# *Btrfs overview, variable blocksize support and others*

*IBM Linux Technology Center*
*Mingming Cao*
*Oct 13th 2012*

# *Agenda*

- Storage challenges
- Btrfs basics
- Btrfs variable blocksize support
- Btrfs fragmentation issue
- Btrfs performance

# *Linux filesystems*

- Data is essential to end users
  - Linux has 50+ filesystems to choose

- Most-active local filesystems are
  - Ext3/4
  - XFS
  - Reiserfs
  - Traditional Unix fs design + improvements
  - Stable, reliable, easy to understand

- New storage trend calling for new filesystem
  - Other OS working on next generation filesystem starting from scratch (ZFS, ReFS etc)

# Storage trend needs

- Better data integrity and high availability
- Scalability and high performance
- Easy data storage Management
- Support for new and existing media
- Virtualization

# *Btrfs and a short history*

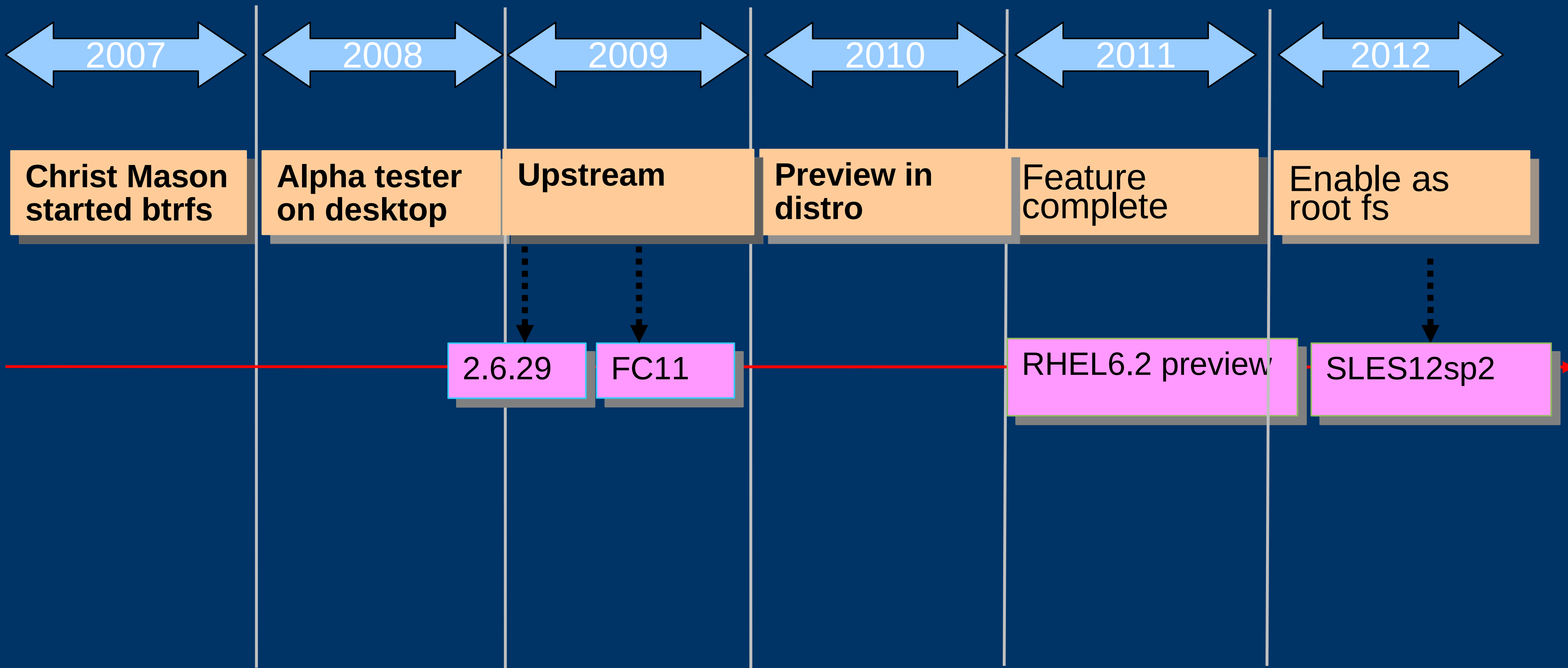
Christ Mason started looking into a new fs in 2007

- After attend LSF workshop
- Inspired by IBM research report about COW-based snapshots
- Many experience and lesson learned from other linux filesystem (reiserfs etc)
- Lot of support from community and multiple companies

# *Btrfs Design target (back in 2007)*

- Storage pools
- Writeable, named, recursive snapshots
- Fast filesystem checking **and** recovery
- Easy large storage management for admins
- Proactive error management
- Better security
- High scalability
  - 128 CPU cores, 256 spindles, hundreds of subvolumes
- Fast incremental backup

# *Btrfs time line target and now*



# Btrfs basics -- btree

- Everything is in a btree
- Three basic on-disk data structures
  - Block header
  - Keys
  - Items
- Every tree block is node or leaf
  - All starts with block header and keys
  - Only leaves stores the actual data
- Copy on write protect tree integrity

Nodes
[header, key ptr0....key ptrN]

Leaves
[header][item0....itemN] [free space][dataN...data0]

