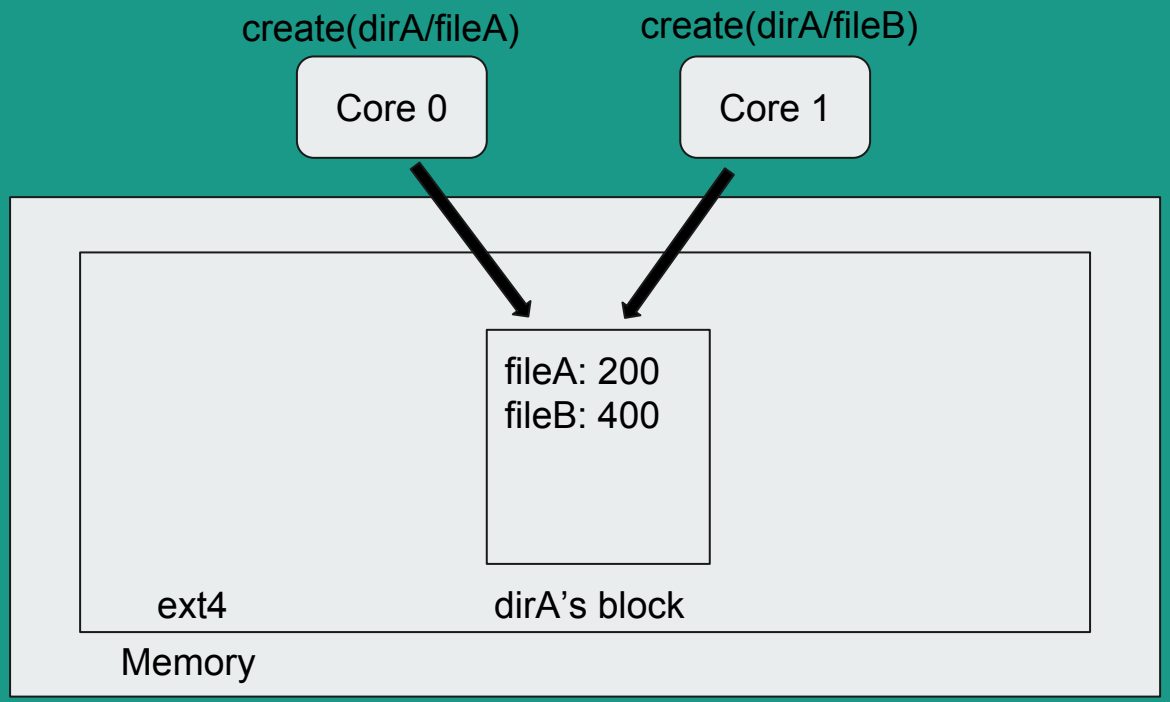
ScaleFS: Scaling a file system to many cores using an operation log

Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich - MIT CSAIL

Presented by: Samarth Kulshreshtha

What's the problem?

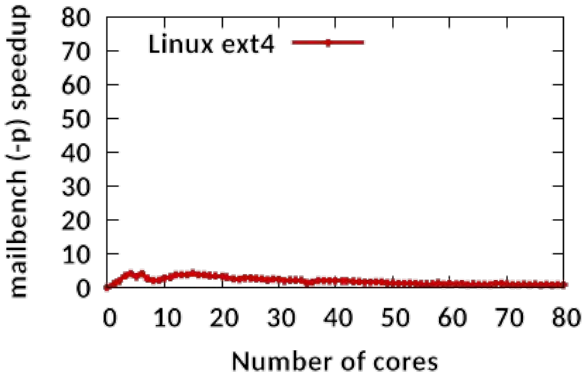


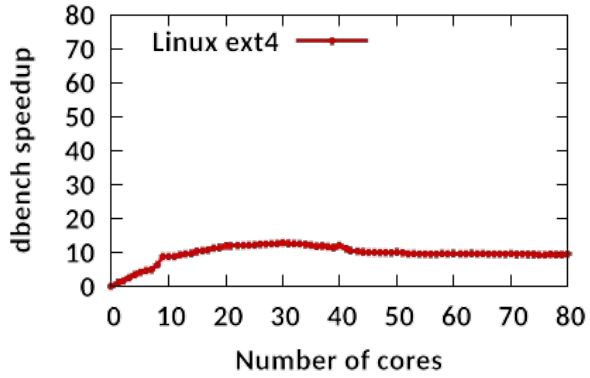
Both cores contend on dirA's block even though these two operations are commutative.

