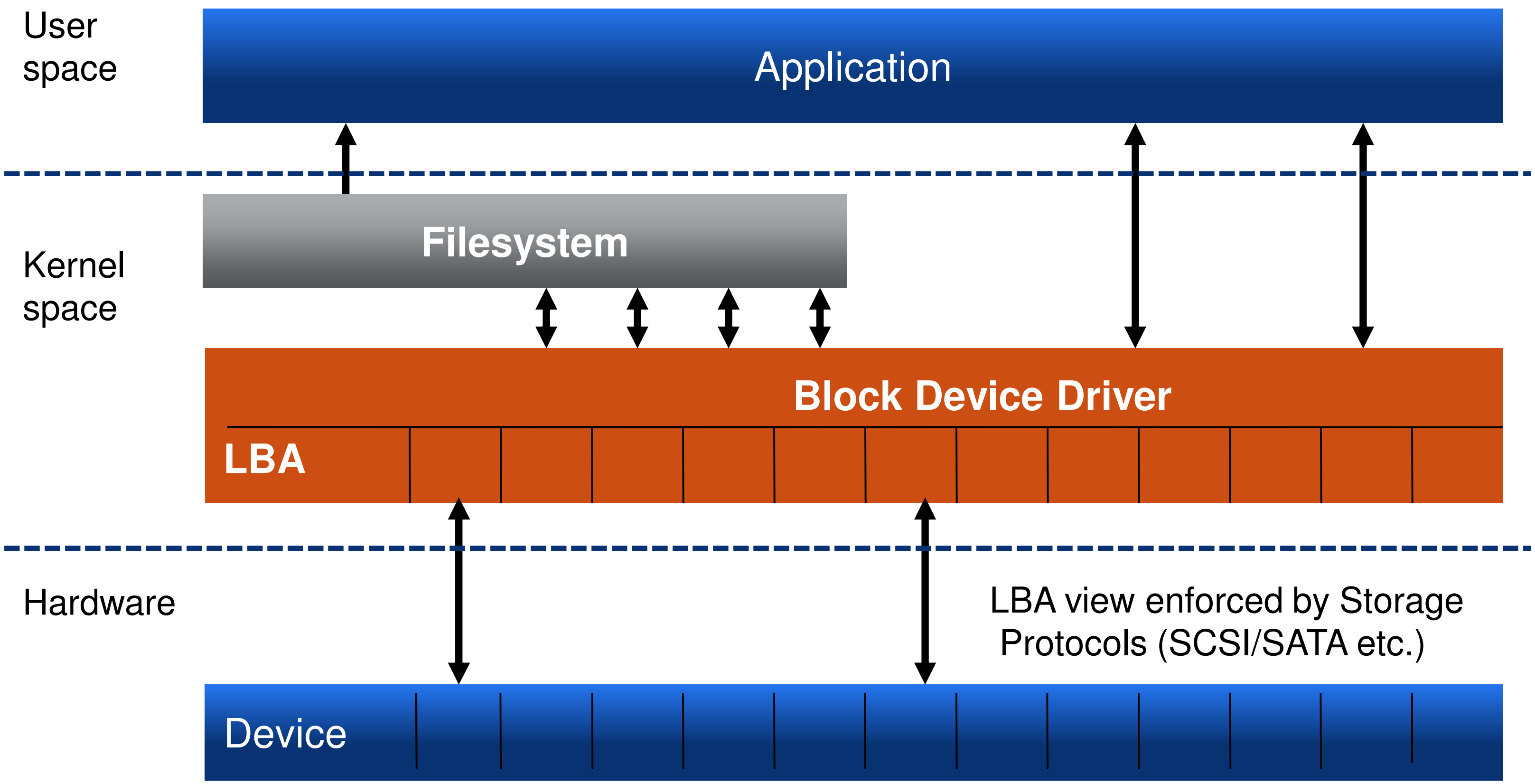
Leveraging host based Flash Translation Layer for Application Acceleration

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. Additional operation (erase)
Sequential vs Random Performance	100x difference. Elevator scheduling for disk arm	<10x difference. No disk arm – NAND die
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	Improve performance

Flash Translation Layer 101

Input

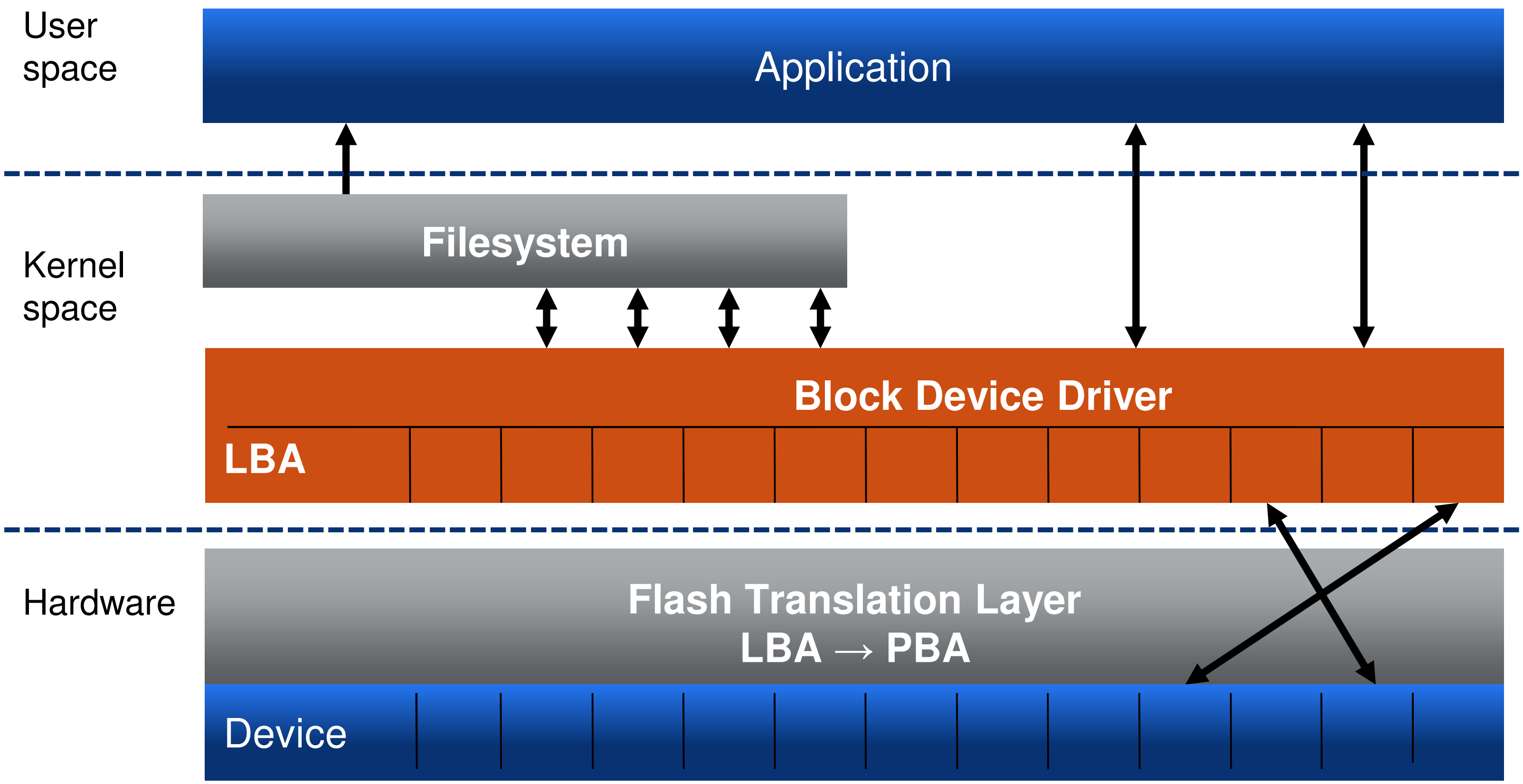
Logical Block Address (LBA)

