Ext4, btrfs, and the others

Jan Kára <jack@suse.cz>

SUSE Labs, Novell

Outline

Challenges to tackle

Design of ext4 and btrfs

Some performance numbers

Other filesystems – reiser4, ocfs2, ubifs

Challenges to tackle

- Storage grows larger, throughput and seek time do not change that much
  - Directories and files grow larger
- Rate of error per sector stays the same
- SSDs
- Demand for new features
  - Snapshots
  - Clustering

Ext4 design

Ext4 basics

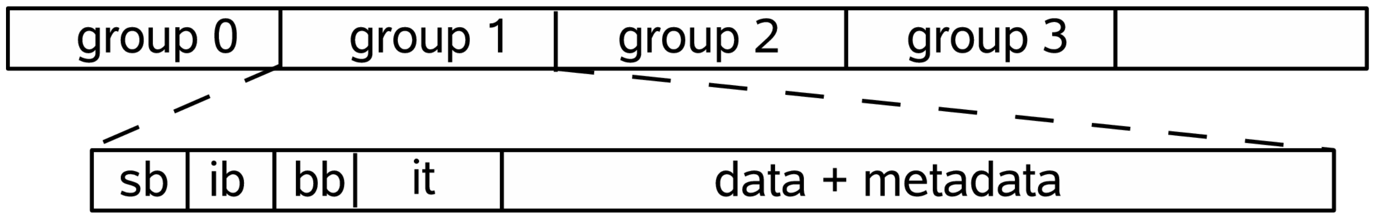
Successor of ext3

Shares 'philosophy' of disk layout with ext3 – standard Unix filesystem

Backward compatible by default

Quite stable, although still less stable than ext3

Global structure



Filesystem divided into groups

- Allocation locality

Each group has its inode table, inode bitmap, block bitmap

- Flexible block groups
- Number of inodes still fixed at filesystem creation time

Some groups have a copy of superblock and group descriptors (sparse super feature)

48-bit block numbers

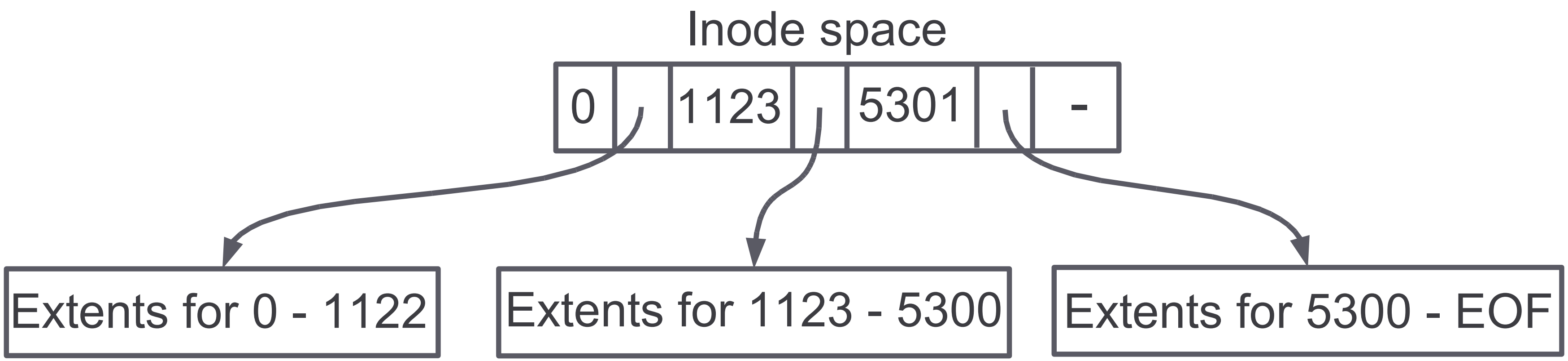
Inodes

- Inode size increased from 128 to 256 bytes
  - Only for newly created filesystems
  - High precision timestamps
  - More space for inline EA
- New way of tracking blocks carrying data – extents

```
struct ext4_extent {
        __le32  ee_block;
        __le16  ee_len;
        __le16  ee_start_hi;
        __le32  ee_start_lo;
};
```

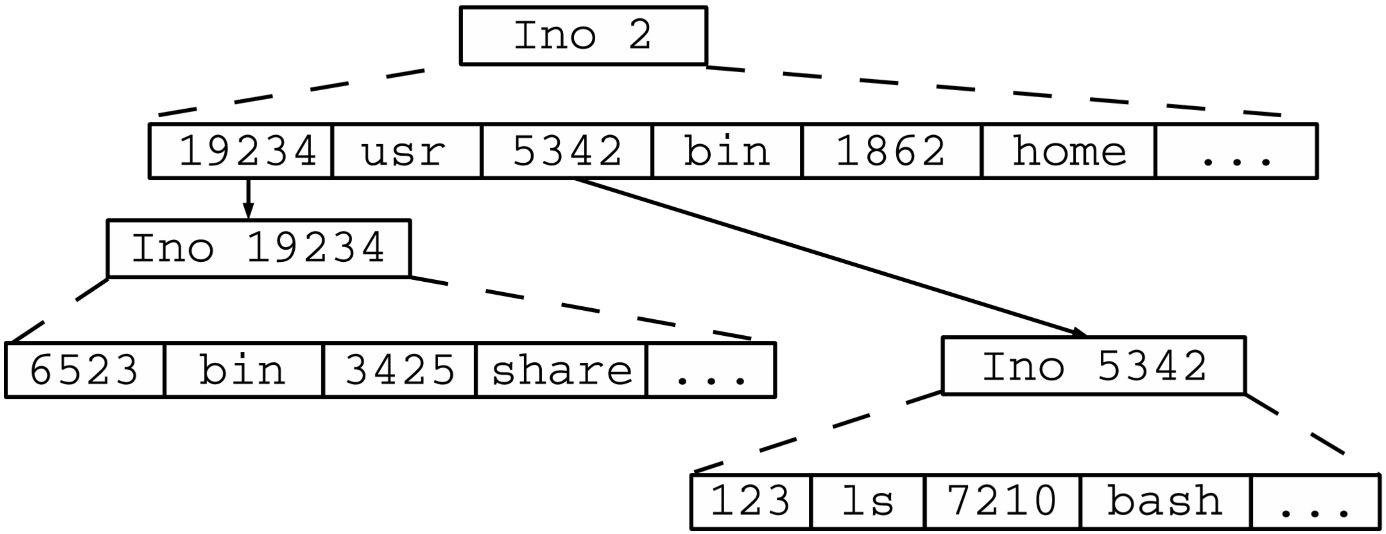
Extent tree

Inode and indirect blocks carry a b-tree of extents



Directories

Inodes containing directory entries



Search tree on top of directory to speed up lookup

- Hidden in special directory entries
- Slow down when scanning whole directories