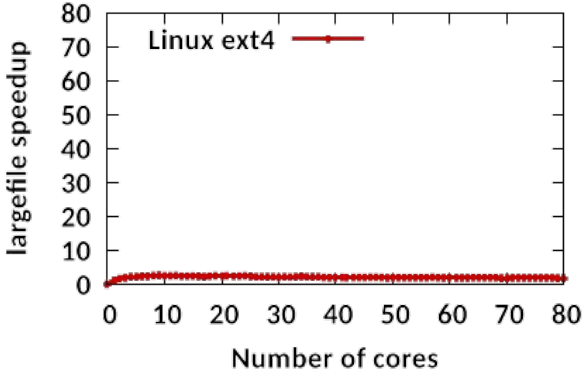
Cache line conflicts -> Scalability issues.

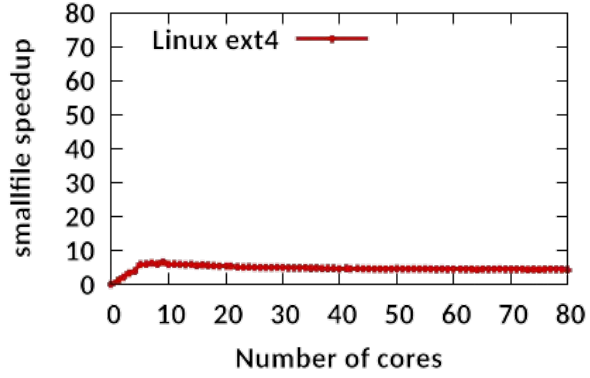
Scalable Commutativity Rule.

Linux ext4 just does not scale with multiple cores









Related work

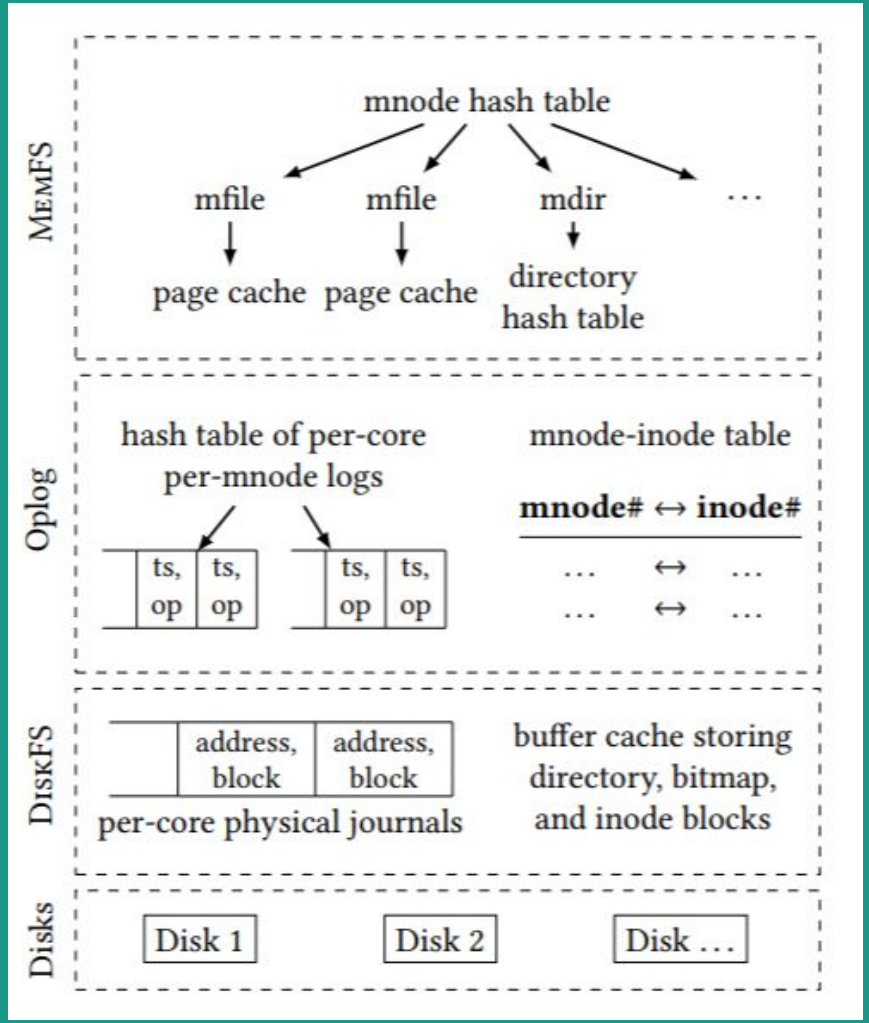
- sv6 = ScaleFS - Crash Safety
- NOVA and iJouranling maintain per-inode logs as compared to per-core logs
- No file system completely decouples the in-memory file system from the on-disk file system
- ReconFS limited to non-volatile memory
- Hare does not provide persistence (only in-memory)
- SpanFS provides persistence to Hare but solves it by distributing files and directories across cores, some operations require two-phase commits