Flash Translation Layer

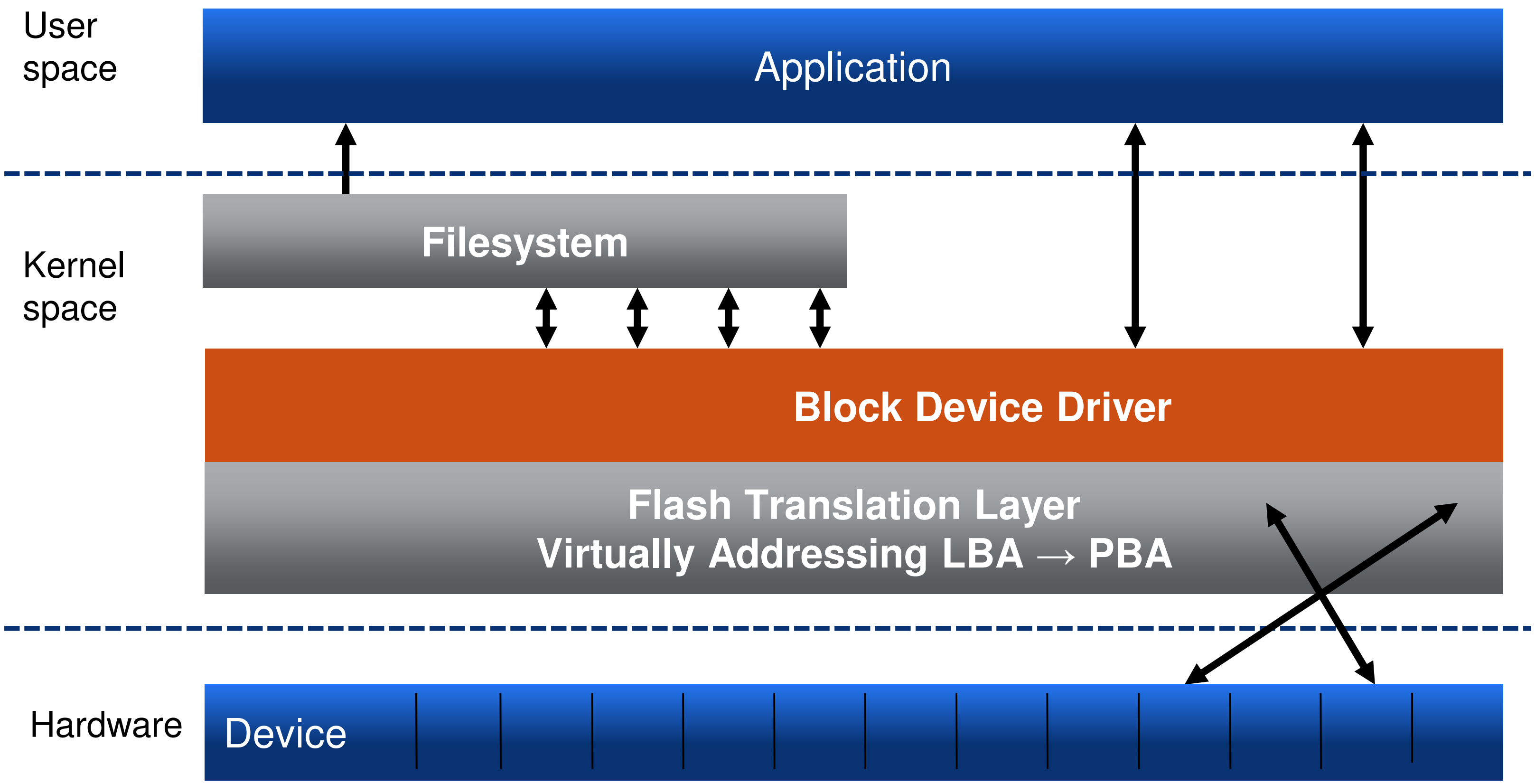
Output

Commands to NAND flash

Flash in Traditional Storage Stacks



Flash as a New Host Based Architecture



Flash Memory SUMMIT Fast Forward

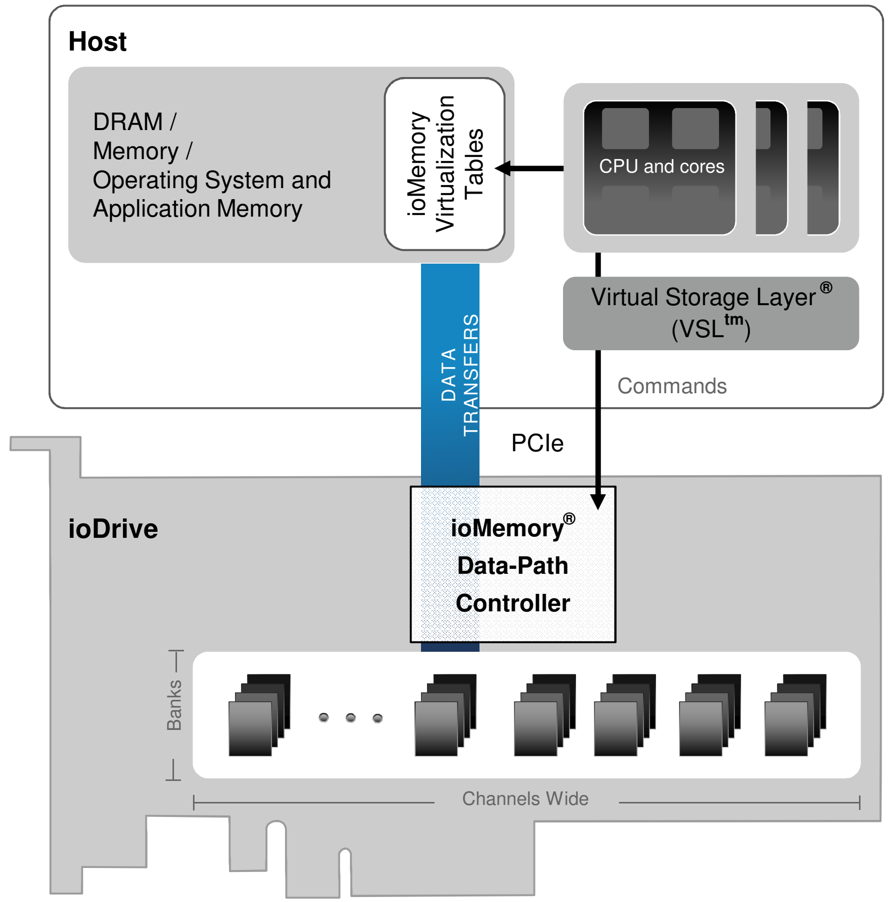
Call to Action

Gary O, Fusion-io, Flash Memory summit 2011

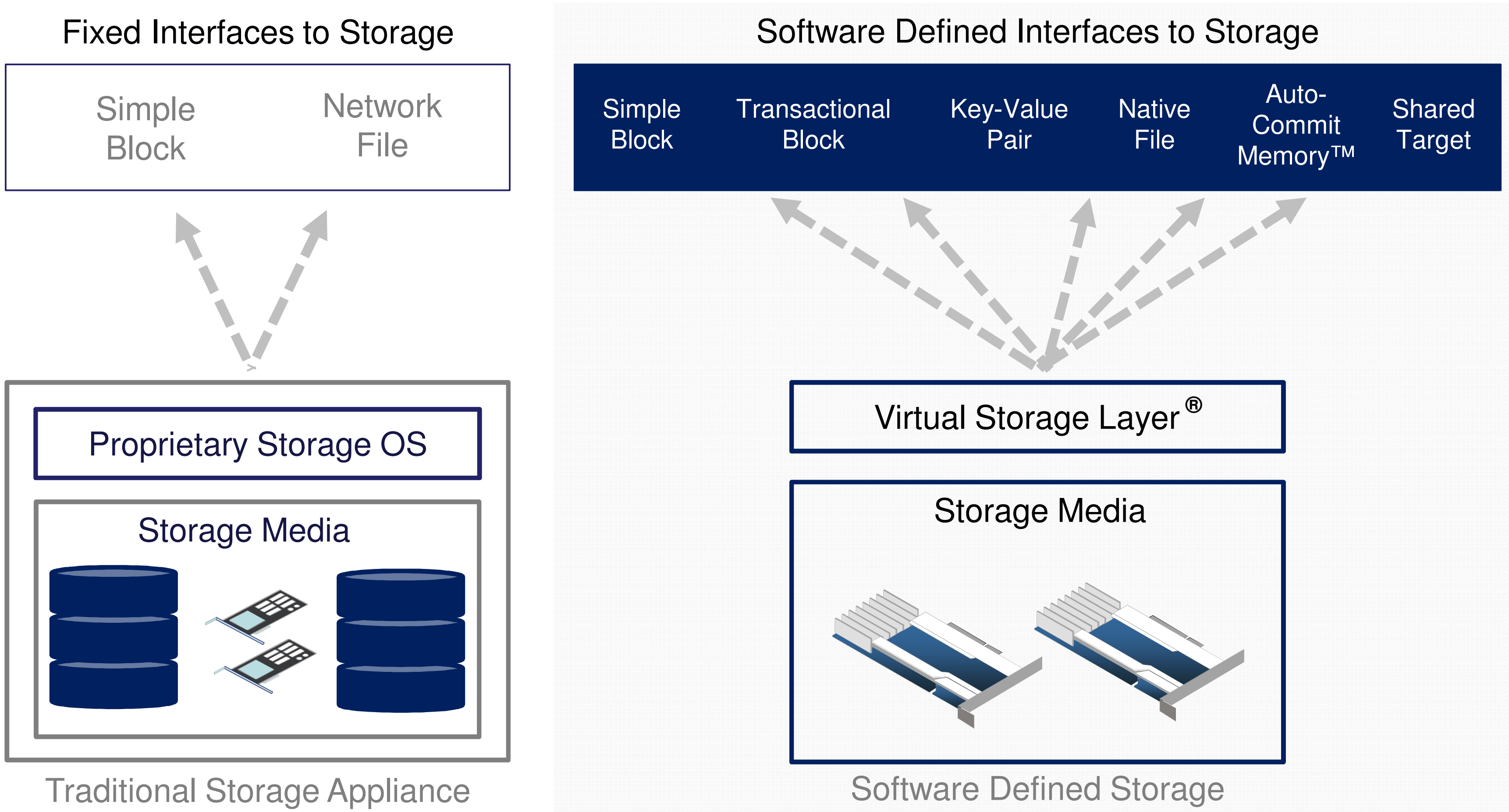
- Host-based FTLs integrate and scale with applications, examples include
  - File Systems
  - Caching
  - Databases
- Power of FTL no longer restricted by traditional block interfaces
