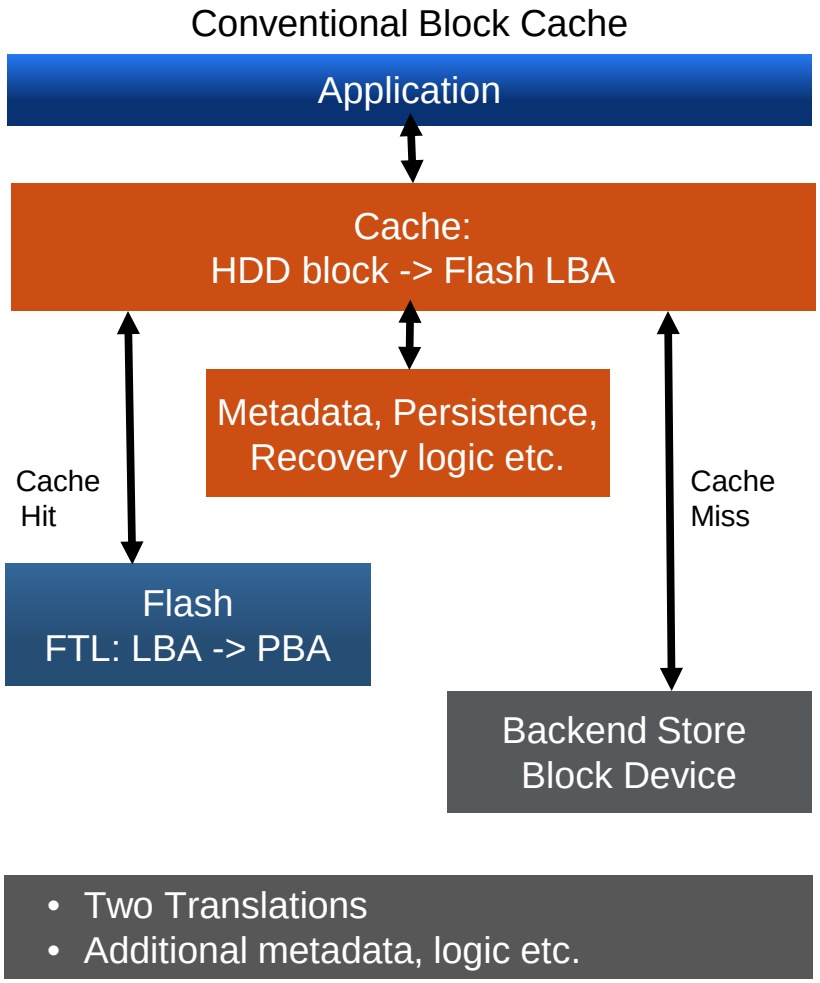
- direct I/O Primitives
  - Sparse Addressing
  - Atomic multi-block Operations
    - Write
    - PTRIM
  - Exists, Range Exists
  - Conditional Write
  - Range Read
- direct Key-Value Store
  - NVM optimized with transactional semantics



