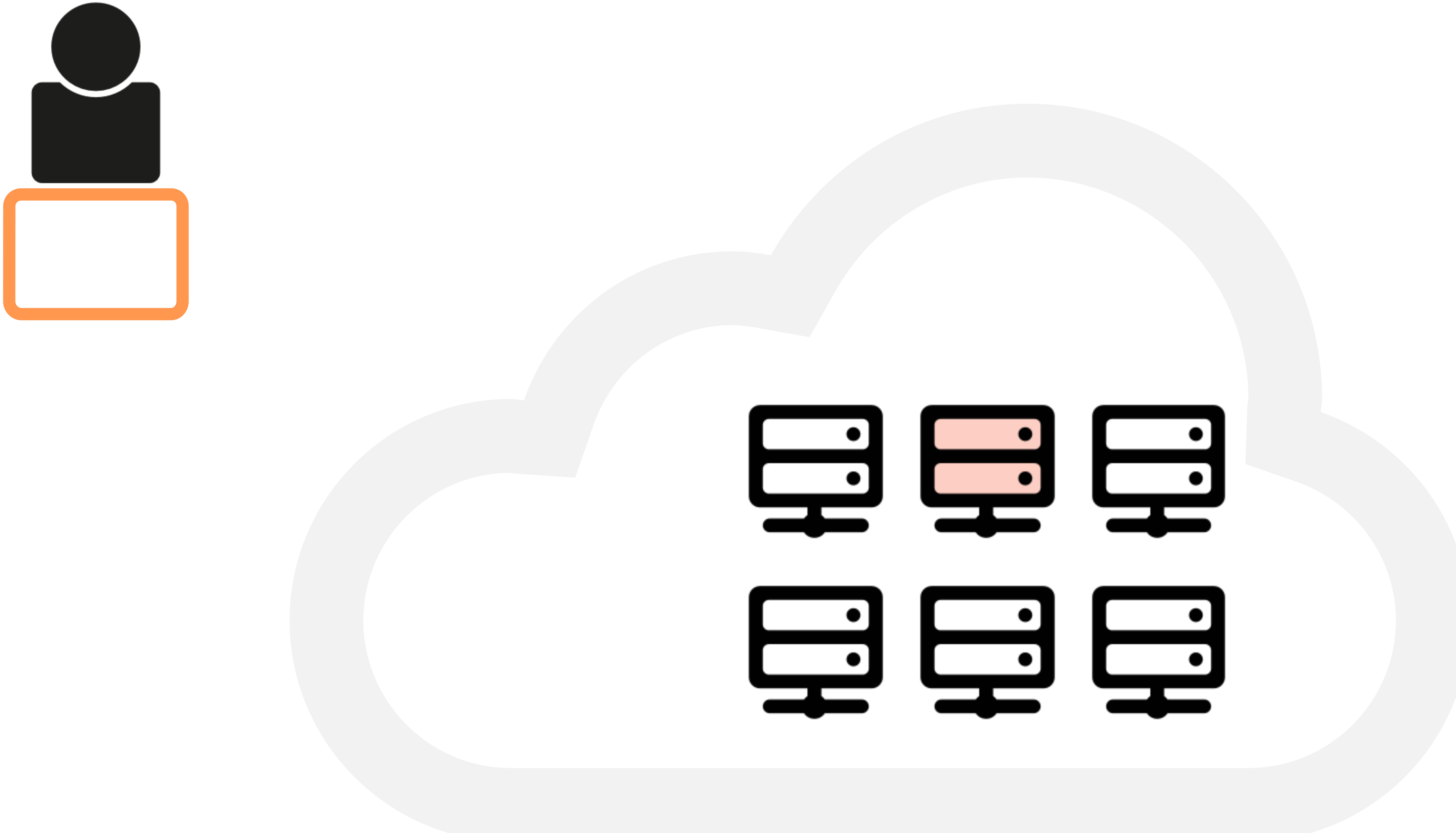
Reducing File System Tail Latencies with *Chopper*

Jun He, Duy Nguyen[+],
Andrea C. Arpaci-Dusseau,
Remzi H. Arpaci-Dusseau

Department of Computer Sciences, [+]Department of Statistics
University of Wisconsin, Madison

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Uncommon tail latencies become common at scale



Uncommon tail latencies become common at scale

DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

SYSTEMS THAT RESPOND to user actions quickly (within 100ms) feel more fluid and natural to users than those that take longer.[3] Improvements in Internet connectivity and the rise of warehouse-scale computing systems[2] have enabled Web services that provide fluid responsiveness while consulting multi-terabyte datasets spanning thousands of servers; for example, the Google search system updates query results interactively as the user types, predicting the most likely query based on the prefix typed so far, performing the search and showing the results within a few tens of milliseconds. Emerging augmented-reality devices (such as the Google Glass prototype[7]) will need associated Web services with even greater responsiveness in order to guarantee seamless interactivity.

It is challenging for service providers to keep the tail of latency distribution short for interactive services as the size and complexity of the system scales up or as overall use increases. Temporary high-latency episodes (unimportant in moderate-size systems) may come to dominate overall service performance at large scale. Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts; we refer to such systems as "latency tail-tolerant," or simply "tail-tolerant." Here, we outline some common causes for high-latency episodes in large online services and describe techniques that reduce their severity or mitigate their effect on whole-system performance. In many cases, tail-tolerant techniques can take advantage of resources already deployed to achieve fault-tolerance, resulting in low additional overhead. We explore how these techniques allow system utilization to be driven higher without lengthening the latency tail, thus avoiding wasteful overprovisioning.

Why Variability Exists?

Variability of response time that leads to high tail latency in individual components of a service can arise for many reasons, including:

Shared resources. Machines might be shared by different applications contending for shared resources (such as CPU cores, processor caches, memory bandwidth, and network bandwidth), and within the same application different requests might contend for resources;

Daemons. Background daemons may use only limited resources on average but when scheduled can generate multi-millisecond hiccups;

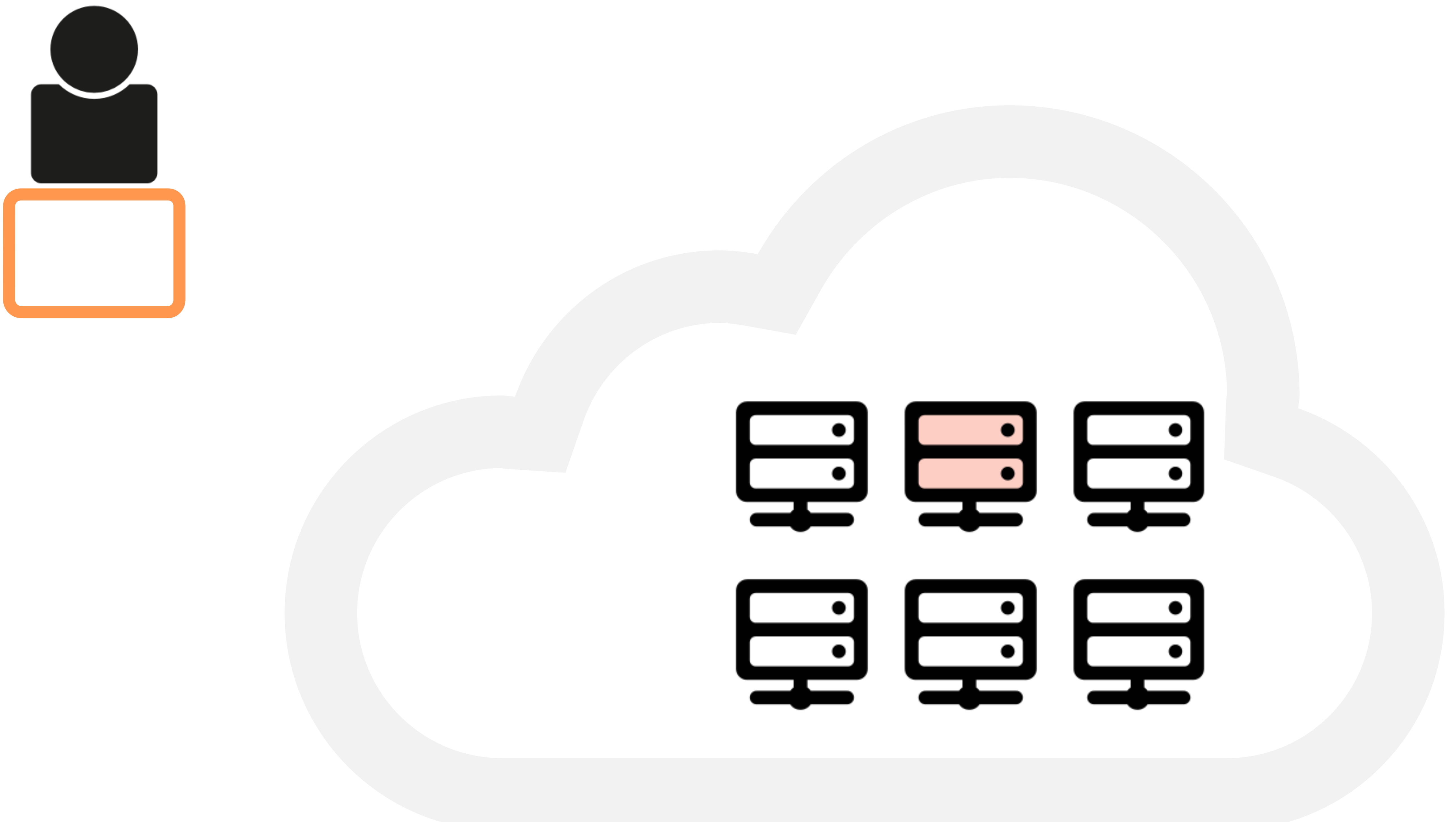
» **key insights**

- Even rare performance hiccups affect a significant fraction of all requests in large-scale distributed systems.
- Eliminating all sources of latency variability in large-scale systems is impractical, especially in shared environments.
- Using an approach analogous to fault-tolerant computing, tail-tolerant software techniques form a predictable whole out of less-predictable parts.