Journaling

- Allows fast filesystem recovery after a crash
- New transaction checksum feature
  - Does not prevent fs corruption, only reduces impact

Delayed allocation

Blocks for data (and metadata) allocated only when kernel decides to write out data to disk

When blocks are written, space and quota is only reserved

More blocks allocated at once

Better coalescing of random writes

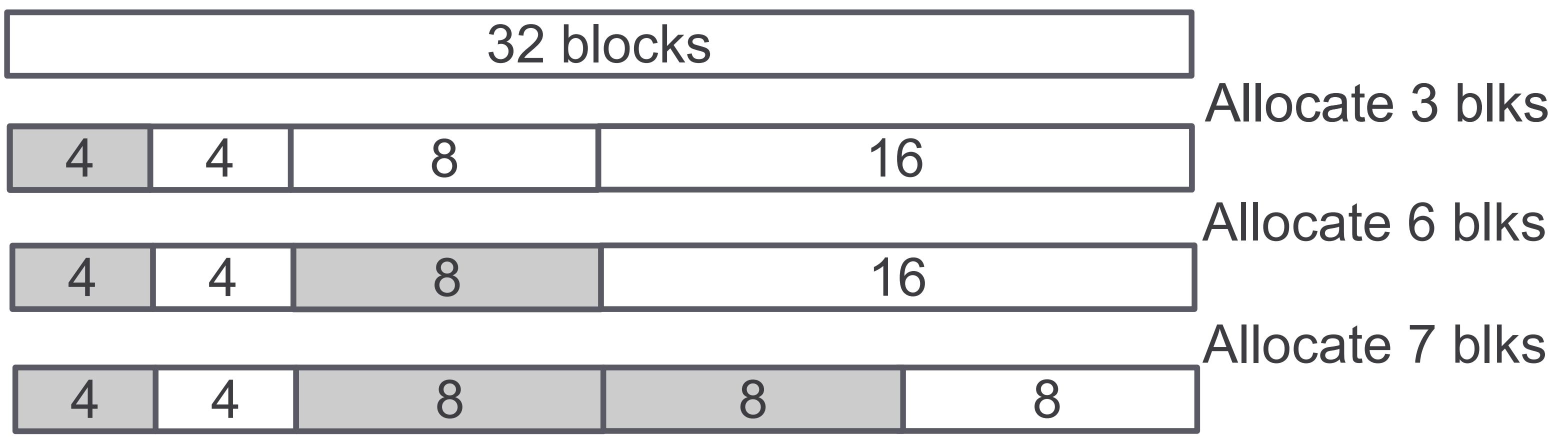
Data gets later to disk

Multiblock allocator

Aims to reduce fragmentation and allocate large chunks of blocks quickly

Buddy allocator in the core

- Allocates aligned chunks of 2^n blocks



Buddy bitmaps only in memory generated from block bitmap

Multiblock allocator (cont.)

- Before allocation we estimate the final file size and continue with allocation for that many blocks
- Buddy allocator enhanced with preallocation lists to use unused space in buddies
  - Per inode preallocation list
  - Per locality group preallocation
    - `/sys/fs/ext4/<dev>/mb_stream_req`
- Logic to handle case when there is no buddy large enough to satisfy the allocation
  - Several rounds of allocation, each round scans groups starting with the goal group
  - In each round we weaken our requirements on the free extent

Multiblock allocator (example)

Assume blocksize 1K, file size is already 15000

Allocate 35 blocks
Estimated file size: 64 K → looking for 45 blocks
Found 64K free buddy, allocate 45 blocks from it
Put 14 blocks left to inode's preallocation list

Allocate next 40 blocks
First 14 blocks are allocated from inode's preallocation
Going to allocate next 26 blocks
Estimated filesize 128K → looking for 64 blocks
Cannot find free buddy of size 64 → next round of scan
Scan all free extents, the best found has 20 blocks.
Next allocation request happens for remaining 6 blocks

Other features

`fallocate`

- Extents just marked as uninitialized, data not written
- Efficient preallocation of blocks to file

Online defragmentation

- `EXT4_IOC_MOVE_EXT` ioctl
- Atomically copies data of a file into provided space (allocated to another file)
- Support for control of allocation under development

Btrfs design

Btrfs basics

Implemented from scratch (started in 2007)

