DATA-INTENSIVE
COMPUTING SYSTEMS
LABORATORY

Modernizing File System through In-Storage Indexing

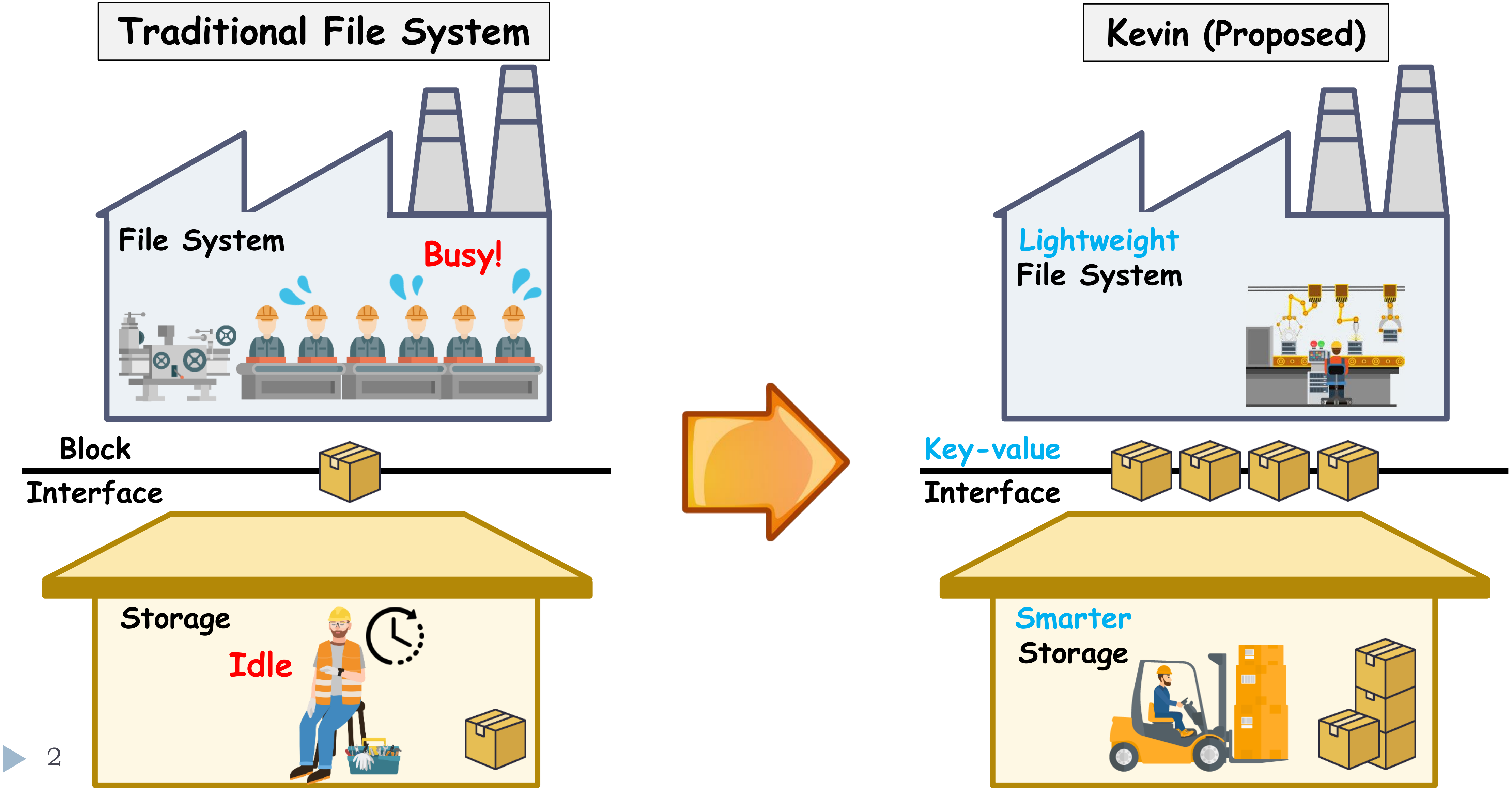
Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park,
Eunji Lee[†], Bryan S. Kim[‡], and Sungjin Lee

DGIST [†]Soongsil University [‡]Syracuse University

15th USENIX Symposium on Operating Systems Design and Implementation

July 14-16, 2021

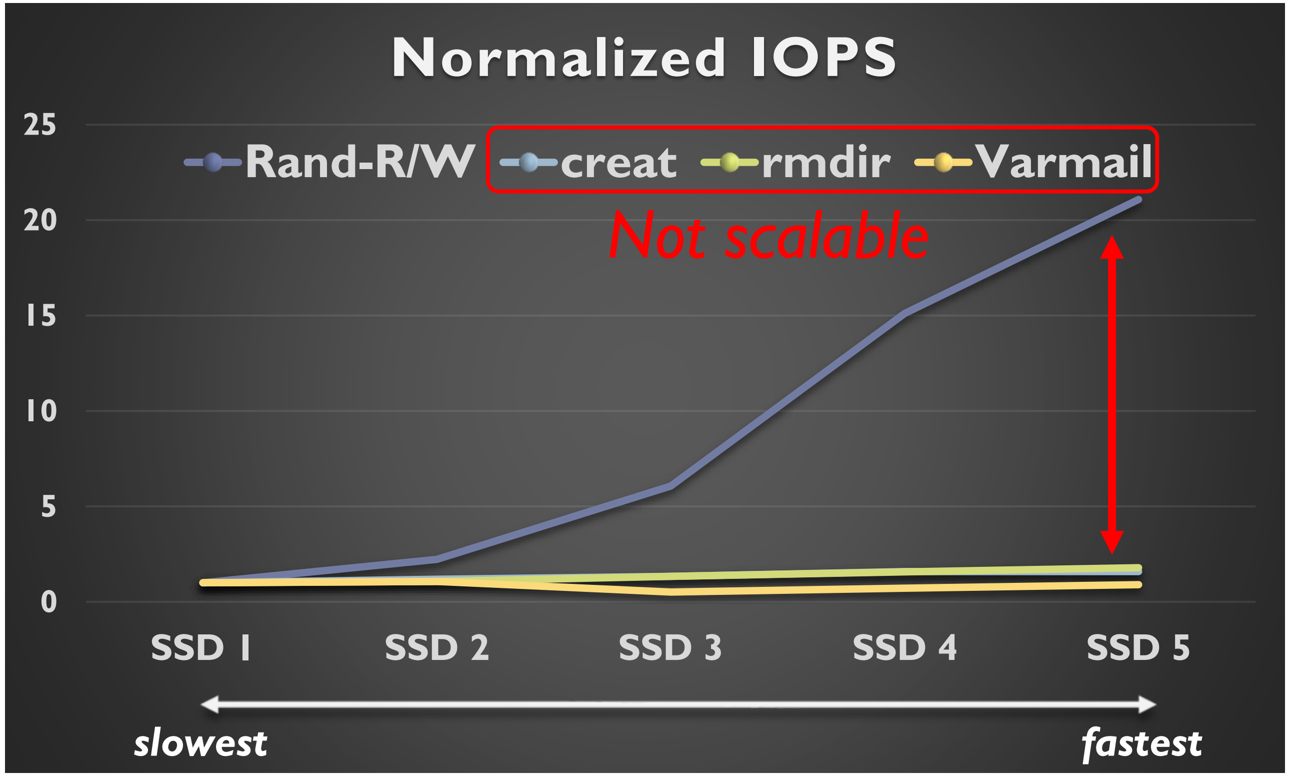
Overview



Traditional File System
File System
Busy!
Block
Interface
Storage
Idle
Kevin (Proposed)
Lightweight
File System
Key-value
Interface
Smarter
Storage

Does the performance of a FS scale with faster SSDs?

- Performance evaluation on EXT4 using 5 SSDs with different performance
  - Rand-R/W (50:50): data-intensive
  - creat, rmdir: metadata-intensive
  - Varmail: fsync-intensive



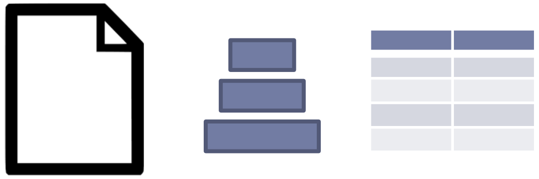
Faster devices *cannot* make file systems perform better!

What does a file system do?

- Abstract files and directories over block devices
  - Metadata management
  - Space management
  - Crash consistency

- Incur extra I/O overhead

Metadata Management



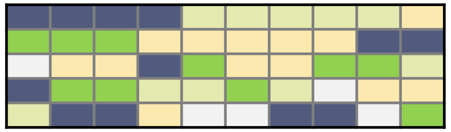
inode

disk pointer (e.g., extent)

dir. entry

I/O Amplification

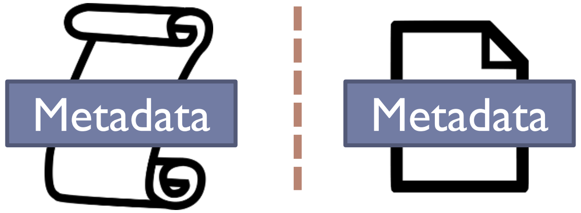
Space Management



bitmaps

Fragmentation

Crash Consistency



journal

barrier

checkpoint!

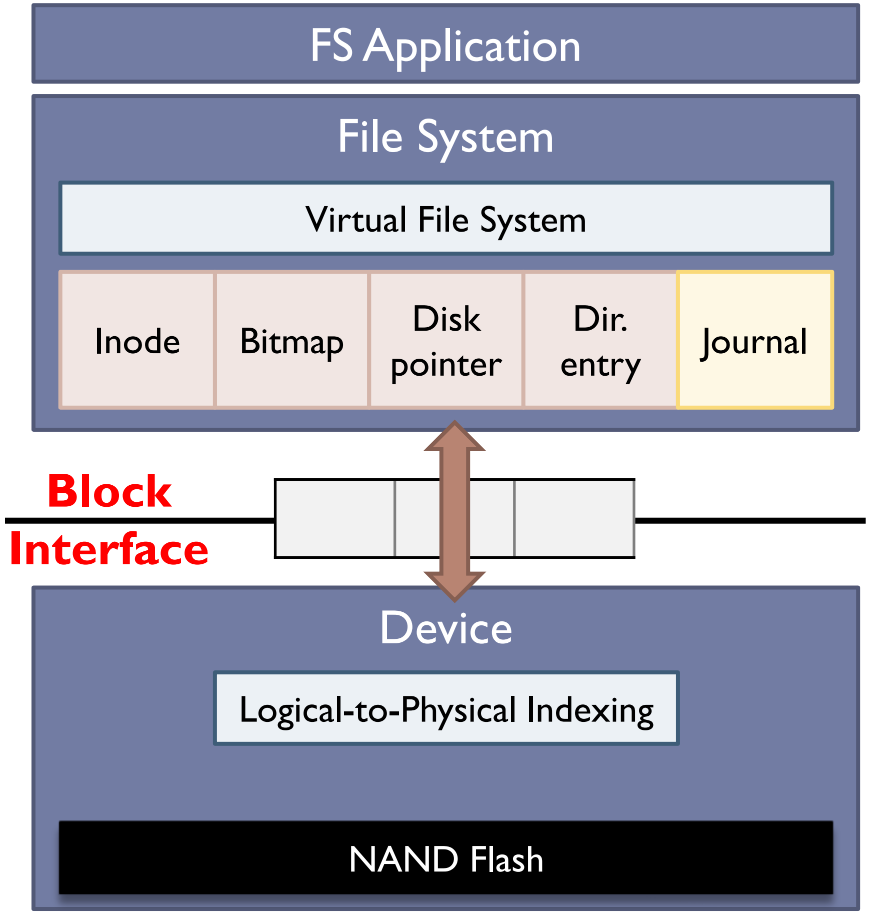
Journaling I/O (double-writes, I/O delay)

Why does file system have to do such tasks?