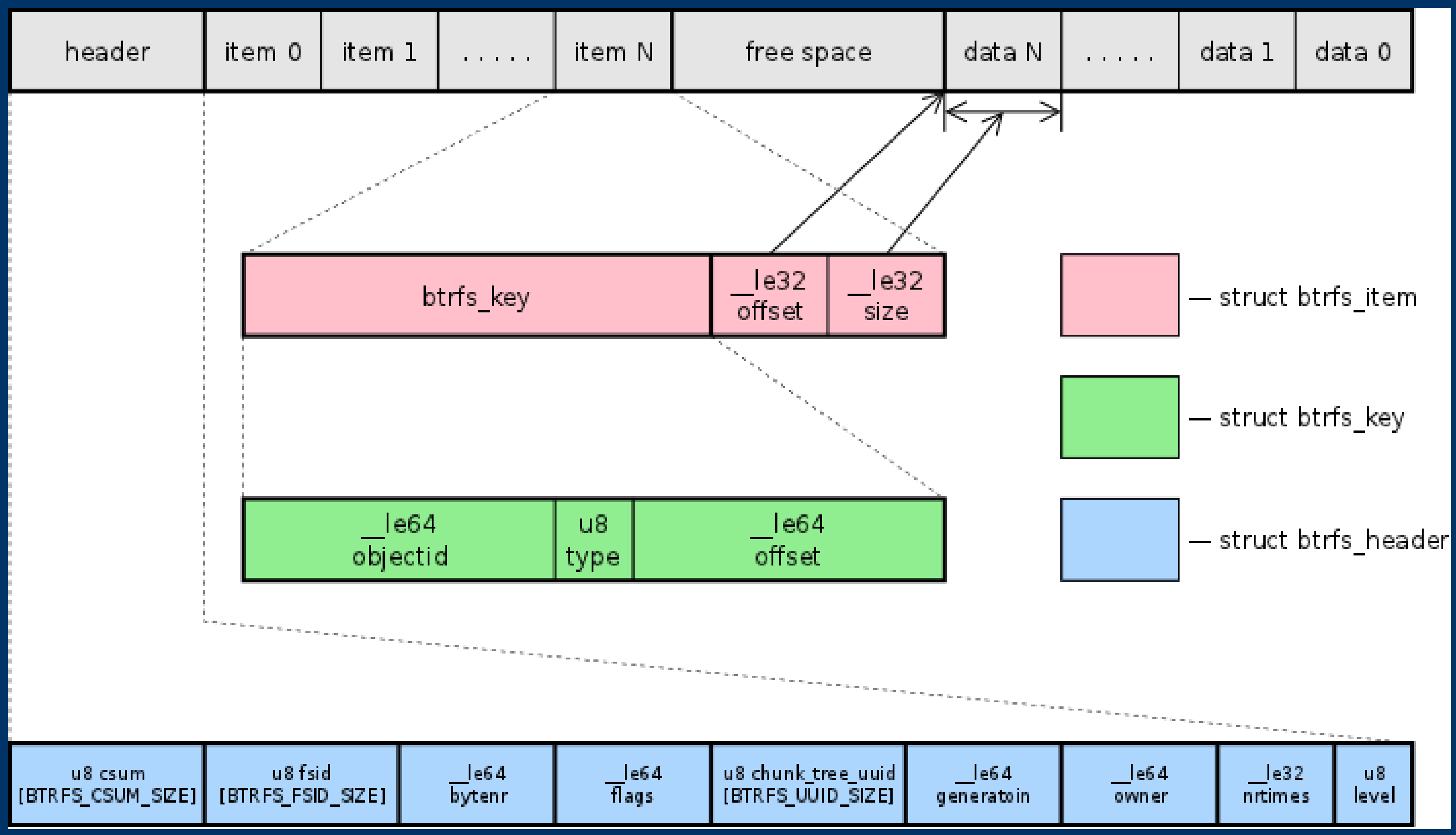
```
struct btrfs_header {
    u8 csum[32];
    u8 fsid[16];
    __le64 blocknr;
    __le64 flags;

    u8 chunk_tree_uid[16];
    __le64 generation;
    __le64 owner;
    __le32 nritems;
    u8 level;
}

struct btrfs_disk_key {
    __le64 objectid;
    u8 type;
    __le64 offset;
}

struct btrfs_item {
    struct btrfs_disk_key key;
    __le32 offset;
    __le32 size;
}
```