Durability Semantics for *fsync*

- fsync's effects are local
- file system can initiate any fsync operations on its own (OOM)
- rename will not cause a file or directory to be lost
- on-disk data structures must be crash safe

All together, the final semantics of fsync are that it flushes changes to the file or directory being fsynced, and, in the case of fsync on a directory, it also flushes changes to other directories where files may have been renamed to, both to avoid losing files and to maintain internal consistency

System overview



File creation

- MemFS
  - Allocates a fresh mnode number
  - Allocates an mfile structure for the file
  - Adds mfile to the mnode hash table
  - Adds an entry to the directory's hash table
  - Adds a logical operation to the directory's oplog
- Multiple cores can create files concurrently without cache-line conflicts even when creating files in the same directory

fsync

- On a directory:
  - MemFS combines the log entries from all per-core logs for the mnode
  - Sends changes in timestamp order to DiskFS
  - mnodes without inodes
    - Allocate on-disk inodes
    - Update mnode-inode table
- On a file:
  - Scans page cache for dirty bits and write to DiskFS
  - Compare in-memory and on-disk length and update the on-disk file
- Orphans!

Background flush

- Periodically flush in-memory changes to disk by invoking *sync*
  - Iterate over all dirty mfiles (dirty bit per file?) and all mnode oplogs
  - Flush them to disk by invoking fsync
  - Combine them to a single physical transaction in DiskFS (maximum allowed size)

Readdir (list directory contents)

- Enumerate the in-memory hash table
- If not present then look up the inode number and read on-disk representation from DiskFS
- Translate inode numbers of files to mnode numbers using mnode-inode table
- Nodes with no mnode numbers get a new mnode number (not accessed yet)

File read

- Look up corresponding page in mfile's page cache and return contents
- If page not present
  - Lookup inode number and ask DiskFS to read in the data for that inode number from the disk

File write

- Update page cache (possibly getting page from disk like in read)
- Mark page as dirty
- If file length extended then adjust mfile's length

Crash recovery

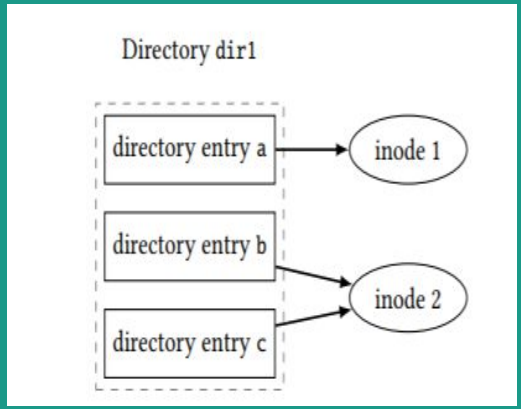
- Post crash, DiskFS recovers the on-disk file system by replaying the on-disk journals after sorting all per-core journals
- Deal with orphan inodes at boot by freeing any inodes with a zero link count

Design Goals!