Some parts resemble reiserfs

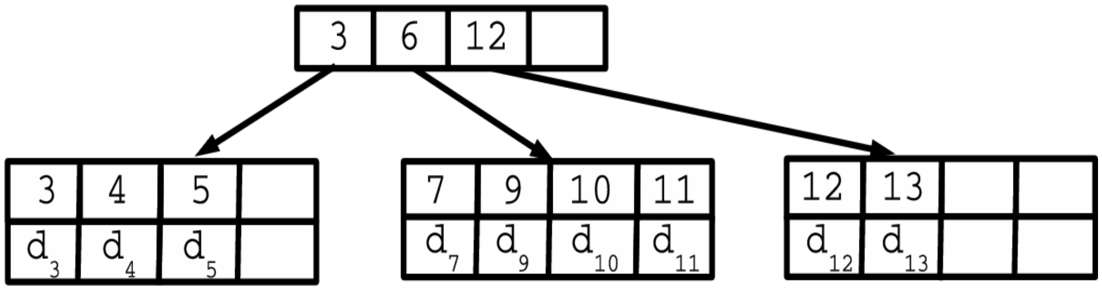
Copy-on-write filesystem

Not completely stable but quite fine

B+trees

Core data structure of the filesystem

Internal nodes contain search indices, leaf nodes items



Several b+trees in the filesystem

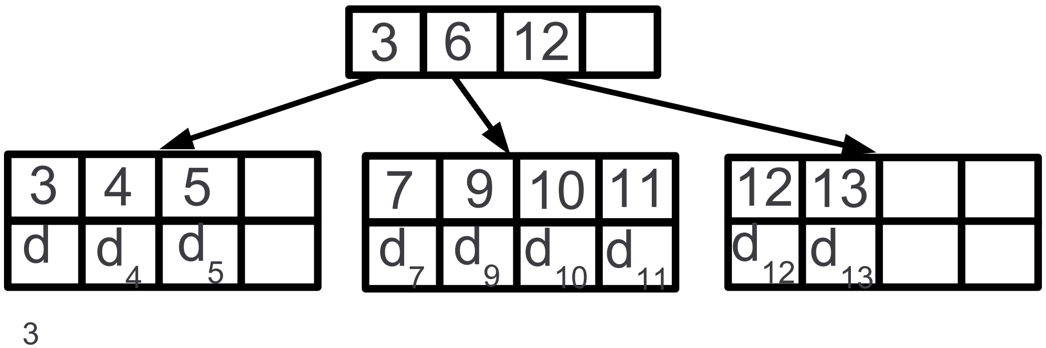
- Main one carrying most of the metadata
- Other 5 trees for special purposes

Key in the tree: `<object id, type, offset>`

- Results in close packing of metadata related to one object

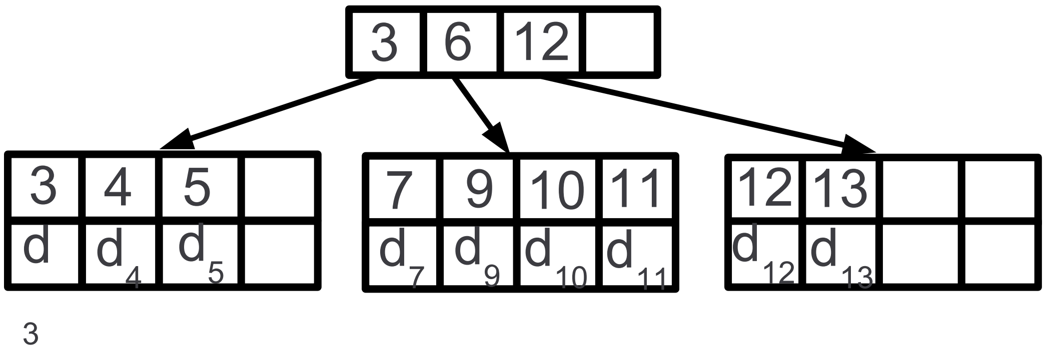
Modifications of b+tree

Modifications handled in copy-on-write manner



Modifications of b+tree

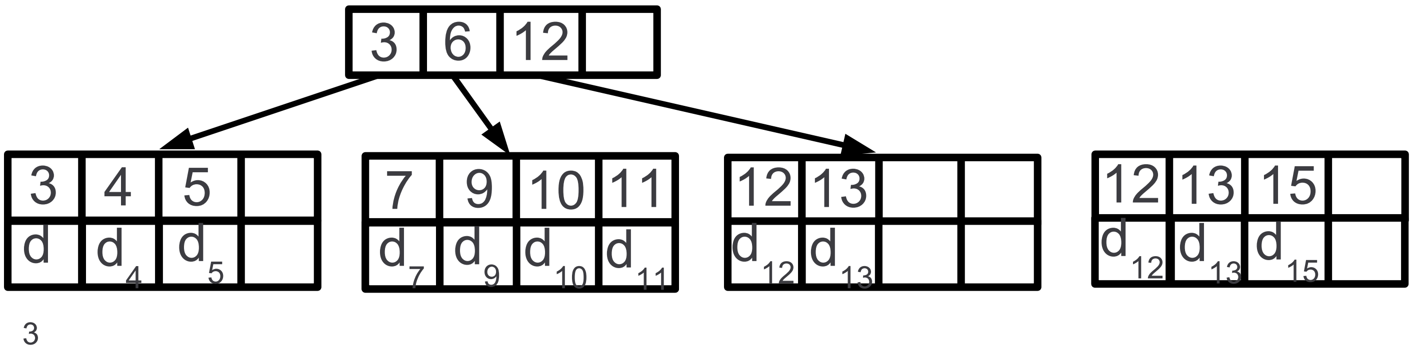
Modifications handled in copy-on-write manner



Add item 15

Modifications of b+tree

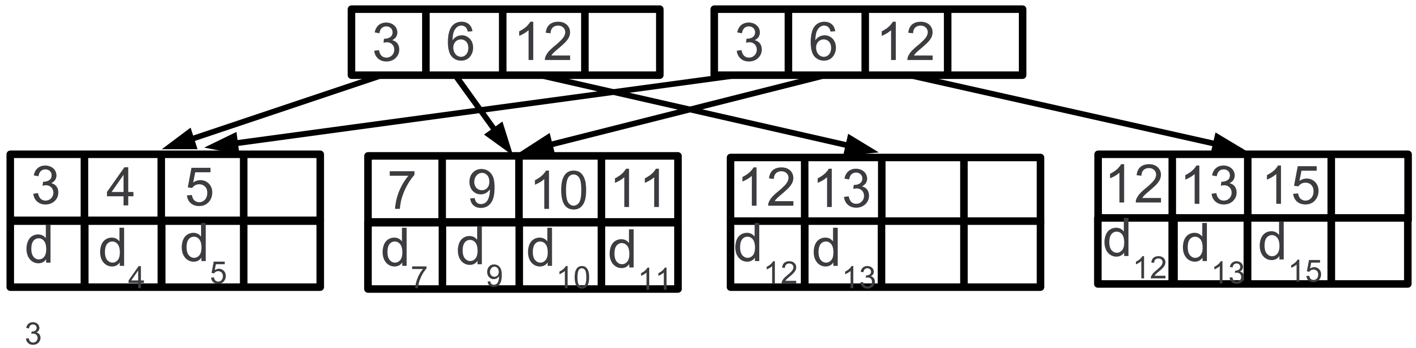
Modifications handled in copy-on-write manner



