Breaking Apart the VFS for Managing File Systems

Kuei (Jack) Sun, Matthew Lakier, Angela Demke Brown, and Ashvin Goel

University of Toronto

HotStorage ‘18, Boston, MA.

File-System Management Applications

- Used by system administrators
  - To maintain, optimize, and administer their file systems
- Examples
  - Defragmentation Tool
  - File System Resizing Tool
  - Garbage Collector
  - File System Aware Data Scrubber
  - File System Upgrade Tool
- Essential for successful deployment of file system

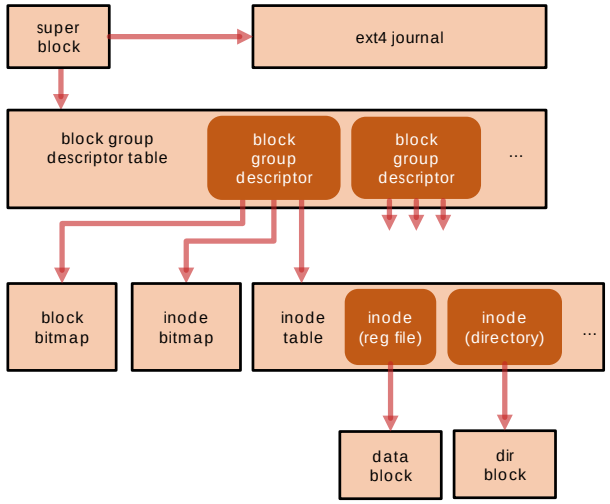
Problem

- These applications operate directly on file system structures
  - E.g., a defragmentation tool moves extents of a fragmented file
- Development requires significant engineering effort
  - Applications have to be developed from scratch for each file system
    - Each file system has its own set of data structures

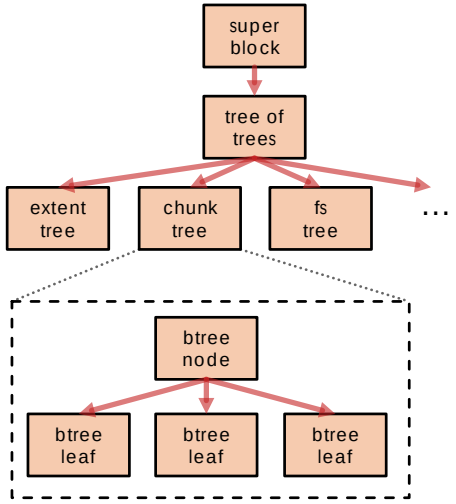
Problem

- These applications operate directly on file system structures
  - E.g., a defragmentation tool moves extents of a fragmented file
- Development requires significant engineering effort
  - Applications have to be developed from scratch for each file system
    - Each file system has its own set of data structures

Ext4 File System



Btrfs File System



Problem

- These applications operate directly on file system structures
  - E.g., a defragmentation tool moves extents of a fragmented file
- Development requires significant engineering effort
  - Applications have to be developed from scratch for each file system
    - Each file system has its own set of data structures
  - Need detailed knowledge of file system format
    - To identify and interpret file system structures
  - File system format is complex and poorly documented
    - Developed by experts
  - Emerging file systems frequently lack these applications
    - Impedes adoption

Goals and Challenges

- Goal
  - Provide generic API for building file system management applications
    - E.g. a defragmentation tool that works for all file systems
- Challenge
  - These applications require fine-grained control over metadata and data
    - E.g. migrate data block to another physical location
  - API needs to provide such control while being generic across file systems

