PADERBORN CENTER FOR PARALLEL COMPUTING

hashFS

Applying Hashing to Optimize File Systems for Small File Reads

Paul Lensing, Dirk Meister, André Brinkmann
Paderborn Center for Parallel Computing
University of Paderborn

Contents

Contents

Scenario

- Web traffic, e.g. profile images, web site archive
  - Internet Archive
    - > 2500 nodes, > 6000 hard disks
  - Similar: Facebook profile images
    - > 6.5 billion images, most images 5-20 KB

- Challenges:
  - Access to small files
  - Multiple IOs per read request
  - ➔ High Overhead
  - ➔ Limits reads/s per disk

PC² PADERBORN CENTER FOR PARALLEL COMPUTING

Motivation

- Small files (4 KB – 20 KB)
- Accesses are almost exclusively reads

- Filenames are randomly generated or calculated
- High number of concurrent users
- → No name or directory locality
- → Accesses are randomly distributed

- RAM/disk ratio low

- Some POSIX features are not important
  - Directory permission
  - Last file access time (atime)

Recursive directory lookup

Example: Read file /var/image/20

1. Read dentries /
2. Read dentries /var
3. Read dentries /var/image
4. Read file /var/image/20

Partial solution

- Store small file data in inode
  - Eliminates 1 IO operation
  - But that operation is not seeking anyway

- Directory lookups are a major problem

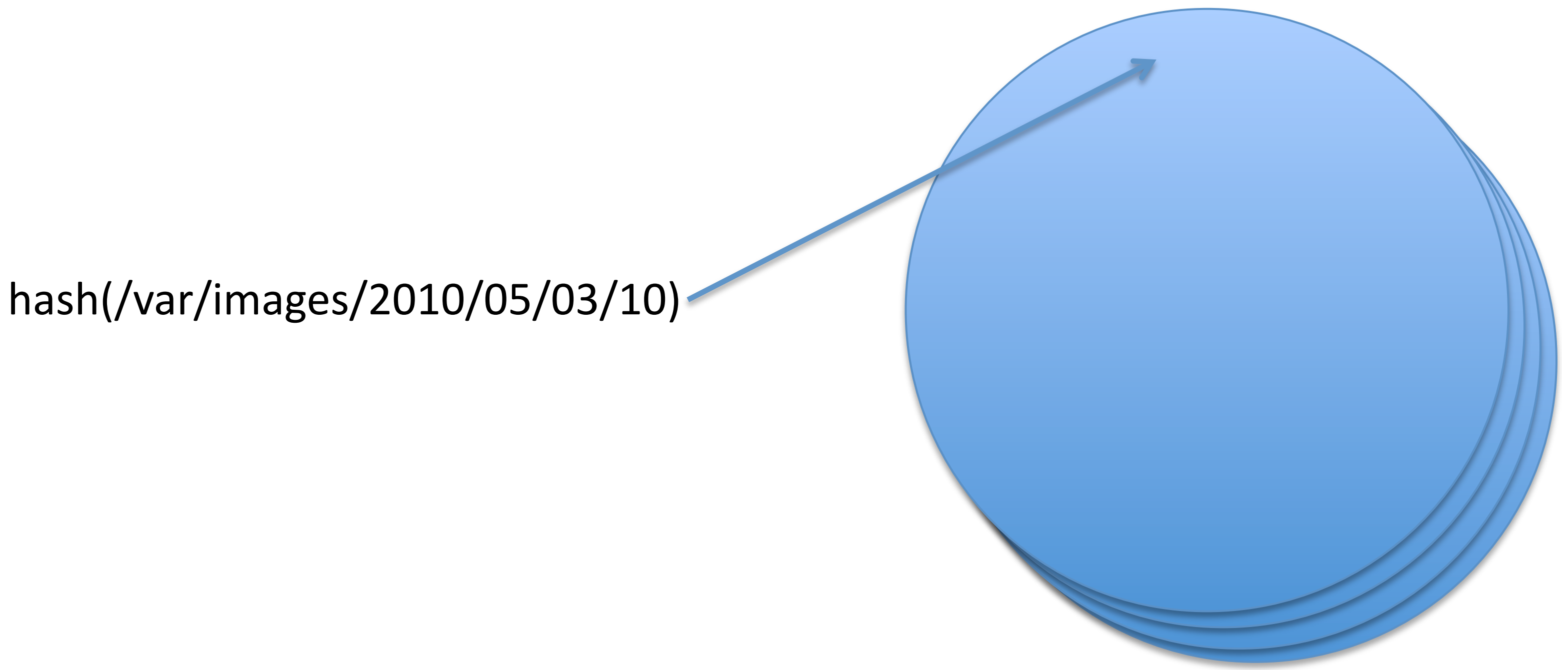
Goal:

1 IO operation per read request
(in web scenario)

Contents

Basic Idea

- Compute location of file
  – No (recursive) lookup
- hash file path to position on disk



Hashing to tracks

- Exact position is not a good approach
  – Collisions

- Hash pathname to “track”
  – Read complete track
    - Overhead, but not prohibitive in random workload
  – Needs geometry data of disk
  – Also possible to use each data region as “track”

Contents

- Motivation and Problem
- Design Idea
- **Implementation**
- Evaluation
- Conclusion

hashFS

- Prototype filesystem
- Based on ext2
- Transparent to application and 3rd party tools
  - No library, etc.
  - Except directory permission checks
- Metadata
  - **Track inode** per hashed file
  - Stored in **track inode block** per track

Track Meta Data

- Every file has additional track inode
  - Only vital information
    - Inode number, size, security,
    - Identity hash, collision bit
    - n direct pointers
- Track inode block
  - Stored all track inodes of track
  - First file system block of track
  - Default: 113 track inodes per track inode block

PADERBORN CENTER FOR PARALLEL COMPUTING

Write new file

```
Hash pathname to track
Hash pathname to identity hash
Write normal inode
Read track inode block
Search track inode via identity hash
If track inode exists: # Collision
    Set collision bit
Else:
    Create track inode
```

Read file

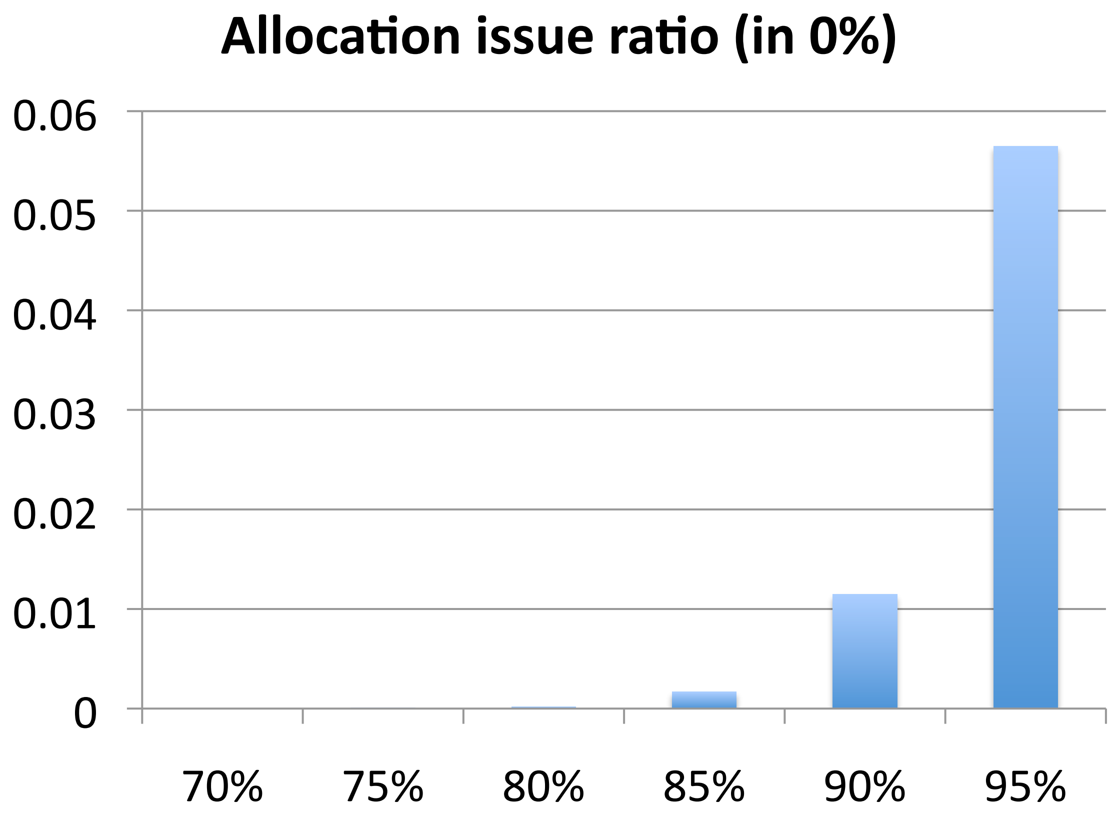
```
Hash pathname to track
Hash pathname to identity hash
Read track inode block
Search for track inode via identity
  hash
If not exists or if collision bit is
  set:
   Fail # Use normal lookup
Else:
   Success
```