- Fundamental limitation of block interface?
  - A linear array of fixed-sized blocks
  - Simple I/O primitives: read, write, and trim
  - Block-level atomicity

- File systems take care of too many things
  - Metadata management
    - Index files and directories over blocks
  - Space management
    - Allocate and free blocks
  - Crash consistency
    - Ensure consistency on multiple blocks

The interface should be changed!



FS Application

File System

Virtual File System

Inode | Bitmap | Disk pointer | Dir. entry | Journal

Block Interface

Device

Logical-to-Physical Indexing

NAND Flash

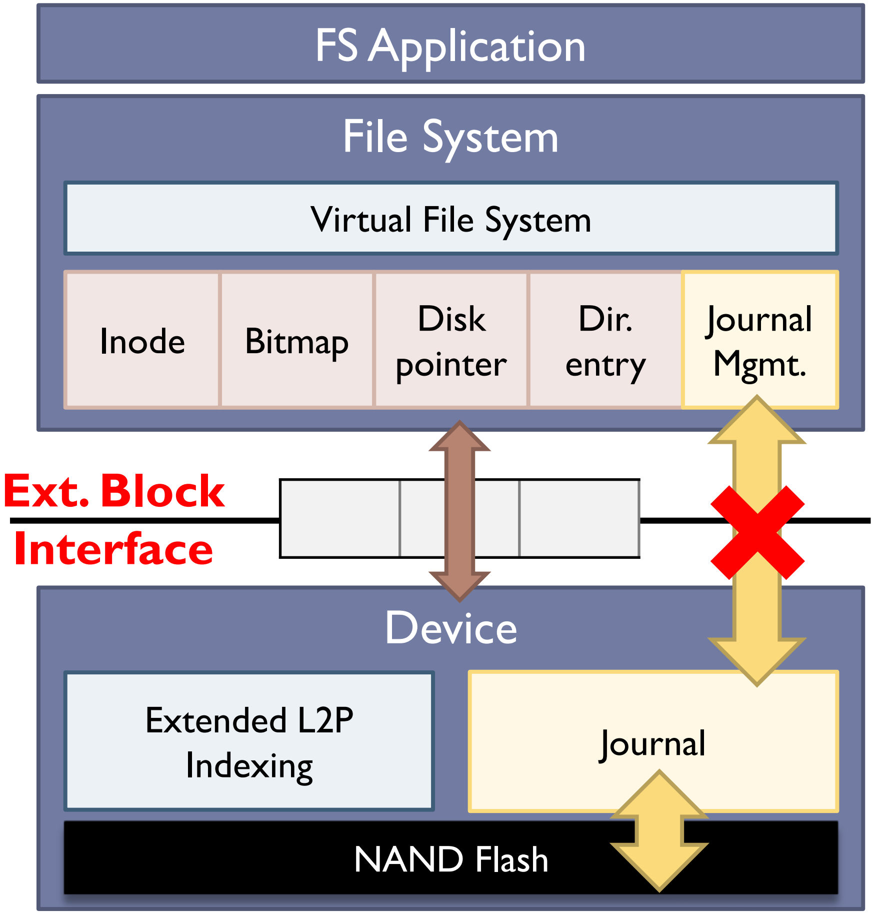
Prior studies – Extended block interface

- Extended block interface
  - BarrierFS, OPTR, SCFTL [FAST'18, ATC'19, OSDI'20]
    - ***Offload journaling*** into the SSD

Checklist

- Reducing Meta I/O Amp.? ✗
- Resilient to Fragmentation? ✗
- Reducing FS Journaling I/O? ✓

- Cannot address all the problems at once



Performing journaling inside the storage
→ No journaling traffic between the FS and the device

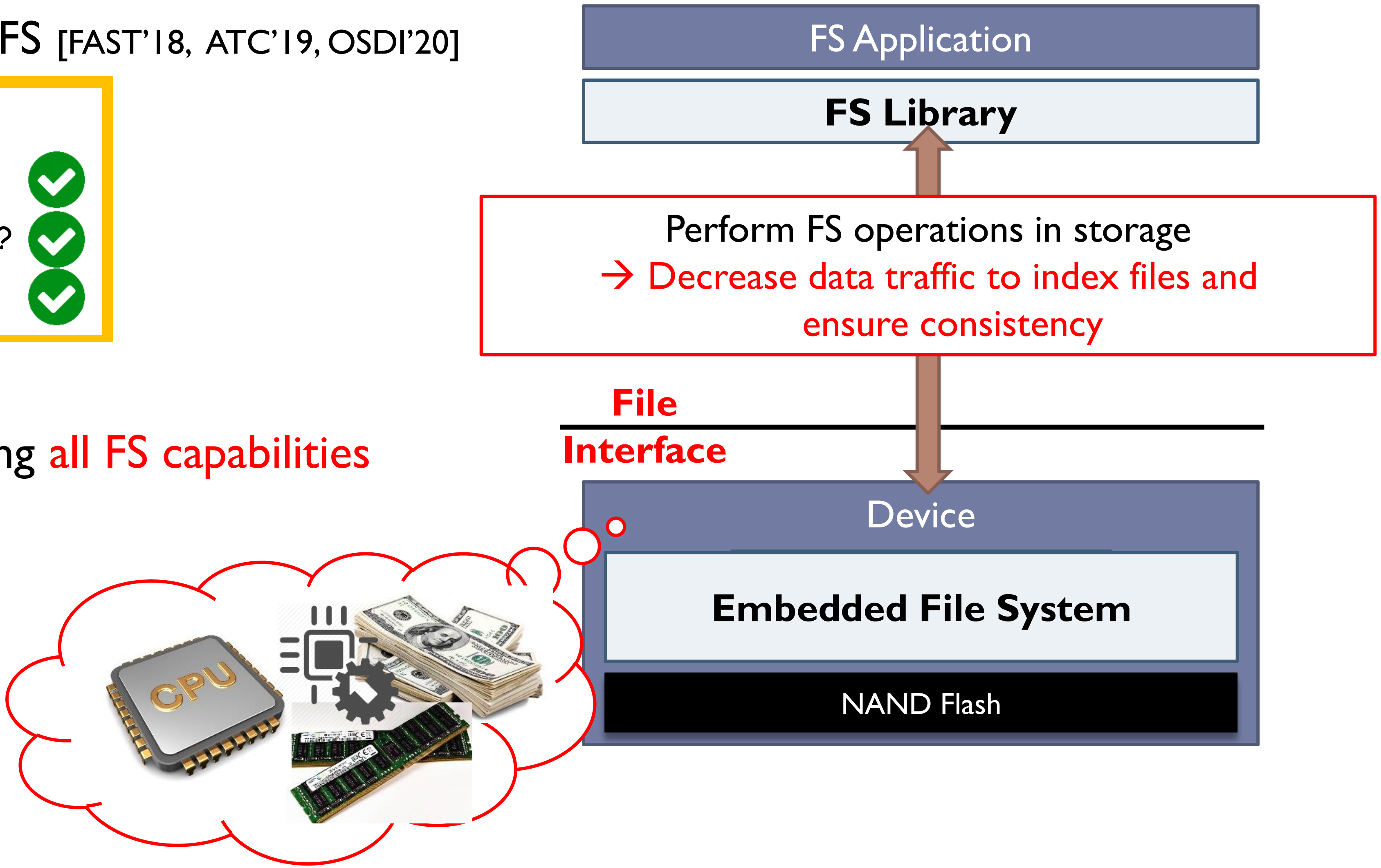
Prior studies – File interface

- File system interface
  - DevFS, INSIDER, CrossFS [FAST'18, ATC'19, OSDI'20]

Checklist

- Reducing **Meta I/O Amp.**? ✓
- Resilient to **Fragmentation**? ✓
- Reducing **Journaling I/O**? ✓

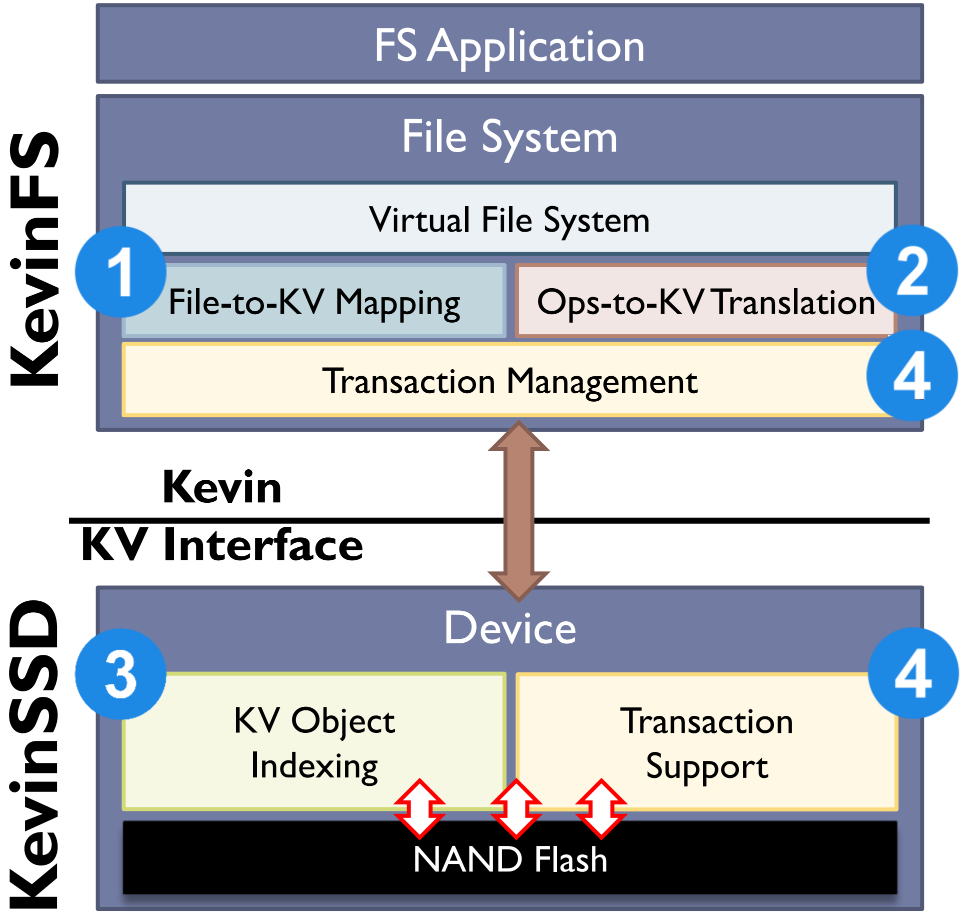
- High cost for maintaining all FS capabilities
  - Caching
  - Backup
  - Snapshot, …



Perform FS operations in storage
→ Decrease data traffic to index files and ensure consistency

Kevin = KevinFS + KevinSSD

- What KevinFS does?
  - Mapping files and directories to KV objects
  - Translate POSIX syscalls to KV commands
  - Ensure consistency with a KV-SSD
- What KevinSSD does?
  - Indexing KV objects in storage
  - Support transaction over multiple KV objects