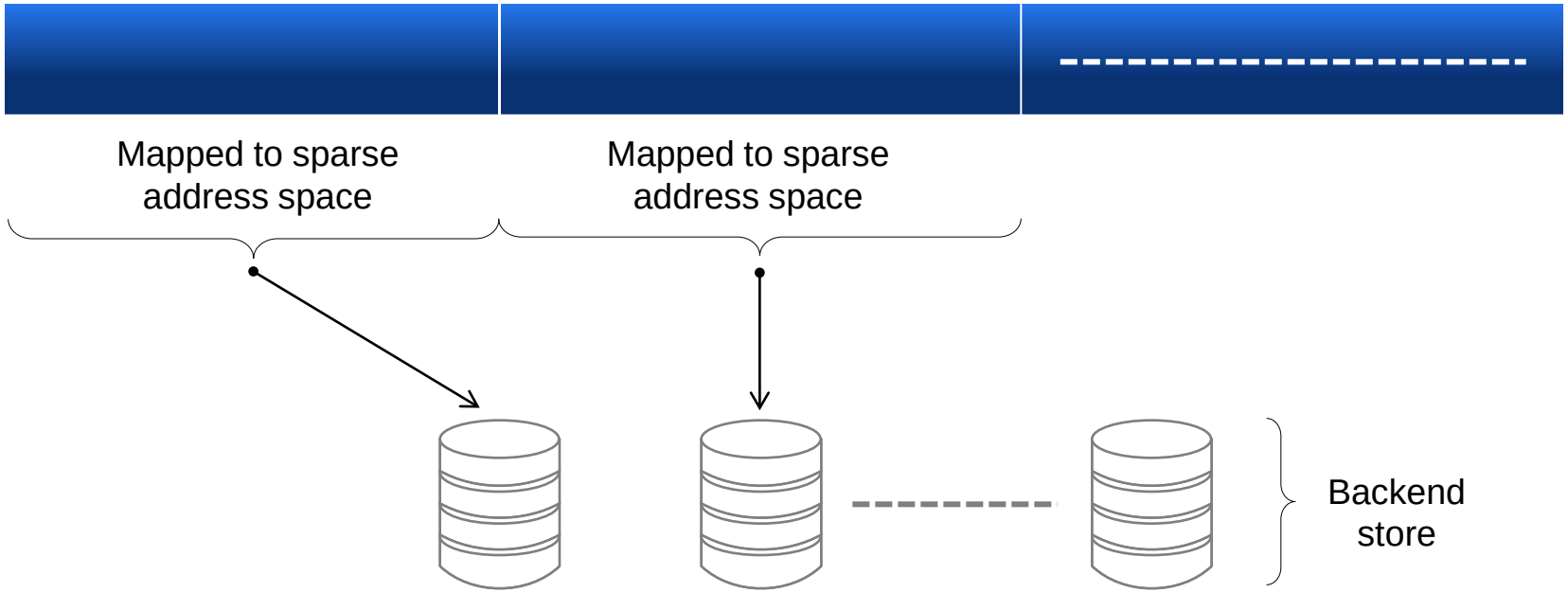
Sparse Addressing



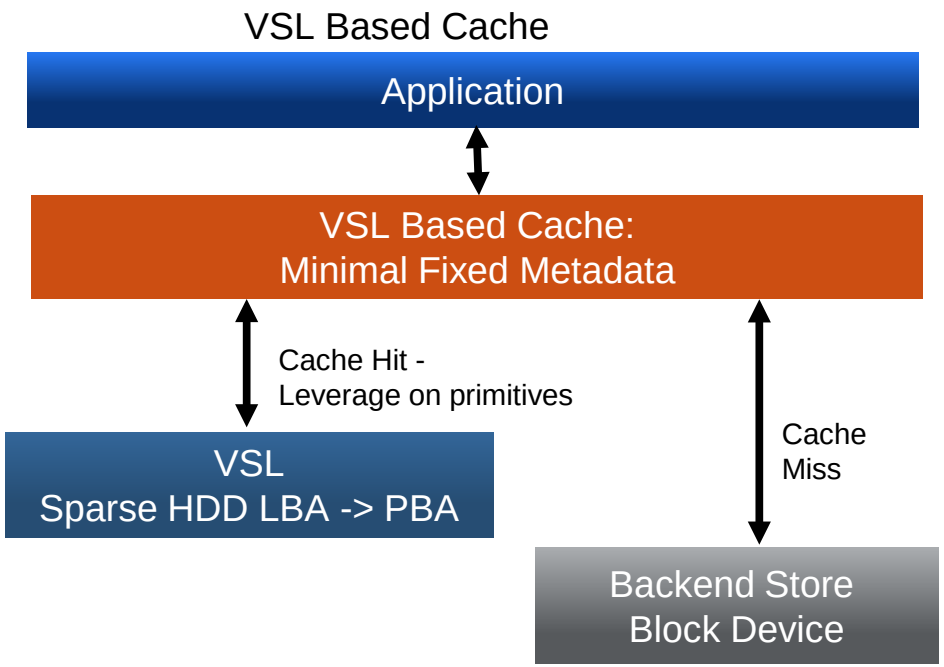
Excess work by conventional cache



Sparse address mapping

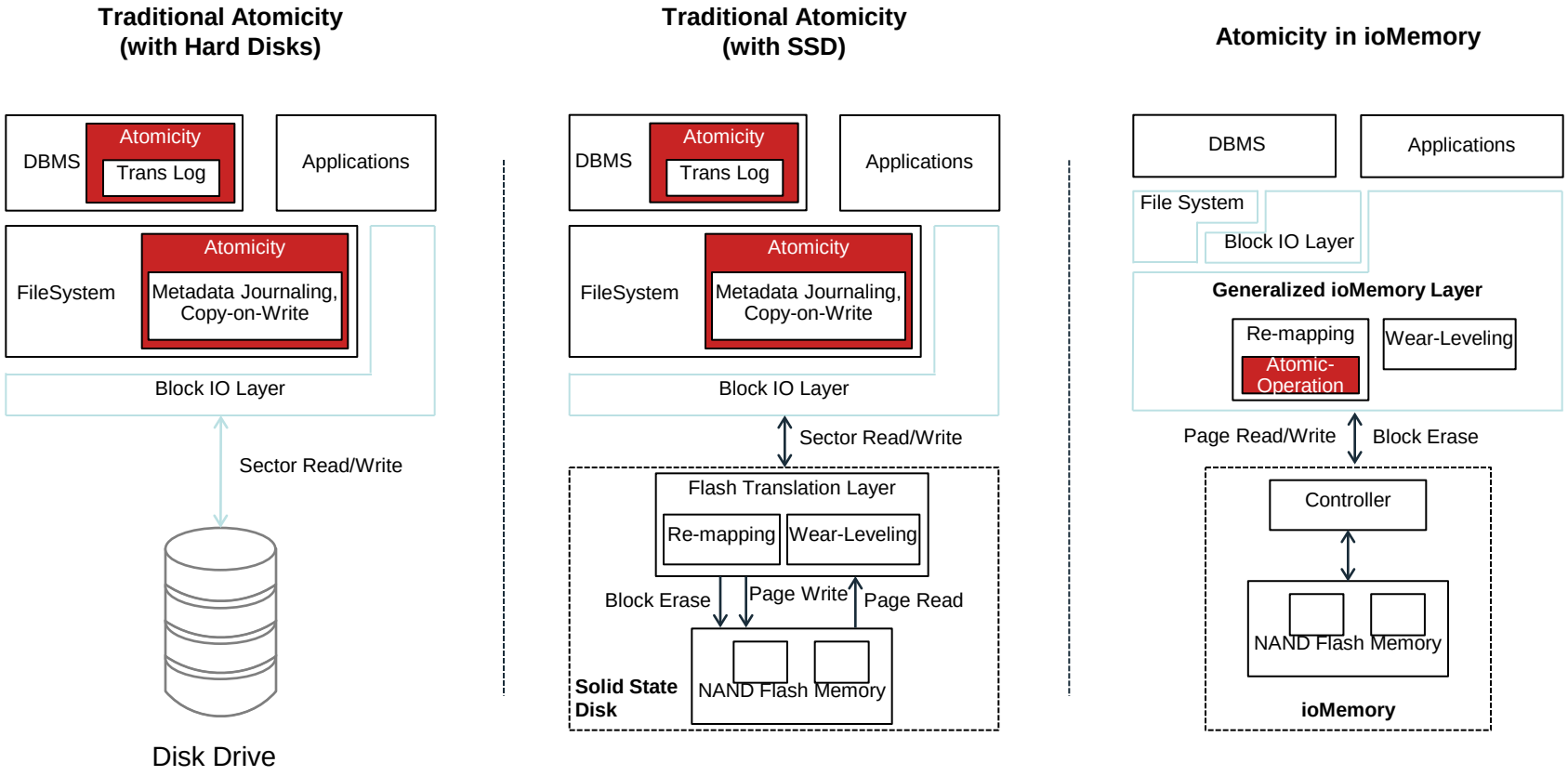


VSL based cache



- Fewer Translations