Block Allocation

- Allocate block for „hashed file“
- Possible issue:
  - No free blocks on hashed track
- Basic approach:
  - Keep ratio of track free for hashed files
  - Depending on file system utilization
  - Use hashing approach only for small files

Other Issues

- Hashed track completely used by fs meta data
  - Prevented by eliminating track from geometry data
- Duplicate identity hashes
- No free track inodes in track inode block
- → Alternative lookup fails
  - Normal lookup necessary as backup



Simulation using file set distribution as on central AFS filesystem at UPB

Integration in Linux

- Modification of VFS layer
  - Bypass recursive directory lookup by pathname method
  - Additional function pointer
    - Transparent to other file systems
  - If pathname lookup has no result
    - Use recursive directory lookup
- hashFS as additional filesystem module
  - geometry data via module options

PADERBORN CENTER FOR PARALLEL COMPUTING

Contents

- Motivation and Problem
- Design Idea
- Implementation
- **Evaluation**
- Conclusion

Evaluation

- Random file access distribution
  - Uniform distribution
  - Zipf-like distribution
- Web traffic is assumed to be zipf-like
  - CDNs, memcache or other caching layers flatten the skewness

Setting

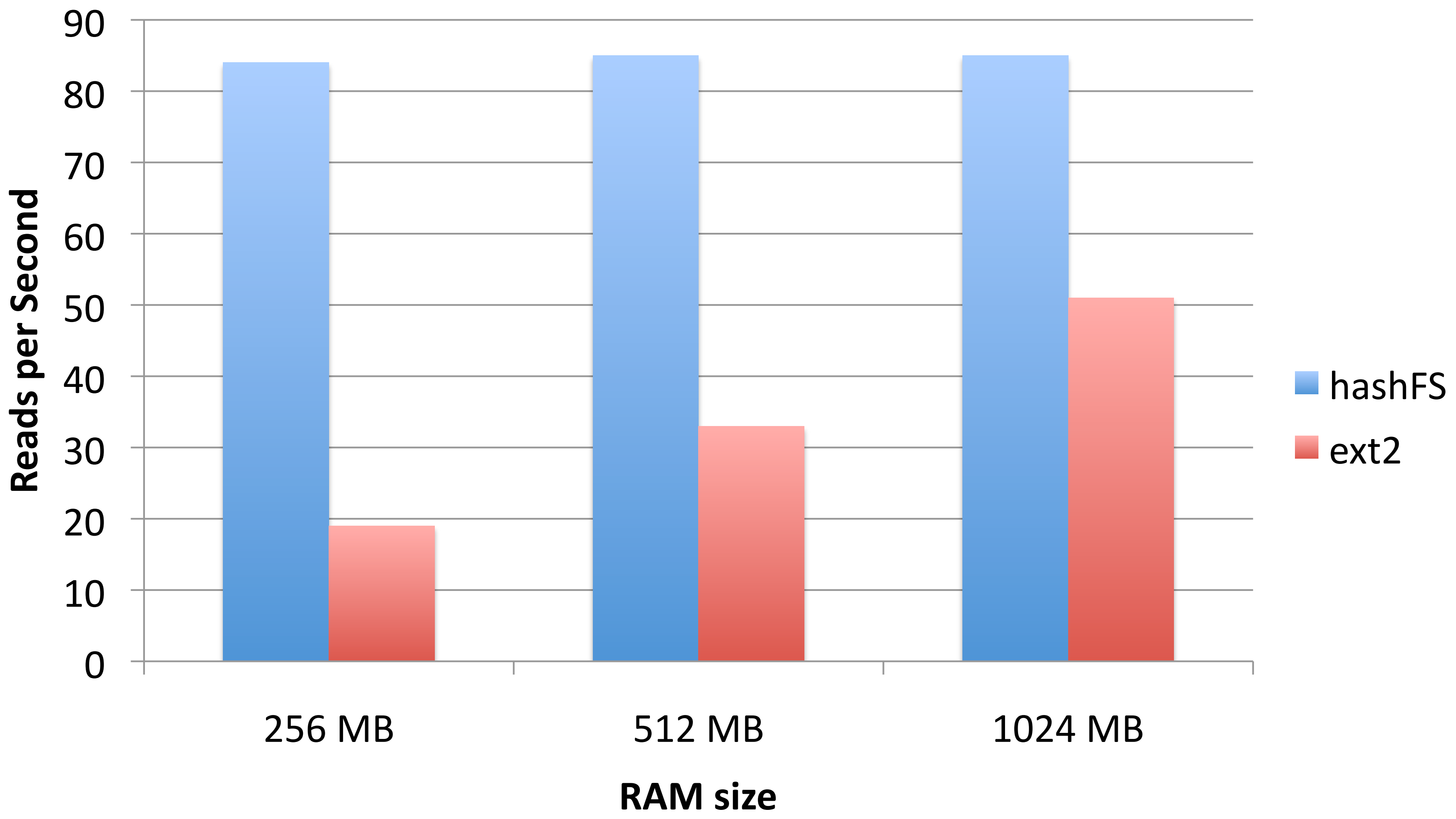
- File size: 4 KB
- File count: 18 Million