# Btrfs leaf nodes

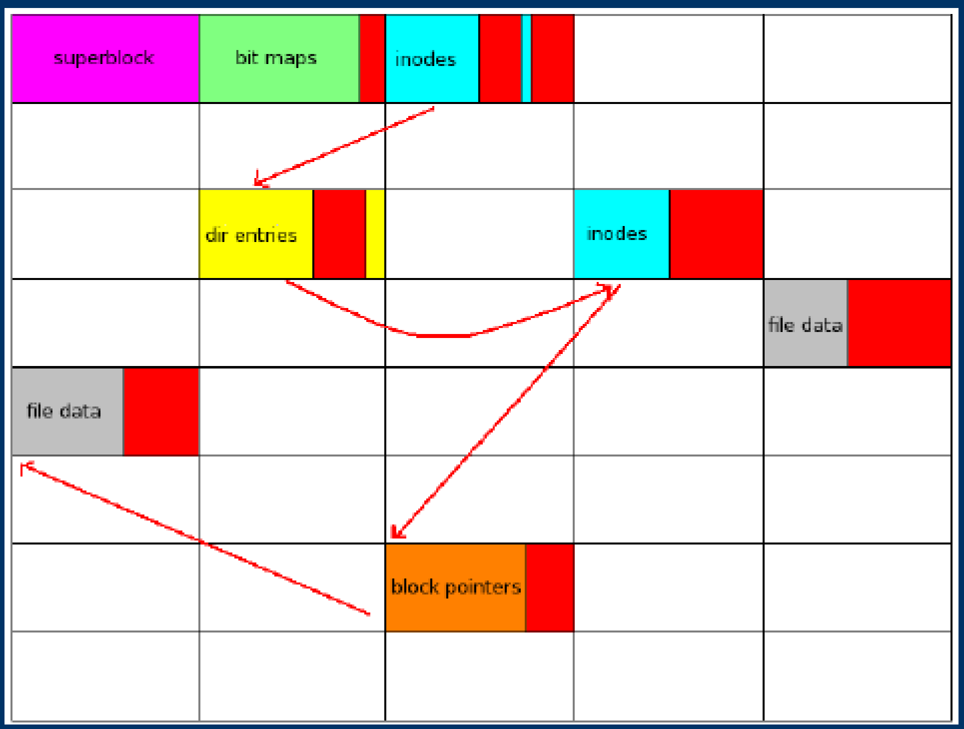


# Btrfs btree

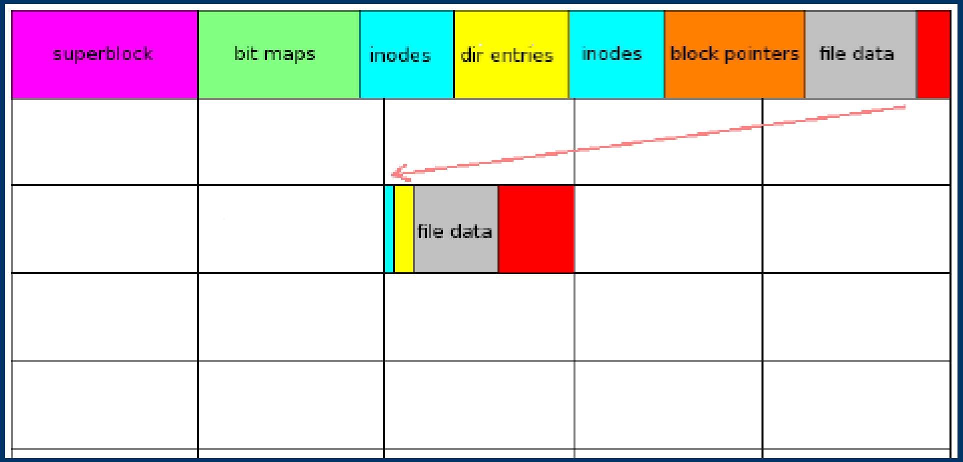
- One main btree plus 6 special purpose btree
  - fs tree (main tree)
  - extent allocation tree
  - chunk tree
  - device allocation tree
  - checksum tree
  - data relocation tree
  - log root tree
- Share single btree implementation code
- Metadata from different files and directories is mixed together in a block