Approach

- Design generic API based on low-level file system abstractions
- Applications operate on abstract file system objects
  - Blocks or extents, inodes, directory entries
- These objects are accessed via abstract operations
  - E.g., Allocate, free, iterate, map
- Example: defragmentation tool
  - Finds fragmented blocks of a file
    - Iterates through logical to physical block mappings
  - Relocates them to contiguous extent
    - Requires allocation of contiguous extent and remapping of the file

eVFS Operations

FS Object	Operations	Description
Global	super_make	Makes a new file system (i.e, *mkfs*)
	super_set	Update file system settings
Extents	extent_alloc / extent_free	Allocate or free an extent
	extent_iterate	Iterate through all extents used by inode
	freesp_iterate	Iterate through all free space extents
	extent_reverse	Returns all inode that maps to extent
Inodes	inode_alloc / inode_free	Allocate or free an inode
	inode_read / inode write	Same as VFS read/write, required for accessing “mangled” data
	inode_map	Maps physical extent to logical file offset
	inode_iterate	Iterate through all allocated inodes

eVFS Operations

FS Object	Operations	Description
Directory Entries	dirent_add	Add an entry to a directory inode
	dirent_remove	Remove an entry from directory inode
	dirent_iterate	Iterate through all entries in directory
Transaction	tx_begin	Provide support for crash consistency
	tx_abort	
	tx_commit	

- Crash consistency
  - Protects file system from corruption and data loss
  - Lacking in most file system management applications

eVFS Implementation

- Written in C++ using Spiffy
- Spiffy
  - Library generated from annotated file system data structures
  - Robust parsing and serialization routines for type-safe access

	Lines of Code	
File System	**Read API**	**Write API**
Ext4	666	N/A
Btrfs	583	N/A
F2FS	199	1953

Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Transaction aborts if journal runs out of free space
  - Written in 1350 lines of C code

: old file system

: new file system

: journal

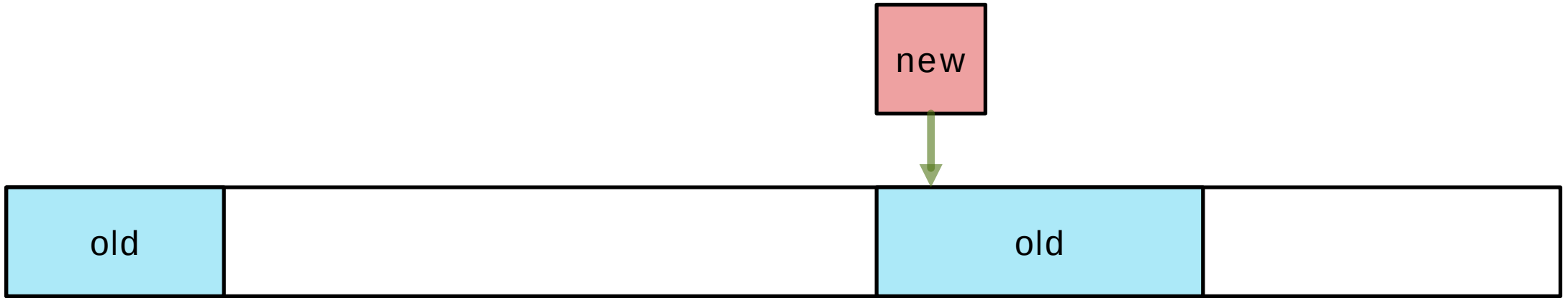
Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code

: old file system

: new file system

: journal



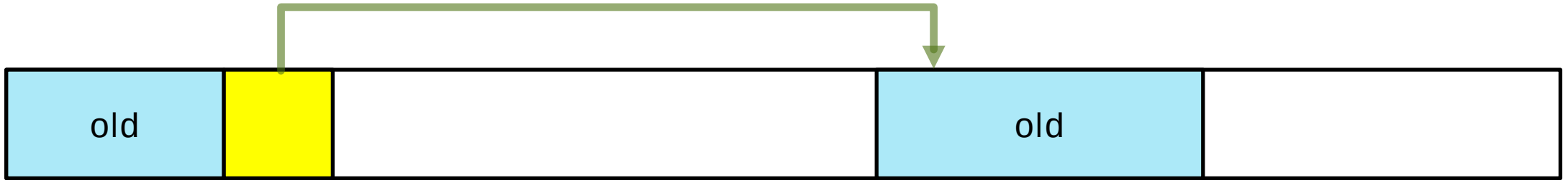
Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code