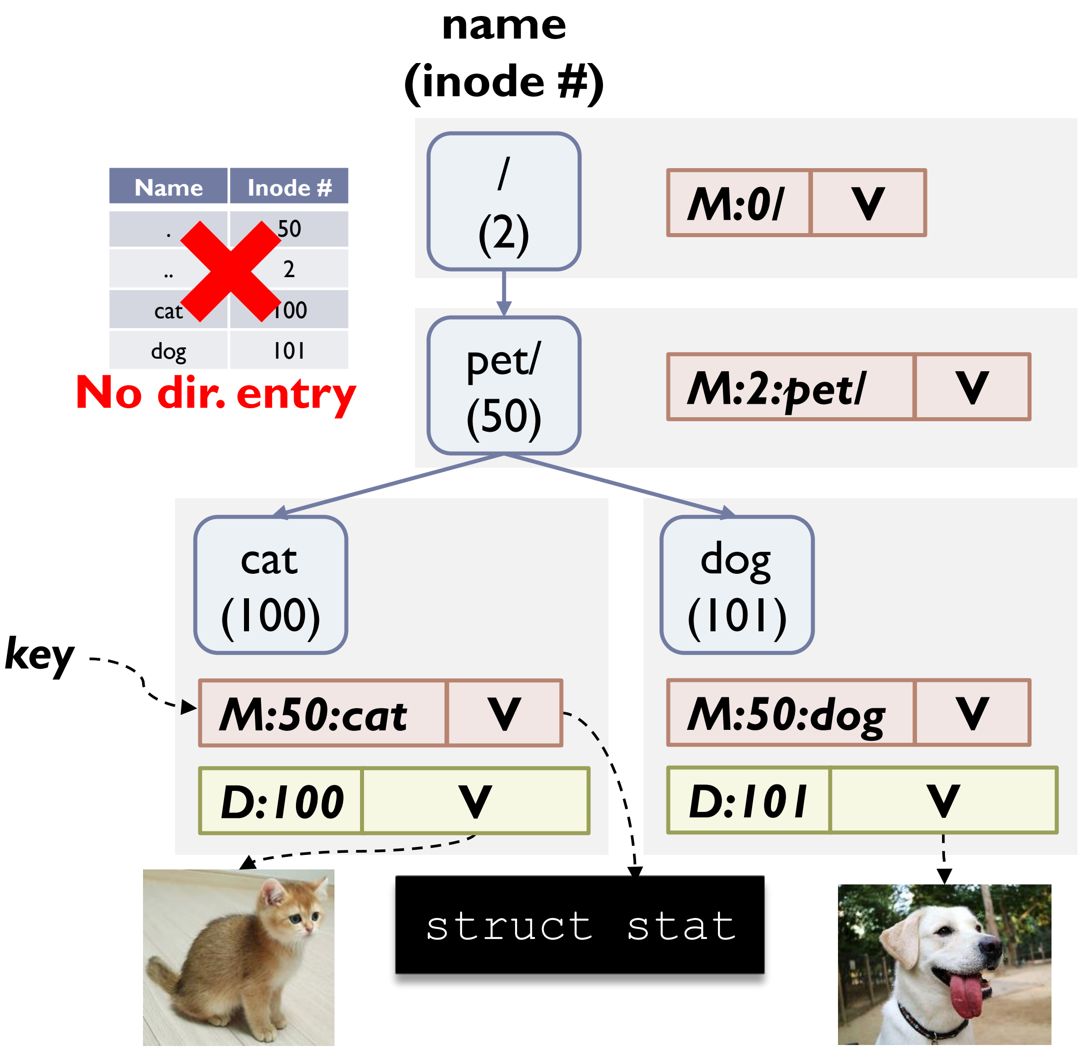
Data transfer for indexing KV objects occurs in storage
→ Lightweight FS with low FS traffic

INDEX

- Introduction
- Kevin
  - File-to-KV mapping
  - Operation-to-KV translation
  - KV object indexing algorithm
  - Transaction management
- Experiments
- Conclusion

File-to-KV mapping

- Map files/directories to two types of KV objects (meta & data objects)
  - 'Inode number' based key naming
    - *cf. full-path based naming*
      - Complicated algo. for handling *rename*
      - Longer key length
- Meta object: store file's information
  - Key: **'M' + parent's inode # + name**
- Data object: store file's contents
  - Key: **'D' + file's inode #**



Name	Inode #
.	50
..	2
cat	100
dog	101

Operation-to-KV translation

- All POSIX-compliant syscalls can be translated into KV operations
  - KevinFS supports 86 syscalls out of 102 currently

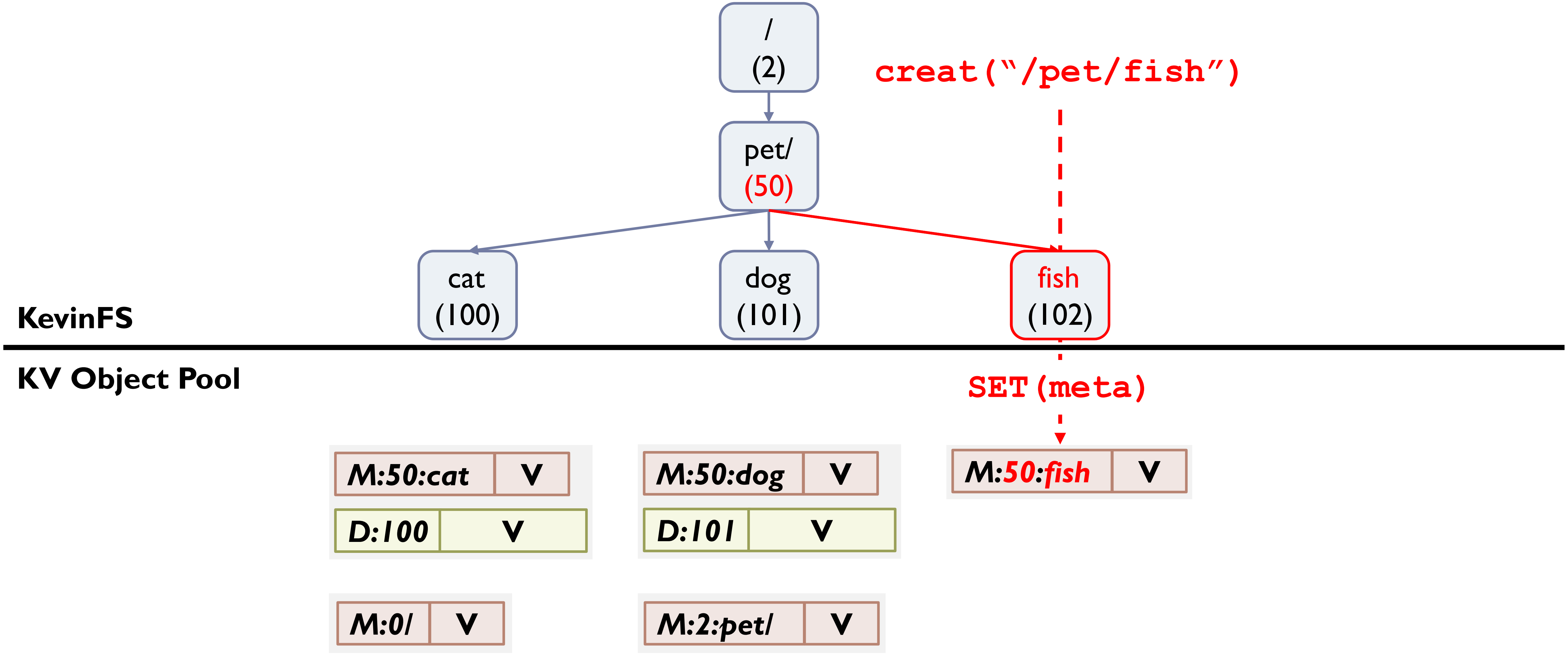
Syscalls	KV
`mkdir()`	`SET(meta)`
`creat()`	`SET(meta)`
`write()`	`SET(data)`
`rmdir()`	`DELETE(meta)`
`lookup()`	`GET(meta)`
`read()`	`GET(data)`
`readdir()`	`ITERATE(meta)`

Operation-to-KV translation (Cont.)

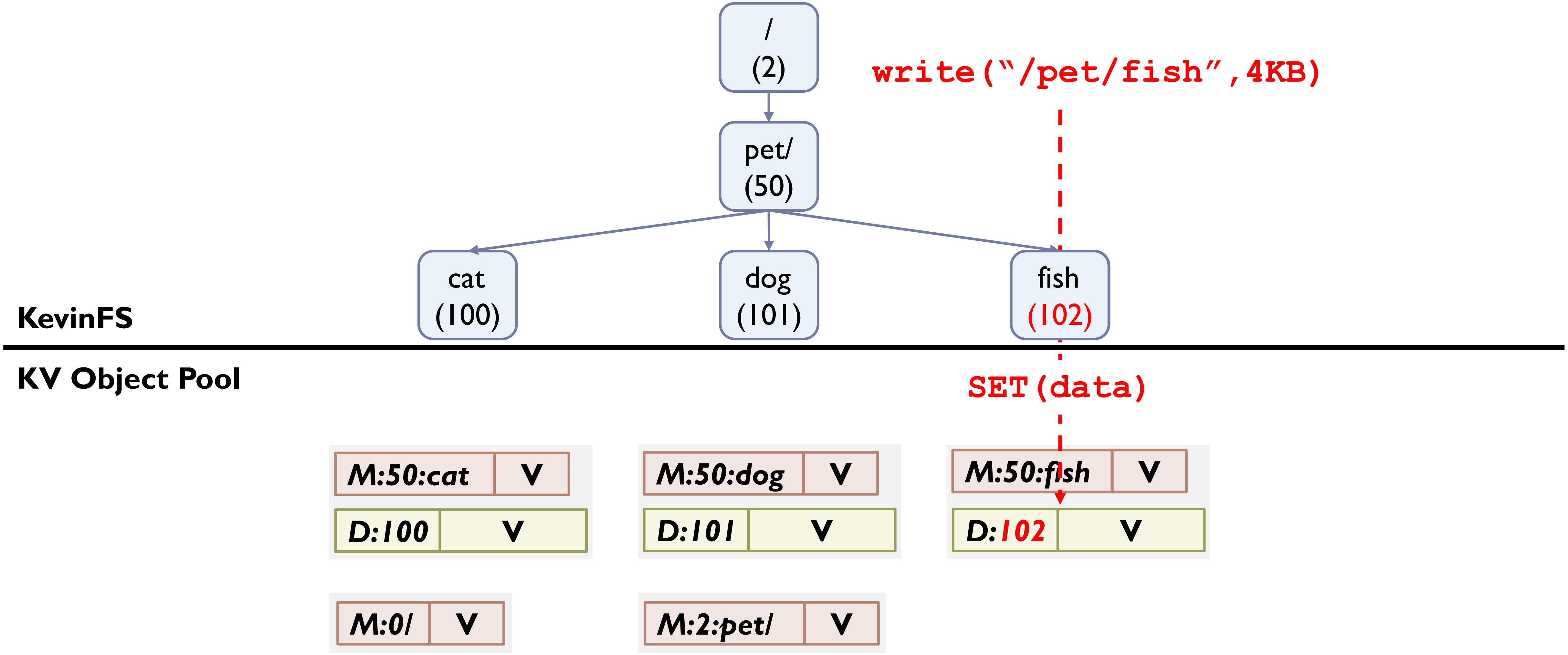
- There are smaller number of I/O requests than block-based file systems because there are no updates on bitmaps, directory entries, and disk pointers