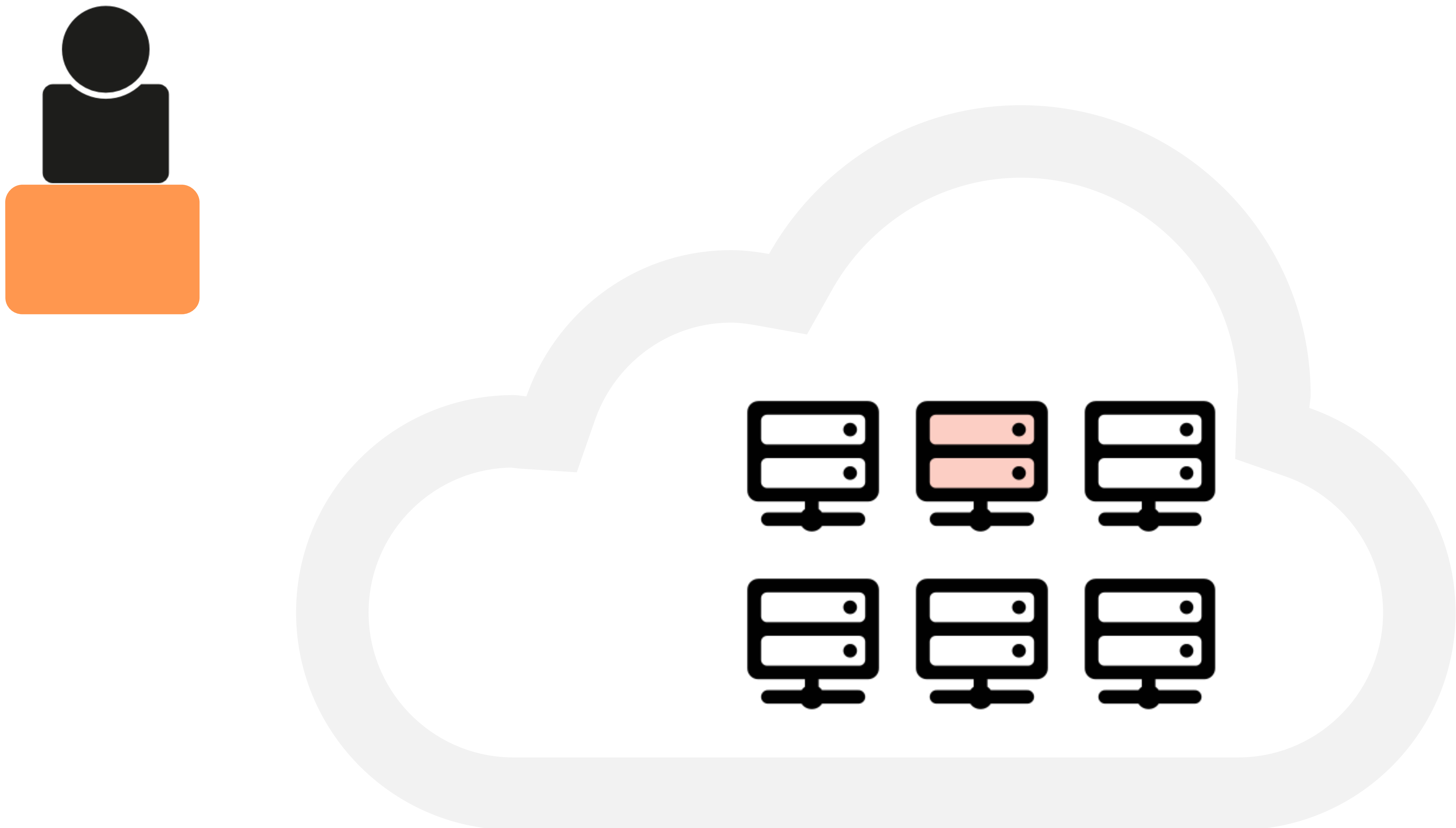
ILLUSTRATION BY GIACOMO MARCHESI

"Temporary high-latency episodes (unimportant in moderate-size systems) may come to dominate overall service performance at large scale."

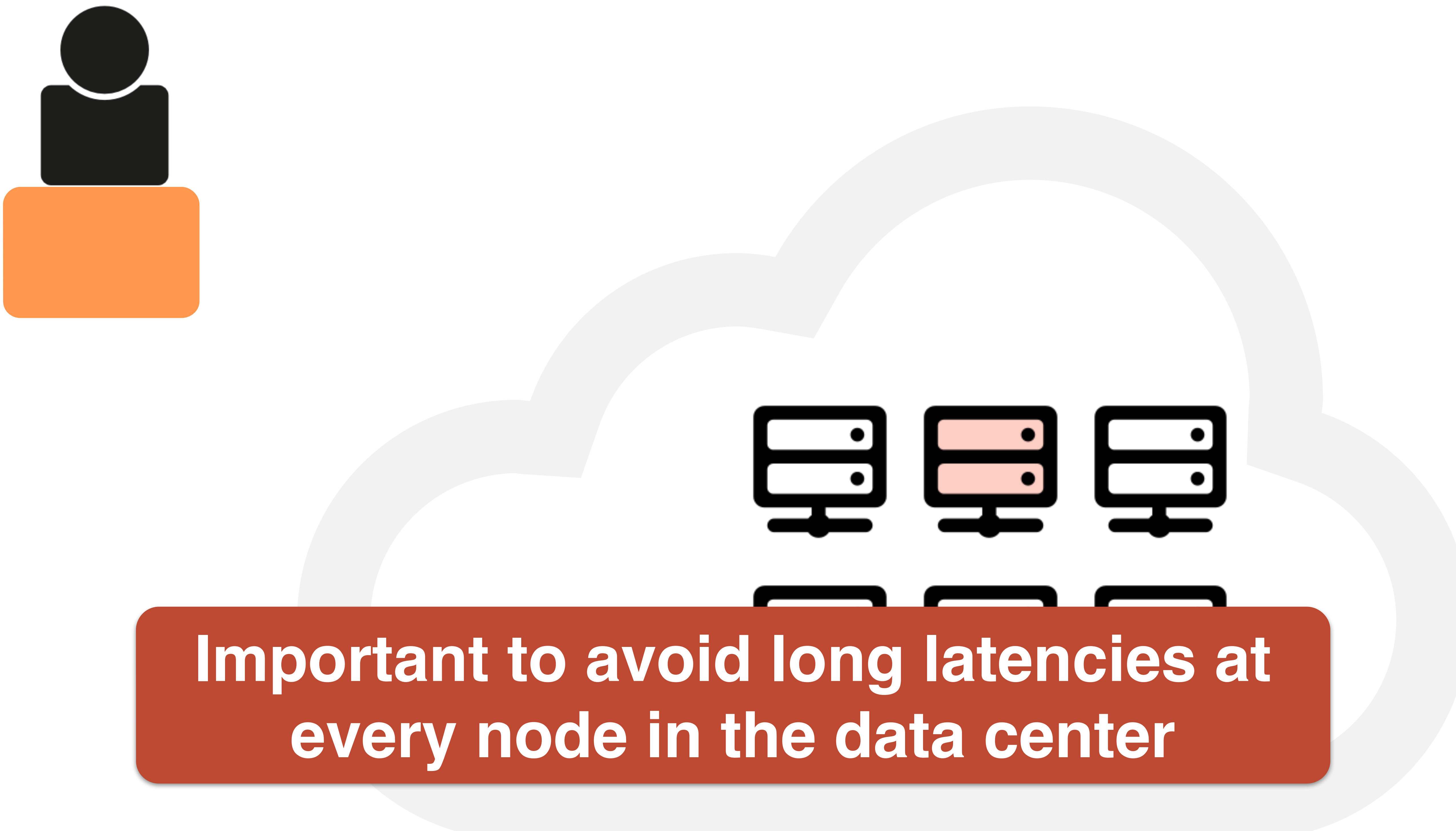
Uncommon tail latencies become common at scale