- Opportunity for performance, simplicity and reliability improvements

Virtual Storage Layer® (VSL^{tm})

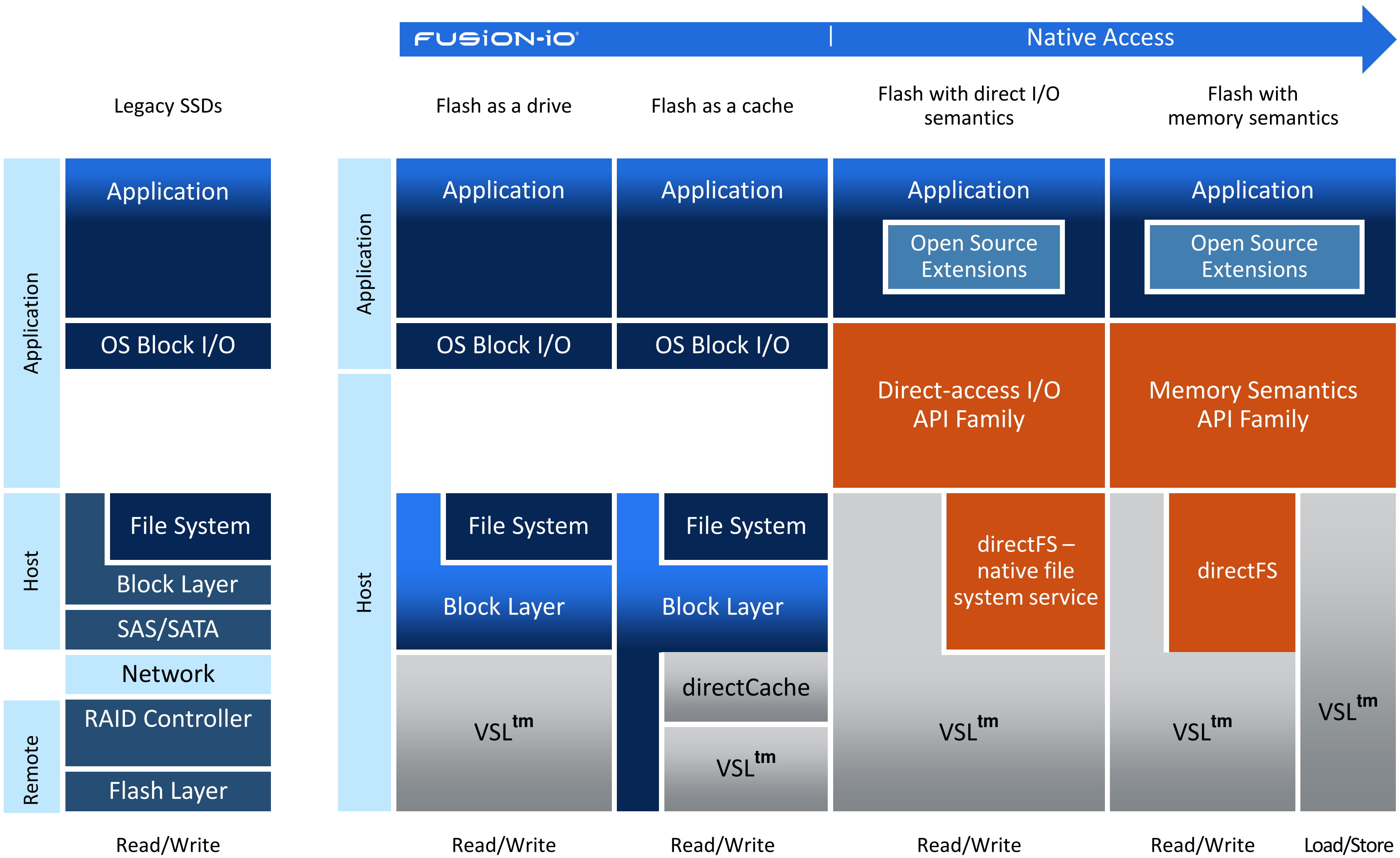
- VSL^{tm} – Fusion-io's host based FTL
- Cut-thru architecture – avoids traditional storage protocols
- Scales with multi-core
- Provide a large virtual address space
- HW/SW functional boundary defined as optimal for flash
- Traditional block access methods for compatibility
- New access methods, functionality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