- Performance [P]
- Correctness [C]

Making operations orderable [P, C]

- MemFS uses lock free-reads which make it difficult to determine the operation order
- Problem:
  - Two threads T1 and T2
  - T1 executes rename(b, c) and T2 executes rename(a, c)
- Solution?
  - Use RDTSCP - timestamp reads are not reordered by the processor
- All directory modifications in the in-memory FS must be linearizable - done by reading timestamp at the appropriate linearization point

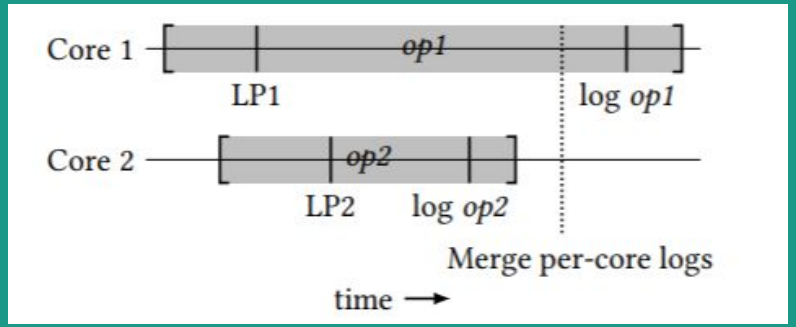


Timestamping lock-free reads [P, C]

- In the previous example rename(b, c)'s linearization point should come first
- How to order lock-free reads with writes?
  - MemFS ensures that read operations happen before any writes in the same operation
  - Use seqlocks when doing a lock-free read and reading timestamp
  - This scales well because it allows read only operations to avoid modifying shared cache-lines

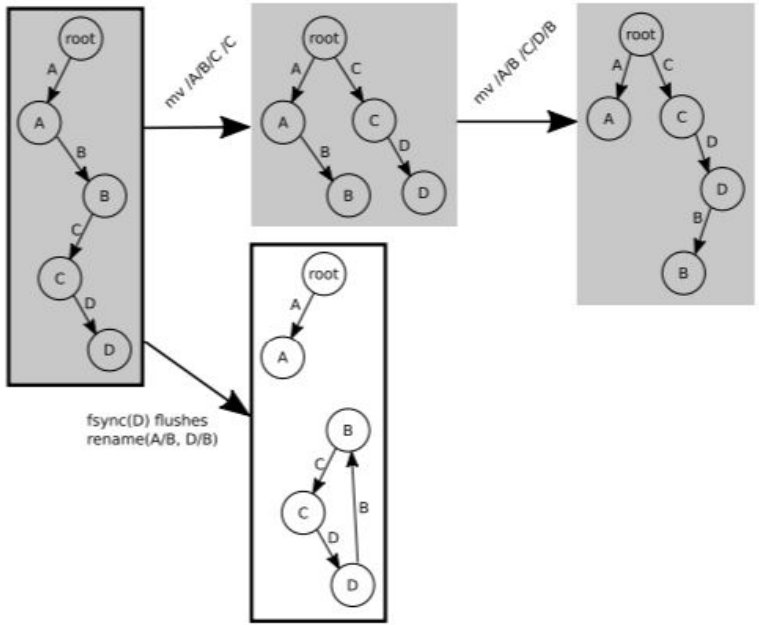
Merging operations [C]

- Timestamps of operations allows operations executed across cores to be ordered
- Per core logs maintained to avoid communication when adding entries
- Merge when fsync or sync is invoked
- In this case op1 will be missing from the log even when LP1 < LP2
- To avoid this, for each core keep track if an operation is currently executing and if so then what is its starting timestamp
- During merge, get timestamp for start of the merge and wait for any running operation



Flushing operation log [P,C]

- Absorption - remove operations that logically cancel each other
- Cross-directory renames to handle moving a file outside a directory and then flushing it
- Internal consistency:
  - Disk links point to initialised nodes on the disk
  - No orphan directories on the disk
  - On-disk FS does not contain a loop after a crash



Implementation

- sv6 (research operating system centered around Scalable Commutativity Rule)
- ScaleFS does not support all FS calls
- Combines oplogged (directory state, file link count) and non-oplogged metadata (file length, modification times) while flushing
- Uses buffer cache to store directory, inode, and bitmap blocks, to speed up read-modify-write operations

ScaleFS component	Lines of C++ code
MemFS (§6.1)	2,458
DiskFS (§6.2)	2,331
MemFS–DiskFS interface	4,094

Figure 5: Lines of C++ code for each component of ScaleFS.