Syscalls	Block	KV
`mkdir()`	`WRITE(bitmap+inode+dir)`	`SET(meta)`
`creat()`	`WRITE(bitmap+inode+dir)`	`SET(meta)`
`write()`	`WRITE(bitmap+ptr+data)`	`SET(data)`
`rmdir()`	`WRITE(bitmap+ptr+dir)`	`DELETE(meta)`
`lookup()`	`READ(ptr+dir+inode)`	`GET(meta)`
`read()`	`READ(ptr+data)`	`GET(data)`
`readdir()`	`READ(ptr+dir+inode)`	`ITERATE(meta)`

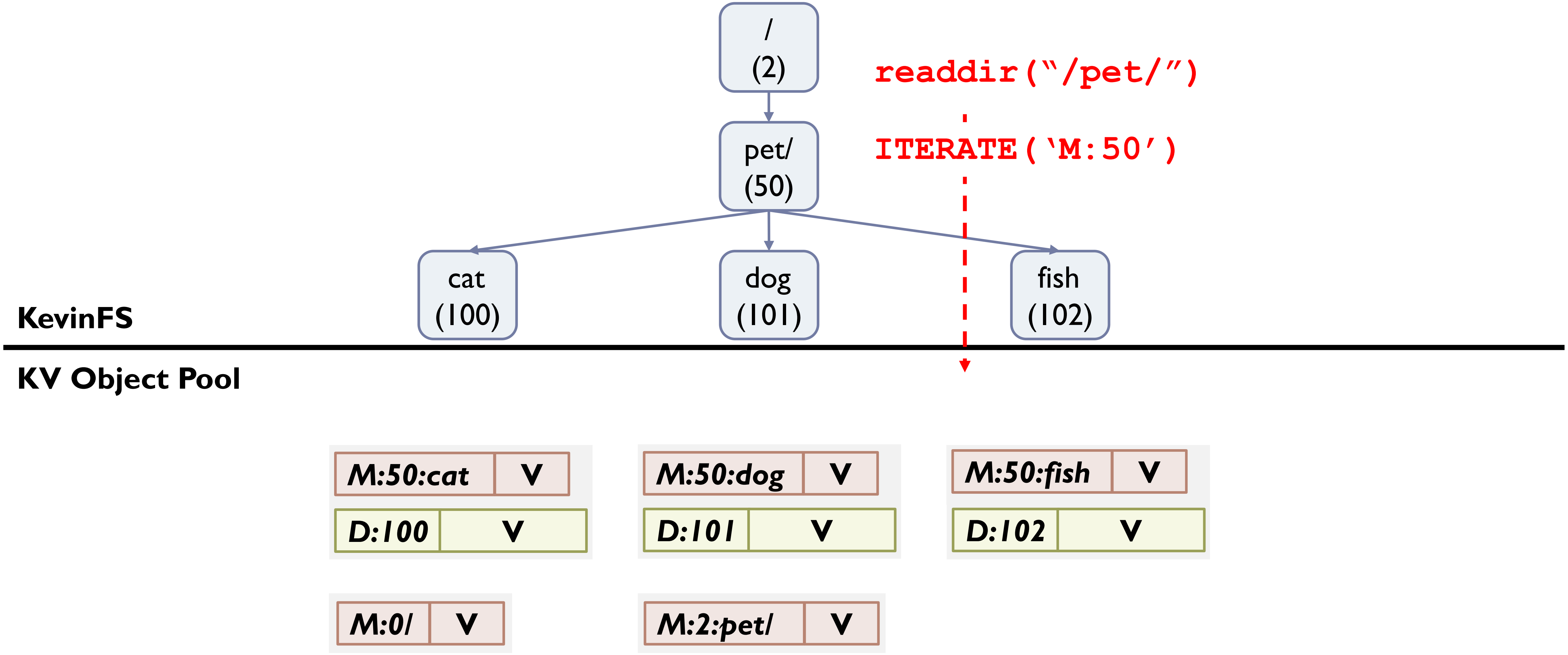
Operation-to-KV translation – Example



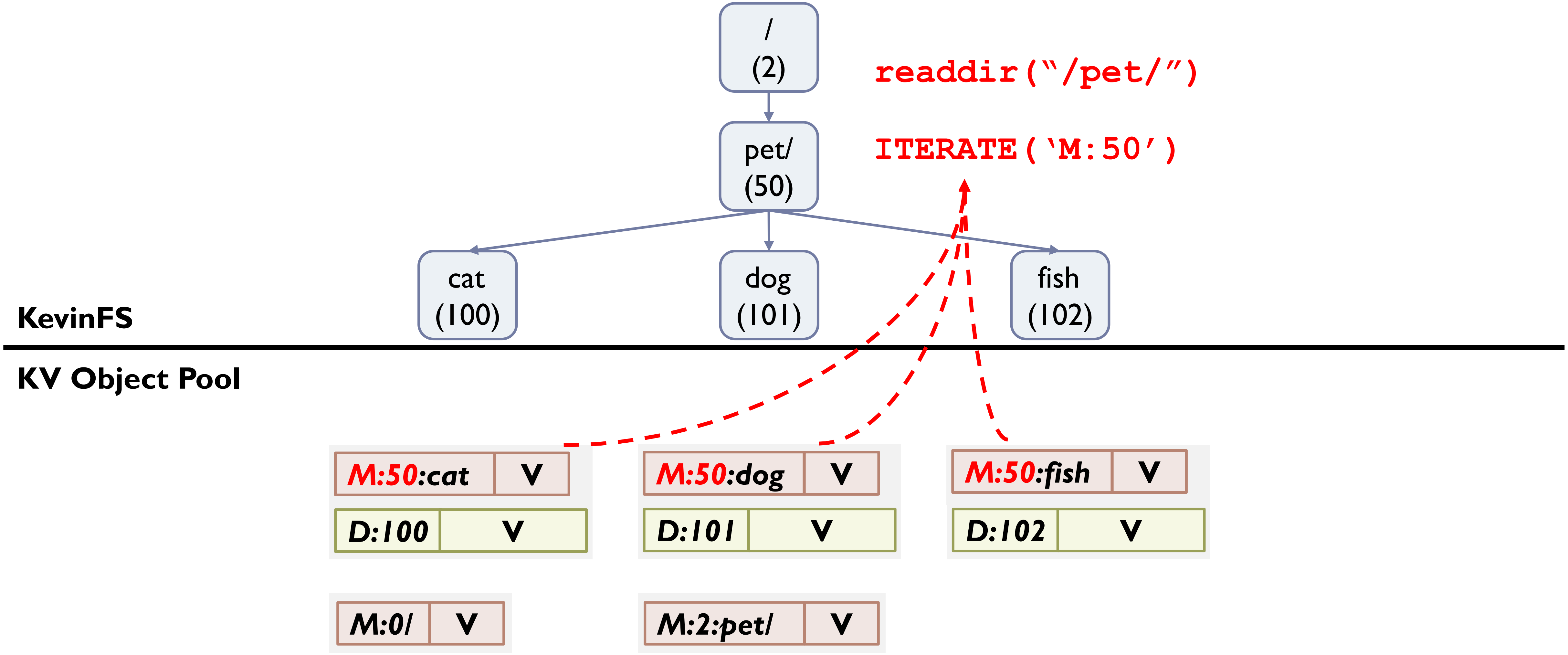
Operation-to-KV translation – Example



Operation-to-KV translation – Example



Operation-to-KV translation – Example



`readdir("/pet/")`

`ITERATE('M:50')`

Operation-to-KV translation – Advantages

- KevinFS performs file operations on meta and data objects only
  - Modifications on bitmaps, dir. entry, and disk pointers are unnecessary
  - Metadata traffic between the file system and the device decreases greatly

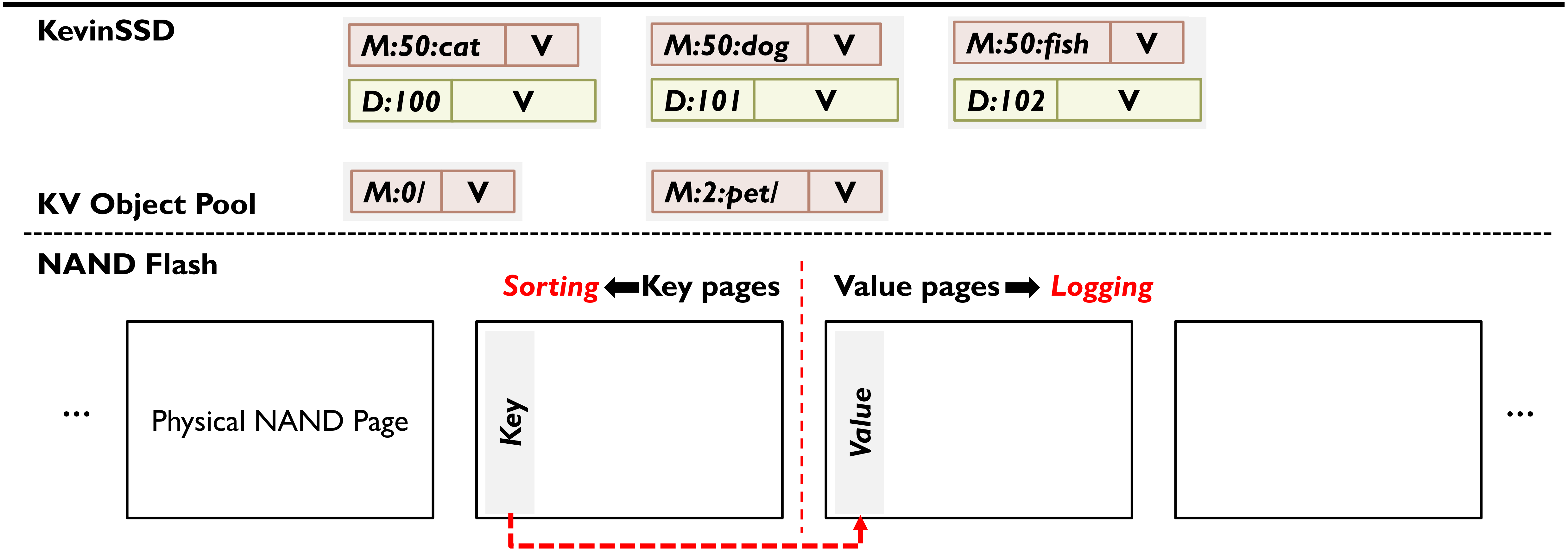
Checklist

- Reducing **Meta I/O Amp**.? ✓
- Resilient to **Fragmentation**?
- Reducing **Journaling I/O**?

How are KV objects mapped to NAND flash pages?

