XFS (64 bit FS by SGI)

- **XFS was originally developed by Silicon Graphics**, Inc.
  - **back in the early 90s**.
- At that time,
  - **SGI found that their existing filesystem** (EFS)
  - **was quickly becoming unsuitable**
  - **for tackling** the **extreme computing challenges of** the **day**.

- **Addressing this problem**,
  - **SGI decided to design a completely new high-performance 64-bit filesystem**
  - **rather than attempting to tweak EFS**
  - to do something that it was never designed to do.

- Thus, **XFS was born**, and **was made available to** the **computing public with** the **release of IRIX 5.3 in 1994.**

XFS (64 bit FS by SGI)

- **To this day**,
  - ☞ **it continues to be used** as the **underlying filesystem**
  - ☞ **for all of SGI's IRIX-based products**,
  - ☞ **from workstations to supercomputers**.

- **And now**, **XFS is also available for Linux**.

- The **arrival of XFS for Linux is exciting**,
  - ☞ **primarily because**
  - ☞ **it provides** the **Linux community**
  - ☞ **with** a **robust, refined,** and **very feature-rich filesystem**
  - ☞ **that's capable of scaling to meet** the **toughest storage challenges**.

XFS design

- In the **"Scalability in the XFS Filesystem" paper**
  - ☞ **featured at USENIX '96**,
  - ☞ the **SGI engineers explain** that
  - ☞ **XFS was designed with**
- a **single main idea**: "**think big**"

- Indeed, **XFS has been designed** to
  - ☞ **eliminate** the **limitations**
  - ☞ **found in traditional filesystems**.

Introducing allocation groups

- **When** an **XFS filesystem is created**,
  - the **underlying block device** is **split into**
  - **8 or more equally-sized linear regions**.

- You can think of them as "chunks" or "linear ranges",
  - **but in XFS terminology**
  - **each region is called** an "**allocation group**".

- **Allocation groups are unique in that**
  - **each allocation group manages**
    - **its own inodes** and
    - **free space**,
  - **in effect turning them into** a **kind of sub-filesystem**
  - **that exists transparently within** the **XFS filesystem proper**.

Allocation groups and scalability

- **So, why exactly does XFS have allocation groups**?
- **Primarily, XFS uses allocation groups**
  - **so that it can efficiently handle parallel IO**.
  - **Because each allocation group is effectively** its **own independent entity**,
  - the **kernel can interact with multiple allocation groups simultaneously.**

- **Without allocation groups**,
  - the **XFS filesystem code could become** a **performance bottleneck**,
  - **forcing IO-hungry processes to "get in line"**
  - to make inode modifications
  - or performing other kinds of **metadata-intensive operations**.

Allocation groups and scalability

- **Thanks to allocation groups**,
- the **XFS code will allow multiple threads and processes**
- **to continue to run in parallel**,
- even if many of them are performing non-trivial IO
  - **on the same filesystem.**

- So, **match XFS with some high-end hardware** and
  - you'll **get high-end results**
  - **rather than** a **filesystem bottleneck**.

- **Allocation groups also help to optimize**
  - **parallel IO performance** on **multiprocessor systems**,
  - **because more than one metadata update**
  - **can be "in transit" at** the **same time**.

B+ trees everywhere (for free space)

- Internally,
  - **allocation groups use efficient B+ trees**
  - **to keep track of important data such as**
    - **ranges of free space** (also called "extents") ,
    - **as well as inodes**.
- The **ability to find regions of free space quickly**
  - **is critical for maximizing write performance**,
  - **which is something that XFS is very good at**.

- In fact, **each allocation group has two B+ trees**
- **used to keep track** of **free space**;
- **1. one B+ tree** **(sizes)**
- **stores:**
  - the **extents** of **free space**
  - **ordered by size**,
- and
- **2. other B+ tree** **(beginning addresses)**
- **has**
  - the **regions ordered** by
  - **their starting physical location on** the **block device.**

B+ trees everywhere (for inodes)

- **XFS is also very efficient**
- **when it comes to** the **management of inodes**

- **Each allocation group allocates inodes as needed**,
  - ☞ **in groups of 64**

- An **allocation group keeps track** of **its own inodes**
  - ☞ **by using** a **B+ tree**
  - ☞ **that records where each particular inode number**
  - ☞ **can be found on disk**.

- You'll find that
- **XFS uses B+ trees as much as possible**,
  - ☞ **due to their excellent performance** and
  - ☞ **tremendous scalability.**

Journaling

- **Like ReiserFS**,
- **XFS only journals metadata**, and
- **does not take any special precautions to ensure that** the **data makes it to disk before metadata is written**.

- **This means** that **with XFS** (just like with **ReiserFS**),
  - it's possible for recently modified data to be lost
  - in the event of an unexpected reboot.

- However, a couple of properties of XFS' journal
- make this issue less common than it is with ReiserFS.

Journaling

- **With ReiserFS**,
  - an **unexpected reboot can result**
  - **in recently modified files**
  - **containing portions** of **previously deleted files**.
- **Besides the obvious data loss**, this could also **theoretically pose** a **security threat**.

- In contrast,
- **XFS ensures that**
  - any **unwritten data blocks are zeroed on reboot**,
  - **when XFS journal is replayed**.
- Thus, **missing blocks are filled** with **null bytes**,
  - **eliminating** the **security hole** –
  - a **much better approach**.

Journaling

- **Now, what about** the **data loss issue itself**?
- **In general**,
- **this problem is minimized with XFS**
- **due to the fact that**
  - **XFS generally, writes pending metadata updates to disk**
  - **much more frequently** than **ReiserFS does**,
  - **especially during periods high disk activity**.