: old file system

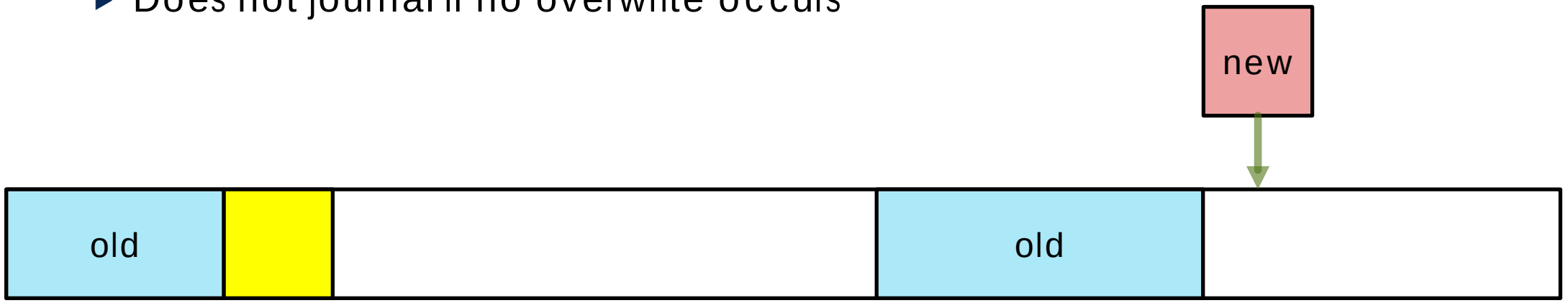
: new file system

: journal



Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code
- Optimization
  - Does not journal if no overwrite occurs



Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code
- Optimization
  - Does not journal if no overwrite occurs