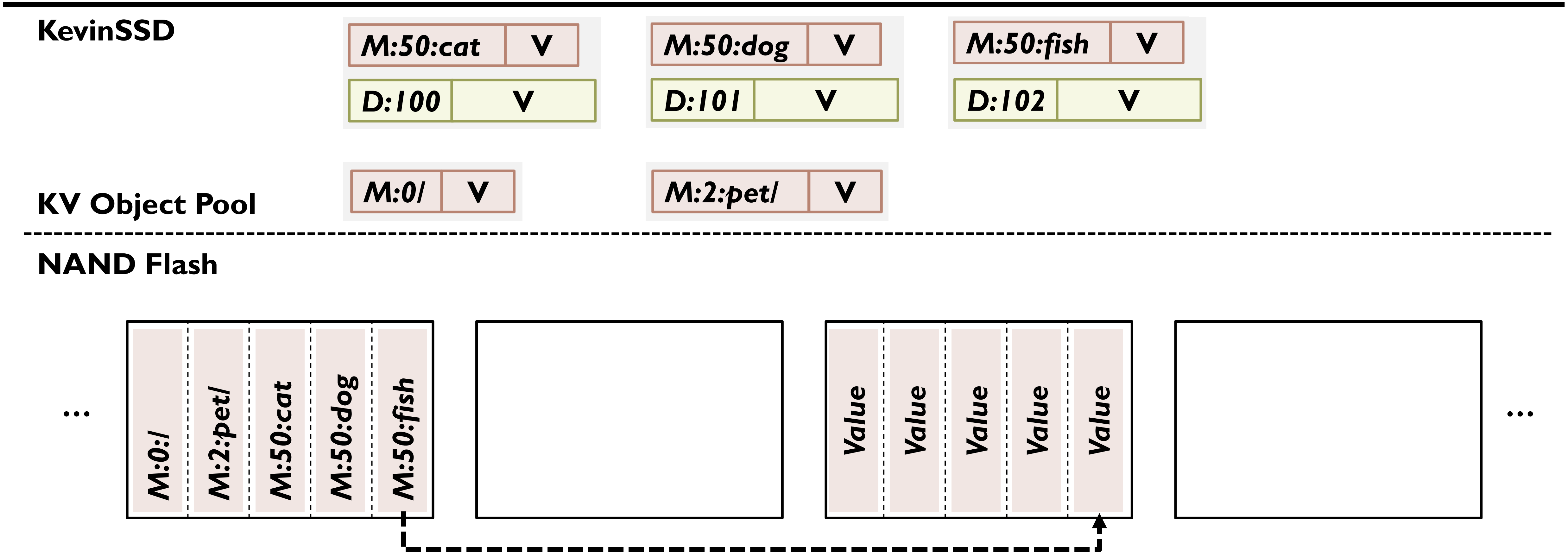
KV object indexing algorithm

- LSM-tree with KV separation
  - Keys are sorted in key pages, and values are logged in value pages



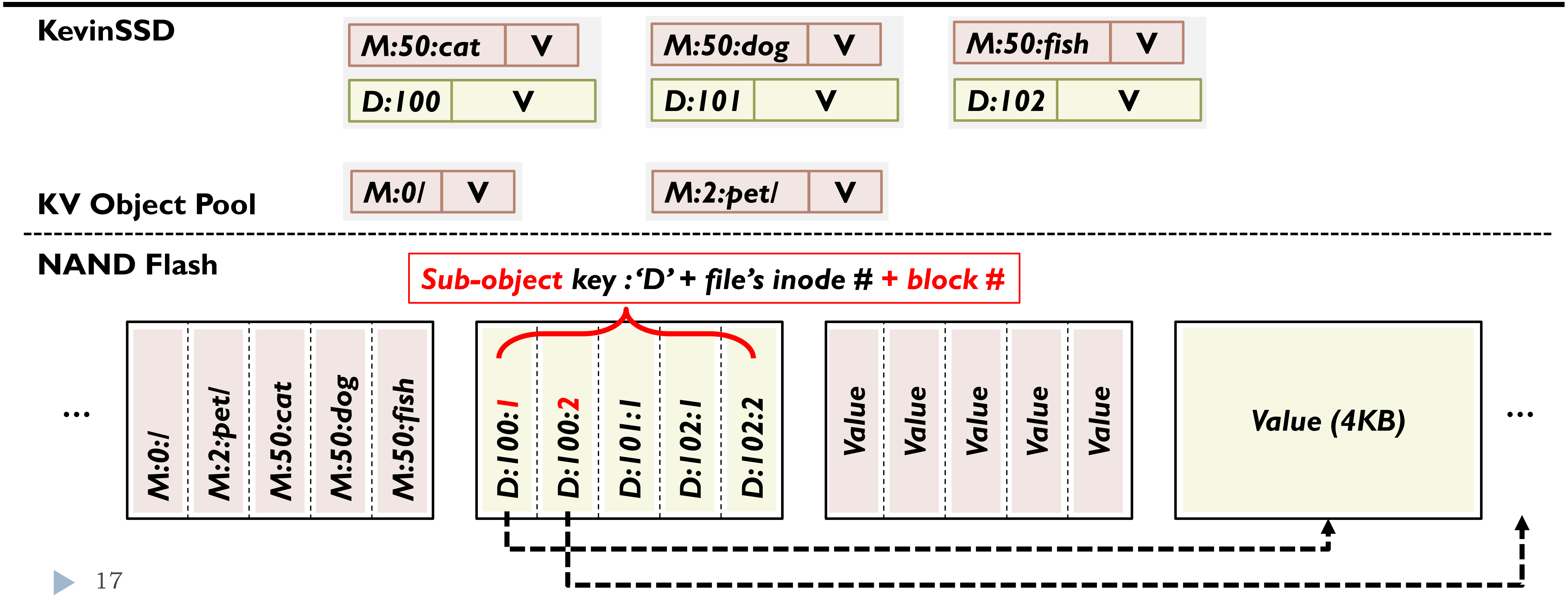
KV object indexing algorithm (Cont.)

- Manage meta and data objects separately



KV object indexing algorithm (Cont.)

- Manage meta and data objects separately
  - ***sub-objects*** for a data object



KevinSSD

M:50:cat V
D:100 V
M:50:dog V
D:101 V
M:50:fish V
D:102 V

KV Object Pool

M:0/ V
M:2:pet/ V

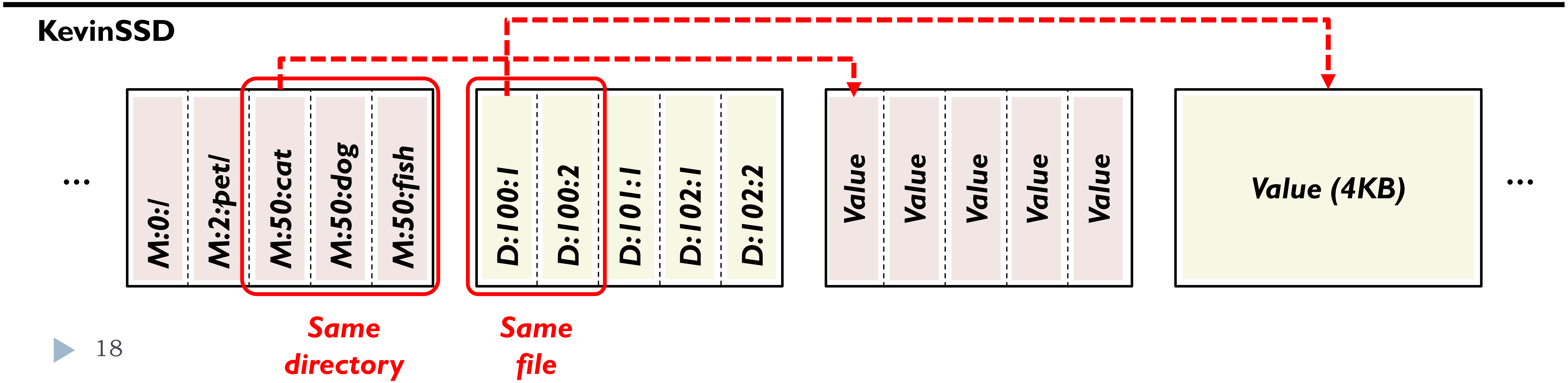
NAND Flash

Sub-object key : ***'D' + file's inode # + block #***

…
M:0:/
M:2:pet/
M:50:cat
M:50:dog
M:50:fish
D:100:1
D:100:2
D:101:1
D:102:1
D:102:2
Value
Value
Value
Value
Value
Value (4KB)
…

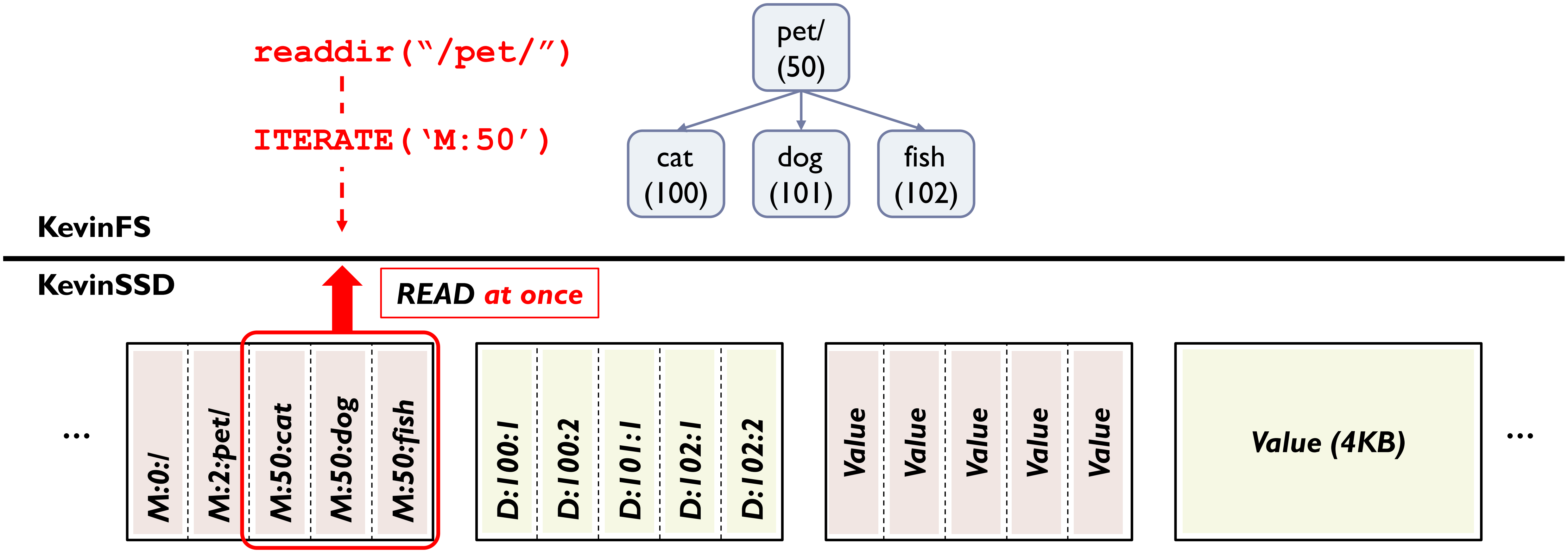
KV object indexing algorithm (Cont.)

- Meta object keys are sorted by parent's inode number ≈ *directory entries*
- Data object keys are sorted by file's inode number ≈ *disk pointers*



KV object indexing algorithm (Cont.)

- Process KV commands efficiently thanks to the sorted keys



`readdir("/pet/")`

`ITERATE('M:50')`

KevinFS

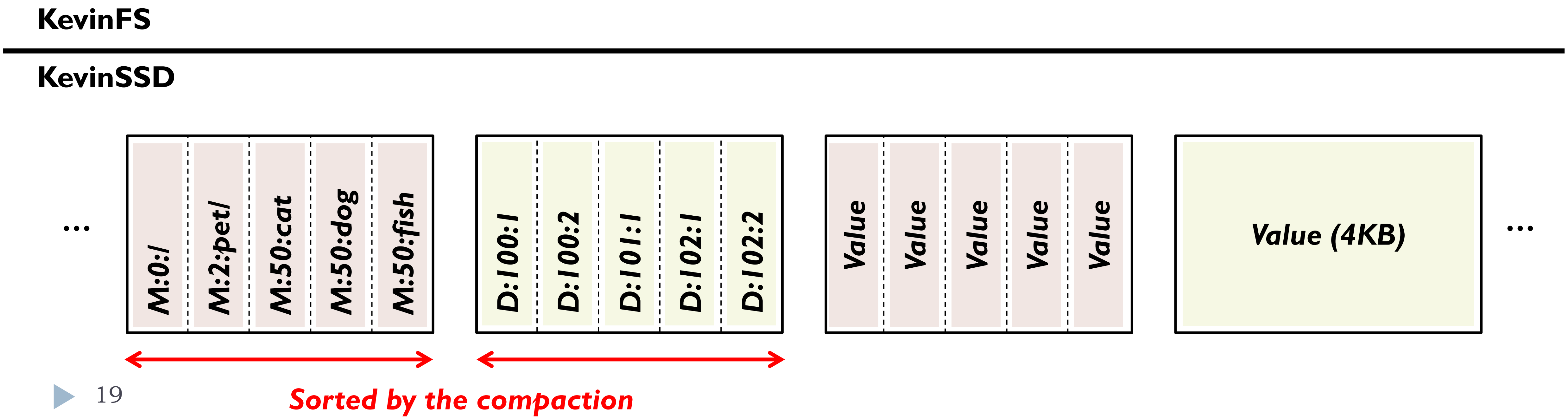
KevinSSD

READ at once

Same directory

KV object indexing algorithm (Cont.)