- Minimal additional metadata, logic etc.

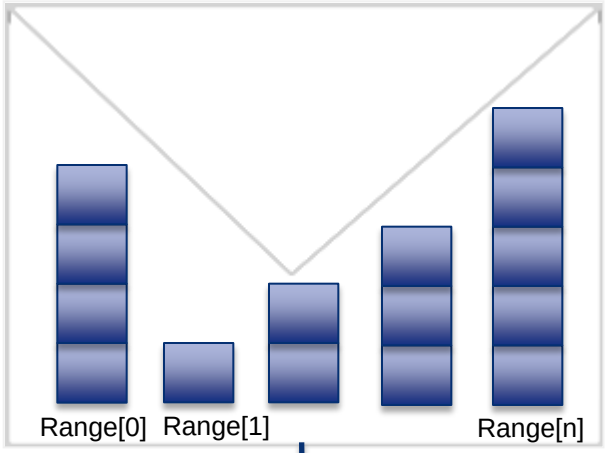
Direct I/O - Atomic Operations



Transactional Block Interface

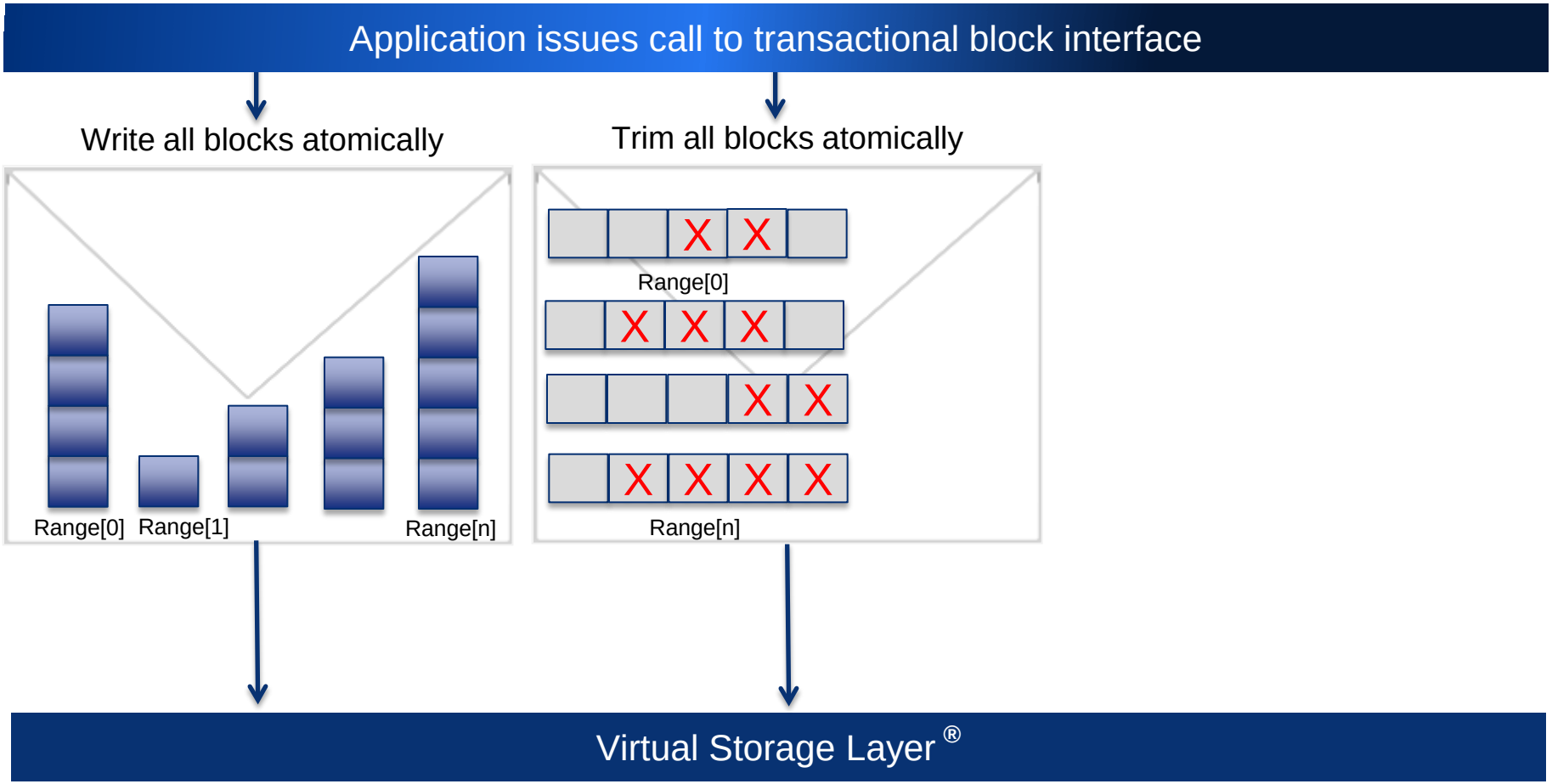
Application issues call to transactional block interface