# *Btrfs efficient metadata packing*

ext3/4




btrfs

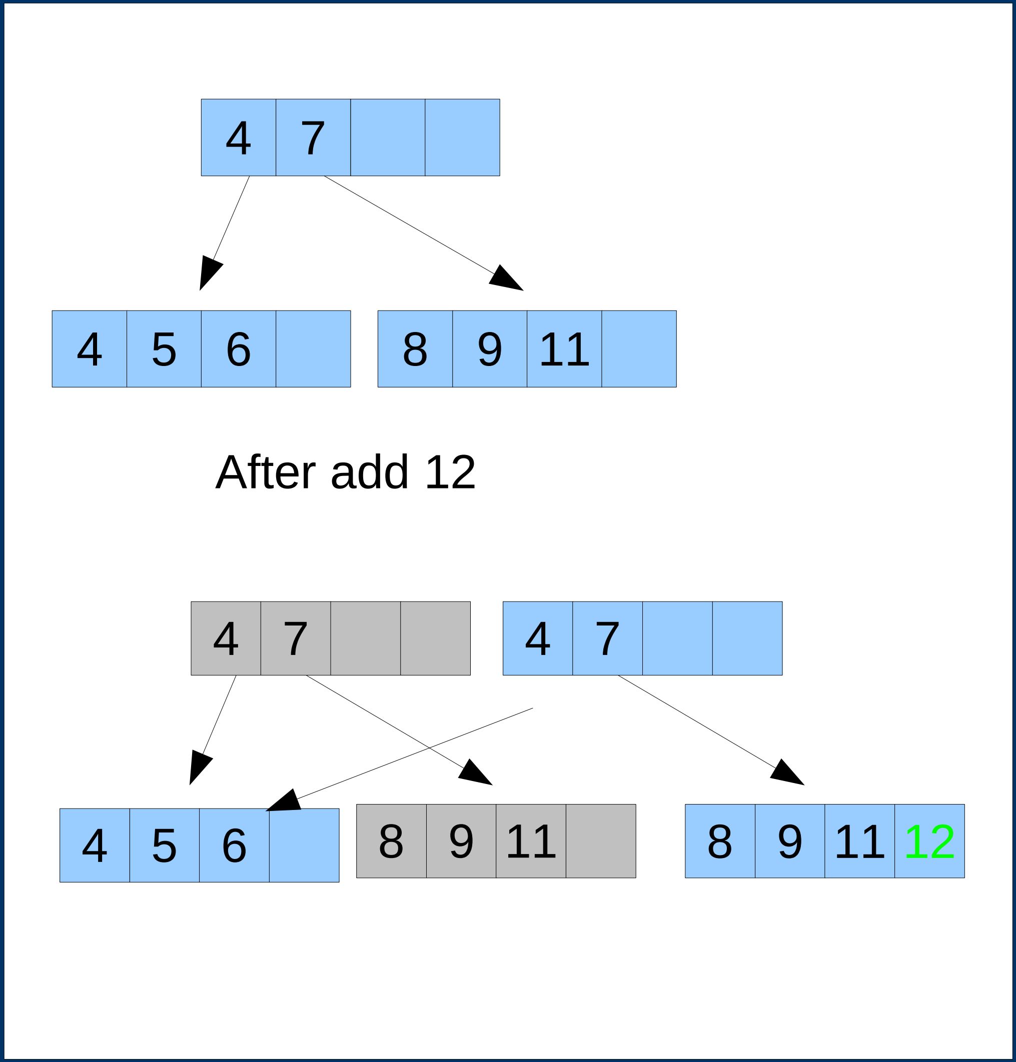



Wasted space

Disk seek

# Btrfs transactions

- No journalling block layer
- No modification in place
- Tree is cloned
- New roots are added into root tree
- Will not commit and link the new root until all data and metadata write to disk
- Original trees/nodes can be deferenced



# Btrfs basic -- snapshot

- Snapshots are readable, recursively writable
- Snapshots in btrfs are really efficient
  - Similar to transaction, implemented just increase reference on the original block
  - Only the modification part is cloned/copied
- Snapshorts are subvolumes
- File clone with cp --reflink

# Btrfs basic-- checksums

- Both data and metadata are checksumed
- Checksums can be read and validated after read from disk
- Uses serveal background threads to offload the work
- Based on checksum, btrfs performs background scrubbing, scanning bothe data and metadata blocks

# Btrfs basic – multi disks support

- Storage pool -- easy to add and remove disks
- Allocates space on its disks by allocating chunks
  - 1 gigabyte chunks for data
  - 256 megabyte chunks for metadata
- All physical devices are hidden, logical devices are created to map a linear address space to multile disks
- Disk type and speed could be mixed
  - Potential to place hot data on fast disks
- Filesystem is still chunked into blockgroups
  - extent allocation tree keeps track of the the fs extent allocation information

# Btrfs offers today

What ext4/xfs does

- Extents
- Journalling
- Online defragmentation
- Delayed allocation
- Preallocation
- Punch hole
- Fiemap
- DIO/AIO
- Quotas
- Extend attributes
- Trim/discard
- Offline system check (sort of)

What ext4/xfs does not

- Intergrited LVM multi-disk support
- snapshot/subvolumes
- Data and metadata checksumming
- Online scrubbing
- Compression
- SSD support
- Space efficient pack of small files
- Offline conversion from ext3/4 to btrfs
- Dynamic metadata usage

# Btrfs challenges

- Performance with sync
  - Frequent commits generates many recow of blocks
  - log tree log only the changed file and directory metadata
  - Helps but still hurts in heavy sync case
- Fragmentation issue
  - Nature caused by COW
  - Reduce the fragmentation by clustering and preallocation
  - Large blocksize could reduce for metadata fragmentation issue
- Variable blocksize support
  - For better performance, less fragmentation
  - For architecture with large pages

# blocksize vs pagesize

- Filesystem are chunked into unit of blocks. Fs handles mapping on disk blocks into memory

- Linux uses buffer_heads structure to
  - map logical to physical blocks
  - And uses to ensure data=ordered journalling mode

- Buffer_head has issues
  - Each page has a buffer_head structure
  - Consumes lots of low memory
  - Hard to reclaim pages when buffer_head no longer referenced
  - Hard to perform large chunk of IO

# *Various blocksize*

- Btrfs initially had the goal of supporting large and small block sizes.  It is now time to bring it back
- Large blocksize
  - Better performance
  - Reduce fragmentation
- Small blocksize (subpage)
  - Space efficiency
  - Better cross architecture enablement

# *Btrfs basic data structures for IO*

- Need some data structures which replace buffer head

- Basic IO data structures
  - Extent_map -- Map
  - Extent state -- Track IO status
  - per extent map tree and extent IO tree
  - extent_buffer -- used only for metadata

# *Btrfs metadata large block support*

- One single extent buffer is attached to multiple pages via page->private
- Pages could be discontinuous in memory, page pointers are saved in extent buffer
- Extent readpage/writepage iterate number of pages to read/write one extent buffer. COW is done at extent buffer unit
- mkfs.btrfs -l 64k
- Generates much less fragmentation for metadata workload

# Extent buffer

```
struct extent_buffer {
        u64 start;
        unsigned long len;
        unsigned long map_start;
        unsigned long map_len;
        unsigned long bflags;
        struct extent_io_tree *tree;
        spinlock_t refs_lock;
        atomic_t refs;
        atomic_t io_pages;
      ....
        struct page *inline_pages[];
        struct page **pages;
};
```

- Large metadata block accepted in 3.3 kernel
- Still missing small blocksize support
  - Unable to mount small blocksize btrfs on large pagesize machines
  - Implies cant do live migration from ext3/4 to btrfs on large pagesize architectures