Uncommon tail latencies become common at scale

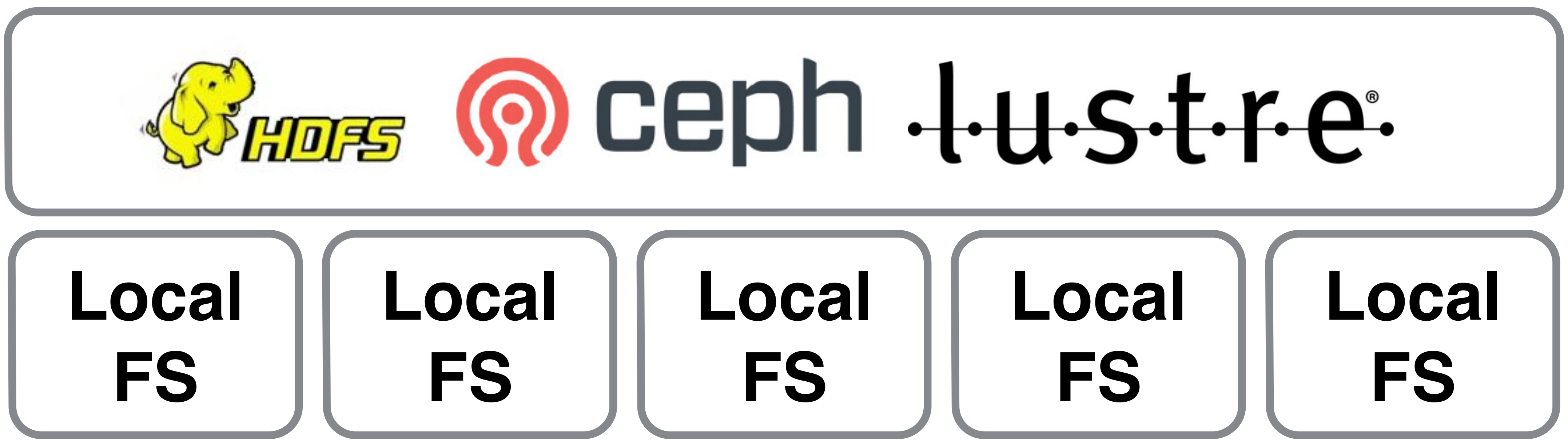


Uncommon tail latencies become common at scale

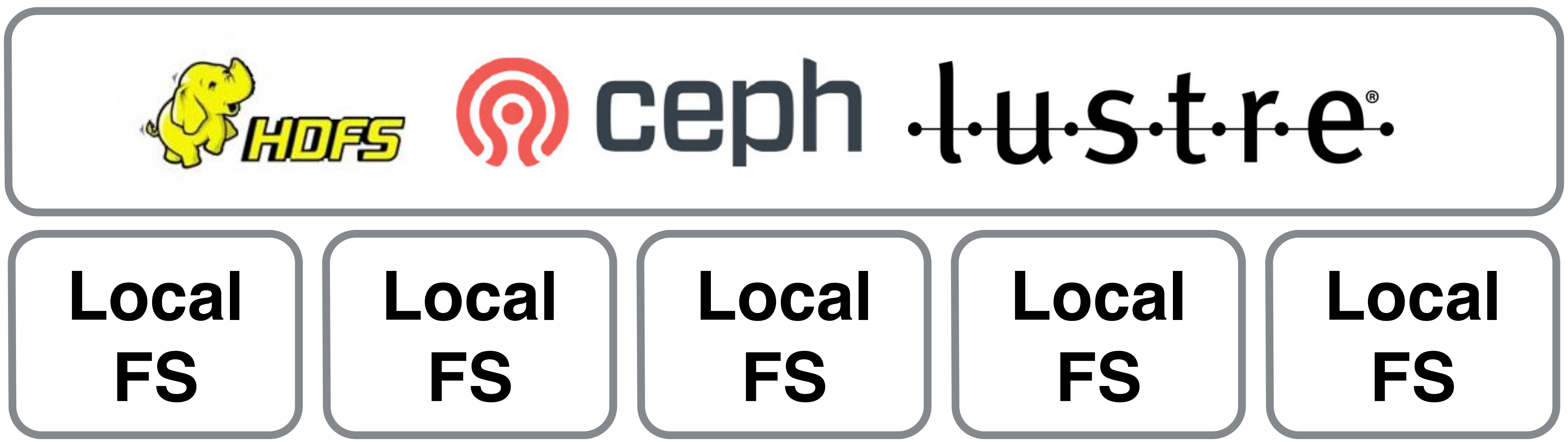


Important to avoid long latencies at every node in the data center

Local file systems contribute to tail latency

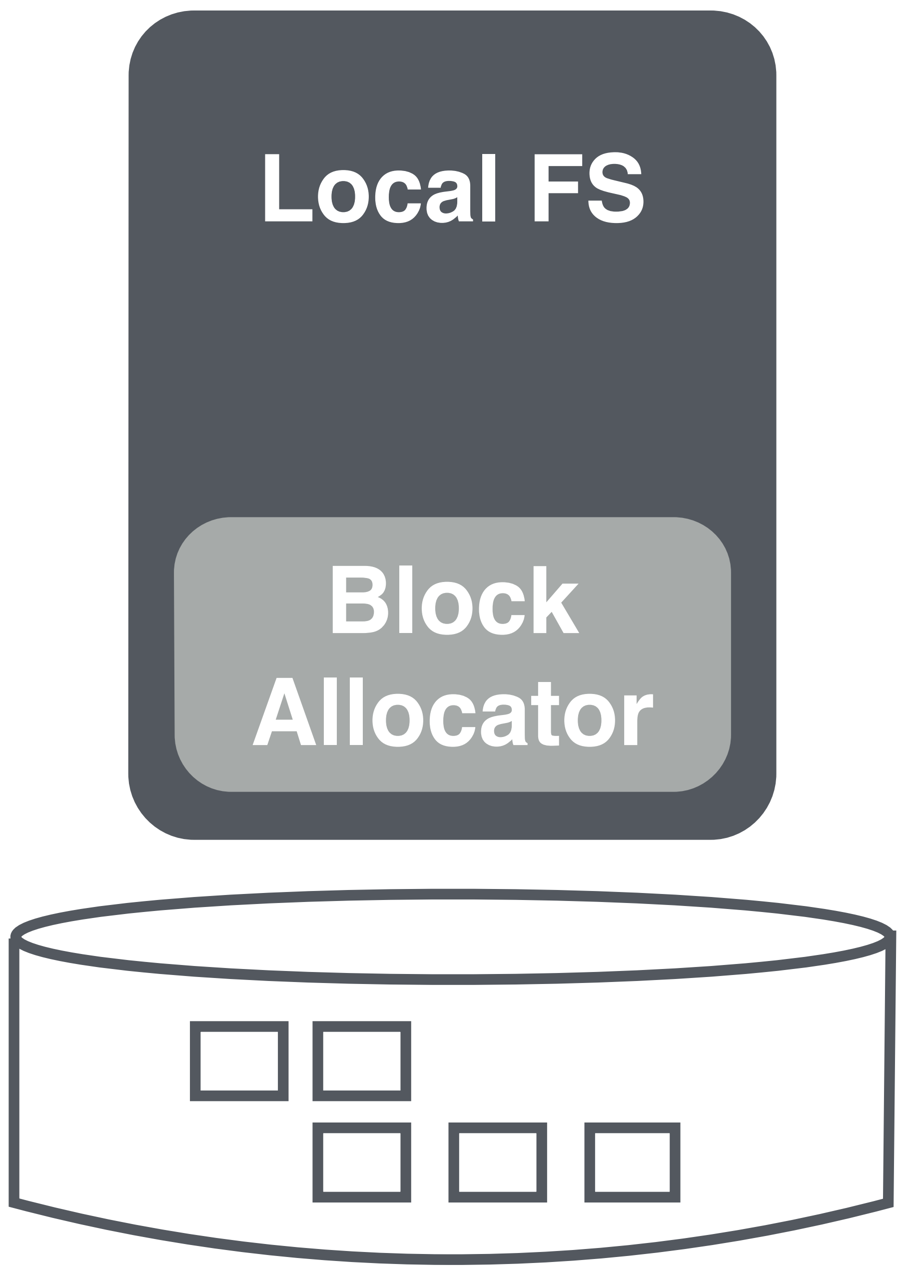


Local file systems contribute to tail latency



Important to avoid long latencies in local file systems

Chopper discovers high-latency operations in local FS



Goal

Find problematic corner cases in file system block allocator

Challenge

File system input space is huge

Chopper explores file systems by statistical techniques

Chopper explores file systems by statistical techniques

We provide an overall analysis of file system block allocations (XFS, ext4)

Chopper explores file systems by statistical techniques

We provide an overall analysis of file system block allocations (XFS, ext4)

We have analyzed unexpected behaviors in detail

Chopper explores file systems by statistical techniques

We provide an overall analysis of file system block allocations (XFS, ext4)

We have analyzed unexpected behaviors in detail

We have found and fixed four allocation issues in ext4 and significantly improved layout quality