Software Defined Interfaces to Storage

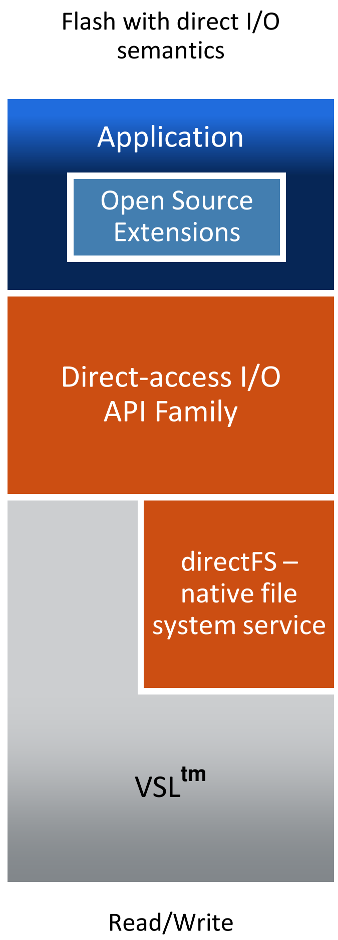


Flash Memory Evolution

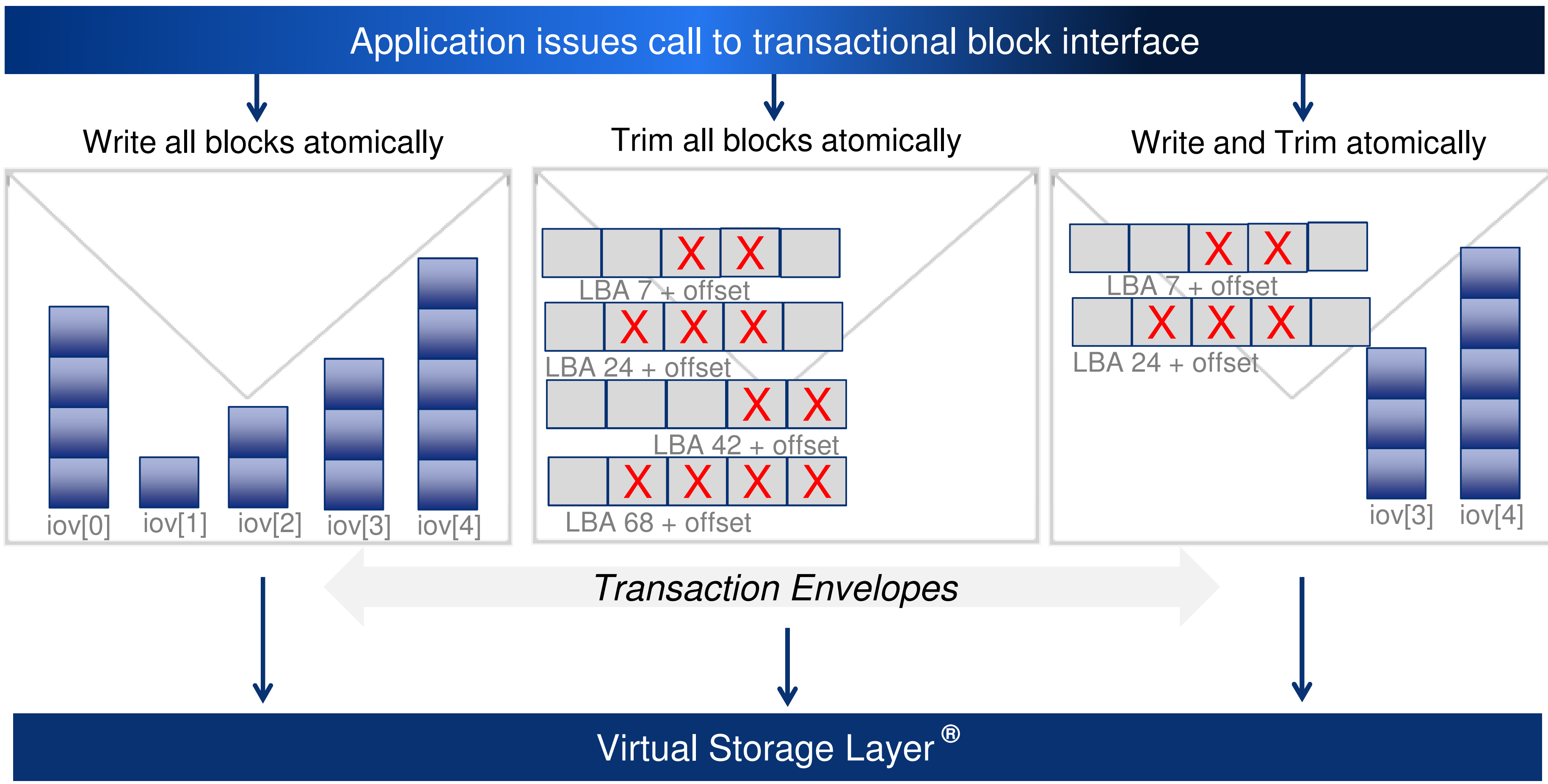


Direct I/O Semantics

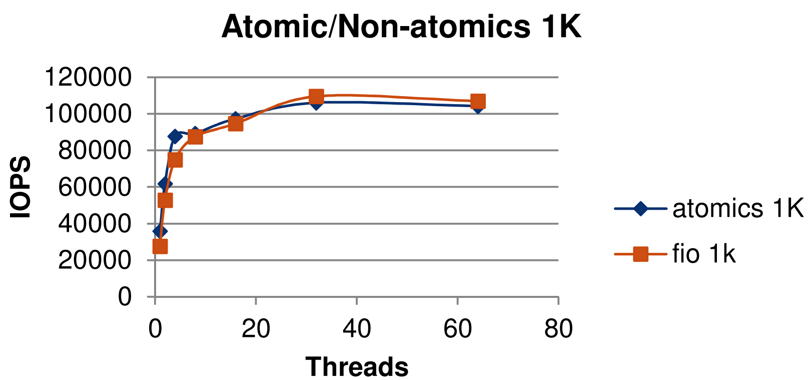
- direct I/O Primitives
  - Atomic Operations
  - Sparse Address Space
  - Exists
- direct Key-Value Store
  - NVM optimized with transactional semantics
- direct FS
  - Near Posix compliant FS implemented natively on flash
- directCache
  - Flash as a cache

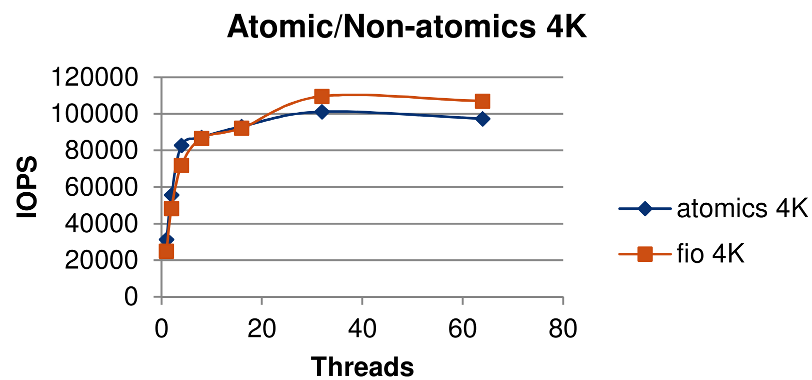


Transactional Block Interface

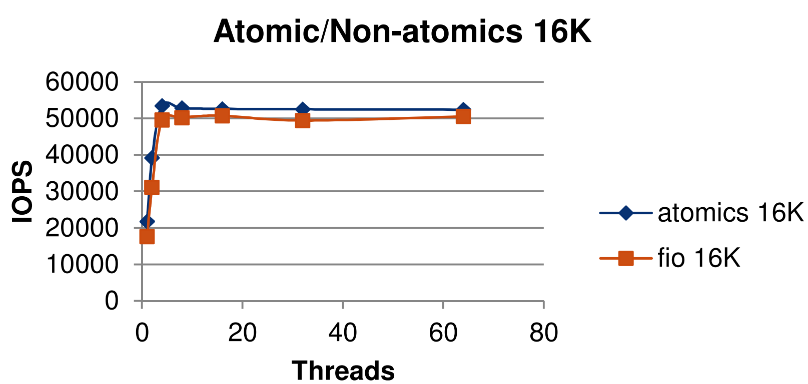


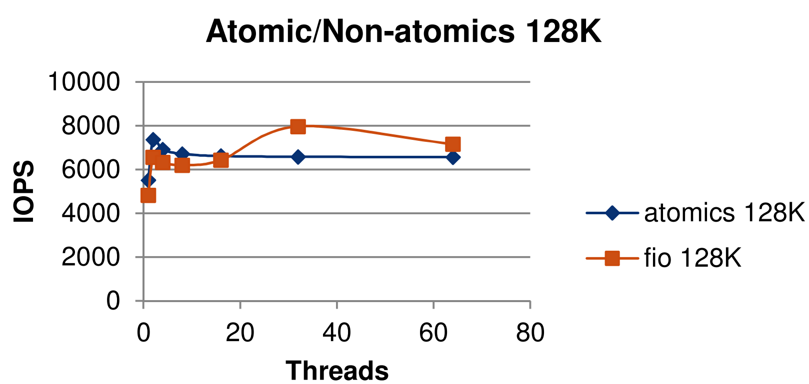
Atomic Micro-Benchmark – Sample Performance