Write all blocks atomically

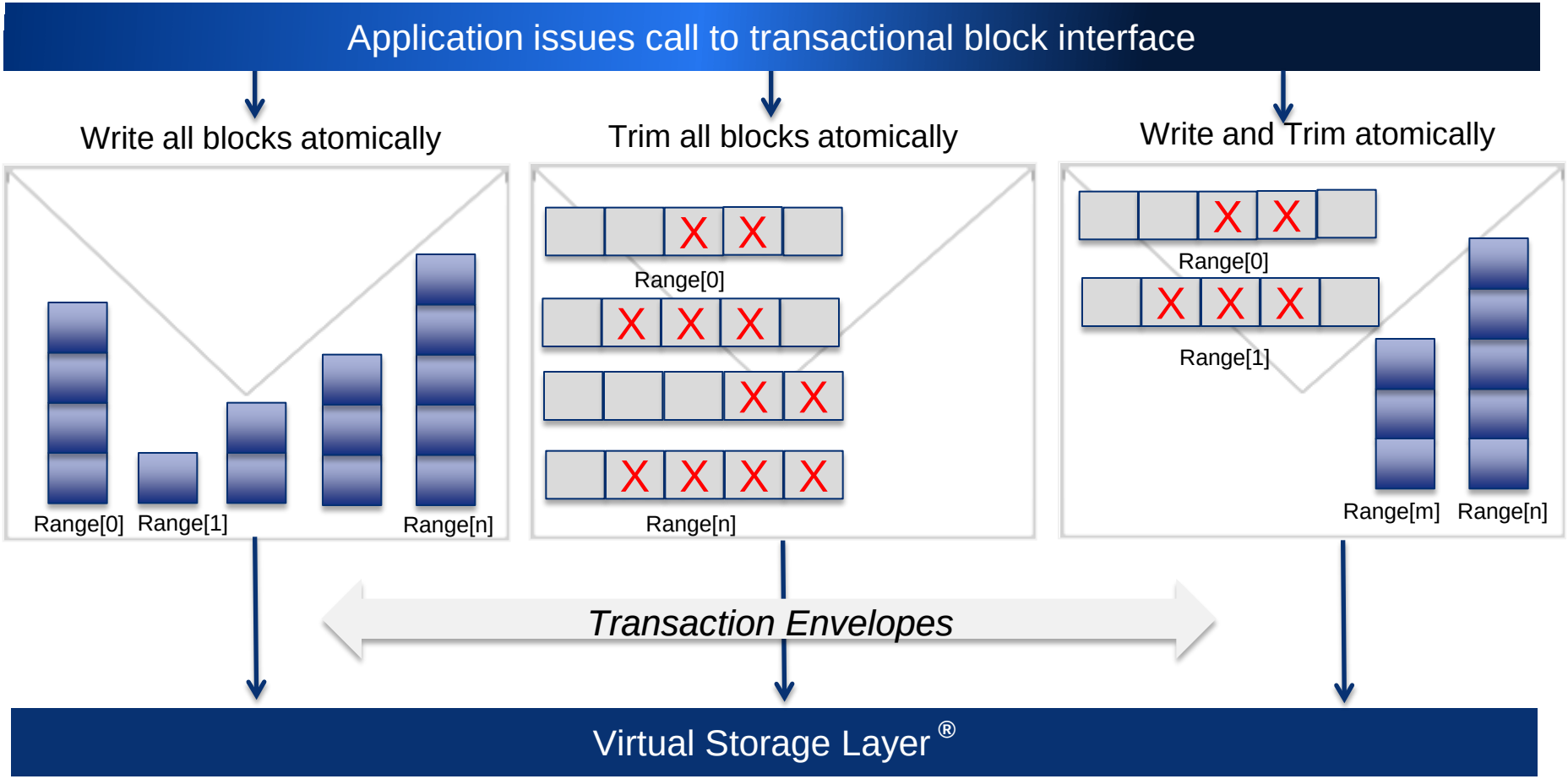


Virtual Storage Layer ®

Transactional Block Interface



Transactional Block Interface



Sysbench Performance With Atomic-Write

MySQL extension for Atomic-Write

43%

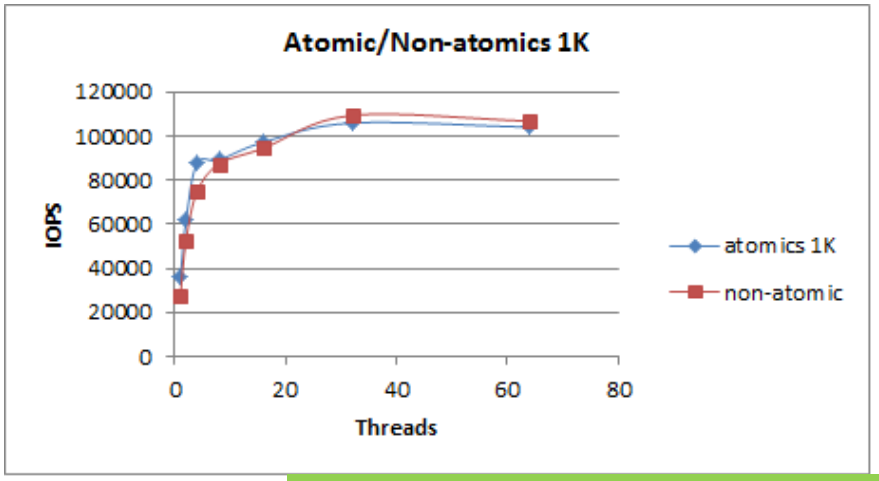
TRANSACTIONS/SEC INCREASE

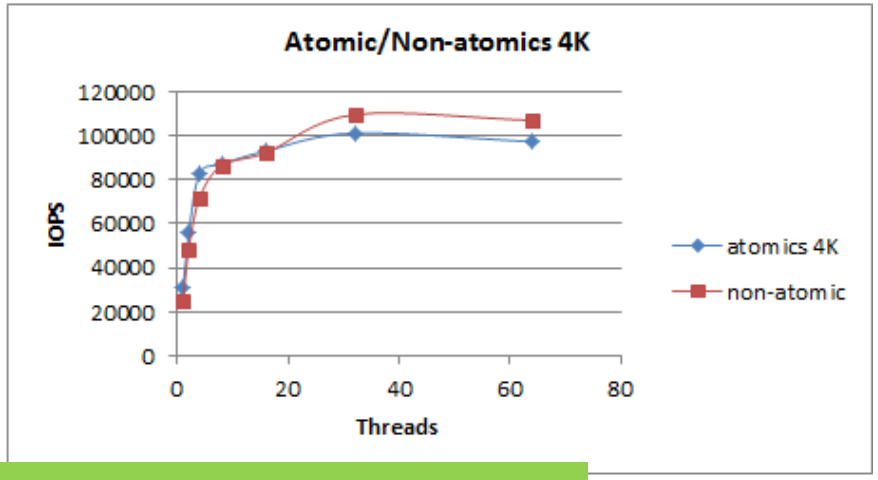
2x

ENDURANCE INCREASE

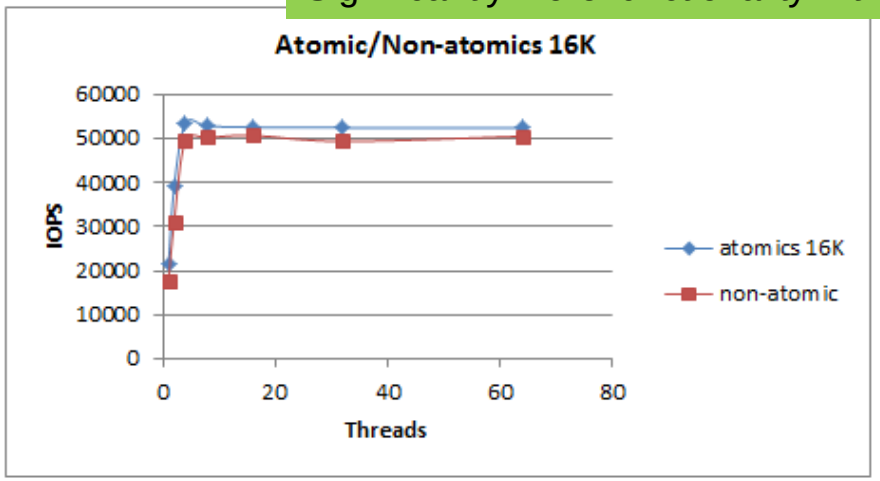
- Processor: Xeon X5472 @ 3.00GHz