- Track size: 465 sectors per track
  - Heuristic: tracks have different sizes on disk
- Partition: 90 GB
  - Small partition to limit evaluation time
  - Adjust cache size accordingly
- RAM size: 256 MB, 512 MB, 1024 MB

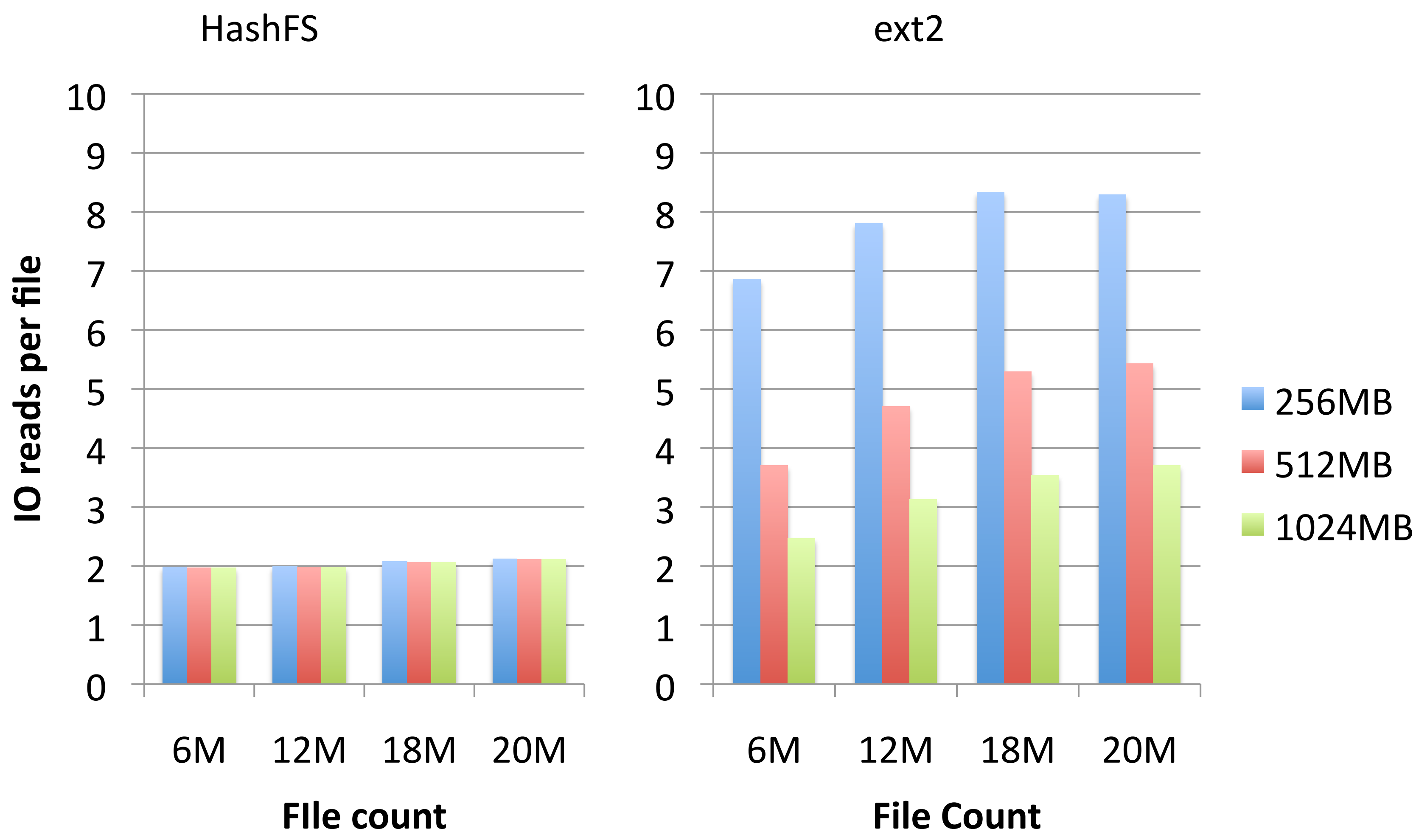
File sets

- Flat File Set:
  - Files per directory: 10,000
  - Max depth of directory: 2
- Deep File Set:
  - Files per directory: 100
  - Max depth of directory: 6
- Deep File Set better for ext2
  - Except with large caches
- Here: Only results for Deep File Set