Evaluation

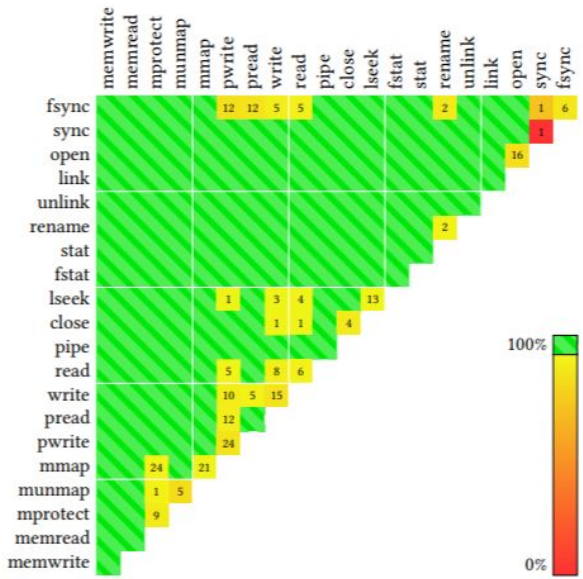


Figure 6: Conflict-freedom of commutative operations in ScaleFS. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 31,551 total test cases generated, 31,317 (99.2%) were conflict-free.

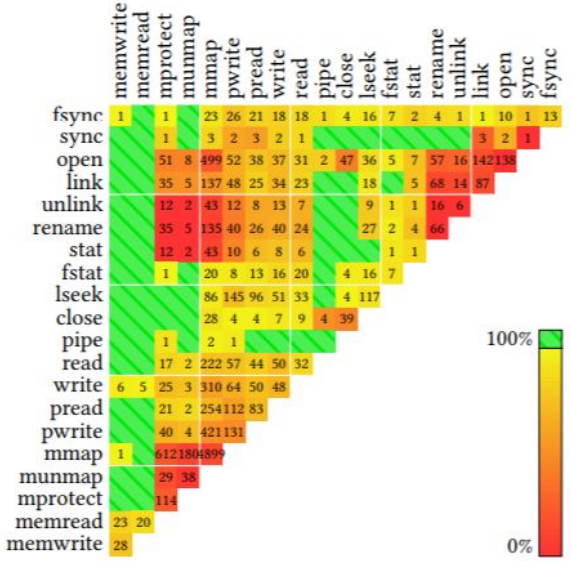


Figure 7: Conflict-freedom of commutative operations in the Linux kernel using an ext4 file system. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 31,551 total test cases generated, 20,394 (65%) were conflict-free.

Does durability reduce conflict freedom?

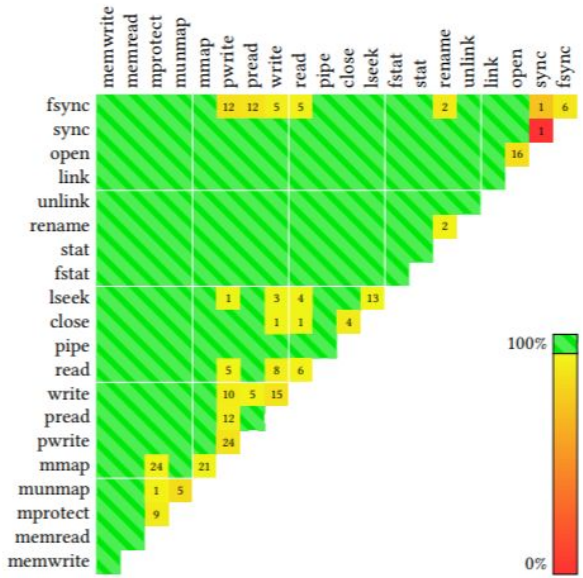


Figure 6: Conflict-freedom of commutative operations in ScaleFS. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 31,551 total test cases generated, 31,317 (99.2%) were conflict-free.