- DRAM: 16GB DDR3 4x4GB DIMMs
- OS: Fedora 14 – Linux kernel 2.6.35
- Sysbench config: 1 million inserts in 8, 2-million-entry tables, using 16 threads

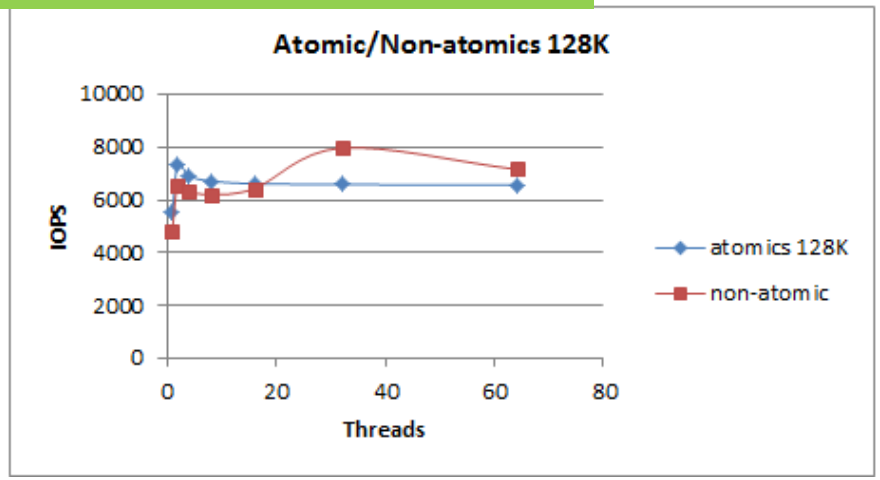
Native raw performance comparison





Significantly more functionality with NO additional performance impact





1U HP blade server with 16 GB RAM, 8 CPU cores - Intel(R) Xeon(R) CPU X5472 @ 3.00GHz with single 1.2 TB ioDrive2 mono