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

Sysbench Performance With Atomic-Write

MySQL extension for Atomic-Write

43%

TRANSACTIONS/S EC INCREASE

2X

ENDURANCE INCREASE

½

95 PERCENTILE LATENCY REDUCTION

- 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

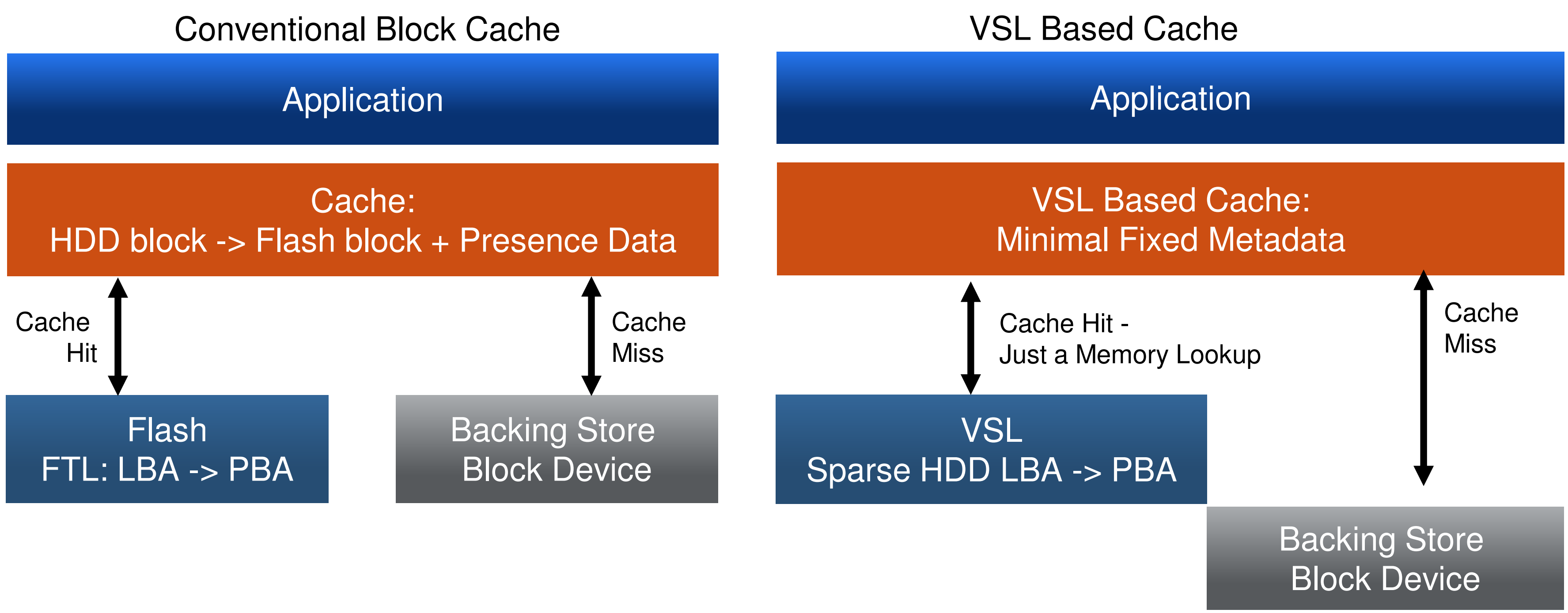
Open Source Enabling and Standardization

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

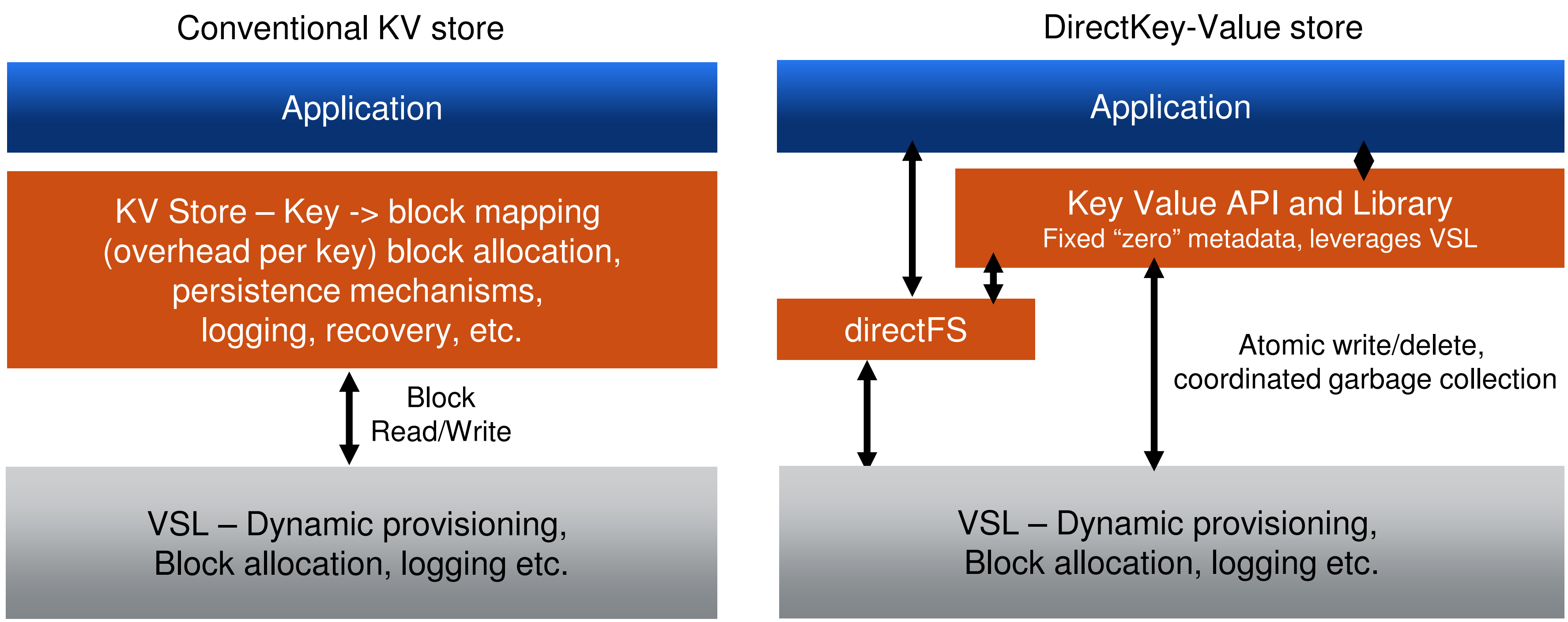
Current standards proposal – Atomic Writes

  - 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 atomichttp://www.t10.org/cgi-bin/ac.pl?t=d&f=12-087r3.pdf