# Btrfs subpage blocksize

```
struct extent_buffer {
        u64 start;
        unsigned long len;
        unsigned long map_start;
        unsigned long map_len;
        unsigned long bflags;
        struct extent_io_tree *tree;
        spinlock_t refs_lock;
        atomic_t refs;
        atomic_t io_pages;
      ....
        struct page *inline_pages[];
        struct page **pages;
        Struct extent_buffer *
next_eb_this_page;
};
```

- Metadata:
  - Lots of assumption in btrfs extent buffer = pagesize
- Chaining up extent buffers inside a page
- extent buffer radix indexed by blocksize
- Teach extent buffer readpage/writepage to iterate multiple extents buffers

# *Subpage data blocks*

- Existing extent map code pretty much setup to support range less than a pagesize

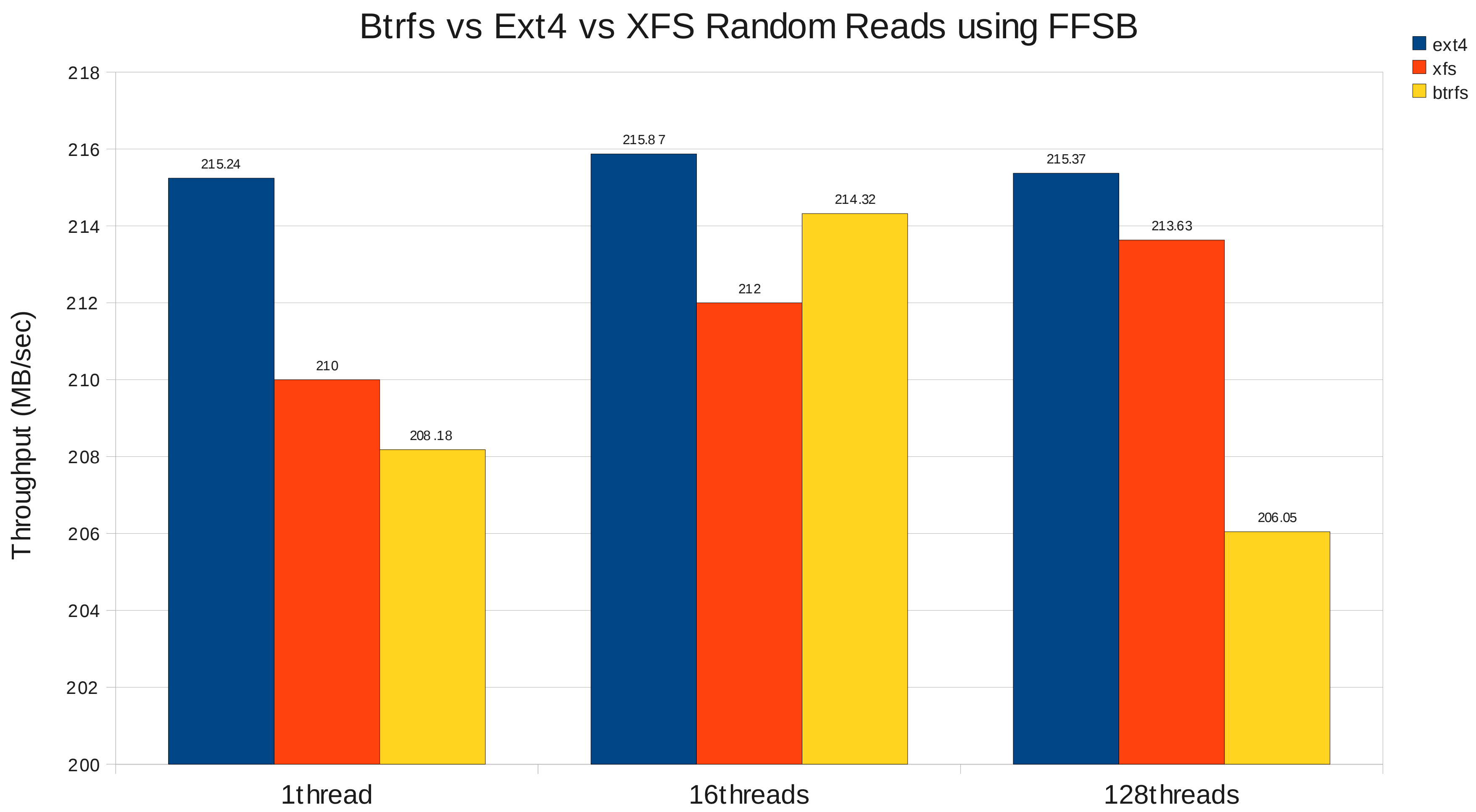
- Just need to link extent map data structures together. Page private points to the first extent map