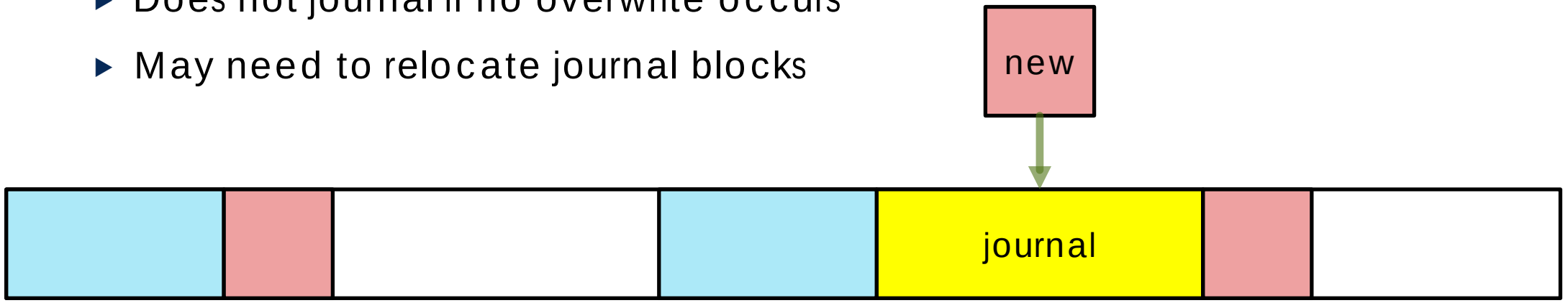
: old file system

: new file system

: journal

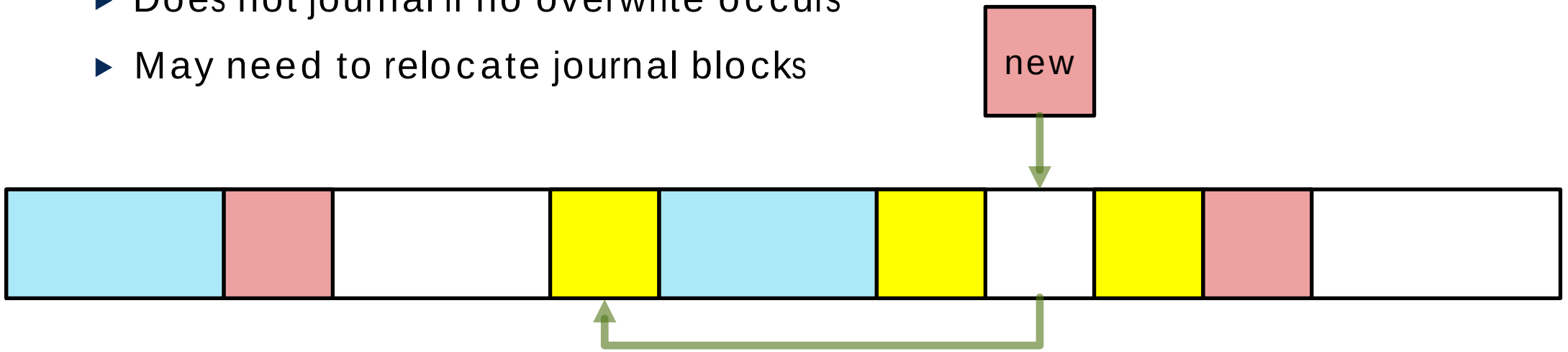
Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code
- Optimization
  - Does not journal if no overwrite occurs
  - May need to relocate journal blocks



Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code
- Optimization
  - Does not journal if no overwrite occurs
  - May need to relocate journal blocks



Journal Implementation

- Variable-sized redo journal
  - Placed in free space of both old and new file system
  - Conversion aborts if journal runs out of free space
  - Written in 1350 lines of C code
- Optimization
  - Does not journal if no overwrite occurs
  - May need to relocate journal blocks

: old file system

: new file system

: journal

File System Conversion Tool

- In-place conversion from one file system to another
  - No backup device necessary
  - Keeps data blocks in original location as much as possible
  - 30 to 50 times faster than copy-based conversion
- Written generically using eVFS
  - Requires only 224 LOC
  - Supports any pair of file systems
- Conversion may result in loss of information
  - E.g. Ext4 does not support immutable snapshots
  - Convert from Btrfs to Ext4 will result in a copy of snapshot

File System Conversion Tool

```
1.  old = fs_open(device, READONLY);
2.  new = fs_open(device, target_fs, UNFORMATTED);
3.  tx = txn_begin(new);
4.  super_make(tx);
5.  for (it = inode_iterate(old); *it != null; it++) {
6.    inode = *it;
7.    i_nr = inode_alloc(tx, inode);
8.    if (i_nr > 0 && S_ISREG(inode.i_mode))
9.      ret = process_regular_file(tx, i_nr, inode);
      …
10.   if (ret < 0) { txn_abort(tx); exit(ret); }
    }
11. txn_commit(tx);
```

File System Conversion Tool

```
   int process_regular_file(tx, i_nr, inode) {
1.   if (inode.i_mangled) {
       /* make a full copy using inode_read/write */
       return 0;
     }
2.   for (it = extent_iterate(inode); *it != null; it++) {
3.     ext = *it;
4.     ret = extent_alloc(tx, ext.phy_nr, ext.size);
       …
5.     ret = inode_map(tx, i_nr, ext.log_nr, ext.phy_nr,
                       ext.size);
       …
     }
  }
```

Benefits of Journaling

- Provides crash consistency
- Small performance overhead of 20%
- Reduces memory overhead of file system conversion tool
  - With journaling, old file system can be read while writing the new file system
  - Without journaling, old file system may be clobbered by new file system
    - Must read entire old file system content into memory before writing new file system