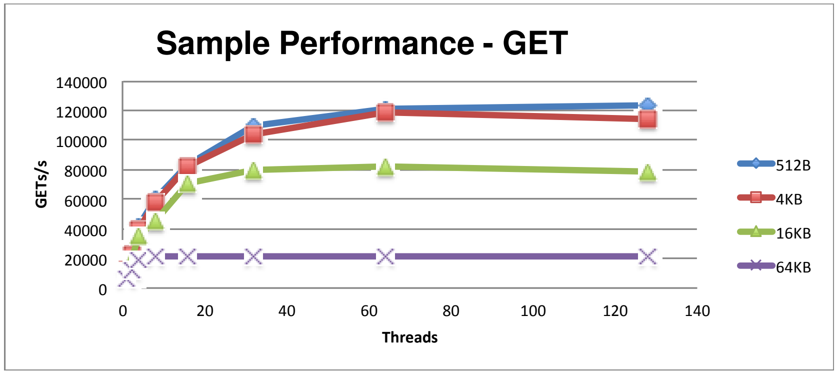
Sparse Addressing Using Example

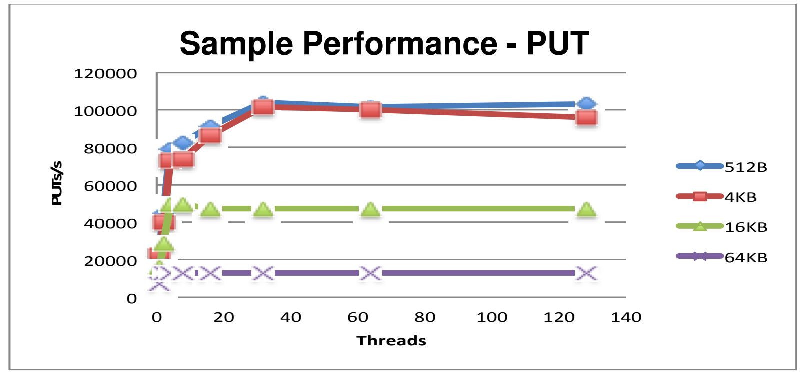


Direct Key-Value Interface

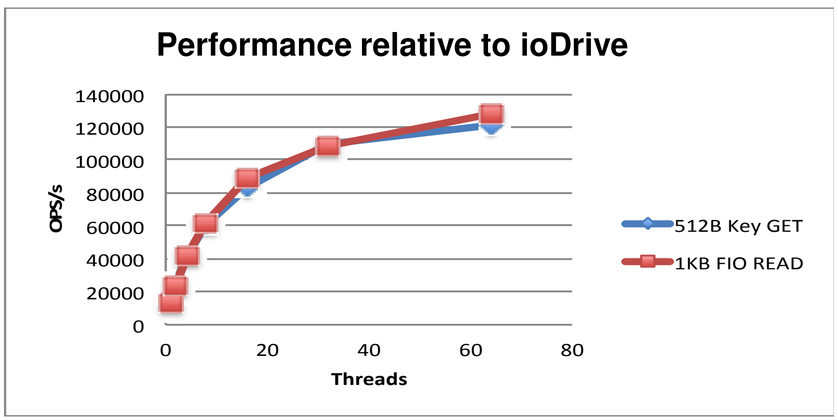


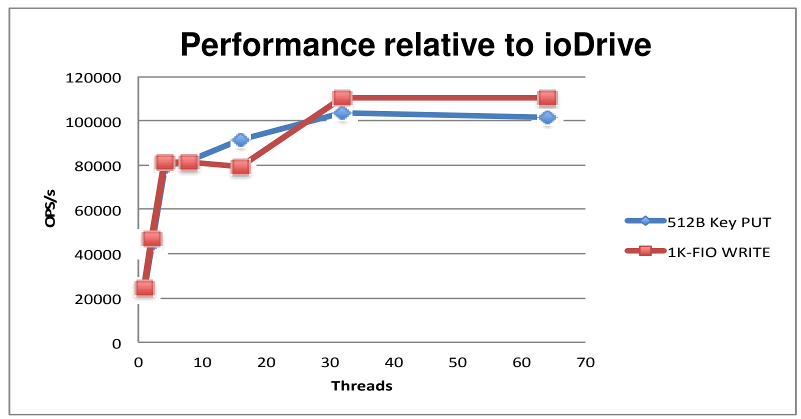
Directkey-Value Store – Sample Performance





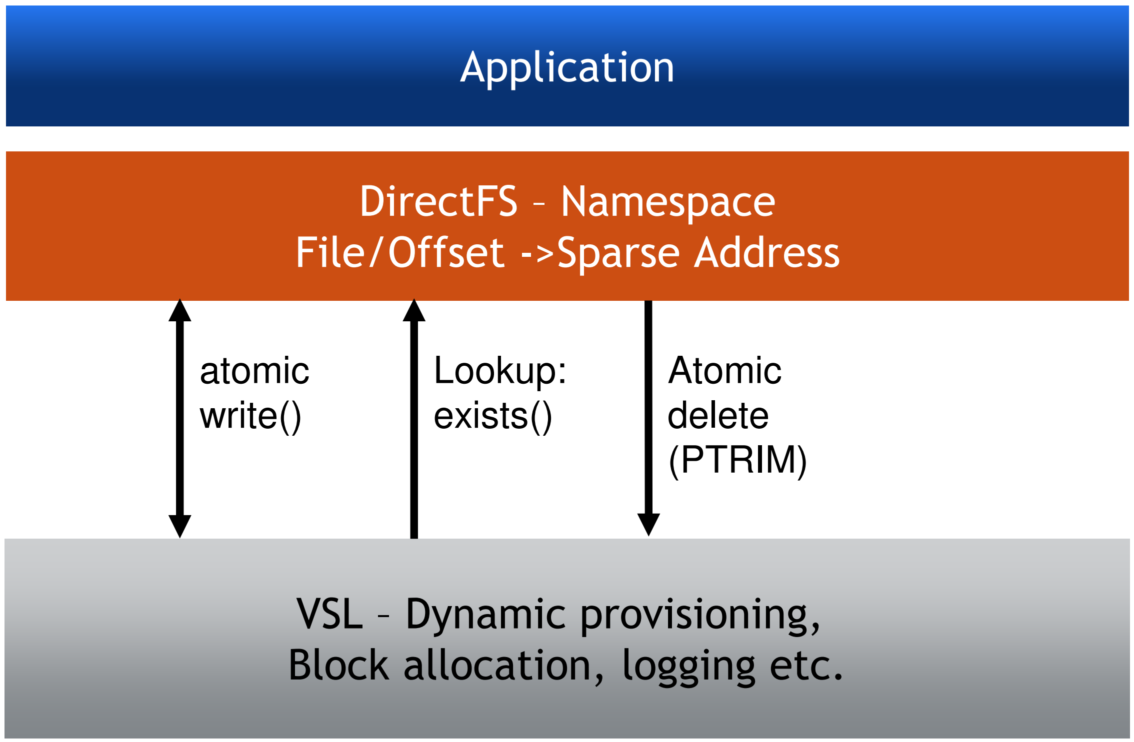
Significantly more functionality with NO additional performance impact





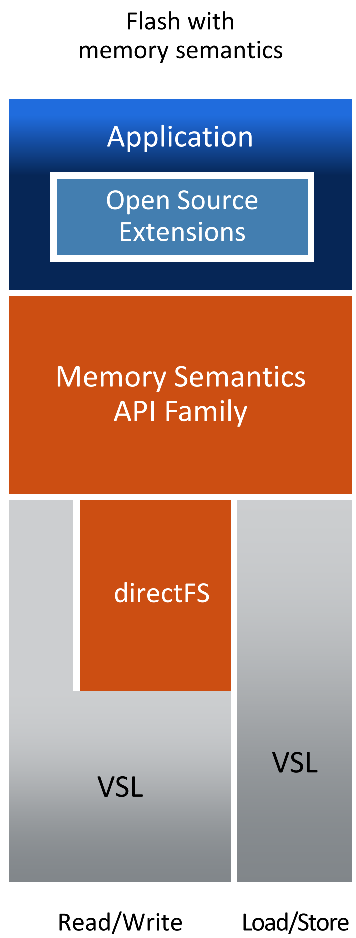
1U HP blade server with 16 GB RAM, 8 CPU cores - Intel(R) Xeon(R) CPU X5472 @ 3.00GHz with single 1.2 TB ioDrive2 mono

DirectFS – Native File Services Layer



Memory-Access Semantics

Extended Memory	**Volatile**	Transparently extends DRAM onto flash, extending application virtual memory
Checkpointed Memory	Volatile with non-volatile checkpoints	Region of application virtual memory which can be persisted to named file on flash
Auto-Commit Memory^{tm}	**Non-volatile**	Region of application memory automatically persisted to flash and recoverable post-system failure