- Process KV commands efficiently thanks to the sorted keys
  - The ***compaction*** operation of LSM-tree sorts keys periodically



KV object indexing algorithm – Advantages

- KevinSSD performs FS-friendly KV object mapping
  - Exploit locality of file system requests with fewer NAND flash accesses
    - Same directory – Meta object keys (directory entries) are sorted
    - Same file – Data object keys (disk pointers) are sorted
  - All keys are automatically sorted by the LSM-tree compaction
    - Resilient to file system aging and metadata fragmentation

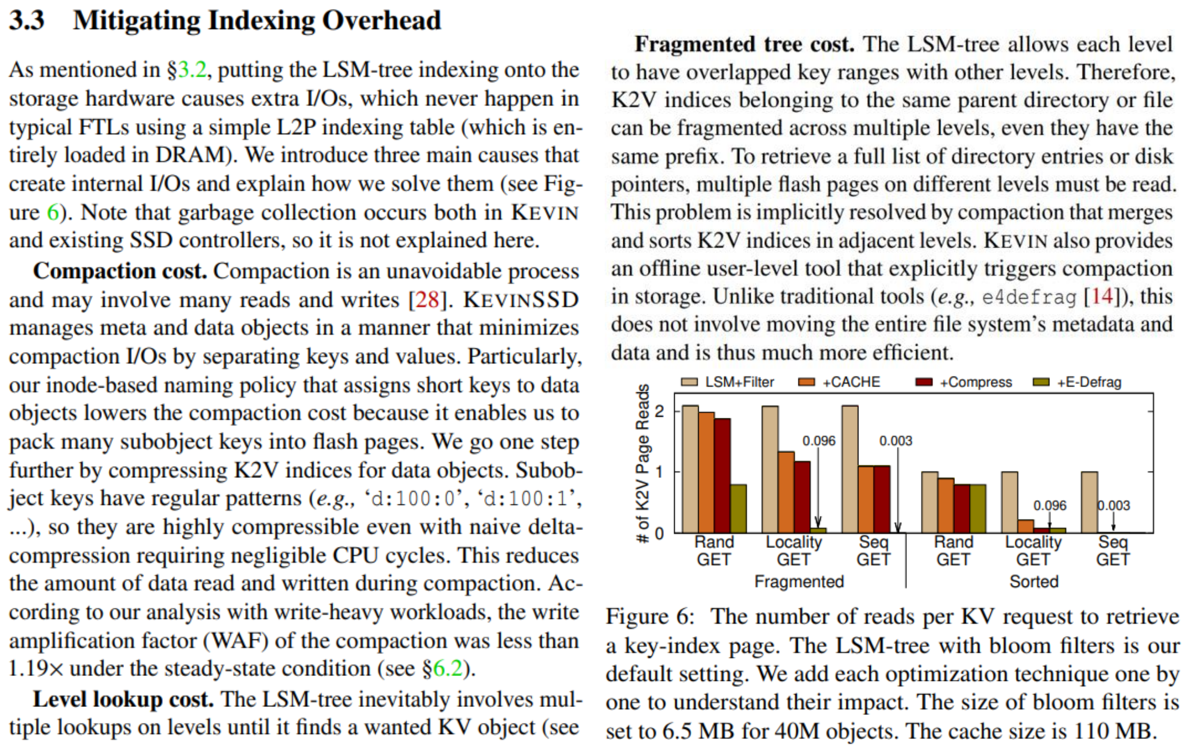
Checklist

- Reducing **Meta I/O Amp**.? ✓
- Resilient to **Fragmentation**? ✓
- Reducing **Journaling I/O**?

Q: Indexing overhead of LSM-tree?

- Overhead of LSM-tree based indexing algorithm
  - Compaction cost → **KV separation + Key compression**
  - Level lookup cost → **Filtering + Caching + Key compression**
  - Fragmented tree cost → **Compaction + Offline-defragmentation**

Please refer to our paper



3.3 Mitigating Indexing Overhead

As mentioned in §3.2, putting the LSM-tree indexing onto the storage hardware causes extra I/Os, which never happen in typical FTLs using a simple L2P indexing table (which is entirely loaded in DRAM). We introduce three main causes that create internal I/Os and explain how we solve them (see Figure 6). Note that garbage collection occurs both in KEVIN and existing SSD controllers, so it is not explained here.

Compaction cost. Compaction is an unavoidable process and may involve many reads and writes [28]. KEVINSSD manages meta and data objects in a manner that minimizes compaction I/Os by separating keys and values. Particularly, our inode-based naming policy that assigns short keys to data objects lowers the compaction cost because it enables us to pack many subobject keys into flash pages. We go one step further by compressing K2V indices for data objects. Subobject keys have regular patterns (*e.g.*, '`d:100:0`', '`d:100:1`', ...), so they are highly compressible even with naive delta-compression requiring negligible CPU cycles. This reduces the amount of data read and written during compaction. According to our analysis with write-heavy workloads, the write amplification factor (WAF) of the compaction was less than 1.19× under the steady-state condition (see §6.2).

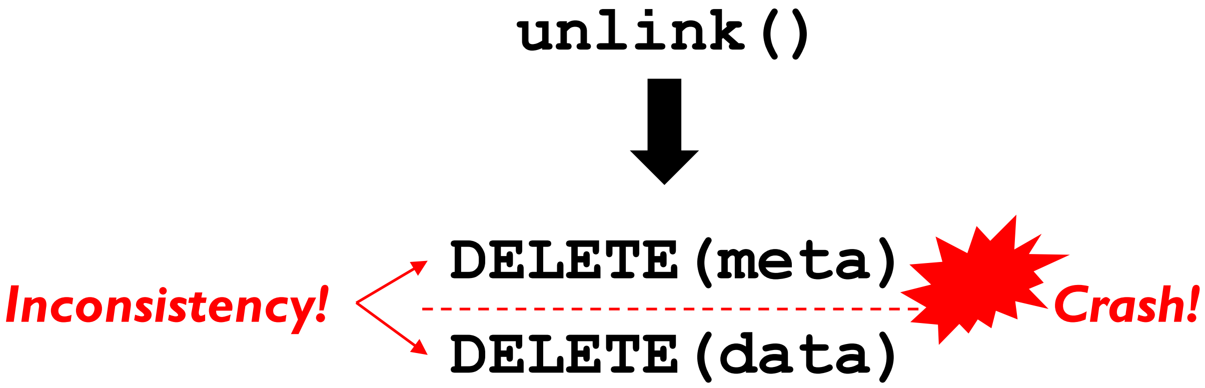
Level lookup cost. The LSM-tree inevitably involves multiple lookups on levels until it finds a wanted KV object (see

Fragmented tree cost. The LSM-tree allows each level to have overlapped key ranges with other levels. Therefore, K2V indices belonging to the same parent directory or file can be fragmented across multiple levels, even they have the same prefix. To retrieve a full list of directory entries or disk pointers, multiple flash pages on different levels must be read. This problem is implicitly resolved by compaction that merges and sorts K2V indices in adjacent levels. KEVIN also provides an offline user-level tool that explicitly triggers compaction in storage. Unlike traditional tools (*e.g.*, `e4defrag` [14]), this does not involve moving the entire file system's metadata and data and is thus much more efficient.

Figure 6: The number of reads per KV request to retrieve a key-index page. The LSM-tree with bloom filters is our default setting. We add each optimization technique one by one to understand their impact. The size of bloom filters is set to 6.5 MB for 40M objects. The cache size is 110 MB.

Transaction management

- Lightweight crash consistency with in-storage transaction support
  - KevinSSD provides transaction APIs: Atomicity & Durability over multiple KV objects
  - KevinFS performs a syscall atomically by wrapping KV commands: FS consistency



unlink()

DELETE(meta)

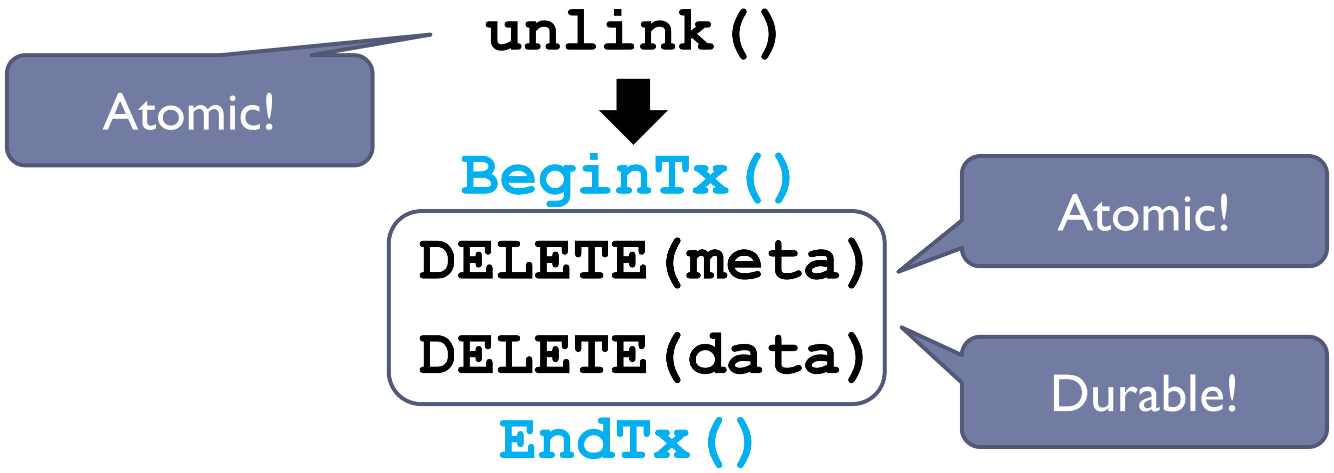
DELETE(data)

Inconsistency!

Crash!