- Thus, **in** the **event of** a **lockup-failure**,
  - you **will generally lose**
  - **fewer of your recent metadata modifications**
  - than you **would with ReiserFS**.
- Of course,
  - **this does not directly address**
  - the **problem of not writing data blocks in time**,
  - **but writing metadata more frequently**
  - **does encourage data to be written more frequently as well**.

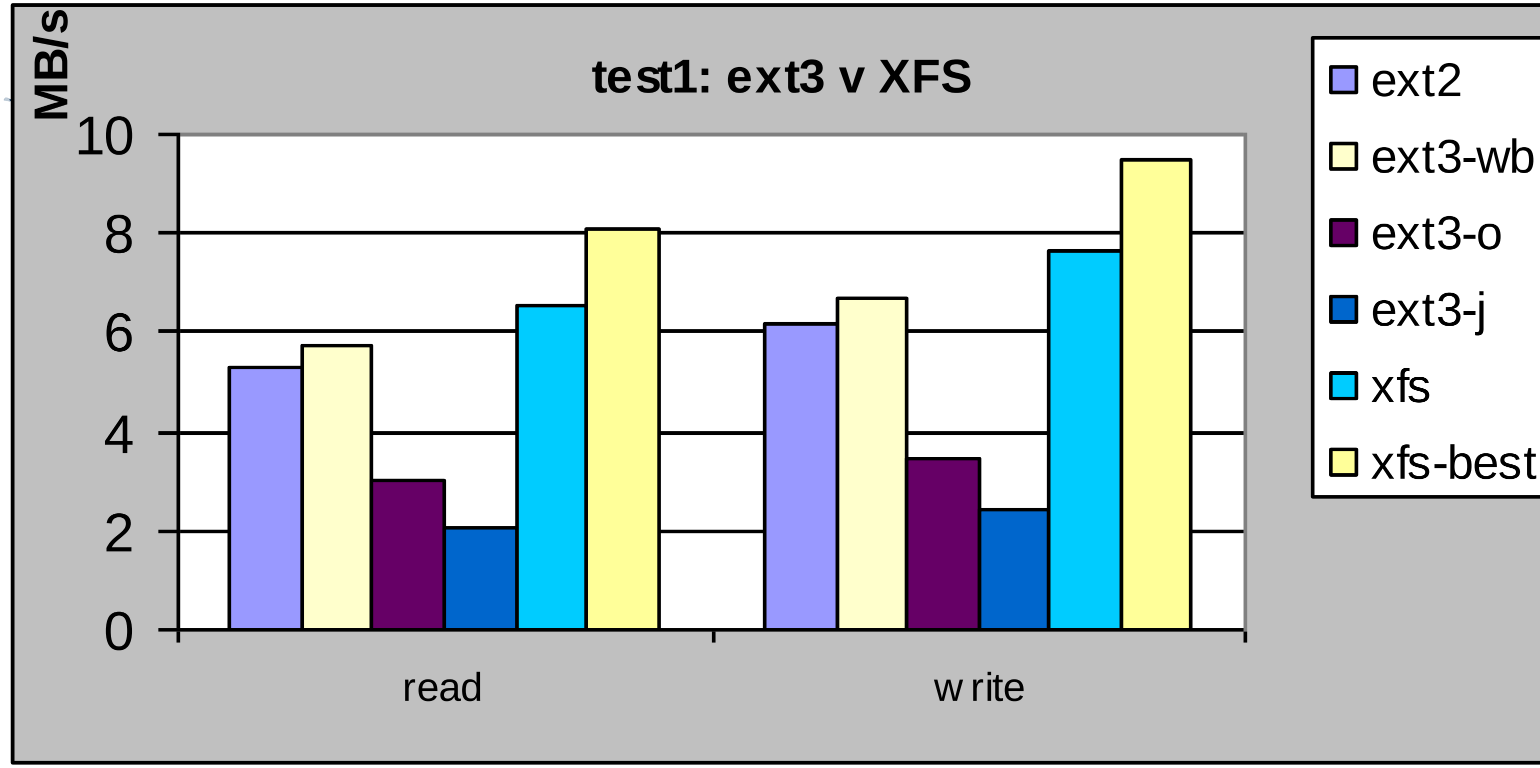
Delayed allocation

- **delayed allocation**, a **feature unique to XFS**.
- the term **allocation refers**
  - **to** the **process of finding regions of free space**
  - **to use for storing new data**.
- **XFS handles allocation by breaking it into** a **two-step process**.
- **First**,
  - **when XFS receives new data to be written,**
  - **it records** the **pending transaction in RAM** and
  - simply **reserves** an **appropriate amount** of **space on the underlying filesystem.**
- **However, while XFS reserves space for** the **new data**,
  - **it doesn't decide**
  - **what filesystem blocks will be used to store** the **data**,
  - **at least not yet**.
- **XFS procrastinates**, (odugovlačiti)
  - **delaying this decision**
  - **to** the **last possible moment**,
  - **right before this data is actually written to disk**

Delayed allocation

- **By delaying allocation**,
  - XFS **gains many opportunities**
  - **to optimize write performance.**
- **When it comes time to write** the **data to disk**,
  - **XFS can now allocate free space intelligently**,
  - **in a way that optimizes filesystem performance**.
- In particular,
  - **if** a **bunch of new data is being appended to** a **single file**,
  - **XFS can allocate** a **single**, **contiguous region on disk**
  - **to store this data**.
- **If XFS hadn't delayed its allocation decision**,
  - it may have unknowingly written the data
  - **into multiple non-contiguous chunks**,
  - **reducing write performance**.

ext3 v XFS: small file performance



ext3 v XFS: ultra small file performance