Extended Memory - Overview

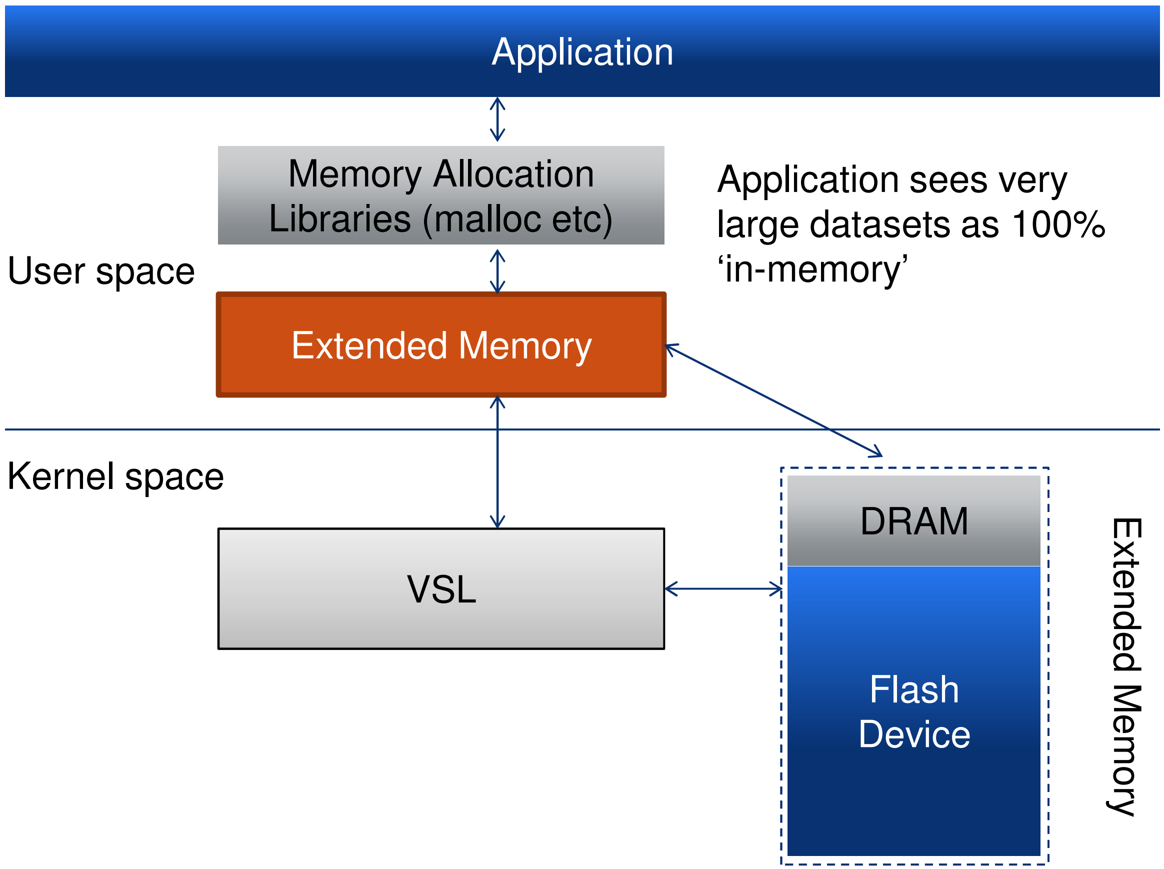
System Memory

Extended Memory Mechanism

Flash as memory

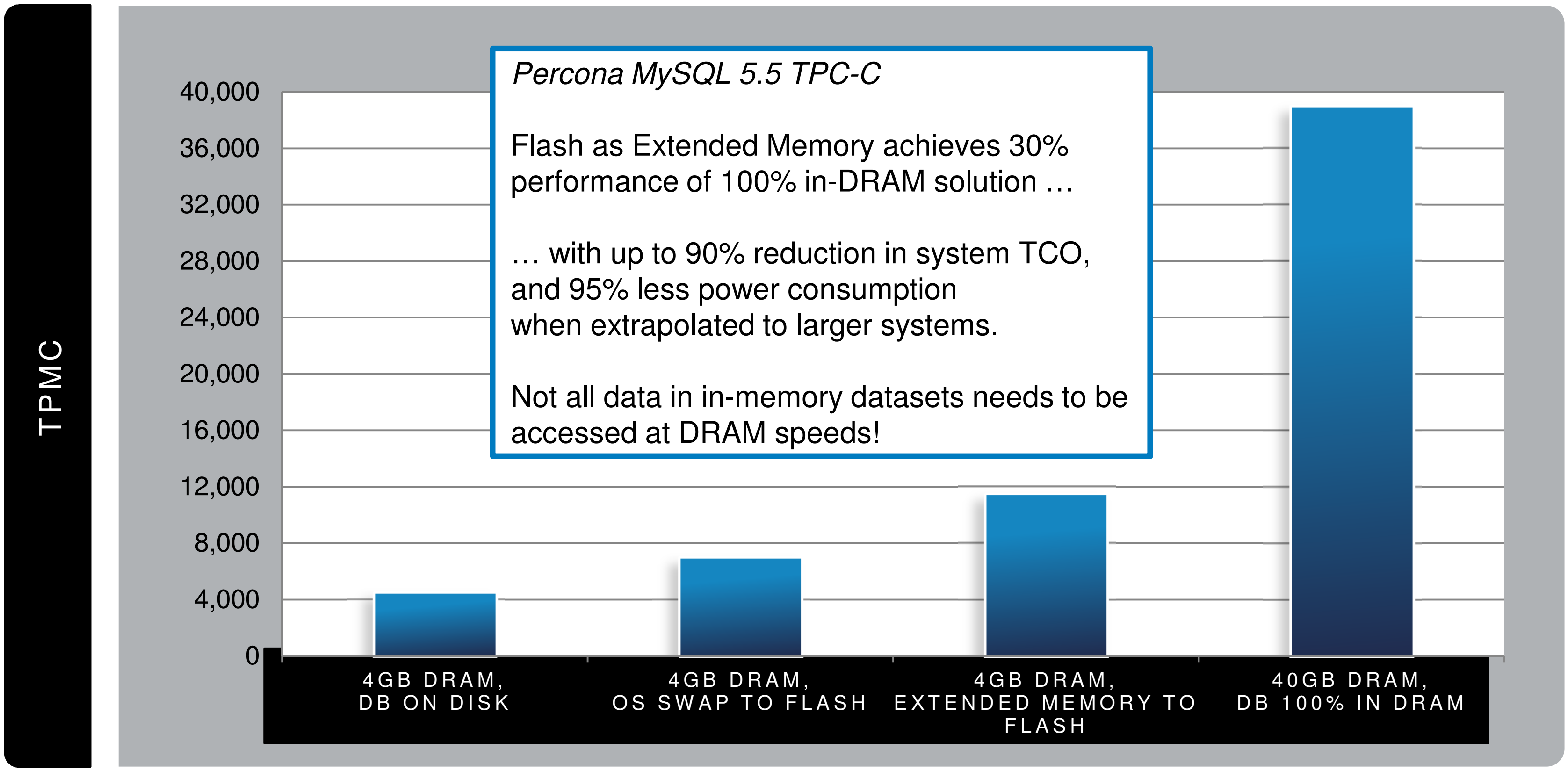
- Layered under existing memory allocation services (malloc(), mmap(), etc.)
- Uses existing memory page pinning and prioritization services (mlock(), madvise(), etc.)
- Leverages OS kernel page usage statistics to determine page eviction policies

Extended Memory – API Library



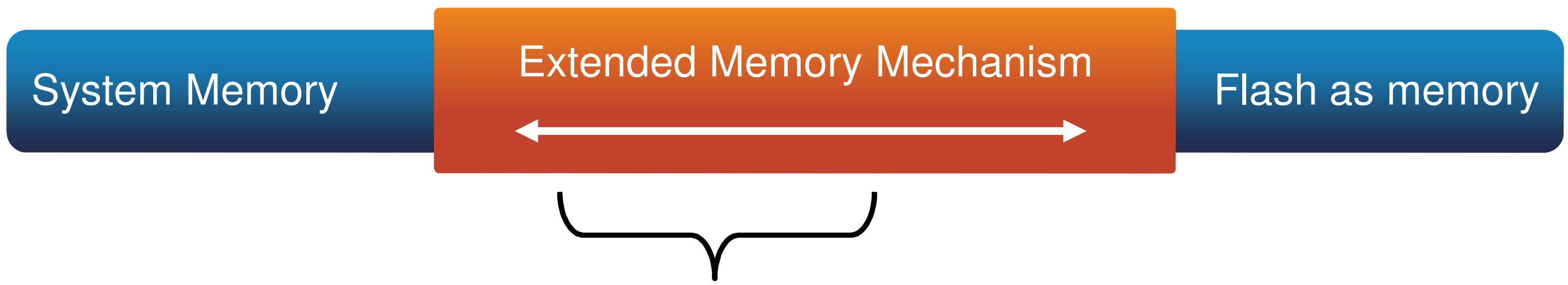
Application sees very large datasets as 100% ‘in-memory’