Add item 15

Modifications of b+tree

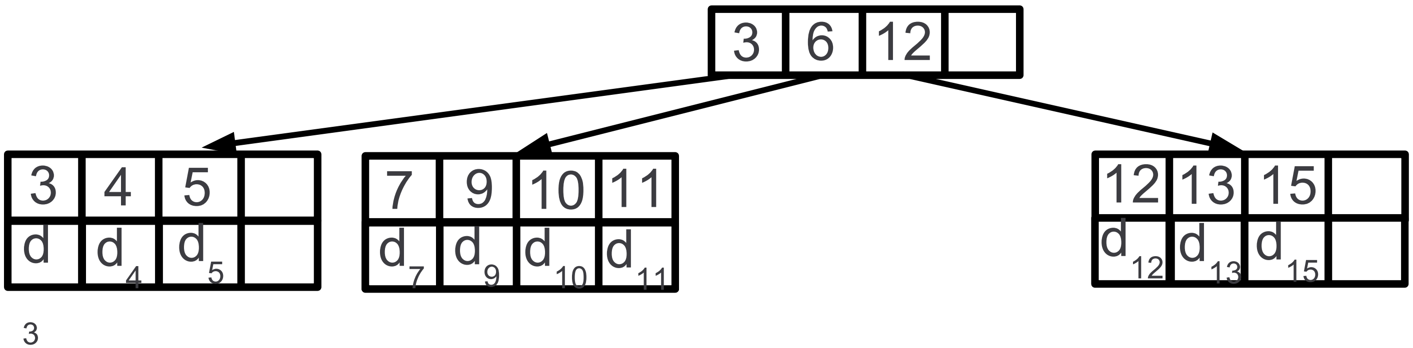
Modifications handled in copy-on-write manner



Add item 15

Modifications of b+tree

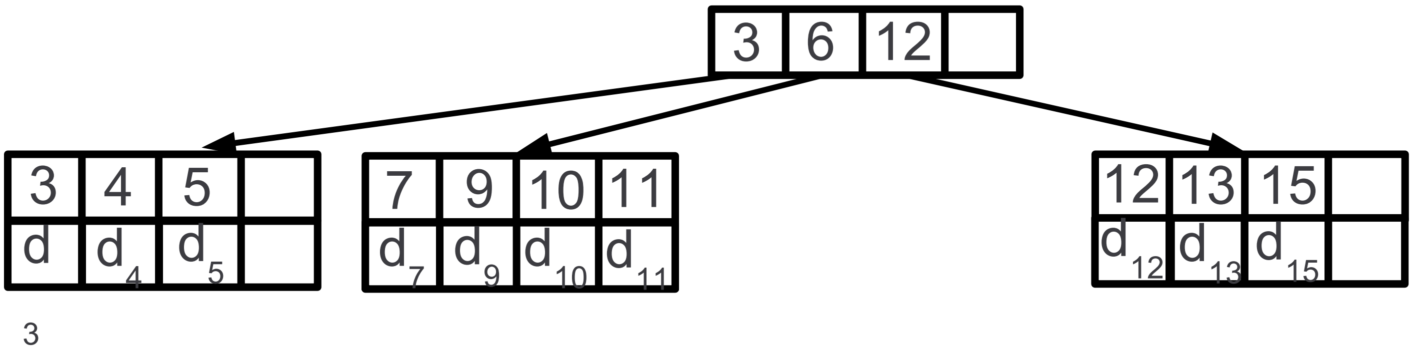
Modifications handled in copy-on-write manner



Add item 15

Modifications of b+tree

Modifications handled in copy-on-write manner



Add item 15

The tree constantly moves as it changes

Files

A file is comprised of:

- Inode item – contains information about file size, permissions, owner, etc.
- Extent items – contain information about extent of data – starting block, length, reference count
- Data item – small files do not have extent items, data is stored in data item instead
  - Data close to metadata → faster read
  - Saves space
- Checksums of data

Directories

- Set of directory items with `objectid` of the directory
- Item contains all entries with the same CRC32 hash
- Natural tree structure → fast lookup and other dir ops
- Directory index items
  - Used to traverse directory on readdir
  - Lists directory entries in creation order (should to be close to disk order of inode items)

Snapshots

- Copy of a filesystem at given point in time stored in a subdirectory of the filesystem
  - Can snapshot also a single directory or even a single file
- Snapshots are writable, modifications to original and snapshot are separate
- Implemented just by referencing snapshotted object (root of the filesystem tree, directory, file)
- Because of copy-on-write handling, unchanged parts are shared
- Reference counting of each extent (tree node or data)
- Recovery after a crash implemented via snapshots

Checksumming

- CRC32 checksum of each tree block
  - Space for 32-byte checksum is reserved
- CRC32 checksum of each data block
  - Stored in a special tree just for checksums indexed by data block number
  - Checksums of several blocks packed into a single item to reduce overhead of item headers

Multiple device support

- Pool of devices to be used by a filesystem
- Space from the pool allocated in a few GB *chunks*
  - Linearly mapped part of a device from the pool
  - Part of a device mirrored to another location on the device
  - Parts of several devices combined via RAID0, RAID1, RAID10
- All devices hidden under a single linear address space
- Special tree storing information about chunks
  - Superblock contains information how to map addresses from the special chunk tree
- Adding and removing chunks online
  - Easier device removal due to backreferences

Tracking free space

- Dedicated tree of free extents on disk
- In memory RB tree of free extents
  - If RB tree would use more than 16 KB / GB, no more extents are added to the RB tree and bitmaps are added instead
  - Total memory use by this structure limited to 32 KB / GB
- Creation of in memory data structure from on disk tree performed by a kernel thread