Transaction management

- Lightweight crash consistency with in-storage transaction support
  - KevinSSD provides transaction APIs: Atomicity & Durability over multiple KV objects
  - KevinFS performs a syscall atomically by wrapping KV commands: FS consistency



Transaction management

- Lightweight crash consistency with in-storage transaction support
  - KevinSSD provides transaction APIs: Atomicity & Durability over multiple KV objects
  - KevinFS performs a syscall atomically by wrapping KV commands: FS consistency
  - KevinFS does not perform journaling for crash consistency

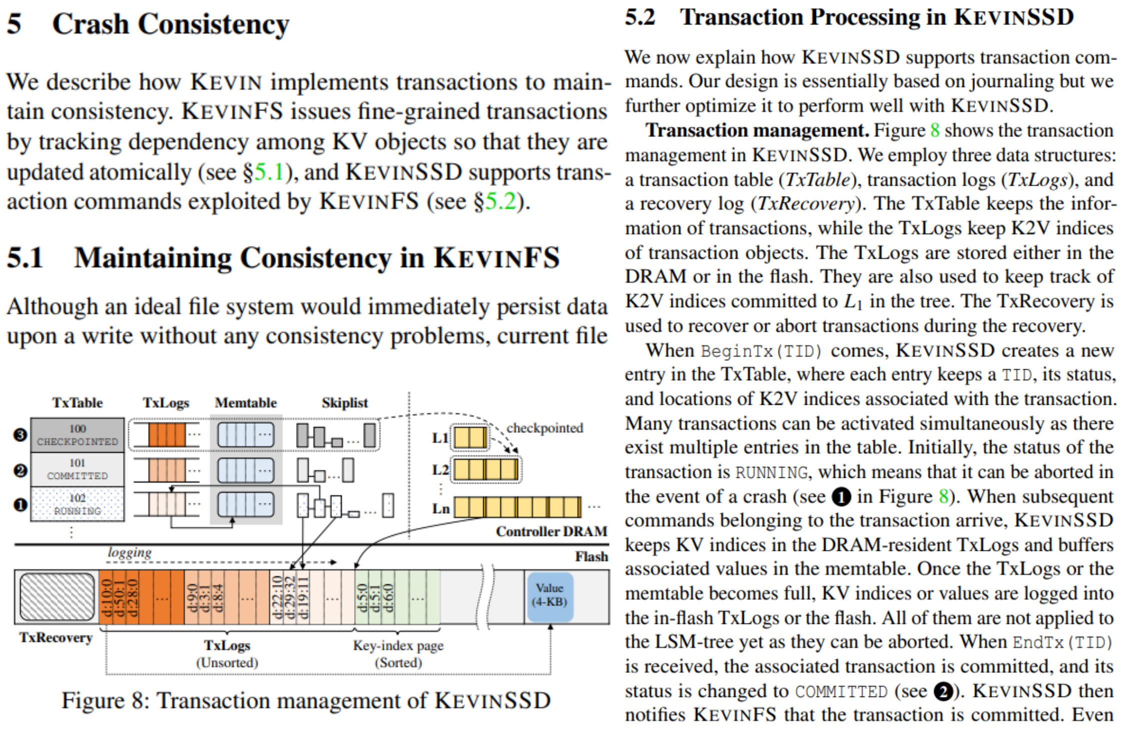
Checklist

- Reducing **Meta I/O Amp.**? ✓
- Resilient to **Fragmentation**? ✓
- Reducing **Journaling I/O**? ✓

Transaction management

- Lightweight crash consistency with in-storage transaction support
  - KevinSSD provides transaction APIs: Atomicity & Durability over multiple KV objects
  - KevinFS performs a syscall atomically by wrapping KV commands: FS consistency
  - KevinFS does not perform journaling for crash consistency

Please refer to our paper



5 Crash Consistency

We describe how KEVIN implements transactions to maintain consistency. KEVINFS issues fine-grained transactions by tracking dependency among KV objects so that they are updated atomically (see §5.1), and KEVINSSD supports transaction commands exploited by KEVINFS (see §5.2).

5.1 Maintaining Consistency in KEVINFS

Although an ideal file system would immediately persist data upon a write without any consistency problems, current file

Figure 8: Transaction management of KEVINSSD

5.2 Transaction Processing in KEVINSSD

We now explain how KEVINSSD supports transaction commands. Our design is essentially based on journaling but we further optimize it to perform well with KEVINSSD.

Transaction management. Figure 8 shows the transaction management in KEVINSSD. We employ three data structures: a transaction table (*TxTable*), transaction logs (*TxLogs*), and a recovery log (*TxRecovery*). The TxTable keeps the information of transactions, while the TxLogs keep K2V indices of transaction objects. The TxLogs are stored either in the DRAM or in the flash. They are also used to keep track of K2V indices committed to L_1 in the tree. The TxRecovery is used to recover or abort transactions during the recovery.

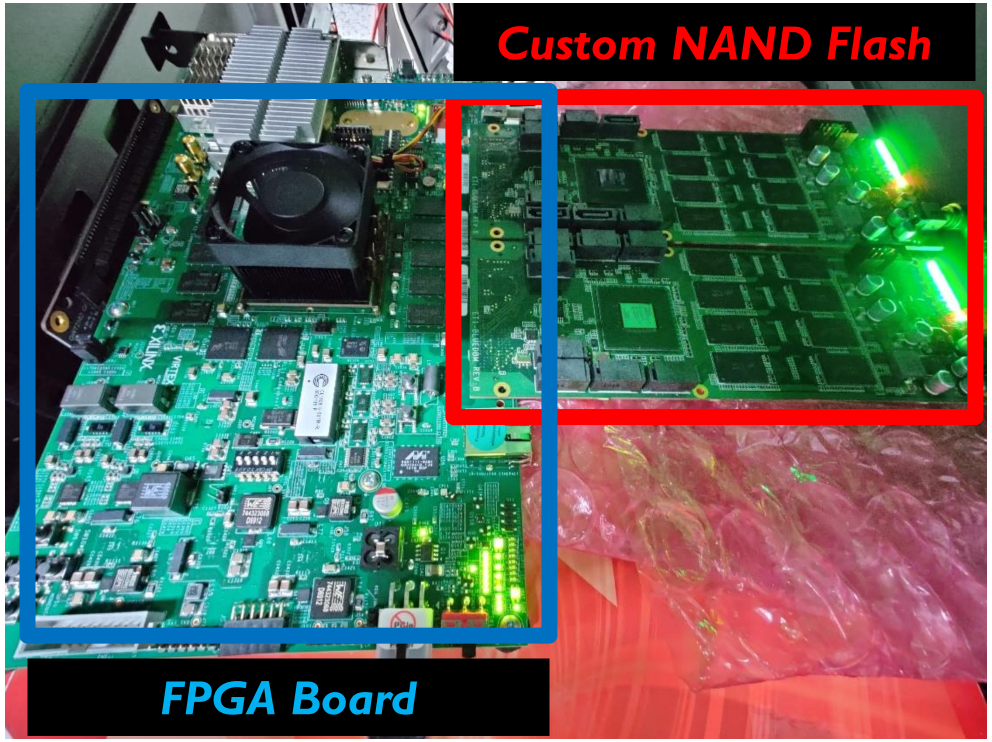
When `BeginTx(TID)` comes, KEVINSSD creates a new entry in the TxTable, where each entry keeps a `TID`, its status, and locations of K2V indices associated with the transaction. Many transactions can be activated simultaneously as there exist multiple entries in the table. Initially, the status of the transaction is `RUNNING`, which means that it can be aborted in the event of a crash (see ❶ in Figure 8). When subsequent commands belonging to the transaction arrive, KEVINSSD keeps KV indices in the DRAM-resident TxLogs and buffers associated values in the memtable. Once the TxLogs or the memtable becomes full, KV indices or values are logged into the in-flash TxLogs or the flash. All of them are not applied to the LSM-tree yet as they can be aborted. When `EndTx(TID)` is received, the associated transaction is committed, and its status is changed to `COMMITTED` (see ❷). KEVINSSD then notifies KEVINFS that the transaction is committed. Even

INDEX

- Introduction
- Kevin
- Experiments
- Conclusion

Kevin prototype

- KevinFS
  - Implemented in Linux v4.15.18 kernel
- KevinSSD
  - Based on Xilinx-VCU108 with custom flash cards



Objectives

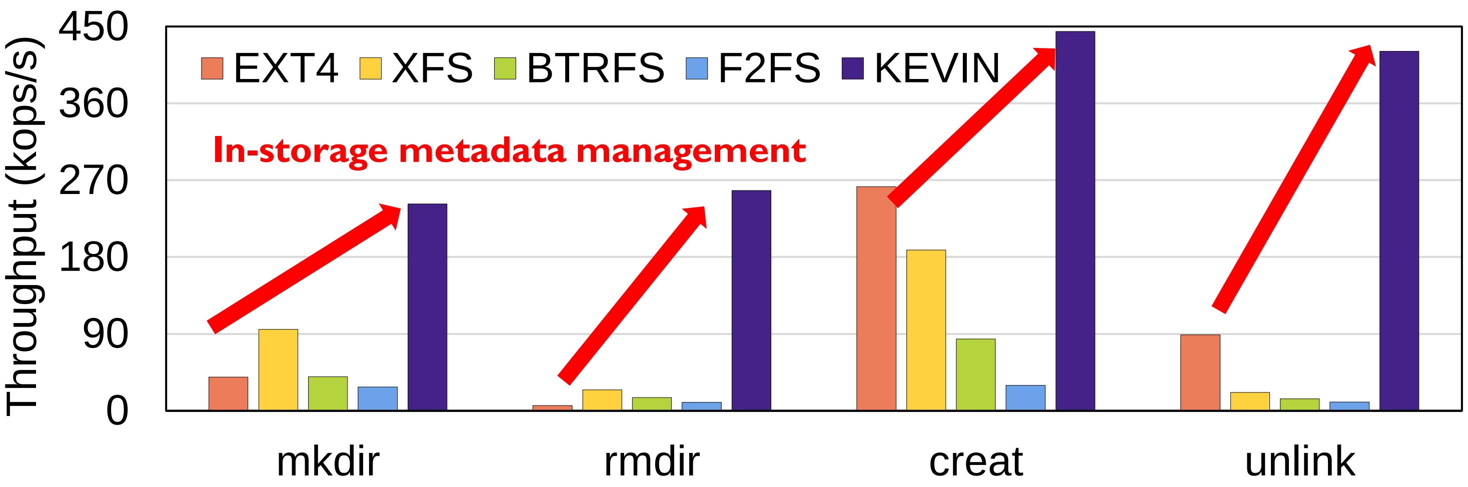
(1) How does Kevin perform under metadata-intensive workloads?

(2) Is Kevin resilient to fragmentation?

(3) How does Kevin perform under fsync-intensive workloads?

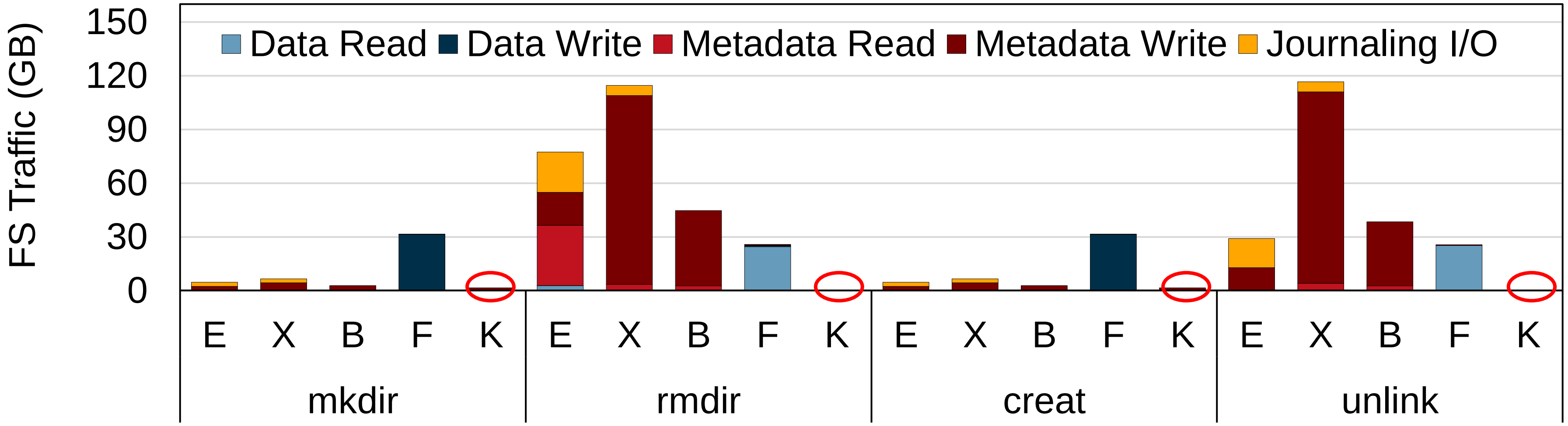
Please check our paper for more results!