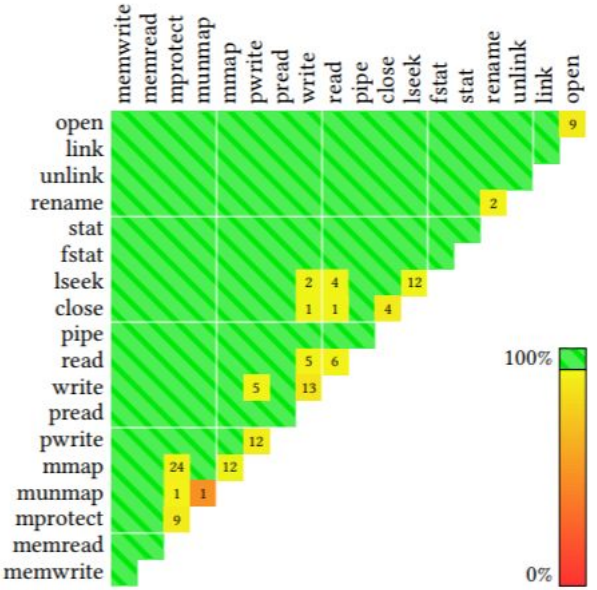
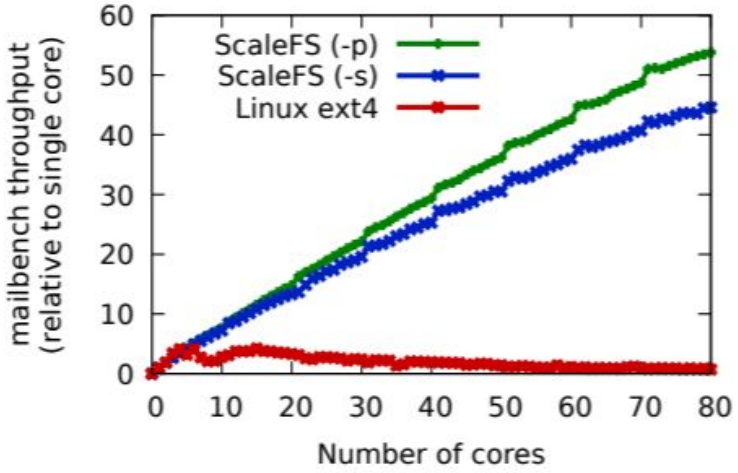
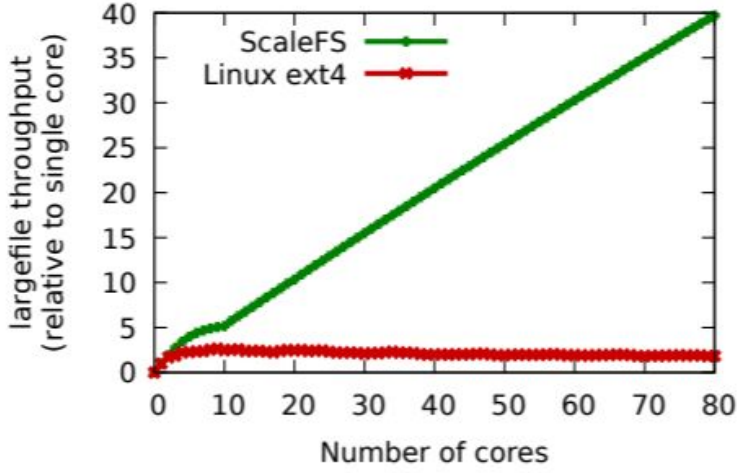
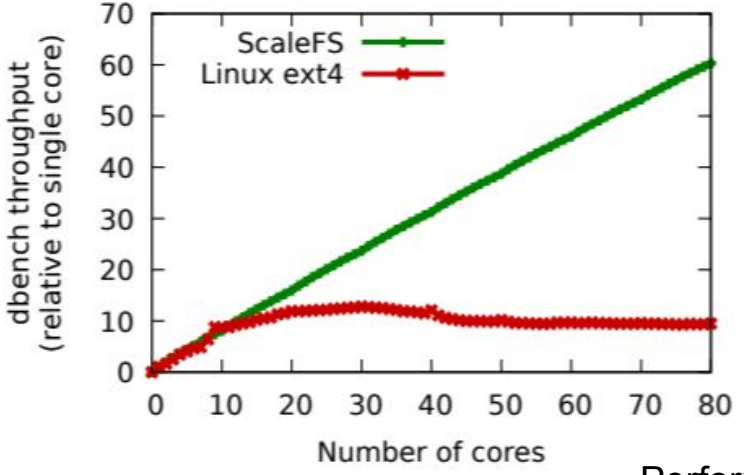
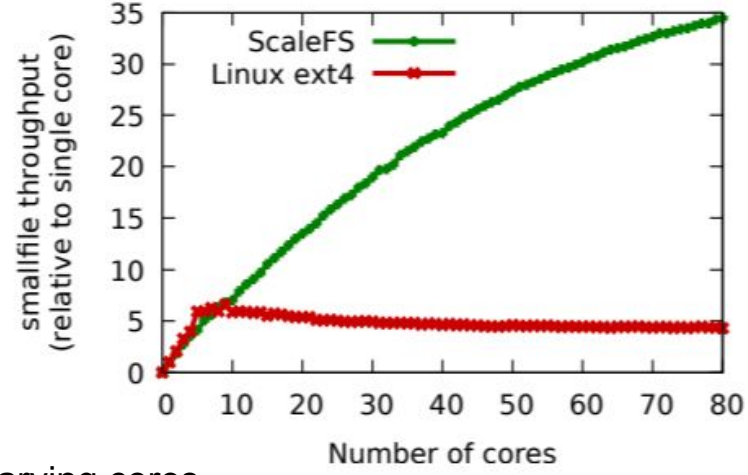