Allocation algorithm

- Delayed allocation
- Search for free blocks quite complex
- Three allocation strategies
  - Rotating media
  - SSD
  - SSD with bad random writes
- Two purposes of allocation – metadata / data
- Depending on purpose and strategy, we pick suitable groups and mode of allocation

Allocation algorithm (cont)

- Three types of allocation groups (chunks)
  - System group – chunk tree
  - Metadata group – nodes of other trees
  - Data group – data blocks
- Several rounds of allocation
  - Groups with cached free space information
  - Groups with partially cached free space information
  - Wait to load free space information
  - Add new chunk to the filesystem
  - Ignore group type

Allocation algorithm (cont)

- Two modes of allocation
  - Simple search for free extent
  - Clustered allocation – look for several nearby extents having more free space, store unused ones for next allocation

	Rotational	SSD	SSD spread
Data	Extent	Cluster (~512K)	Cluster (~2M)
Metadata	Cluster (64K)	Cluster (~128K)	Cluster (~2M)

- In the last round, we just do simple extent search
- When everything fails, restart allocation procedure looking for a single free block

Other features

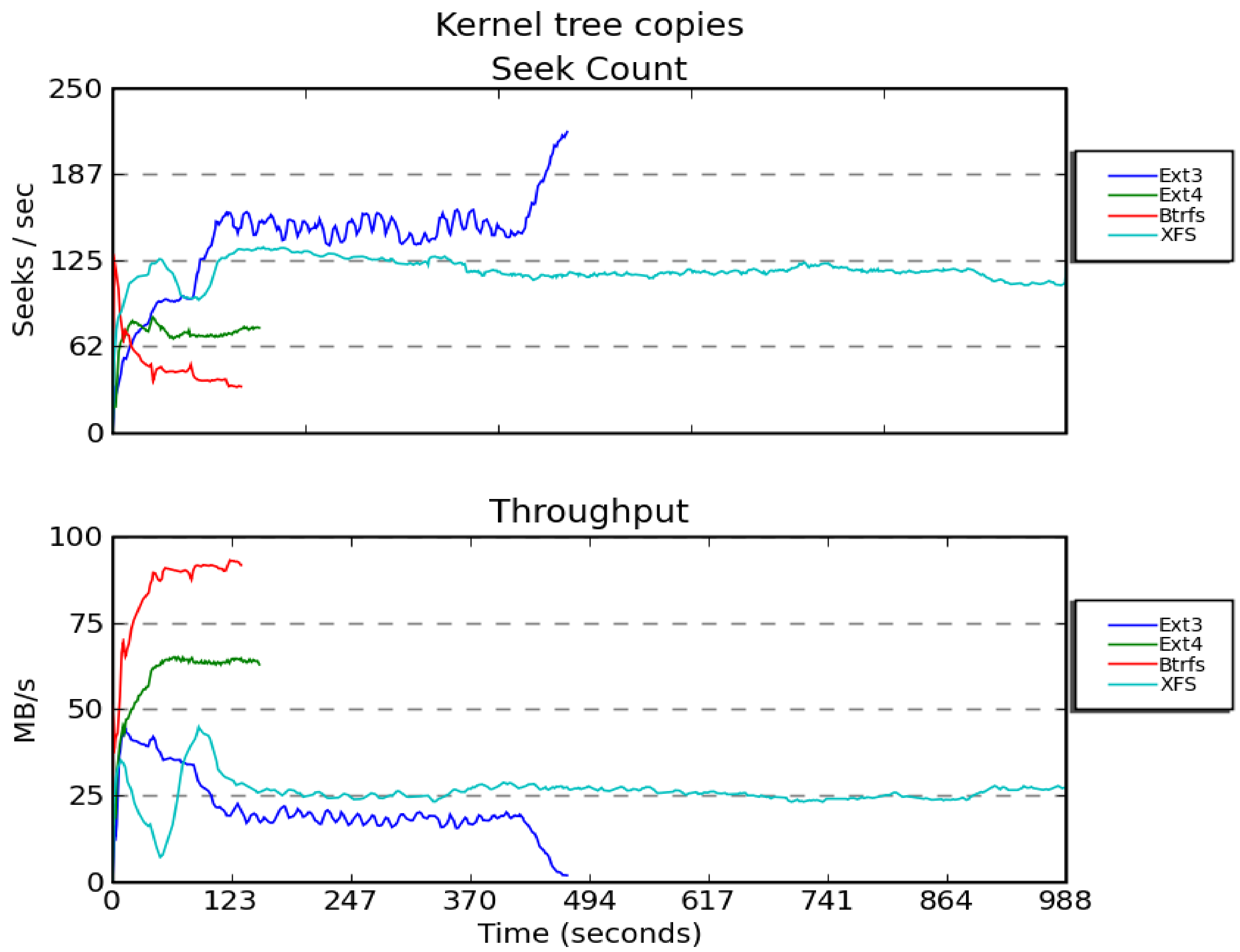
Fallocate (similarly as ext4)