Outline

Part 1

Collect Data

Part 2

Analyze Data

Part 3

Understand File System

Outline

Part 1

Collect Data

Part 2

Analyze Data

Part 3

Understand File System

We quantify and qualify everything

INPUT { Workload
File system

↓

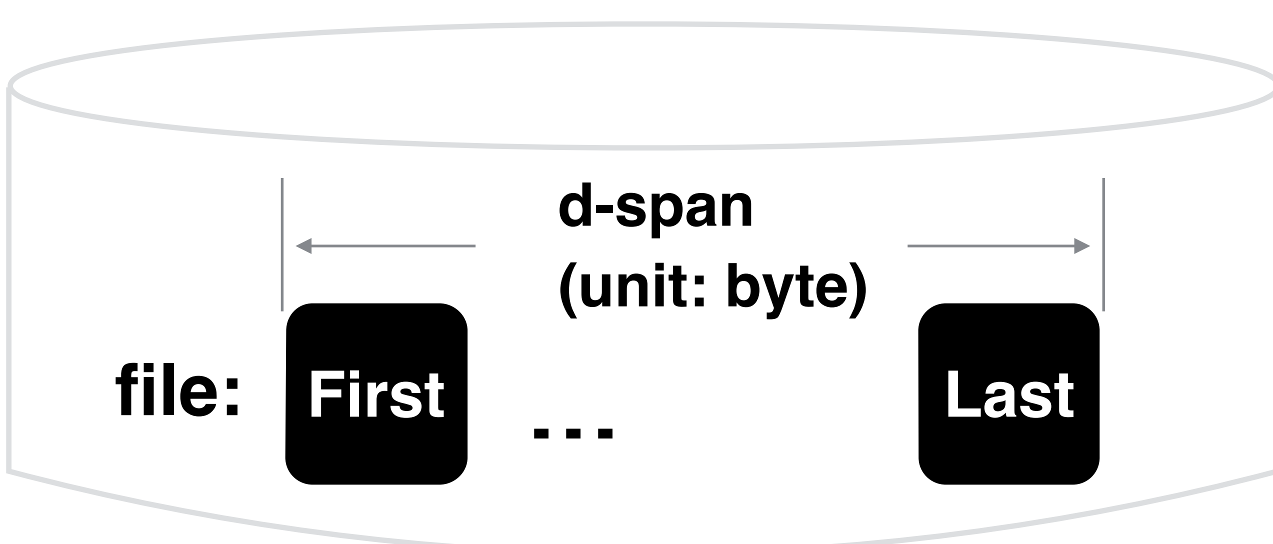
OUTPUT Layout quality

We quantify and qualify everything

INPUT { Workload
File system

↓

OUTPUT Layout quality

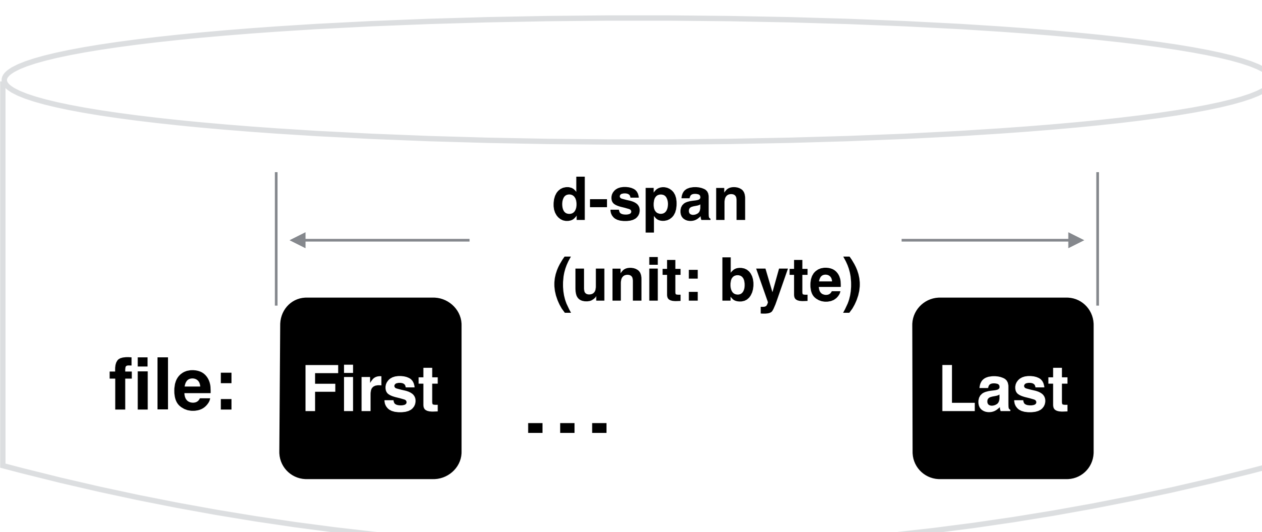


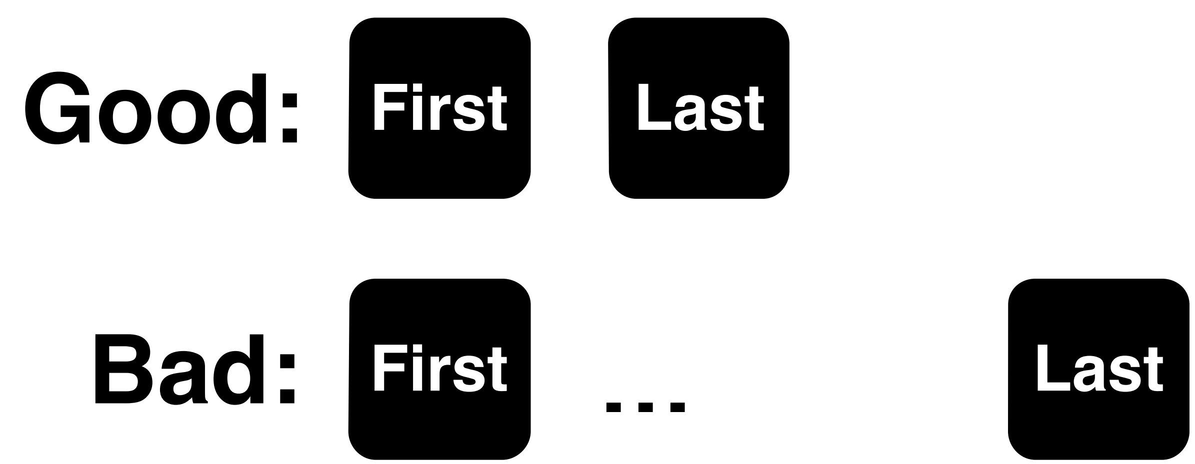
We quantify and qualify everything

INPUT { Workload
File system

↓

OUTPUT Layout quality





What values to pick for the factors?

File System

- Disk Size
- Used Ratio
- Fragmentation
- CPU Count

Workload

- File Size
- Chunk Count
- Internal Density
- Chunk Order
- Fsync
- Sync
- File Count
- Directory Span

What values to pick for the factors?

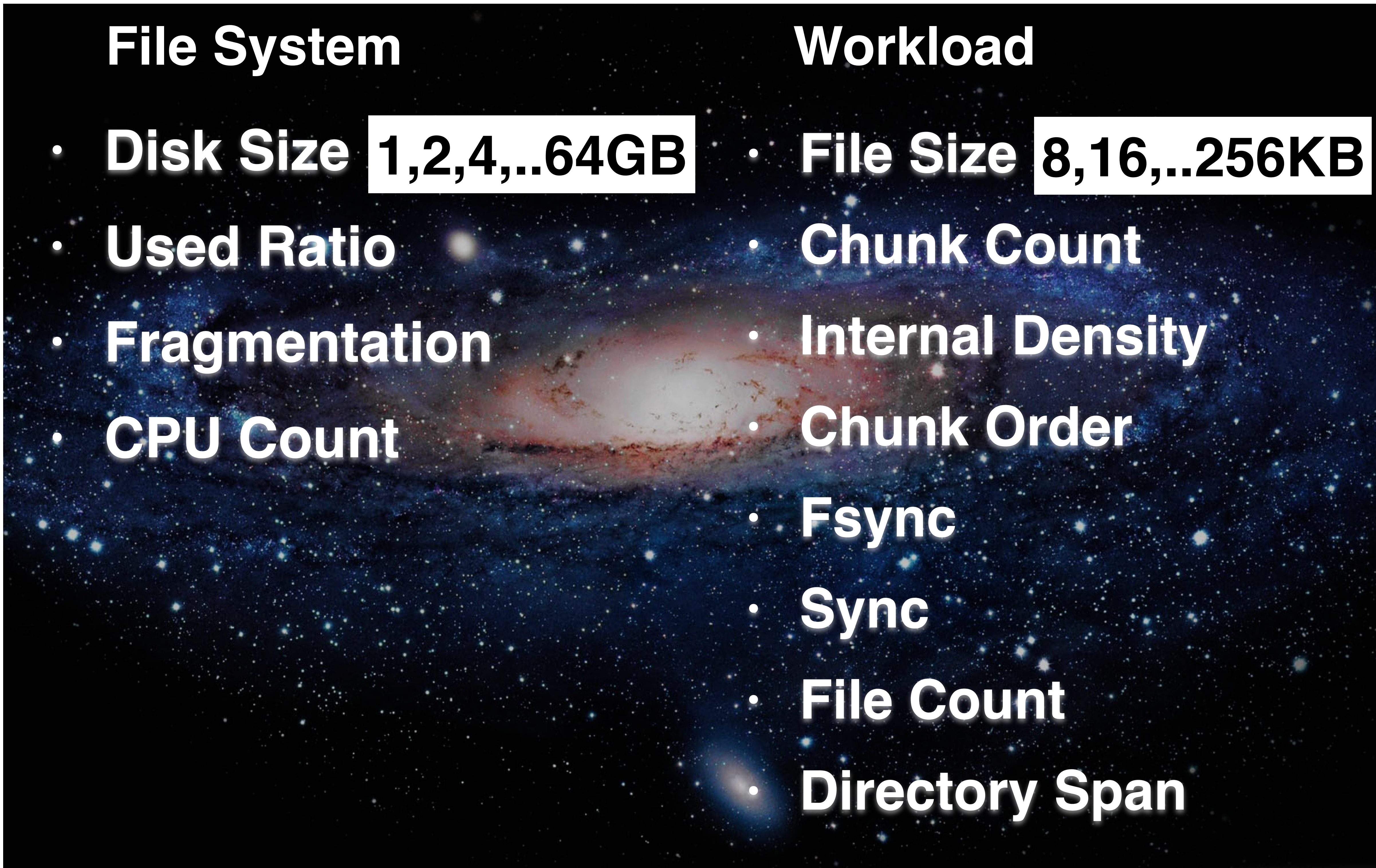
File System

- Disk Size 1,2,4,..64GB
- Used Ratio
- Fragmentation
- CPU Count

Workload

- File Size
- Chunk Count
- Internal Density
- Chunk Order
- Fsync
- Sync
- File Count
- Directory Span

What values to pick for the factors?



What values to pick for the factors?