Direct I/O Primitives – Persistent Trim and Exists

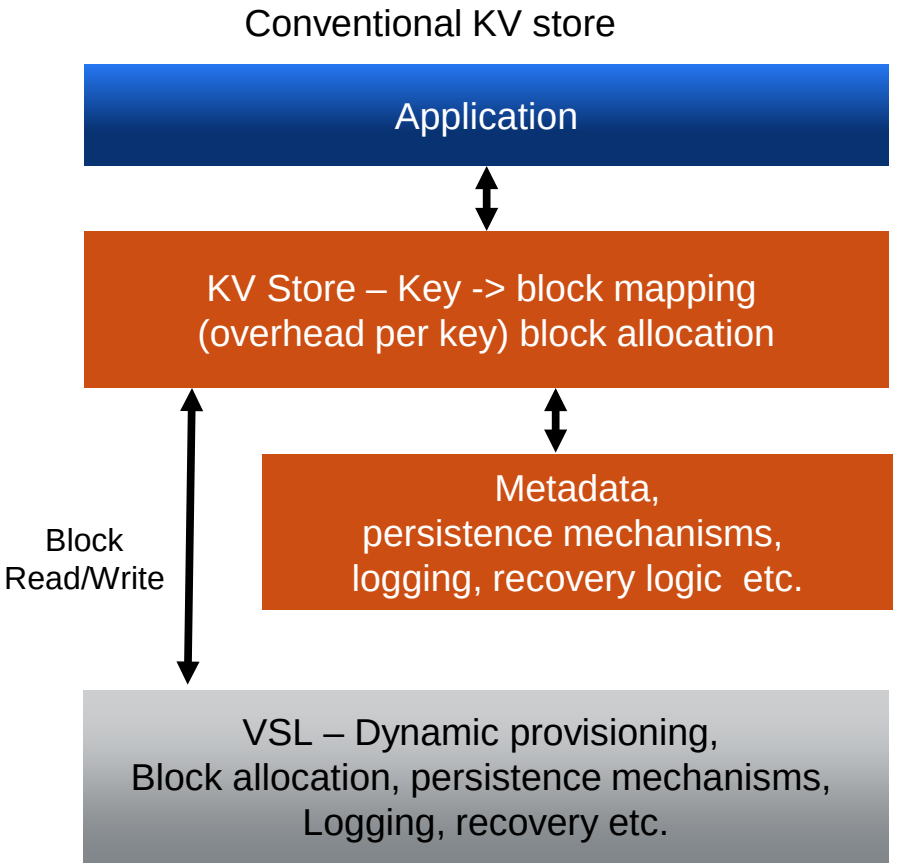
Persistent TRIM (Virtual Address)

- Has all the positive properties of TRIM
  - Improves wear leveling
  - Improves write performance
- However – well defined with respect to failures
  - Deterministic return of zeros for read
  - Survives power failures (transactional)

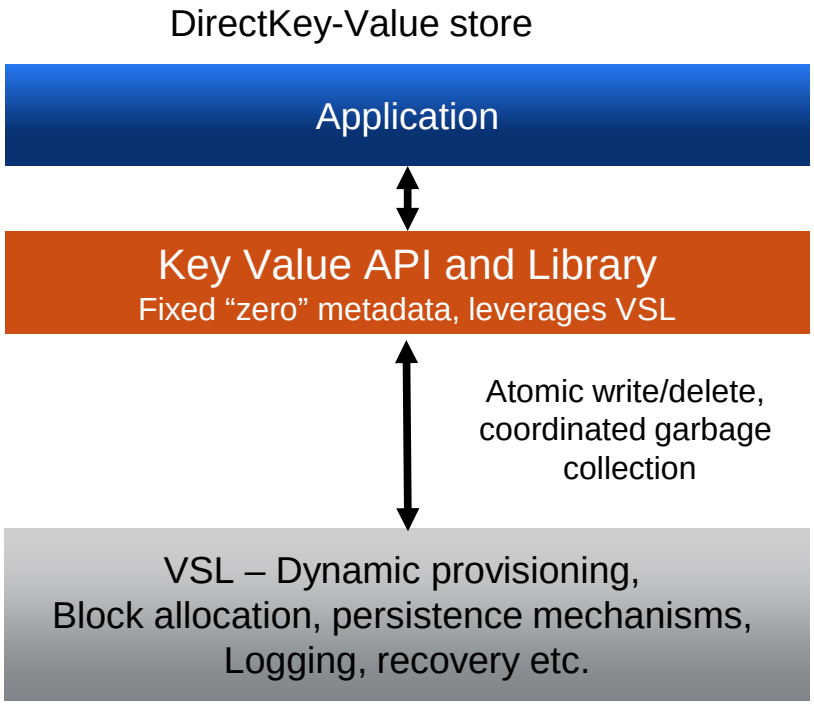
EXISTS (Virtual Address)

- Query the existence of a particular element
- Enables sparse stores with well defined “presence” semantics