Simple Database Example



24 core Xeon, 140G Fusion-io NAND-flash, 40G DB size

Checkpointed Memory Persistence Path



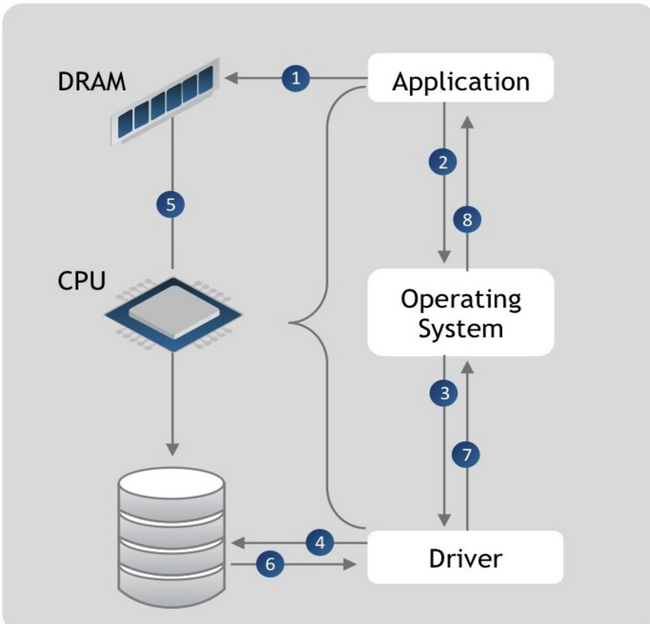
1. Application designates virtual address space range to be checkpointed
   a. Causes creation of independently-addressable linked clone of the checkpointed address range (no data moves or copies)
   b. Checkpoint appears as addressable file in the directFS native filesystem namespace.

2. Application can continue manipulating contents of designated virtual address range without affecting contents of persisted checkpoint file.

3. Application can load or manipulate persisted checkpoint file at a later time

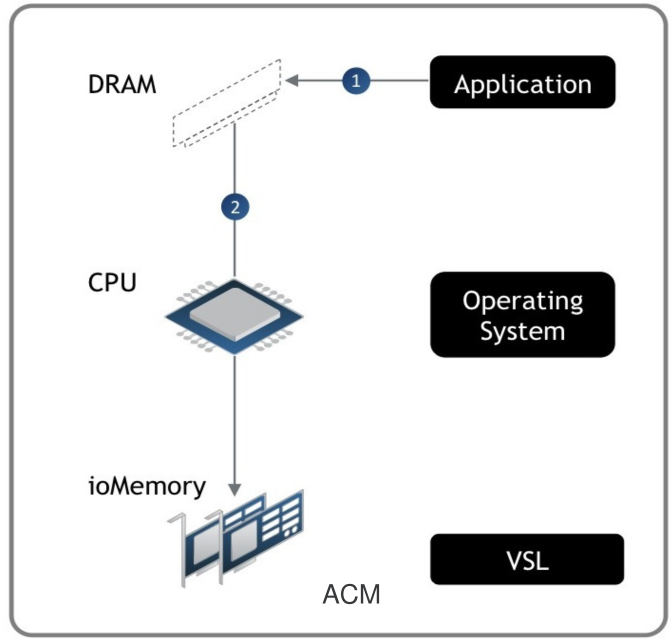
Auto-Commit Memory™ Persistence Path

Conventional I/O



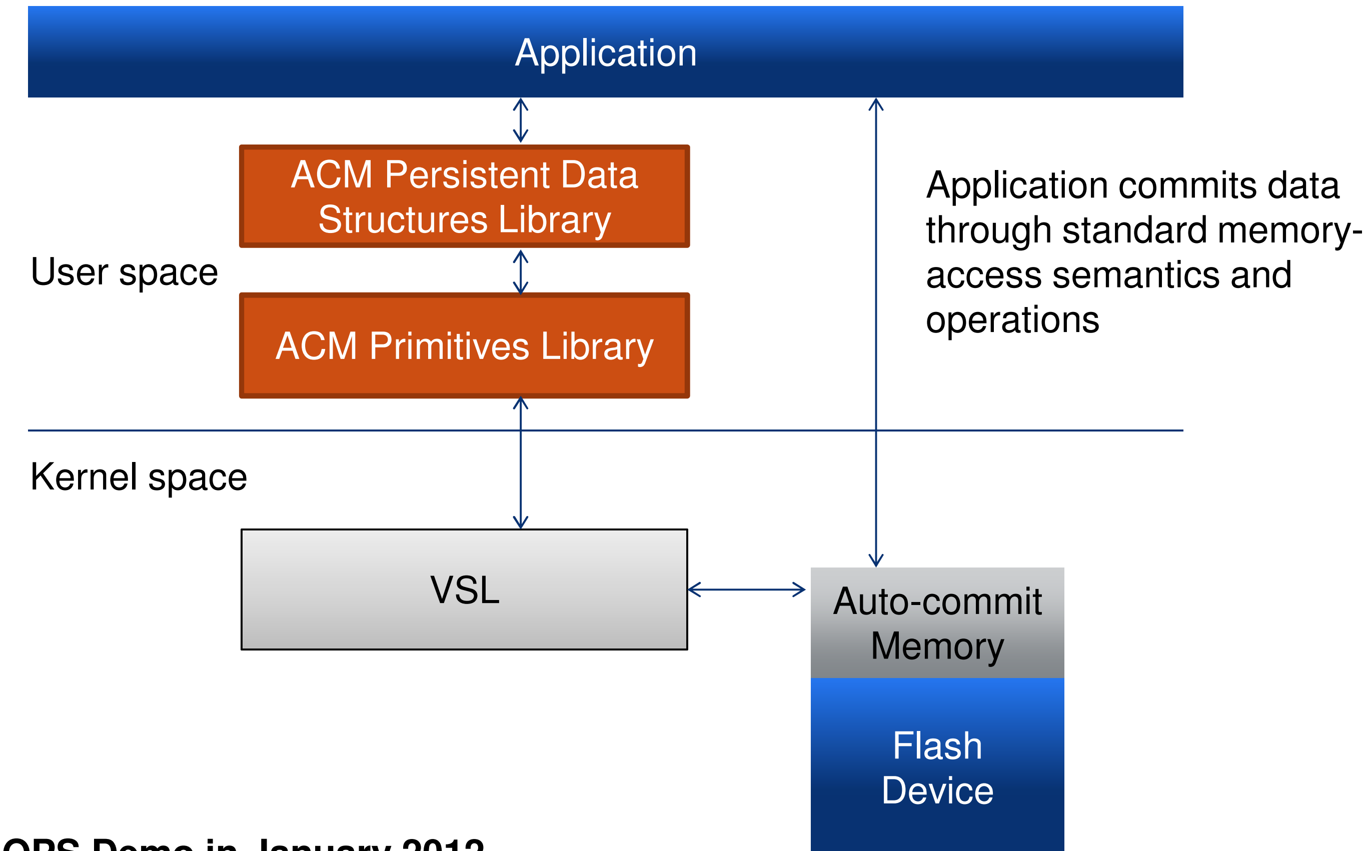
1. Data written to DRAM
2. Application calls OS to persist data
3. OS calls storage driver
4. Storage driver transmits command to I/O device
5. DMA transfers data from memory to I/O device
6. I/O device sends completion
7. Driver sends completion
8. OS sends completion

Auto-Commit Memory



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

Auto-Commit Memory™ – API Library



Billion IOPS Demo in January 2012 – ACM with 64 ioDrives, 8 HP servers

Conclusion

Host based FTL

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.

Thank you!

Ashish Batwara

Fusion-io

abatwara@fusionio.com

Direct I/O Primitives – Sparse Address Space

- A capability of the Virtual Storage Layer®
- Capacity dynamically allocated upon write
- LBA address space size can be far larger than actual capacity
- Capability is extended to upstream software via the sparse address space
- Higher level software usage via primitives
- Supports conventional block usages while enabling new usages in cache, file systems, etc.

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

EXISTS (Virtual Address)

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