File System

- Disk Size 1,2,4,..64GB
- Used Ratio
- Fragmentation
- CPU Count

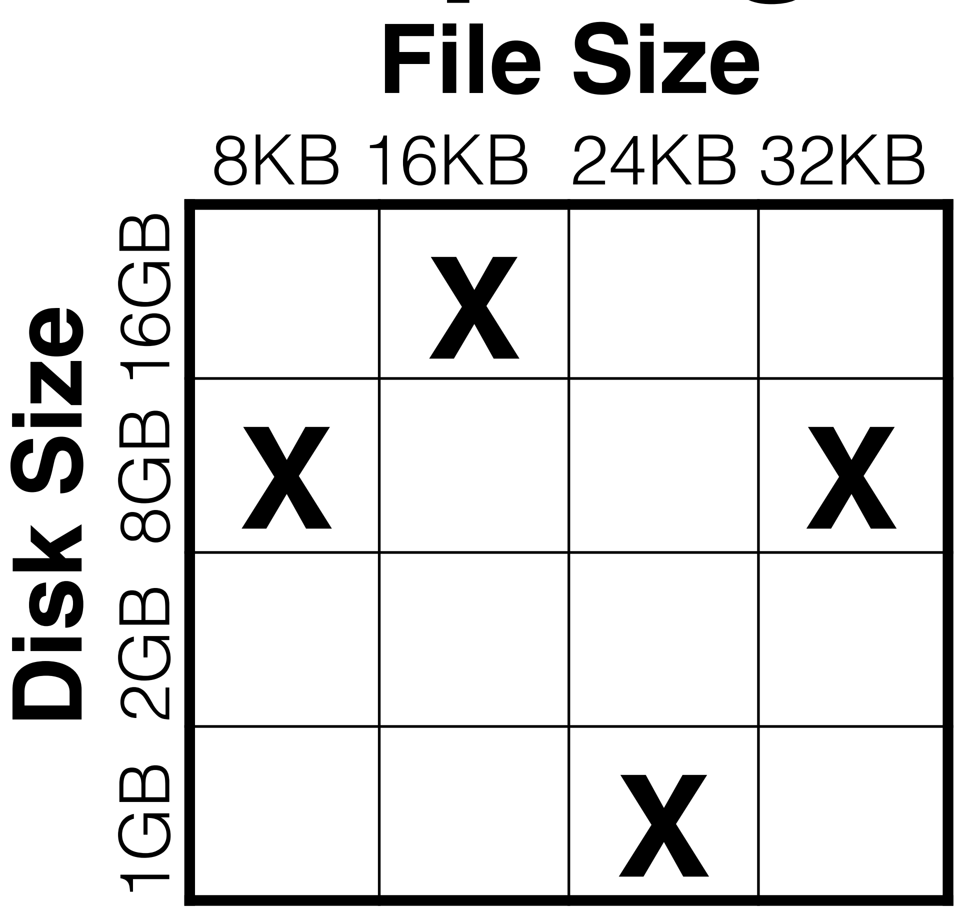
Workload

- File Size 8,16,..256KB
- Chunk Count
- Internal Density
- Chunk Order
- Directory Span

After refining,
250 years to explore all combinations

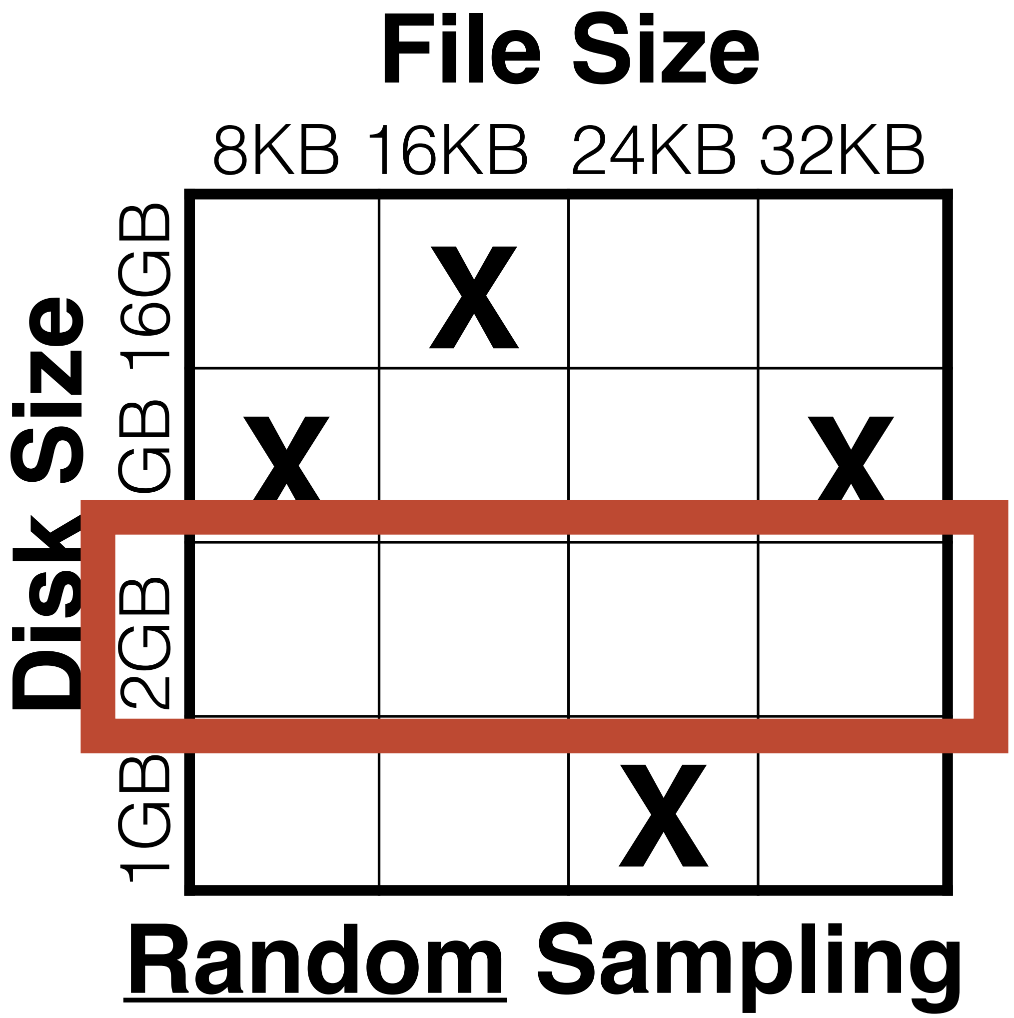
We use Latin Hypercube Sampling to search efficiently