- Using extent io tree for tracking IO status
- Patch work in progress

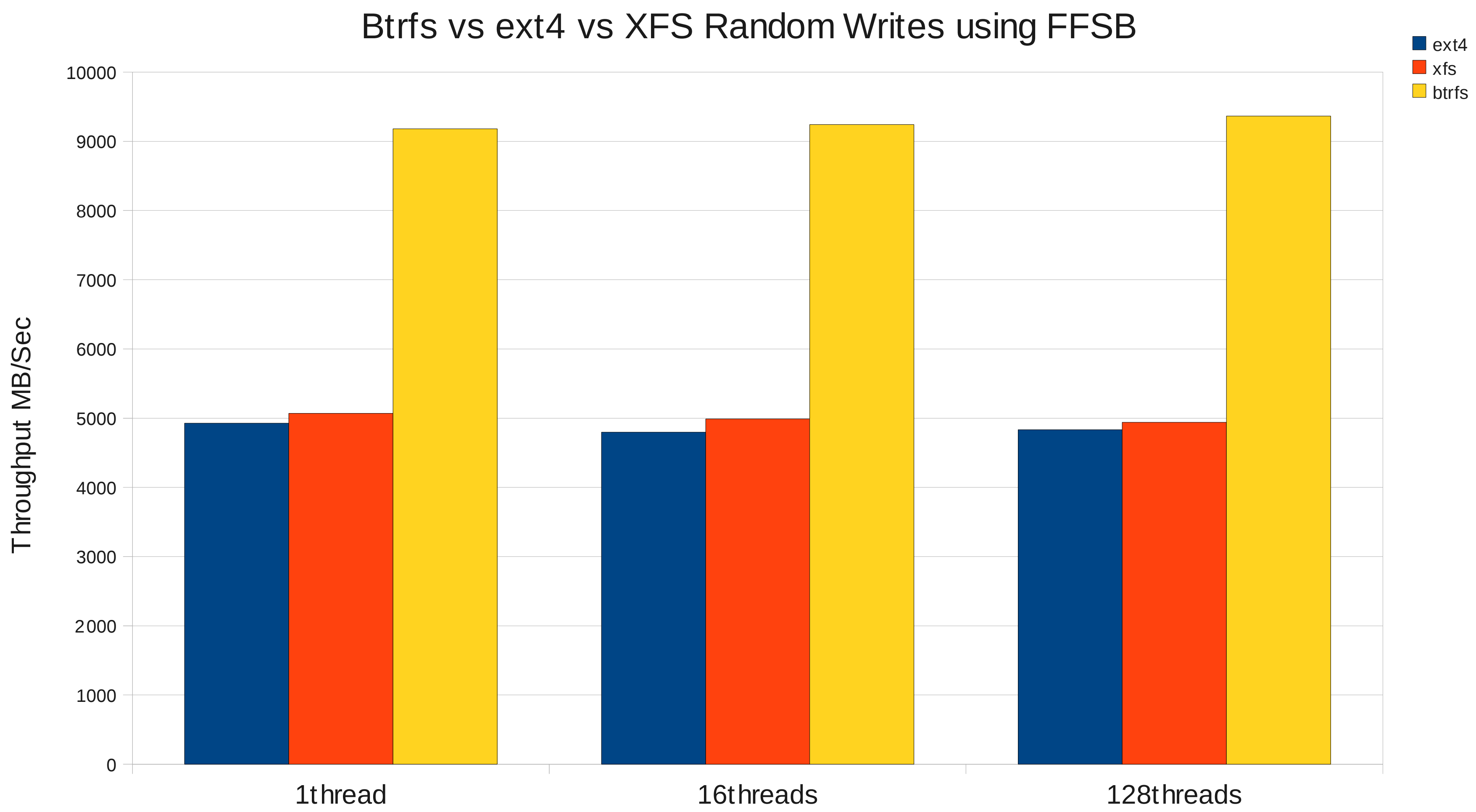
# Btrfs performance

- Overall good for general workload
  - FFSB runs random/sequential read/write tests
  - Fast on random IO operations
  - Slow on metadata-intensive workloads (e.g. Mailserver)
  - Lock contention become visible with multiple threads
  - Tests are done in 2 sockets quad core Intel SAN with LVM