Experimental results: metadata-intensive



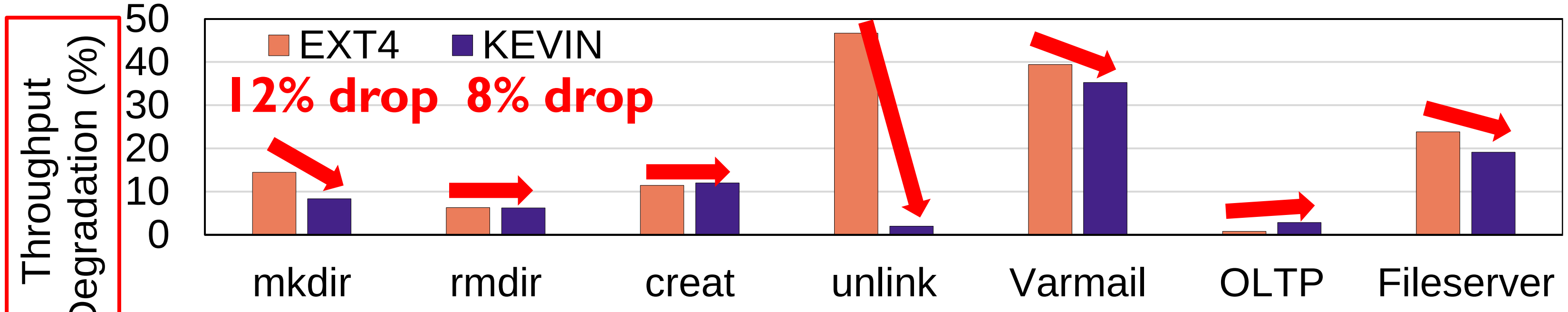
In-storage metadata management

Performance **x6.2 higher on average**

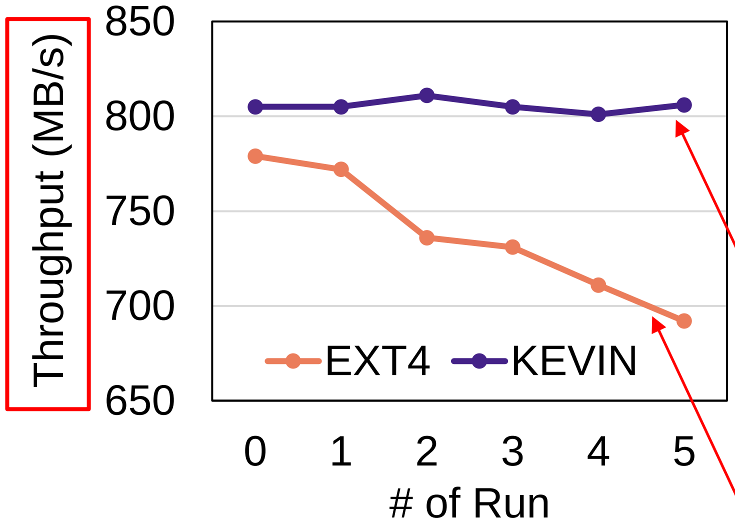


FS traffic **74% lower traffic**

Experimental results: aged file system



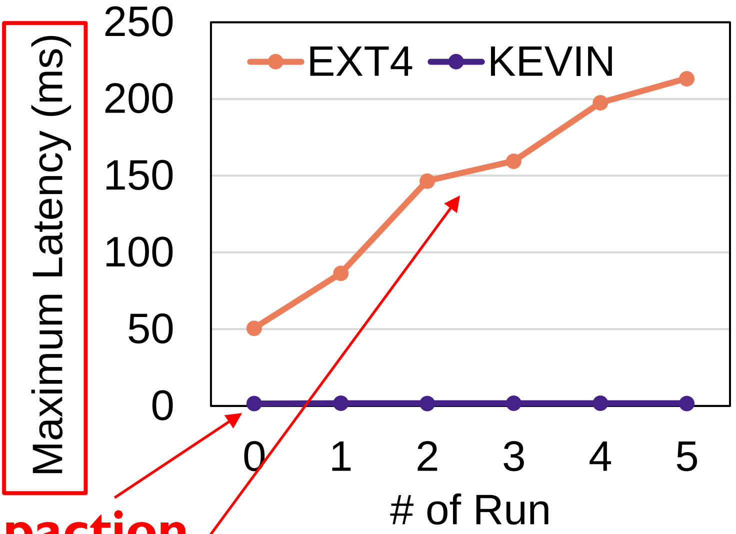
Write-dominant workloads



Sequential read

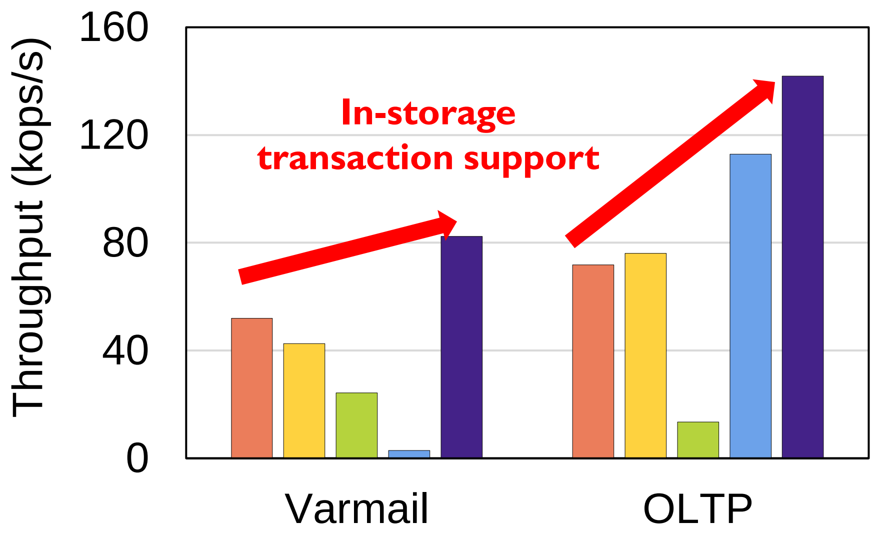
Compaction

Fragmentation



Readdir

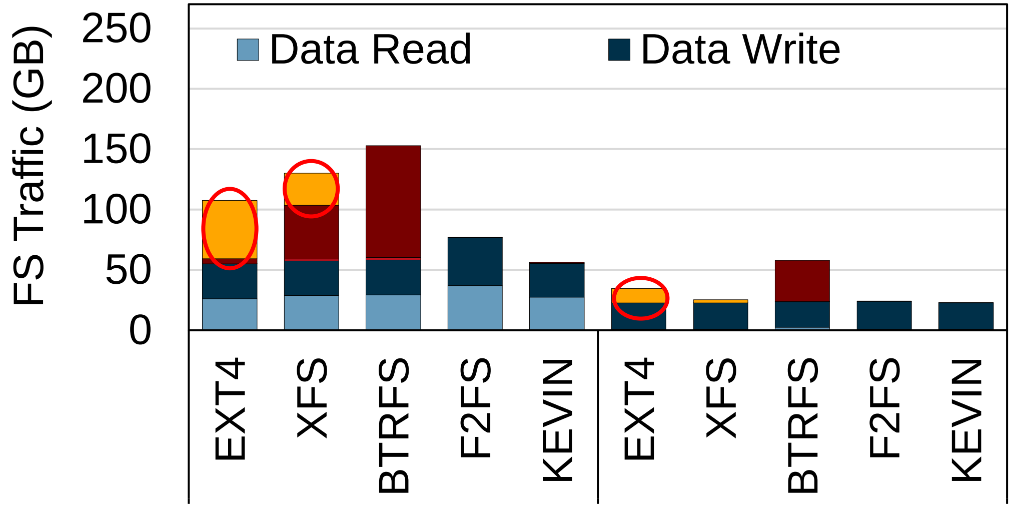
Experimental results: fsync-intensive



In-storage transaction support

Performance

68% higher on average



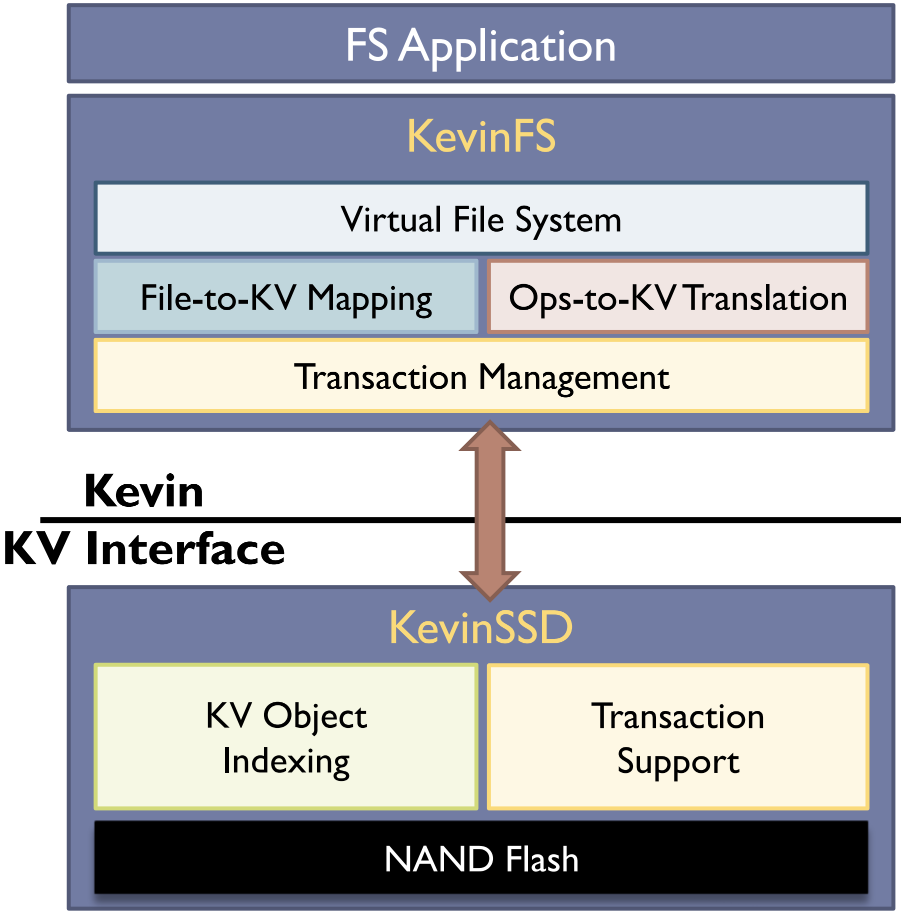
FS traffic

No FS journaling I/O

Conclusion

- Traditional file systems cannot exploit fast storage performance because of fundamental limitations of the block interface

- Kevin
  - Offloading indexing capability to the storage device
  - VFS-to-KV translation
  - In-storage KV indexing
  - Transaction support



Thank You !

Jinhyung Koo (jhk5361@dgist.ac.kr)