Discussion

- Generic API
- Provides least control

- FS Specific API
- Provides greatest control

VFS

- **File-level API**
- **Supports all file systems**

eVFS

- **File object API**
- **Supports most file systems**

Spiffy

- **Data structure API**
- **Type-safe access**
- **Simplifies traversal**
- **File-system specific**

Manual

- **No API**
- **File-system specific**

Limitations

- Generic API
- Provides least control

- FS Specific API
- Provides greatest control

- File system may only support a subset of eVFS
  - E.g. Ext4 cannot efficiently implement reverse mapping of extent to inodes
- Does not support file-system specific tools
  - E.g. file system checkers operate on file-system specific structures
  - E.g. eVFS does not support Btrfs RAID and volume manager

Future Work

- Online Support for eVFS
  - Current implementation works for offline only
  - Must handle concurrency
- Idea
  - Reuse file system's locking protocols to ensure atomicity
    - Existing applications should not see inconsistent file system state

Conclusion

- Extended Virtual File System Interface (eVFS)
  - Operates on abstract file system objects
  - Supports fine-grained operations
  - Enables file system management applications to be written generically
  - Application works across file systems

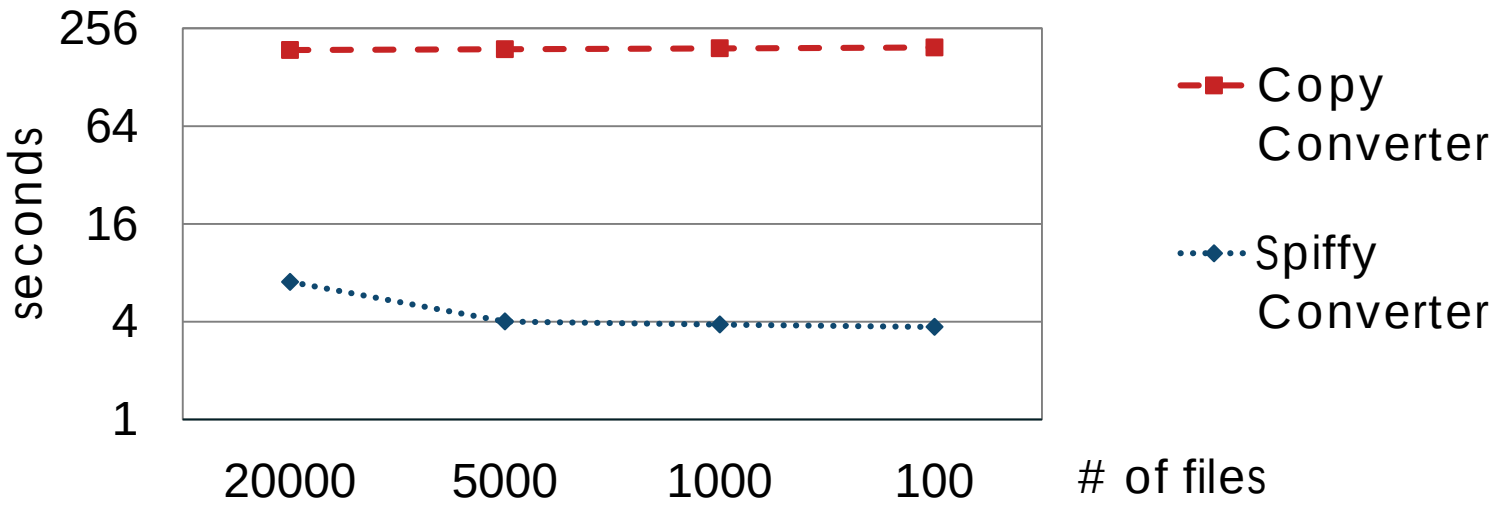
Breaking Apart the VFS for Managing File Systems

Questions?

Presented by Kuei (Jack) Sun

Evaluation

- Compare copy-based converter vs. Spiffy converter
  - Copy converter copies data to local disk, reformat, then copies back
- Converts 64GB file system with 16GB of data on SSD



- Copy converter 30~50 times slower