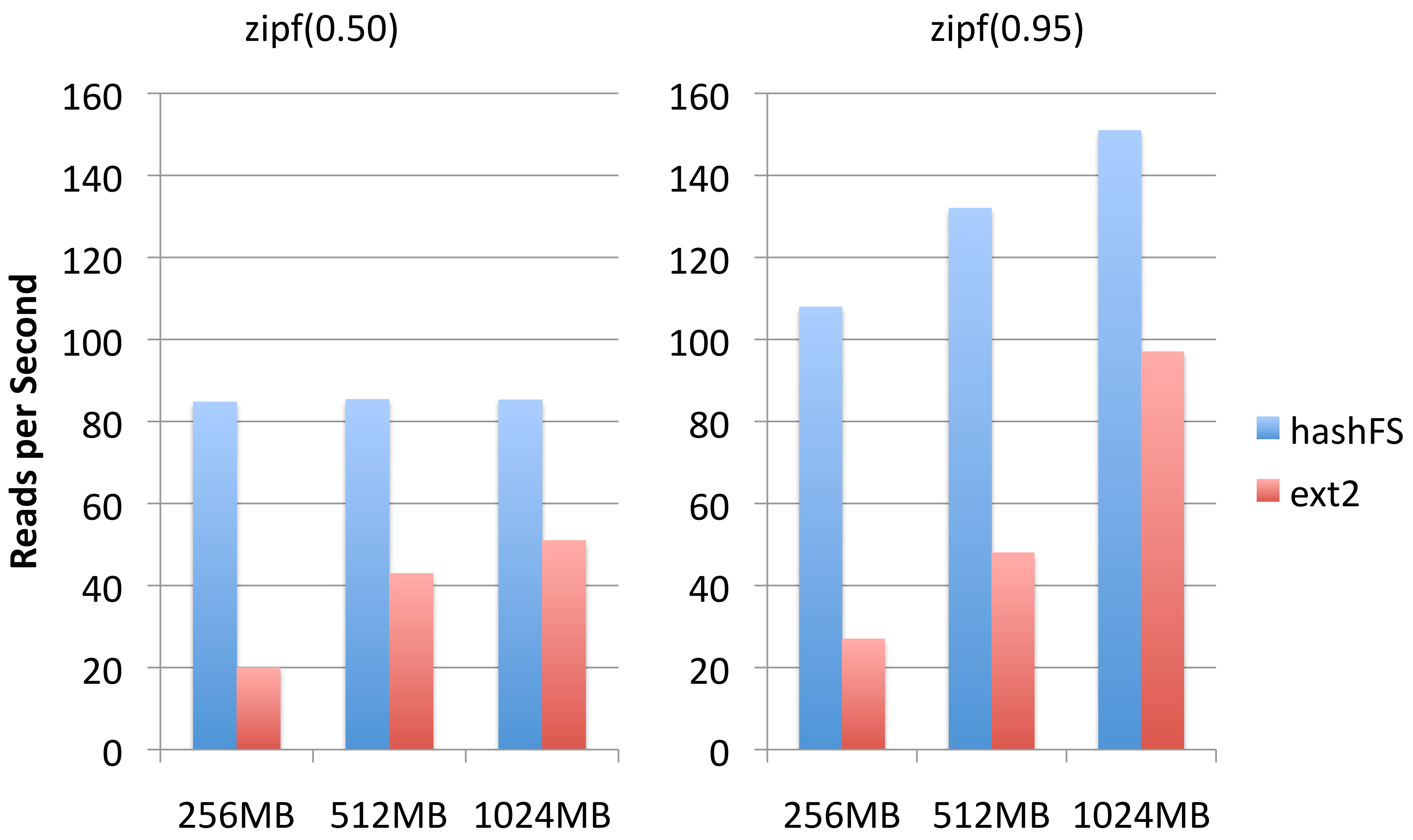
Uniform distribution



IO operations per file



Zipf distribution



Evaluation Summary

- hashFS
  – Nearly constant access time
  – Only influenced by allocation problems due to utilization
- ext2
  – Very different performance values
  – Depends on
    - Cache size and cache state
    - Number and depth of directories
    - Number of files