We use Latin Hypercube Sampling to search efficiently

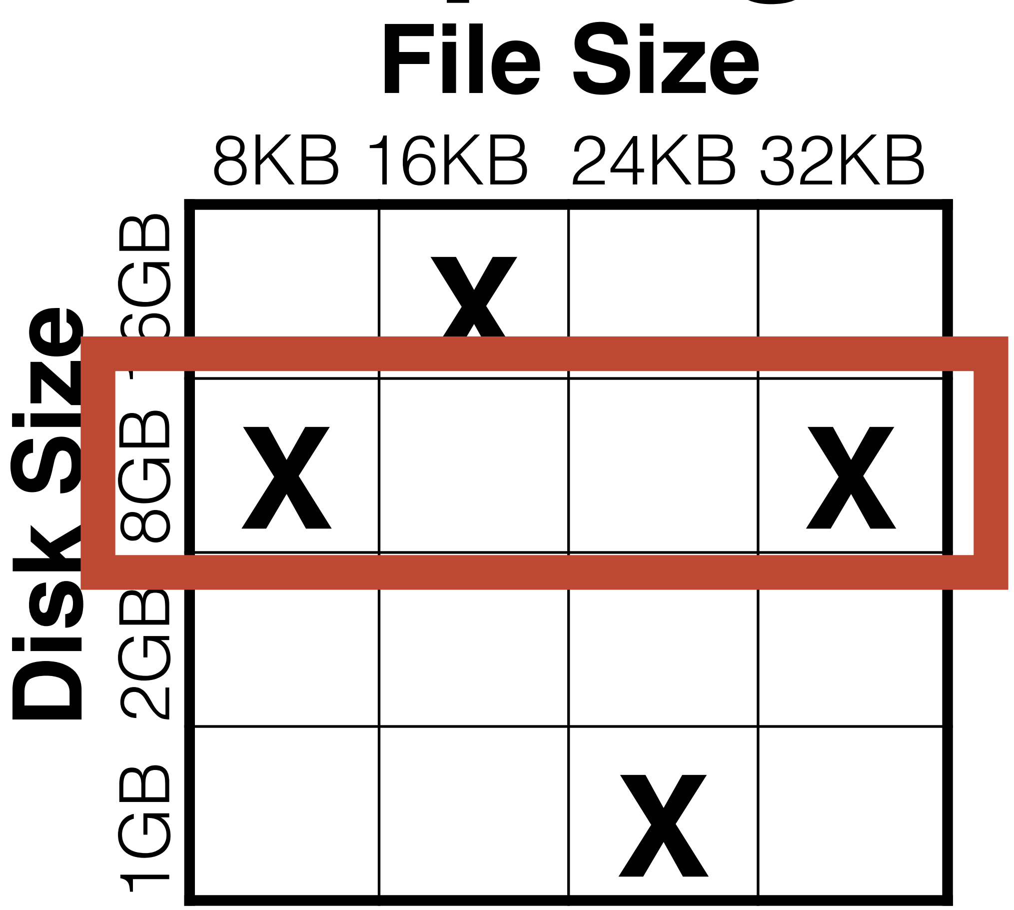


<u>Random</u> Sampling

We use Latin Hypercube Sampling to search efficiently

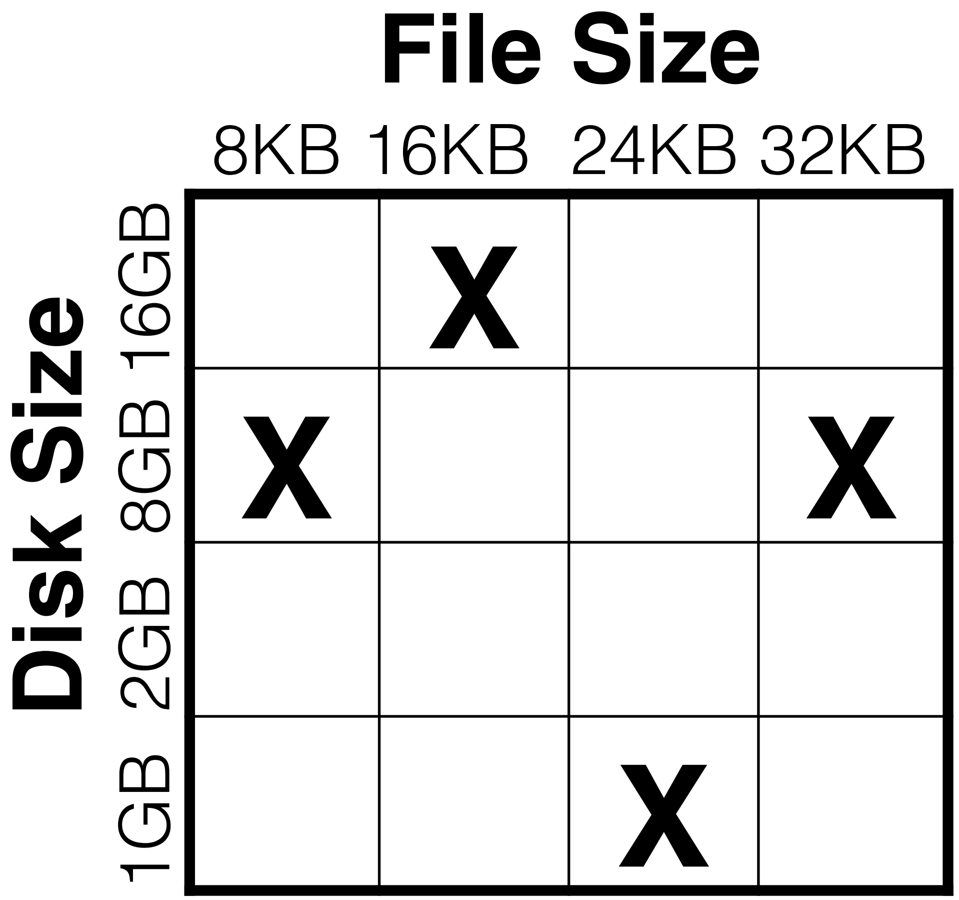


We use Latin Hypercube Sampling to search efficiently



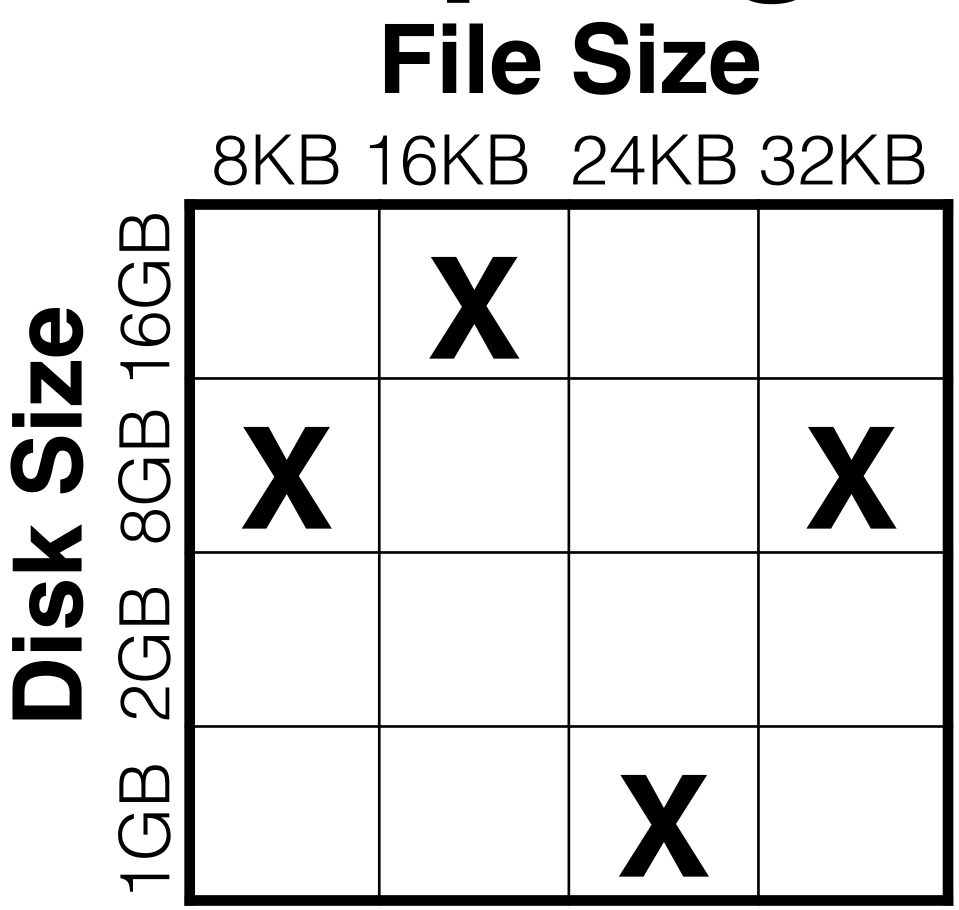
Random Sampling

We use Latin Hypercube Sampling to search efficiently

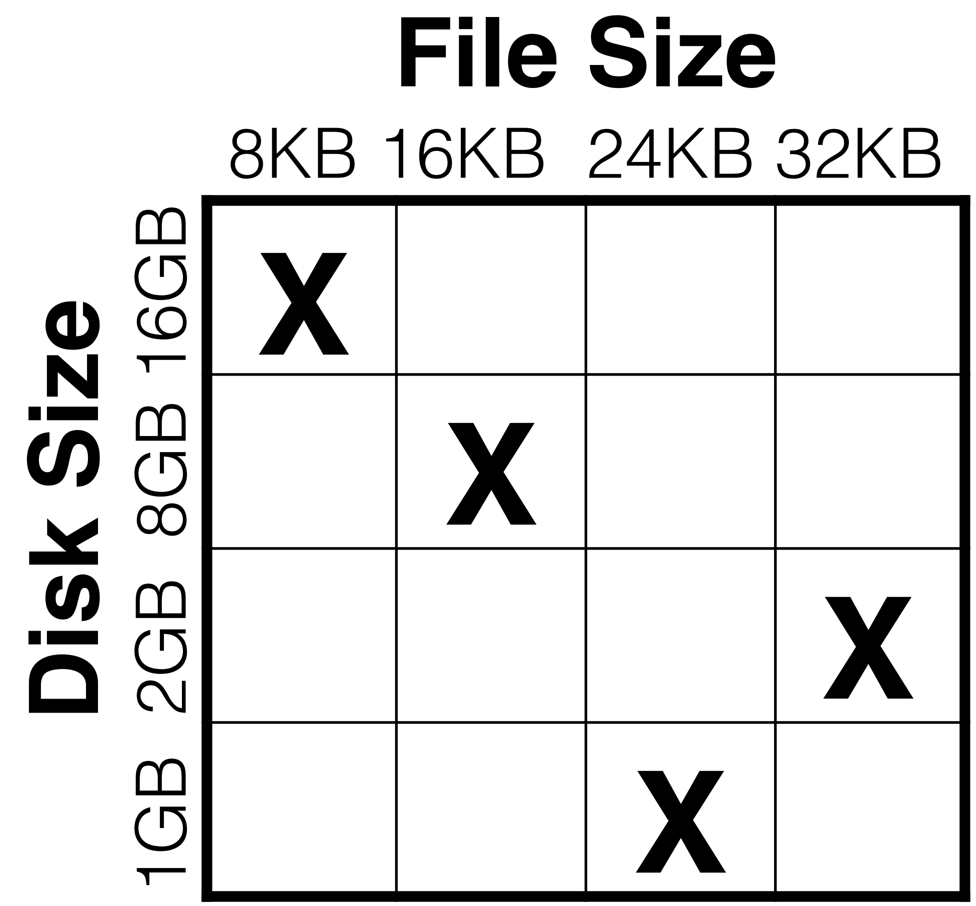


<u>Random</u> Sampling

We use Latin Hypercube Sampling to search efficiently