Conventional Key-Value Store



VSL based Direct Key-Value Store

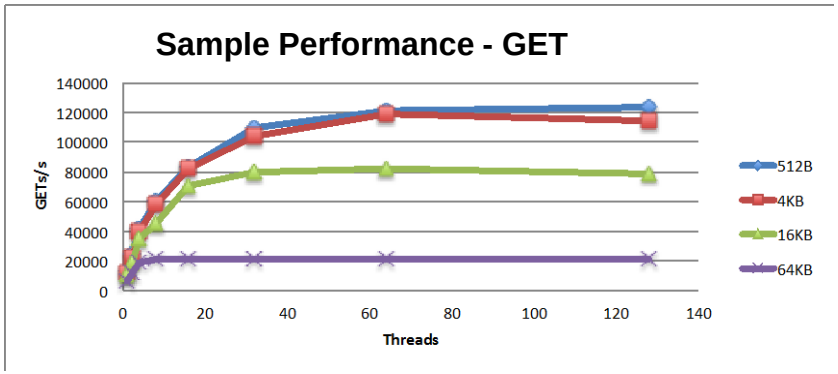


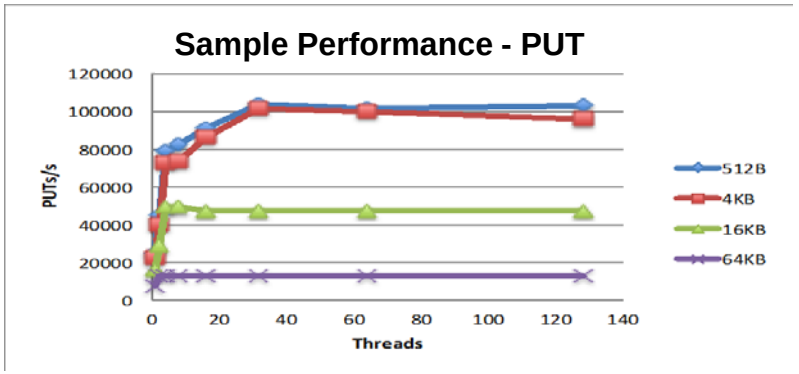
directKey-Value Store API Overview

- **KV Store Administration**
  - kv_create()
  - kv_pool_create()
  - kv_pool_delete()
  - kv_open()
  - kv_get_store_info()
  - kv_get_key_info()
  - kv_register_notification_handler()
  - kv_close()
  - kv_destroy()
  - kv_get_pool_info()

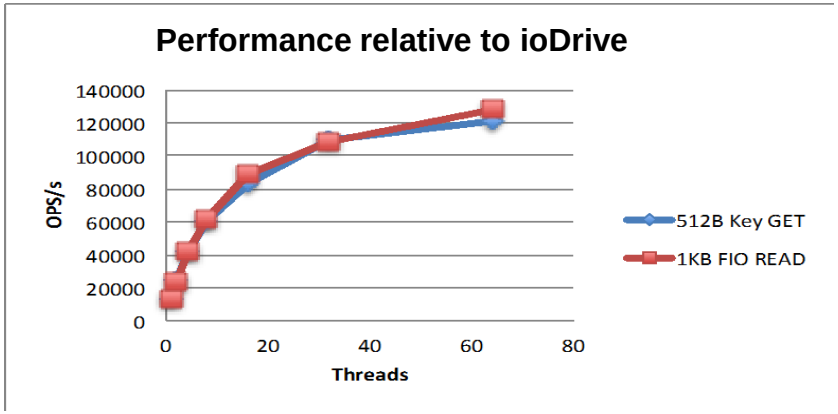
- **KV Store Data Operations**
  - kv_put()
  - kv_batch_put()
  - kv_get()
  - kv_batch_get()
  - kv_delete()
  - kv_delete_all()
  - kv_batch_delete()
  - kv_begin()
  - kv_next()
  - kv_get_current()
  - kv_exists()

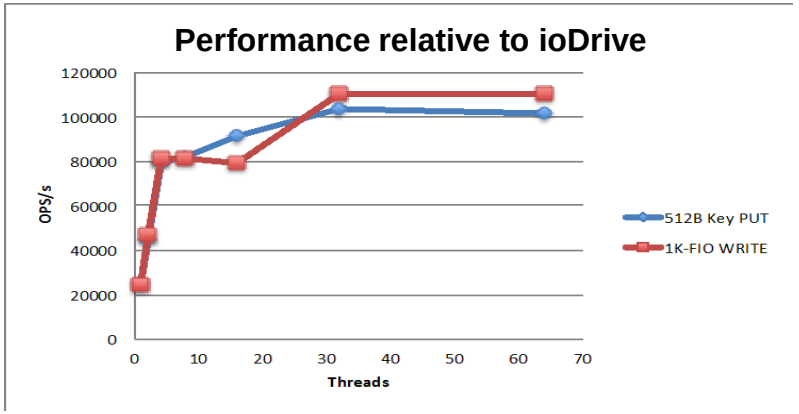
Directkey-Value Store – Sample Performance





Significantly more functionality with NO additional performance impact





1U HP blade server with 16 GB RAM, 8 CPU cores - Intel(R) Xeon(R) CPU X5472 @ 3.00GHz with single 1.2 TB ioDrive2 mono

Advantages of Native Flash Access

1. Helps accelerating applications
2. Eliminates redundant functionality
3. Leverages FTL mapping and sparse addressing
4. Optimizes garbage collection
5. Delivers transactional properties
6. Provides direct I/O as well as memory semantics.

Open Source Enablement and Standardization

- MySQL InnoDB extension (GPLv2)
- Standardization of primitives in T10

  Current standards proposal – Atomic Writes

  - SBC-4 SPC-5 Atomic-Write
    http://www.t10.org/cgi-bin/ac.pl?t=d&f=11-229r5.pdf
  - SBC-4 SPC-5 Scattered writes, optionally atomic
    http://www.t10.org/cgi-bin/ac.pl?t=d&f=12-086r3.pdf
  - SBC-4 SPC-5 Gathered reads, optionally atomic
    http://www.t10.org/cgi-bin/ac.pl?t=d&f=12-087r3.pdf

Thank you!