Figure 8: Conflict-freedom between commutative operations in sv6 with only an in-memory file system. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 13,664 total test cases generated, 13,528 (99%) were conflict-free.







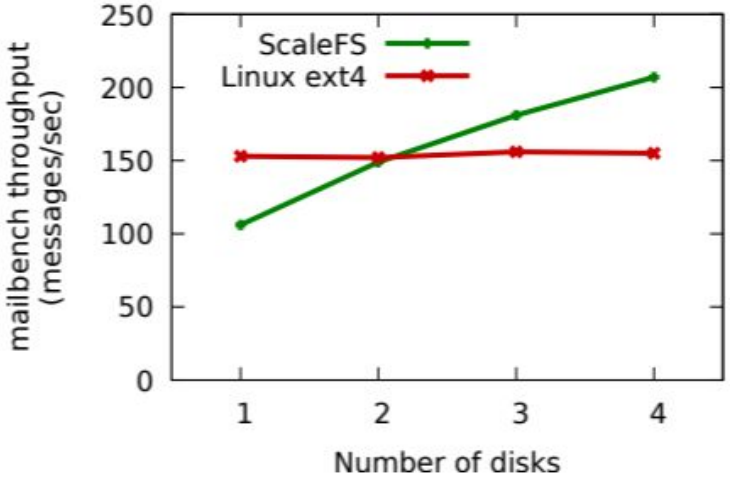


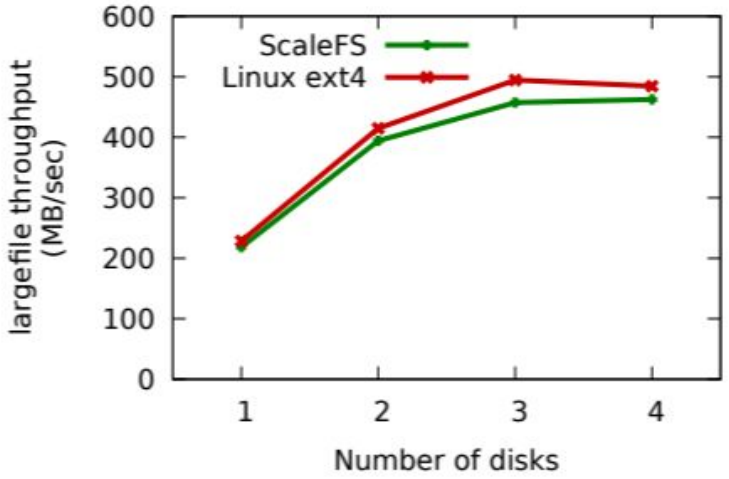
Performance with varying cores

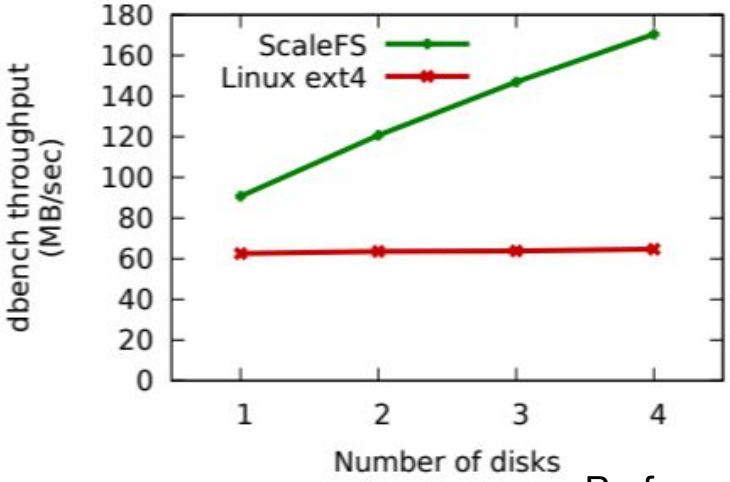
Single core performance

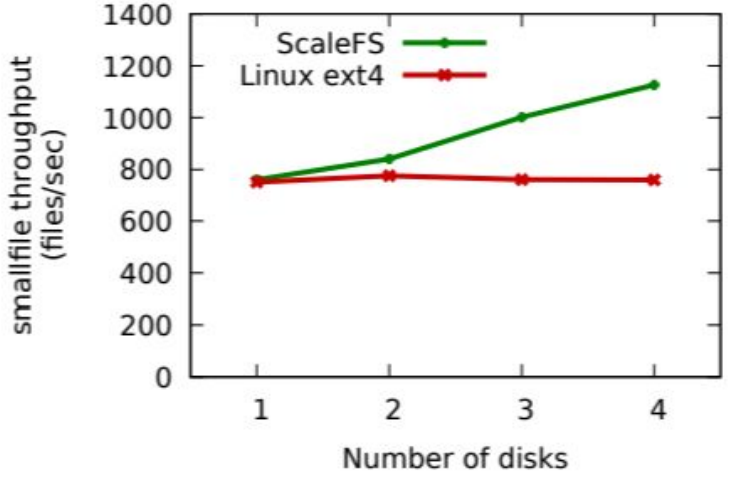
Disk	Benchmark	ScaleFS	Linux ext4
RAM disk	mailbench-p	641 msg/sec	675 msg/sec
	dbench	317 MB/sec	216 MB/sec
	largefile	331 MB/sec	378 MB/sec
	smallfile	7151 files/sec	3553 files/sec
SSD	mailbench-p	61 msg/sec	66 msg/sec
	dbench	47 MB/sec	28 MB/sec
	largefile	180 MB/sec	180 MB/sec
	smallfile	364 files/sec	277 files/sec
HDD	mailbench-p	9 msg/sec	9 msg/sec
	dbench	7 MB/sec	5 MB/sec
	largefile	83 MB/sec	92 MB/sec
	smallfile	51 files/sec	27 files/sec

Figure 10: Performance of workloads on ScaleFS and Linux ext4 on a single disk and a single core.









Performance with varying disks [4 cores]

Memory overhead

Benchmark	Memory use in MemFS	Memory use in ScaleFS	ScaleFS overhead
mailbench-p	12 MB	21 MB	75%
mailbench-s	515 MB	770 MB	49.5%
dbench	84 MB	94 MB	11.9%
largefile	106 MB	107 MB	0.9%
smallfile	296 MB	561 MB	89.5%

Figure 12: Peak memory use in MemFS and ScaleFS during the execution of different benchmarks.

Comments

- Quite thorough and thoughtful design
- Better techniques for absorption
- Why not lock the lowest common ancestor instead of a global lock?
- How much do page faults cost? (latency between MemFS and DiskFS)
- Is it practical?
- Security aspects?