Online compression and decompression

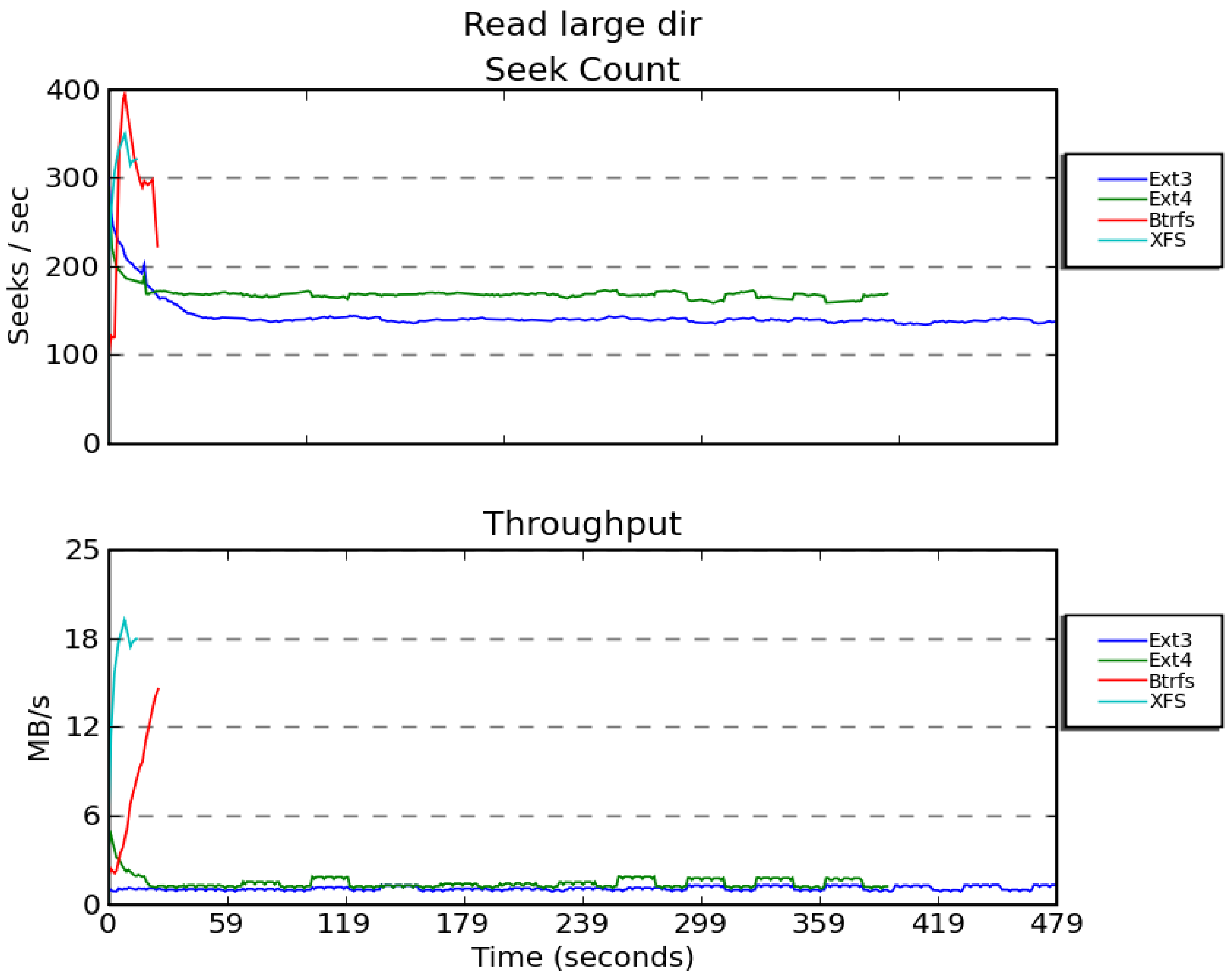
Simple online defragmentation (reallocate file in the new location)