Ashish Batwara

Fusion-io

abatwara@fusionio.com

Extended Memory Non-Persistence Path

Fusion-io has developed a subsystem technology called Extended memory, which enables developers to take advantage of NAND Flash memory as an extension of DRAM.

The idea is to move frequently accessed data pages to DRAM while rarely accessed data pages are transferred from DRAM to Flash. Thus, the overall available capacity for DRAM can indirectly be increased.

Fusion-io said that the technology, which was created in collaboration with Princeton University researchers, allows software developers to simply assume that their entire data set is kept in-memory all the time as NAND is a much more cost-effective memory solution and can reach much greater capacities than DRAM.

“The Fusion ioMemory architecture is uniquely suited to innovation like the Extended Memory subsystem,” said Chris Mason, Fusion-io director of kernel engineering and principal author of the Btrfs file system for Linux, in a prepared statement. “Since Fusion ioMemory has moved beyond legacy disk-era protocols, we can integrate new features like the Extended Memory subsystem to truly advance application performance for enterprise computing in ways that are simply not possible with traditional SSDs.”

Developers can access the Extended Memory feature via Fusion-io's developer community.

http://www.tomshardware.com/news/dram-memory-flash-nand-fusion-io,16254.html?utm_source=dlvr.it&utm_medium=twitter#xtor=RSS-181

Auto-Commit Memory: Cutting Latency by Eliminating Block I/O

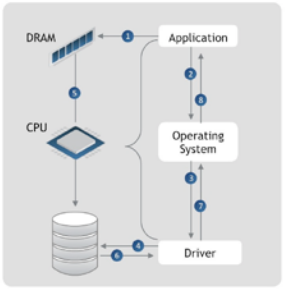
Auto-Commit Memory: Cutting Latency by Eliminating Block I/O
Our recent demonstration of one billion IOPS showcased a new paradigm for storing data through Fusion-io Auto-Commit Memory (ACM). ACM isn't just about making NAND Flash storage devices go faster, although it does that too. It's about introducing a much simpler and faster way for an application to guarantee data persistence.

When Simplicity Meets Speed
For decades, the industry norm for persisting data has been the same – an application manipulates data in memory, and when ready to persist the data, packages the data (sometimes called a transaction) for storage. At this point, the application asks the operating system to route the transaction through the kernel block I/O layer. The kernel block layer was built to work with traditional disks. In order to minimize the effect of slow rotational-disk seek times, application I/O is packaged into blocks with sizes matching hard disk sector sizes and sequenced for delivery to the disk drive. As Linux block maintainer (and Fusion-io chief architect) Jens Axboe points out, most real-world I/O patterns are dominated by small, random I/O requests, but are force-fit into 4k block I/Os sequenced by the block layer to match the characteristics of rotating disks. Note the number of steps in this pathway – each one contributes to latency. Even more steps are introduced in this pathway when the block storage device is at the other end of a network, behind various bus adaptors, and controllers. As long as memory is volatile, this type of I/O pathway will be the norm.

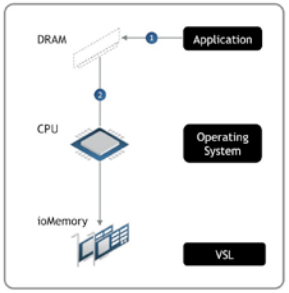
But, what if an application could designate a certain area within its memory space as persistent, and know that data in this area would maintain its state across system reboots? This application would no longer have the burden of following the multi-step block I/O pathway to persist that data. It would no longer need smaller transactions to be packaged into 4k blocks for storage. It would just place selected data meant for persistence in this designated memory area, and then continue using it through normal memory access semantics. If the application or system experienced a failure, the restarted application would find its data persisted in non-volatile memory exactly where it was left. To illustrate, how much faster could real-world databases go if the in-memory tail of their transaction logs had guaranteed persistence without waiting on synchronous block I/O? How much faster could real-world key-value stores (typical in NoSQL databases) go if their indices could be updated in non-volatile memory and not block while waiting on kernel I/O? That is the simplicity of Auto Commit Memory. Itreduces latency by eliminating entire steps in the persistence pathway.

Addressing Both Halves of the Problem
Block storage benchmarks such as throughput and IOPS are certainly important, but only address half of the problem. The other half of the problem is the work the application and kernel I/O subsystems must do to package and route data for storage. Applications can be accelerated by addressing either or both halves of this problem. However, note that, at some point, the overhead incurred by packaging and routing data through the kernel block storage subsystem will become the bottleneck. Breaking through that barrier was the goal of this technology demonstration. Give applications the software mechanisms to avoid this block storage packaging and routing latency and complexity. Let them spend more time processing data in memory, and less time packaging and waiting for that data to arrive at a block storage destination.

Fusion-io does indeed make very fast block I/O devices. What's most exciting about last week's demonstration for us is looking beyond fast block I/O devices to show what is possible when you address this this fast device as memory rather than block storage.



1. Data written to DRAM
2. Application calls OS to persist data
3. OS calls storage driver
4. Storage driver transmits command to I/O device
5. DMA transfers data from memory to I/O device
6. I/O device sends completion
7. Driver sends completion
8. OS sends completion



1. Data written to a designated DRAM region
2. Data transparently persisted to flash

http://www.fusionio.com/blog/auto-commit-memory-cutting-latency-by-eliminating-block-i/o/