Contents

Limits and State

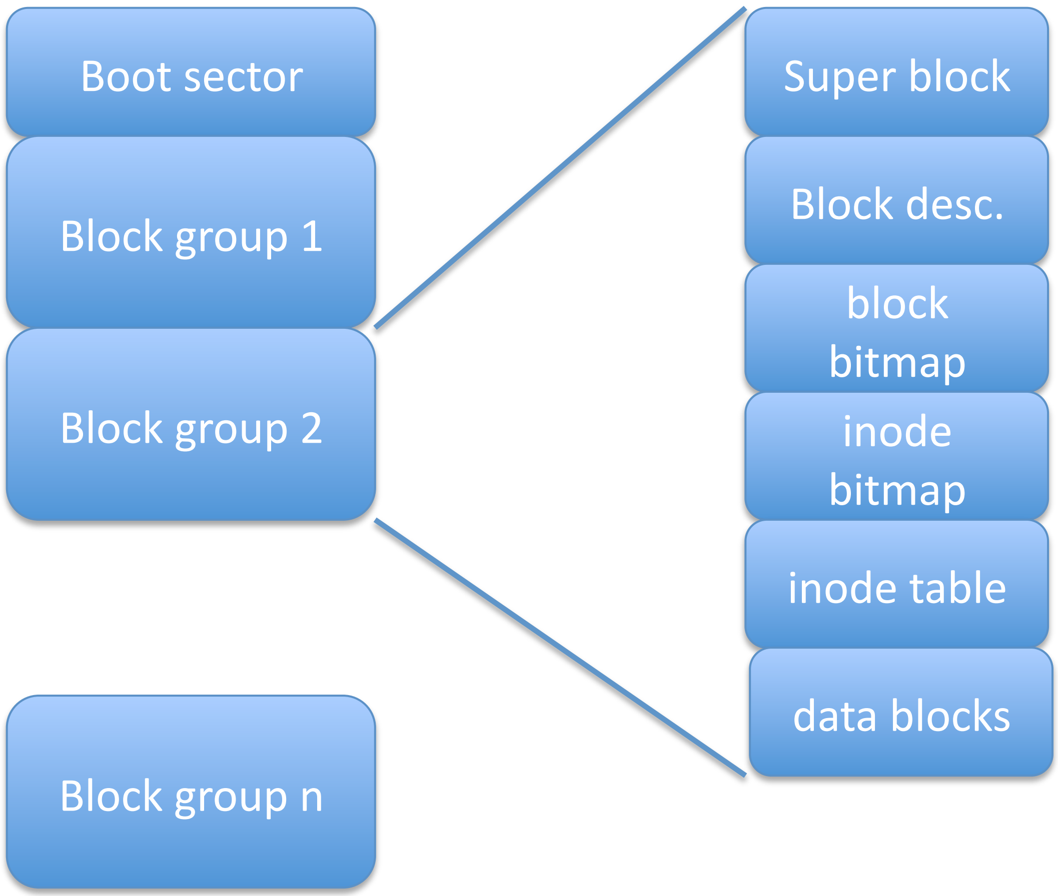
- No full POSIX conformity
  - Directory access permissions are not checked
  - noatime implicit
- Negative performance impact for
  - Large file reads
  - Writes (update to track metadata)
- Current state
  - Prototype implementation in Linux 2.6.18
  - Writes are implemented poorly
  - Evaluation in more complex settings