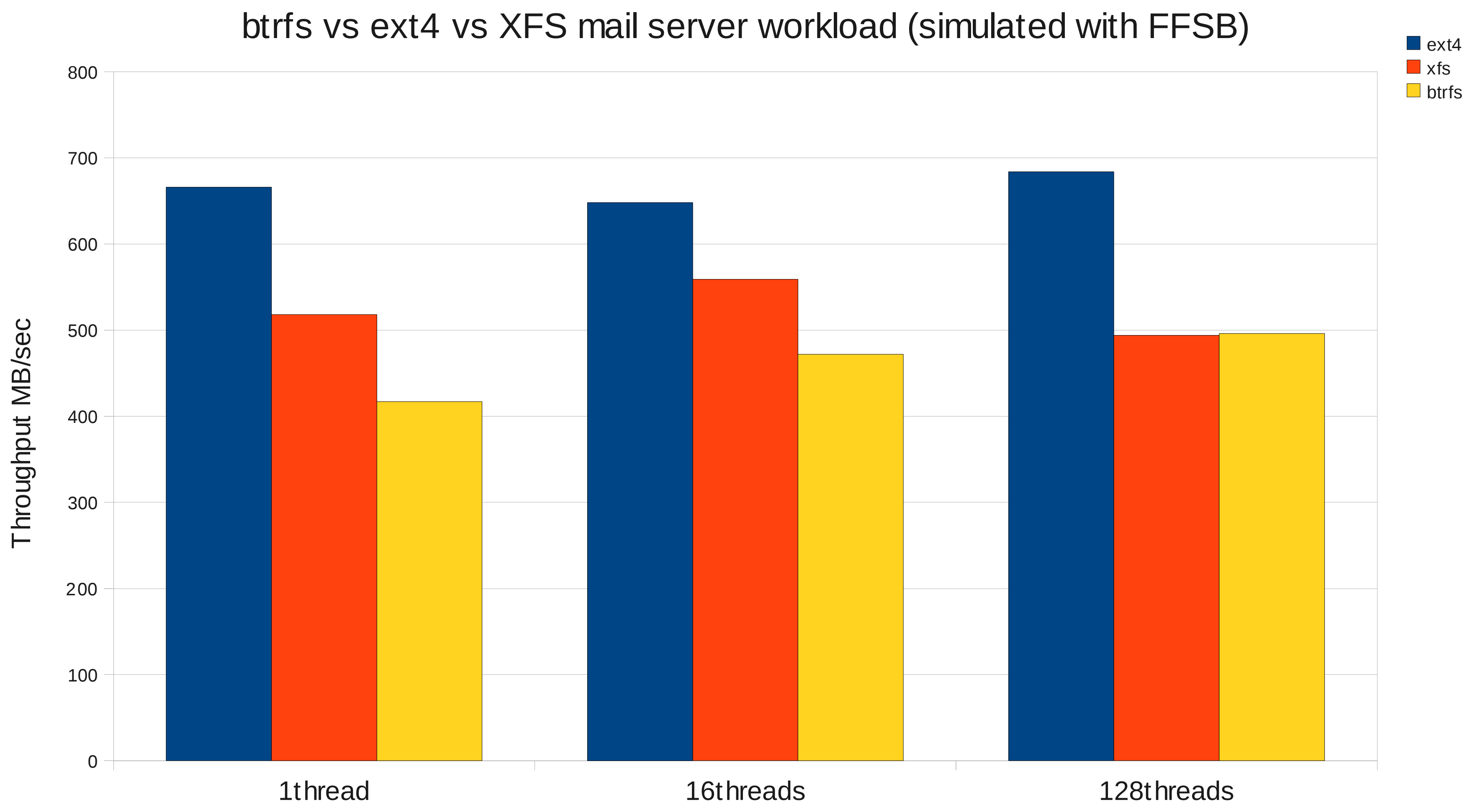
# *Btrfs performance – random reads*



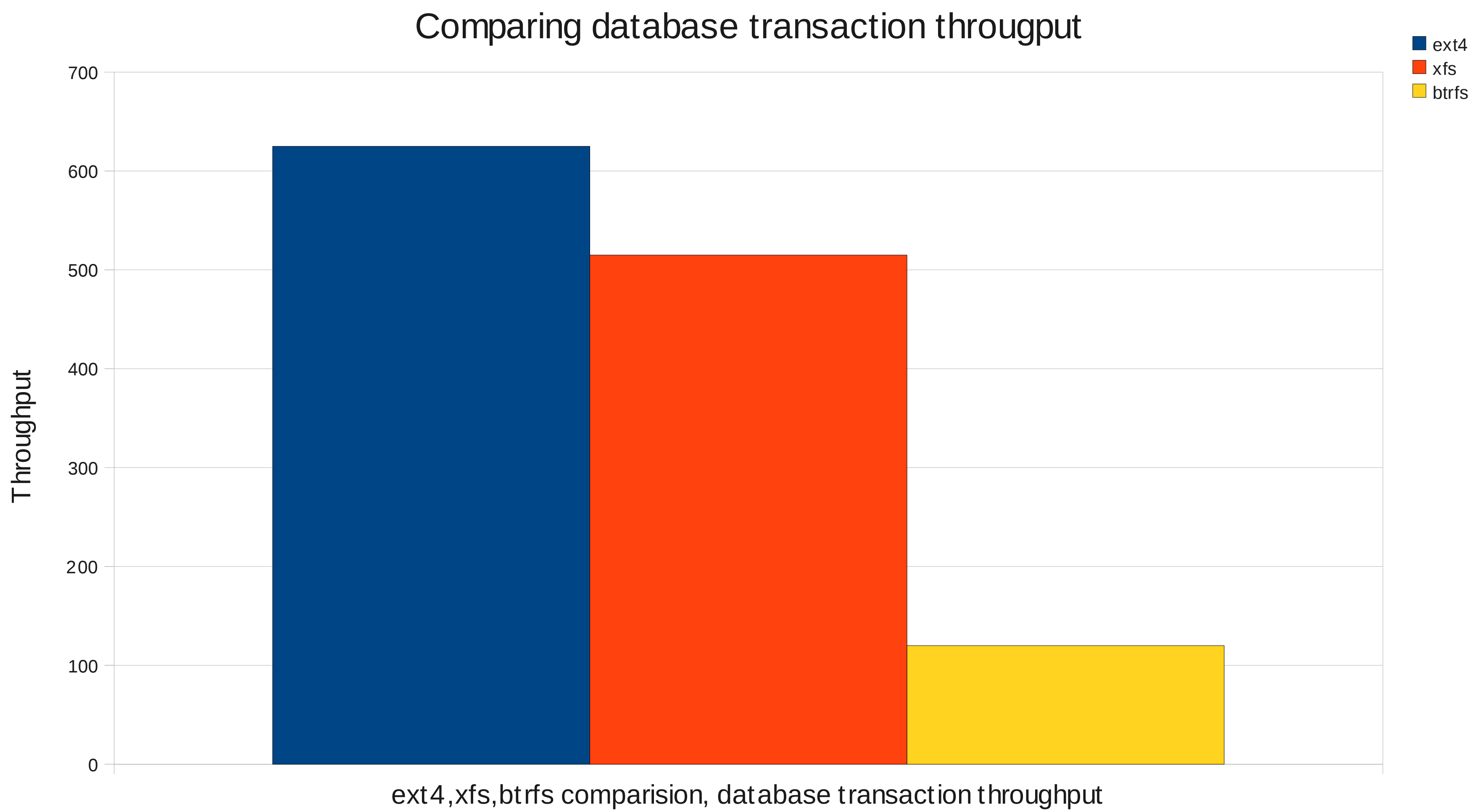
# Btrfs performance – random writes



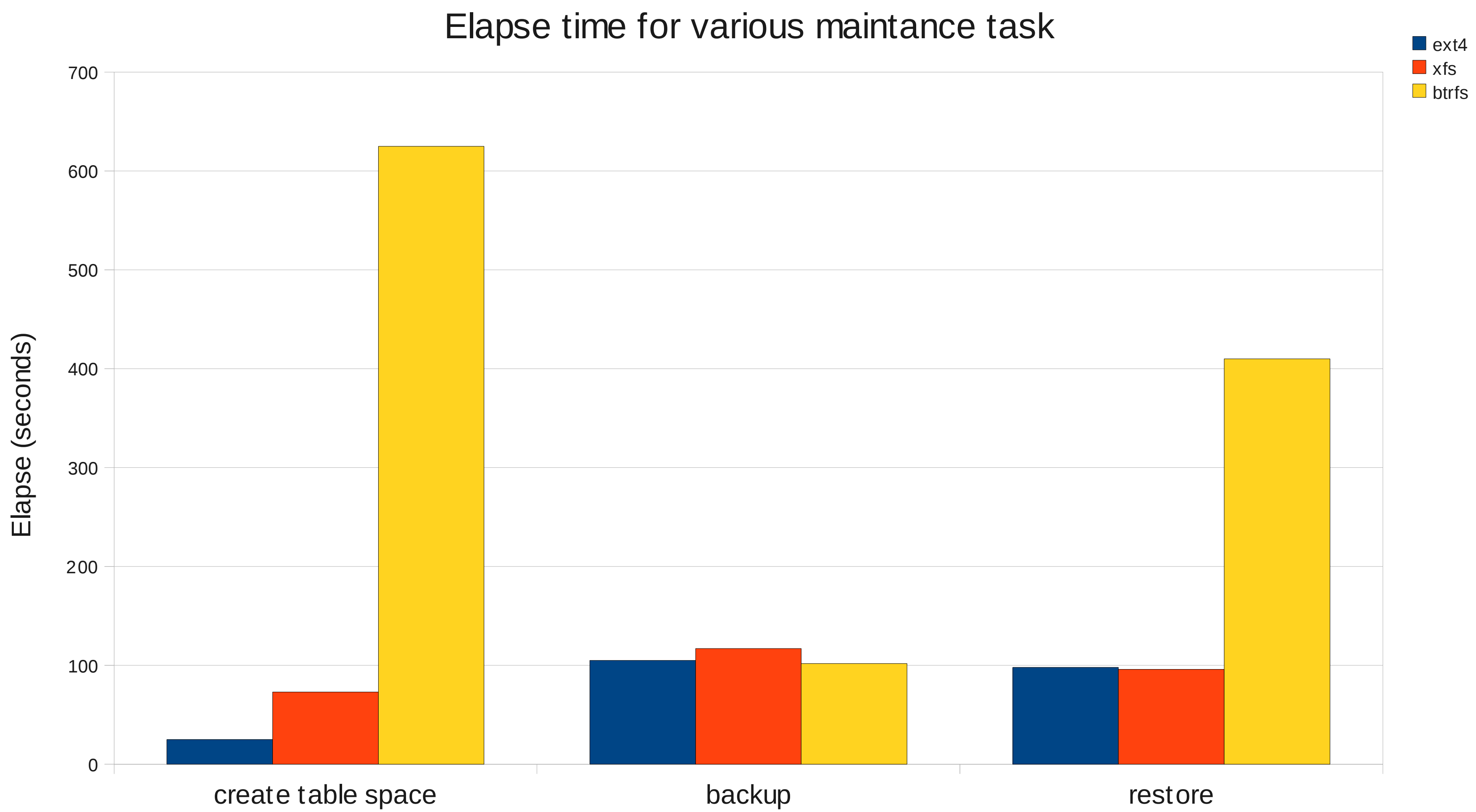
# Btrfs performance – Mail server



# Btrfs performance database workload study



# Btrfs performance database maintenance tasks



# Btrfs database workload study

- Run small OLTP workload on SSDs
  - Btrfs does not perform so well
  - COW causes bad fragmentation
  - Expensive sync operations still
  - DIO, preallocation does not respond well enough with nocow at first

- Further investigate fragmentation issue
  - Large random IO write with fallocate and DIO
  - Using pre-allocation and direct IO does not means nocow all the time
  - The first time fill the pre-allocated space, nocow is used
  - After that the space back to COW, fragmentation starts

# *Btrfs future work*

- RAID 5/6 support
- Efficient Offline fsck
- De-Duplication for btrfs
- ENOSPC issue
- Tiered storage

# *Links*

- Btrfs wiki page

  https://btrfs.wiki.kernel.org/index.php/Main_Page
- Contact cmm@us.ibm.com
- Thanks!