Performance comparison

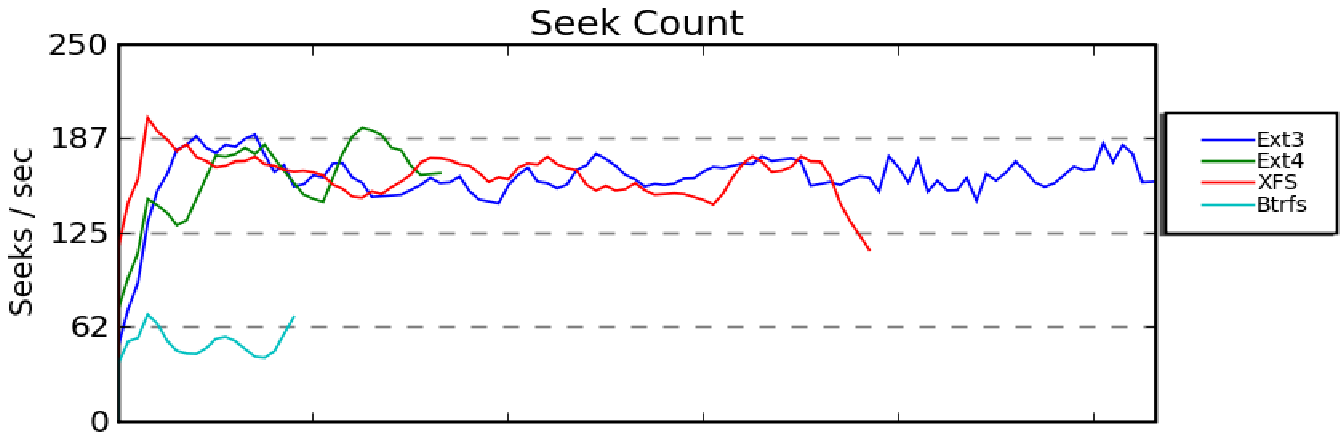
Kernel tree copy

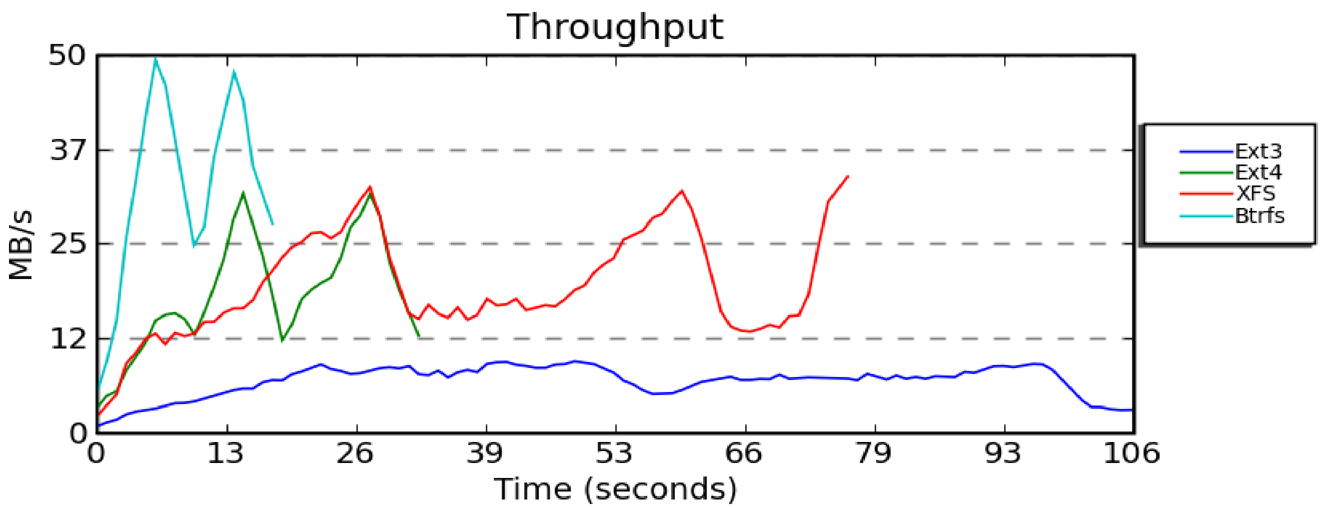


Large directory

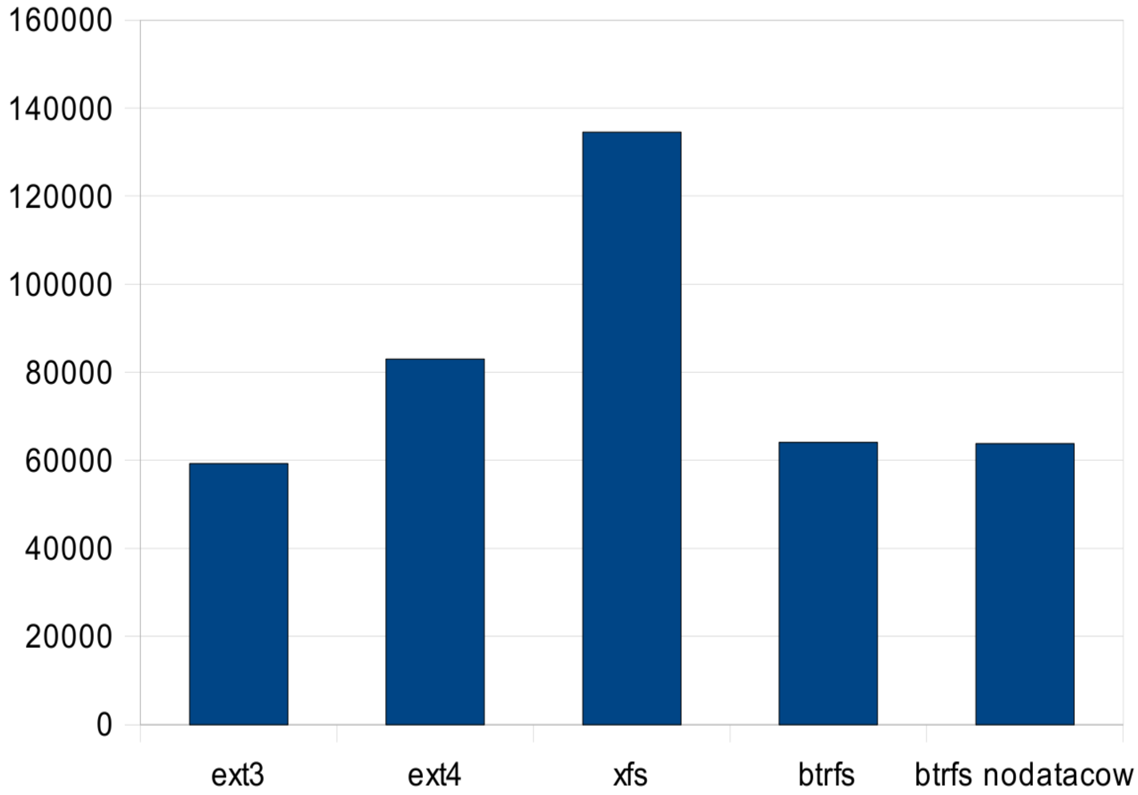


Syncing test



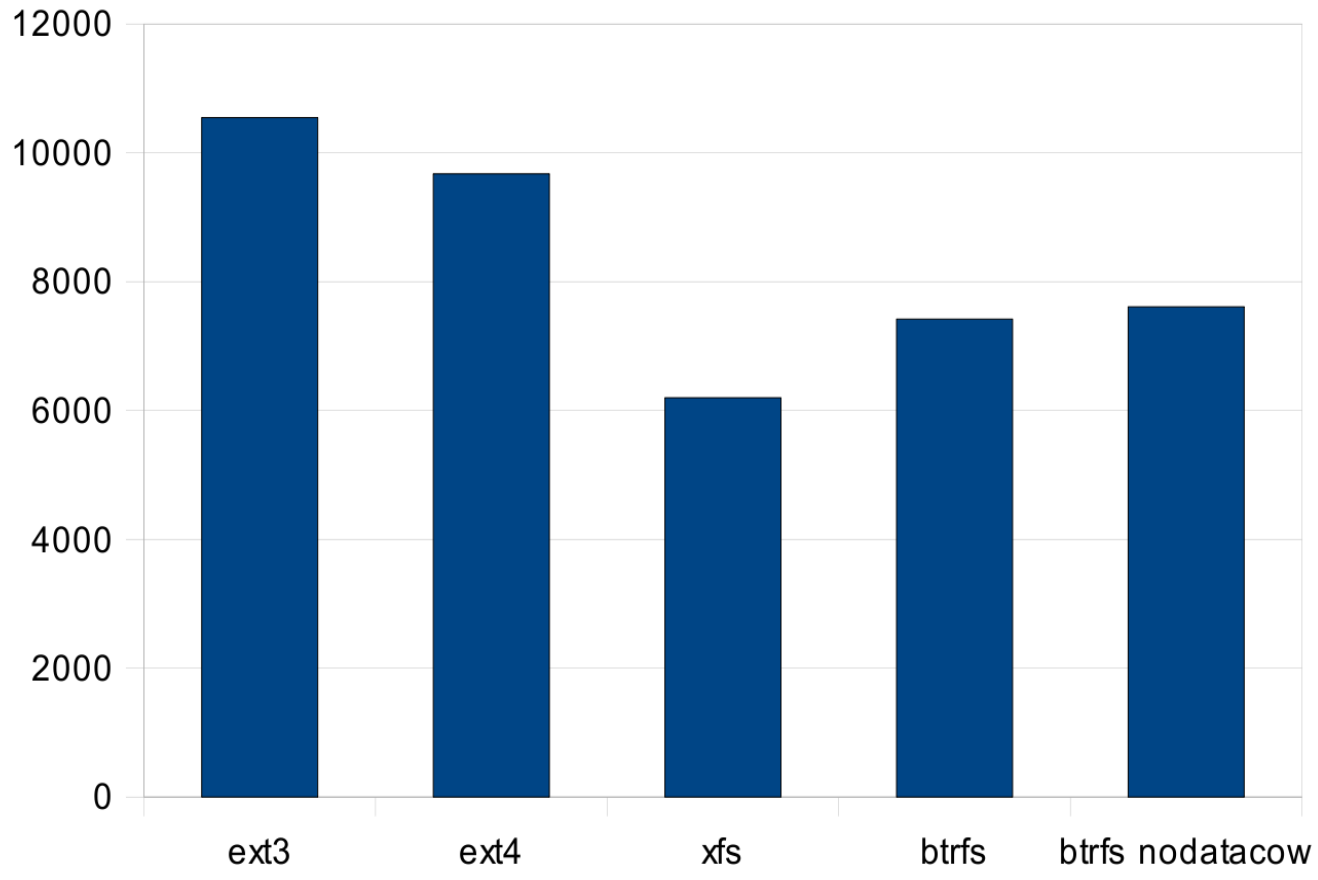


16 streaming writes

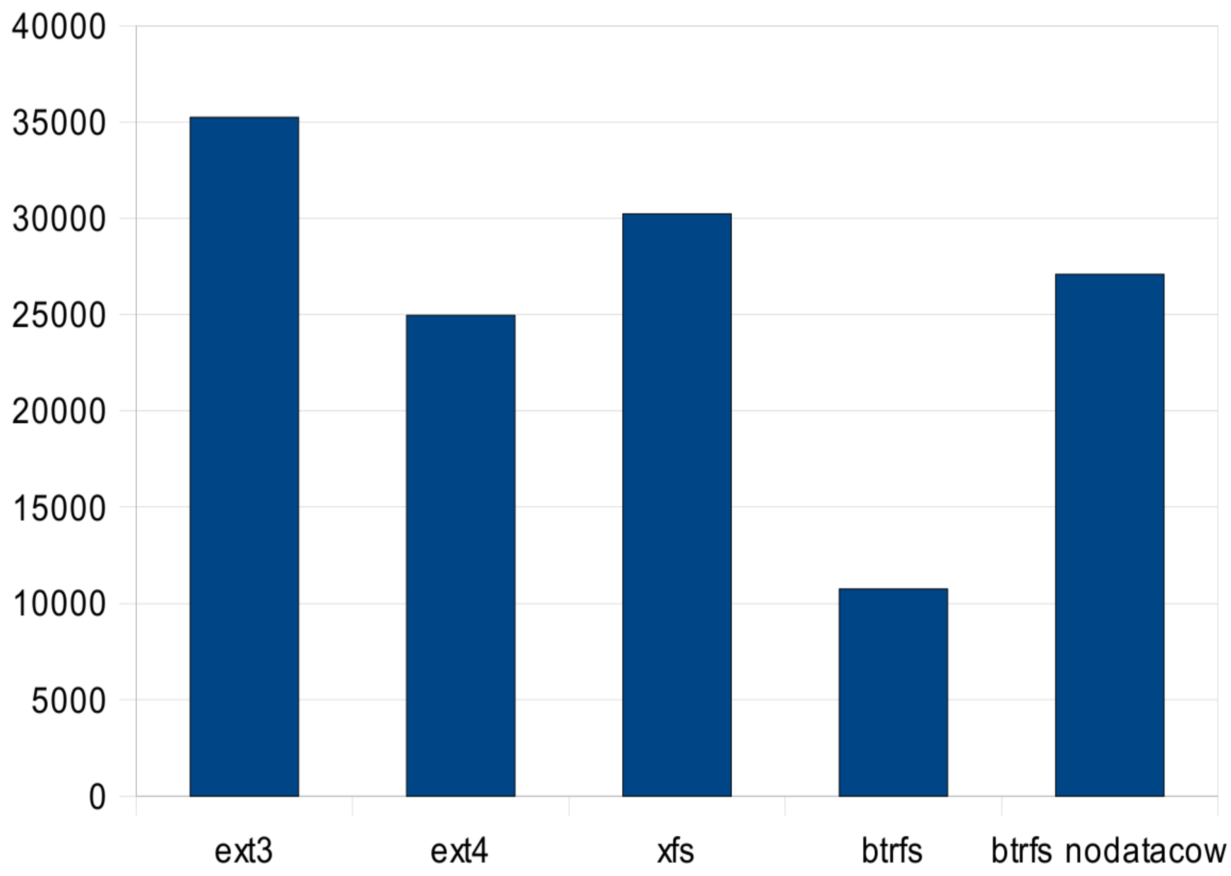


With nocow, btrfs matches xfs

Mail server


12000
10000
8000
6000
4000
2000
0
ext3
ext4
xfs
btrfs
btrfs nodatacow

Random writes



Other filesystems

Reiser4

Successor of reiserfs

Uses b+trees as the core structure

Combination of journaling and copy-on-write (wandering trees)

Not certain whether / when it will be finished

Transparent encryption, compression

Modular design

OCFS2

- Cluster filesystem
- Quite close to traditional unix design
  - Dynamic inode allocation
  - Extent trees
  - Journaling
- Node local on disk structures to improve concurrency

UBIFS

- New flash filesystem – not for block devices
- UBI layer to handle wearlevelling
- No scalability issues of JFFS2 (mount time and memory consumption independent of filesystem size)
- UBI layer still takes time linearly growing with device size to setup – work in progress to fix it
- B+trees modified in copy-on-write manner
- Online compression, checksumming

Thank you