Conclusion

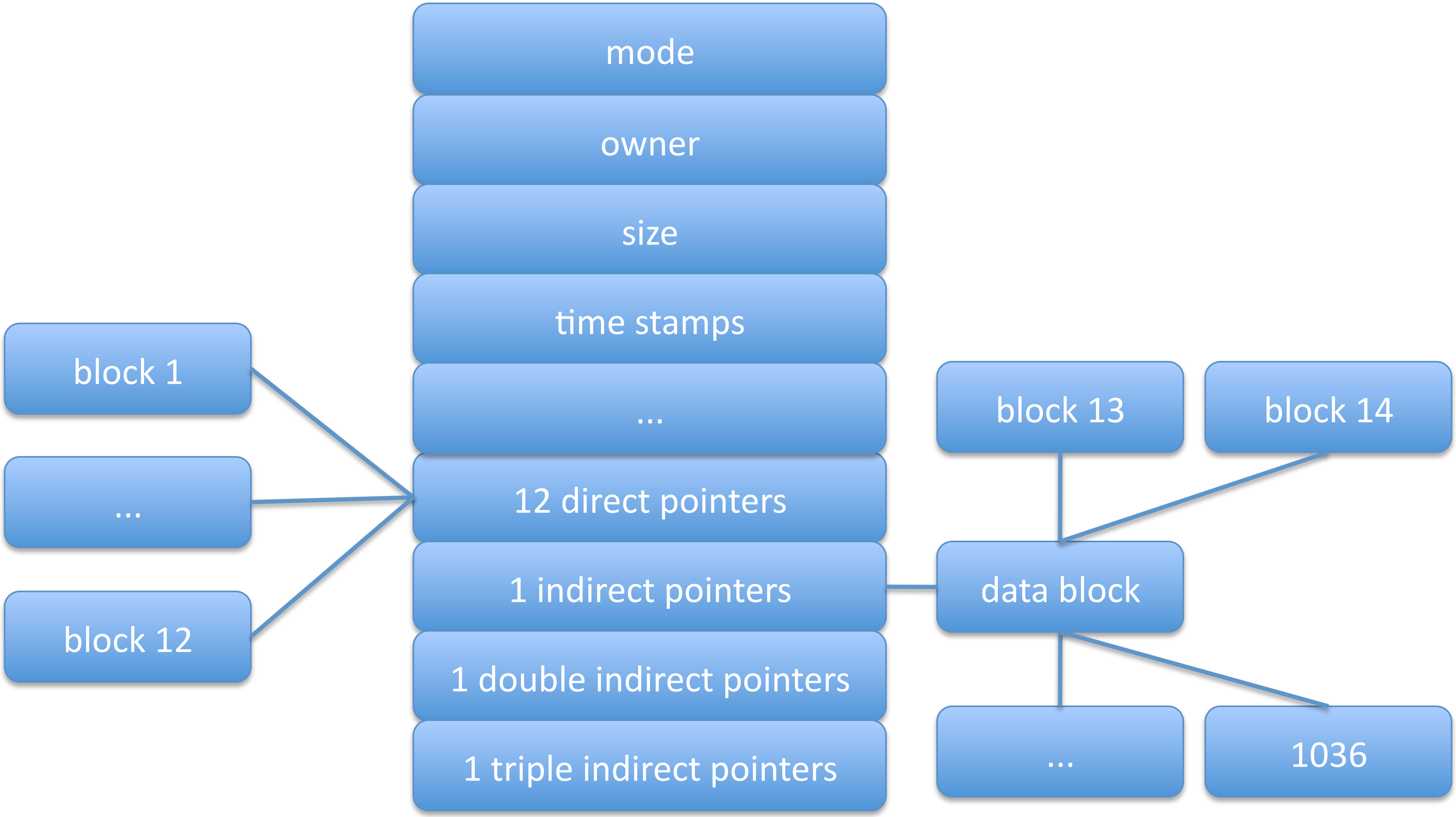
- Hashing pathnames is viable approach
  - limited to certain situations
- Approach not limited to ext2
  - All general-purpose file systems use recursive directory lookup approach
  - IO operations for lookup at cache miss