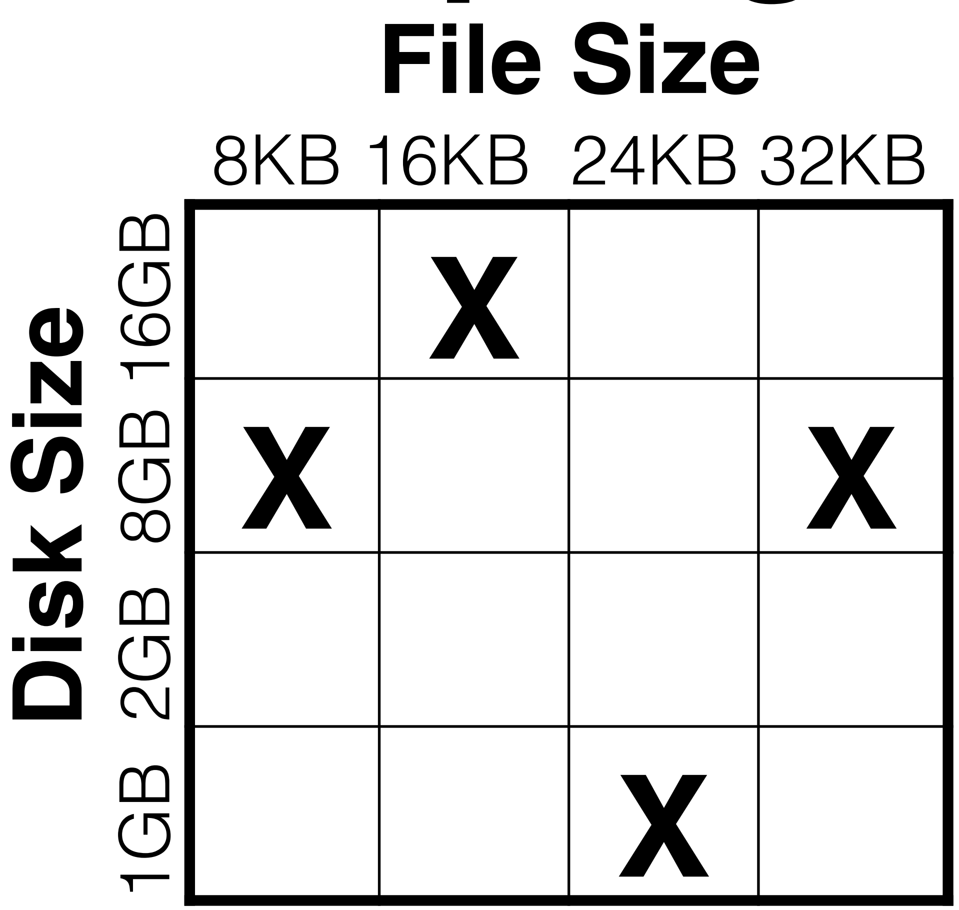


<u>Random</u> Sampling

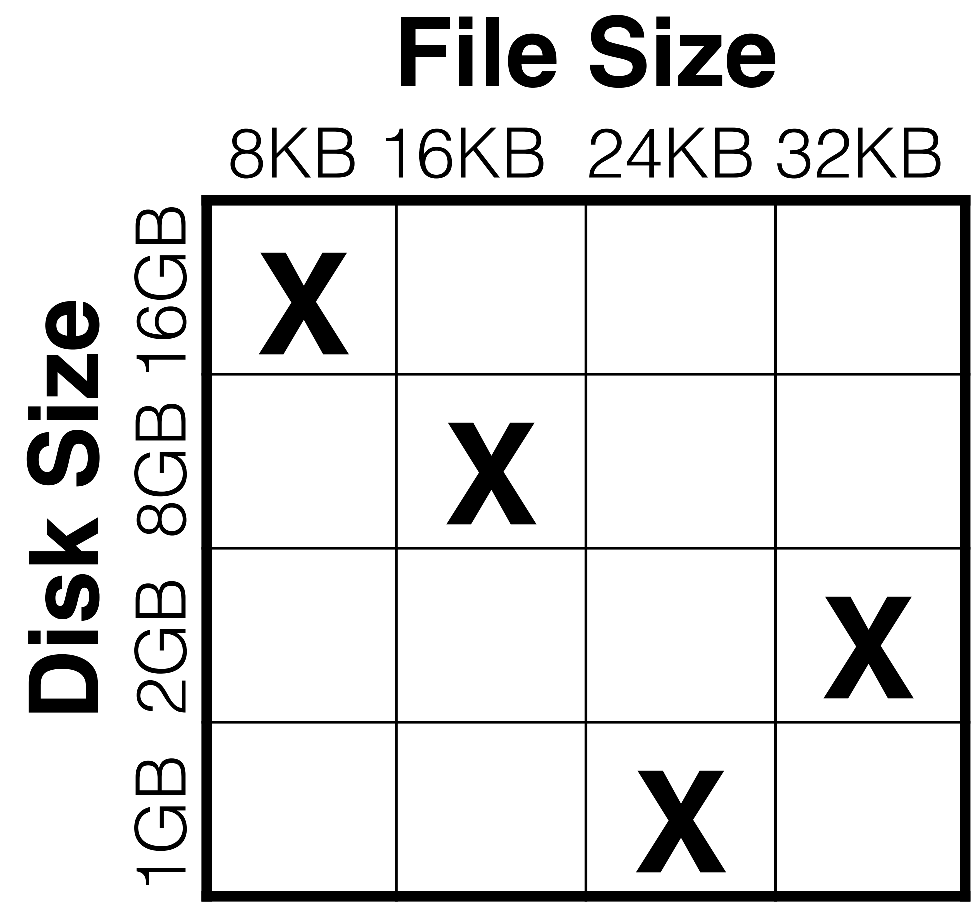


Latin Hypercube Sampling

We use Latin Hypercube Sampling to search efficiently

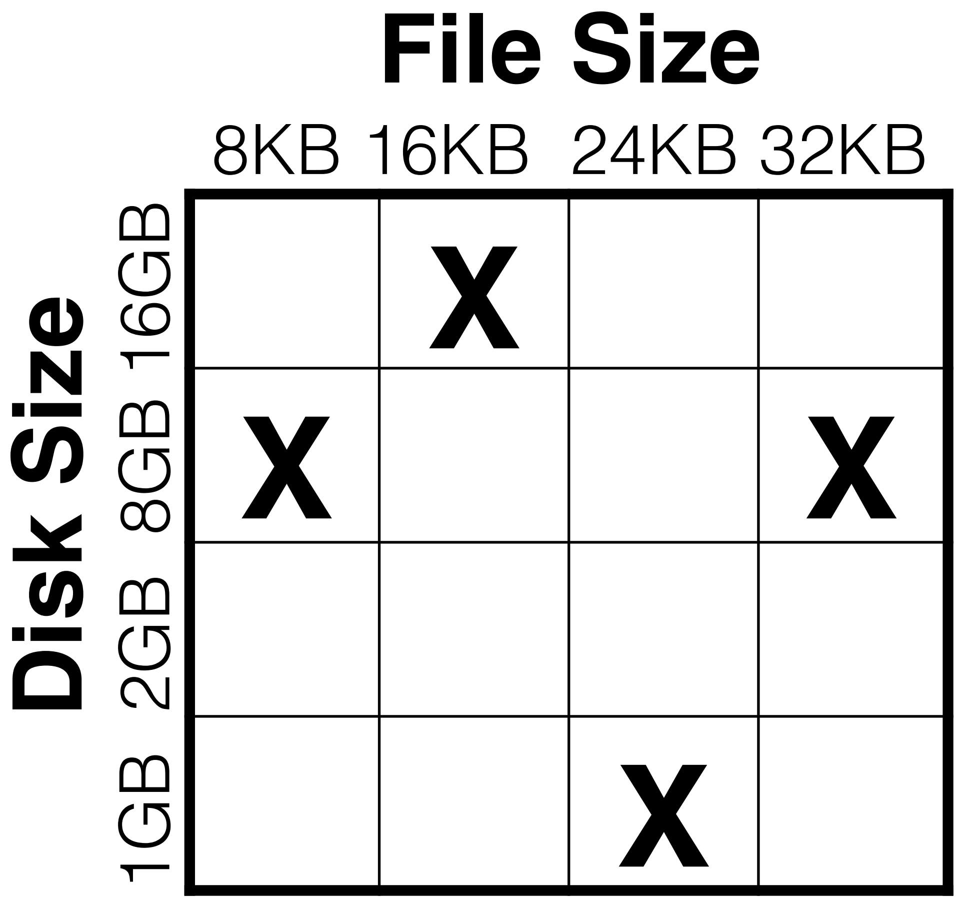


Random Sampling

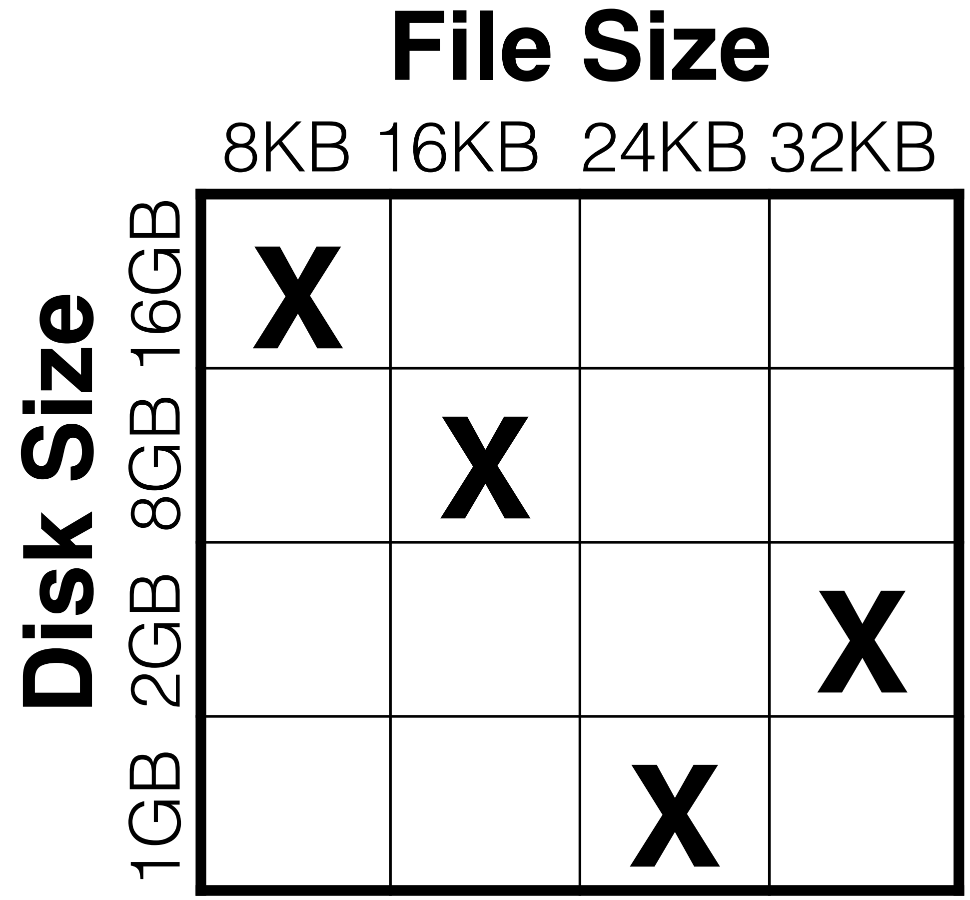


Latin Hypercube Sampling

We use Latin Hypercube Sampling to search efficiently



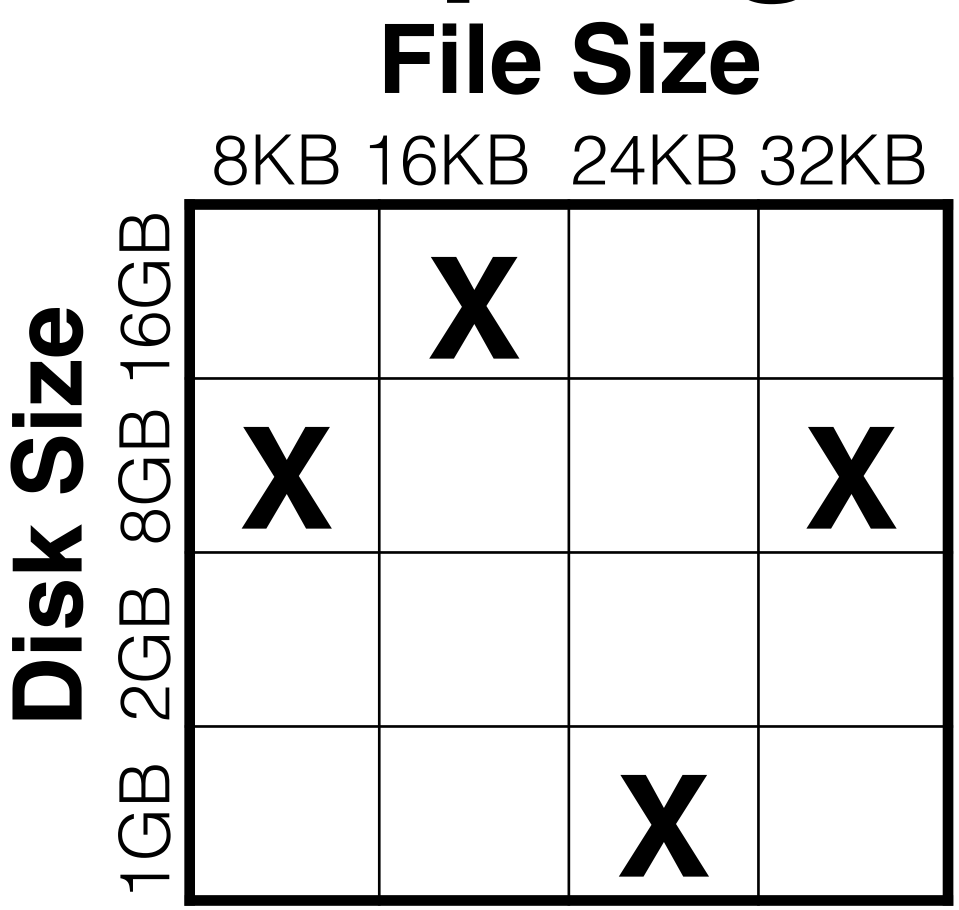
Random Sampling



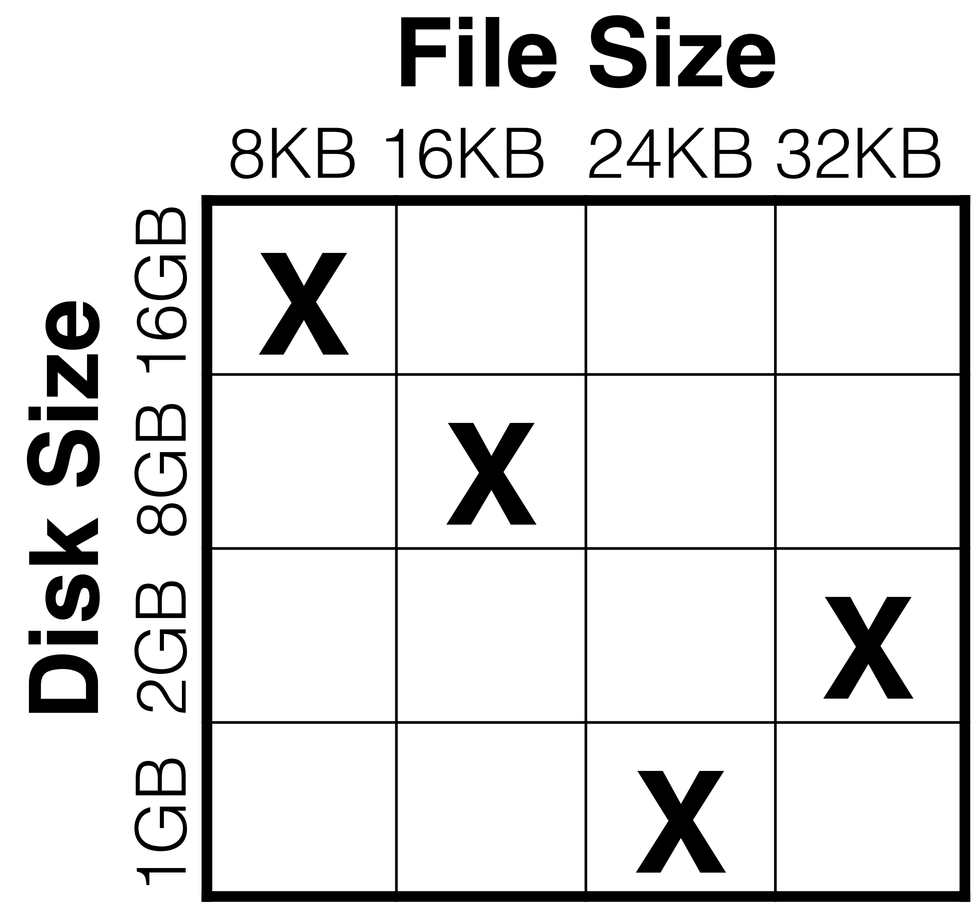
Latin Hypercube Sampling

- Explores space evenly