Questions

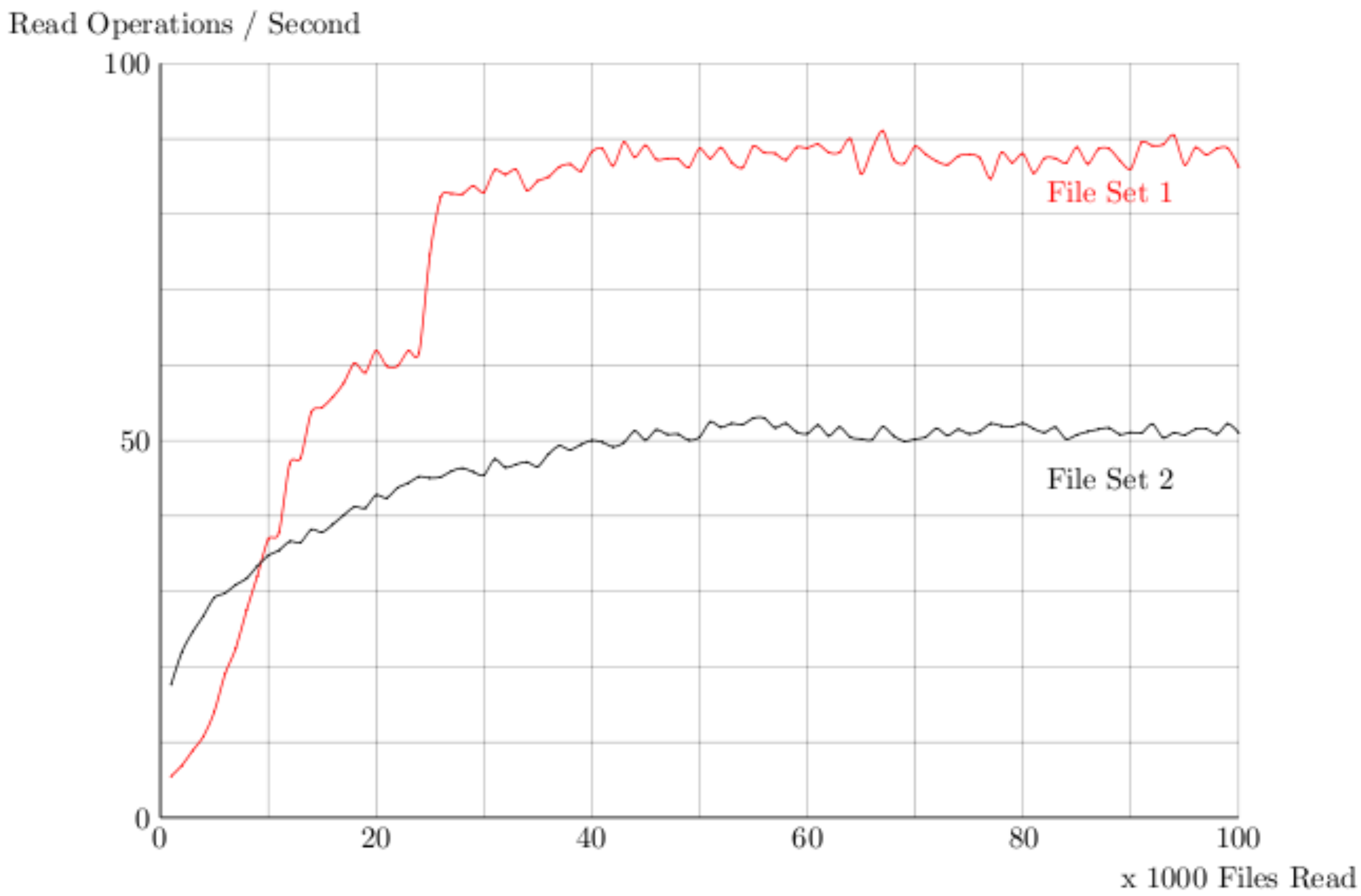
Background:ext2



Background: inodes



Influence of Cache State (ext2)



Warm-up Phase

Track inode issues

number of files	Disk Utilization	Track inodes issues (average)	Ratio (average)
1-13 million	<69.6%	0	0%
14 million	74.7%	1.9	< 0.0001%
15 million	79.8%	25.7	0.0002%
16 million	84.9%	277.9	0.0017%
17 million	90.0%	1946.6	0.0115%
18 million	95.1%	10,170.0	0.0565%

Simulation using file set distribution as on central AFS filesystem at UPB