- Aids visualization
- Explores interactions between factors well

We use Latin Hypercube Sampling to search efficiently



<u>Random</u> Sampling



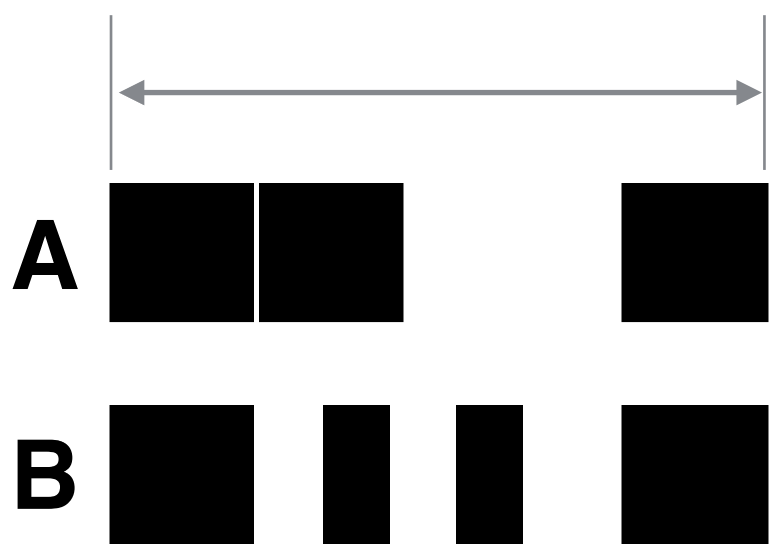
Latin Hypercube Sampling

- Explores space evenly
- Aids visualization
- Explores interactions between factors well

16384 samples, 30 mins with 32 machines

d-span is a signal of block allocation problems

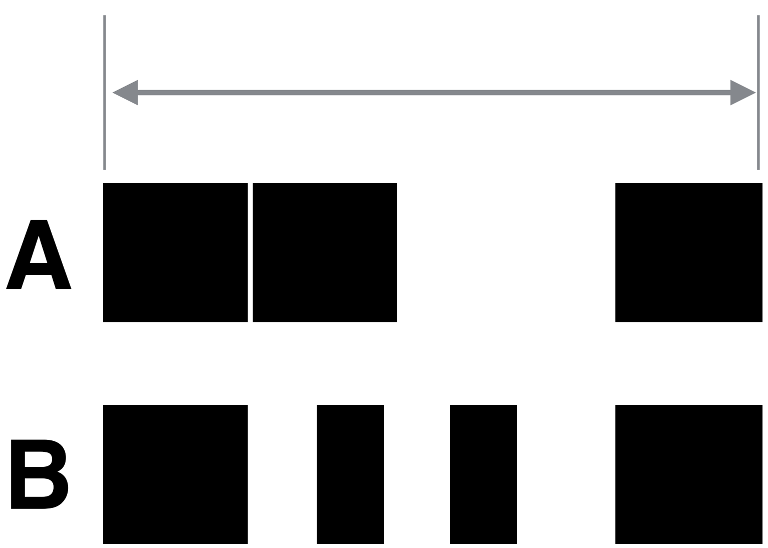
We use
d-span



d-span is a signal of block allocation problems

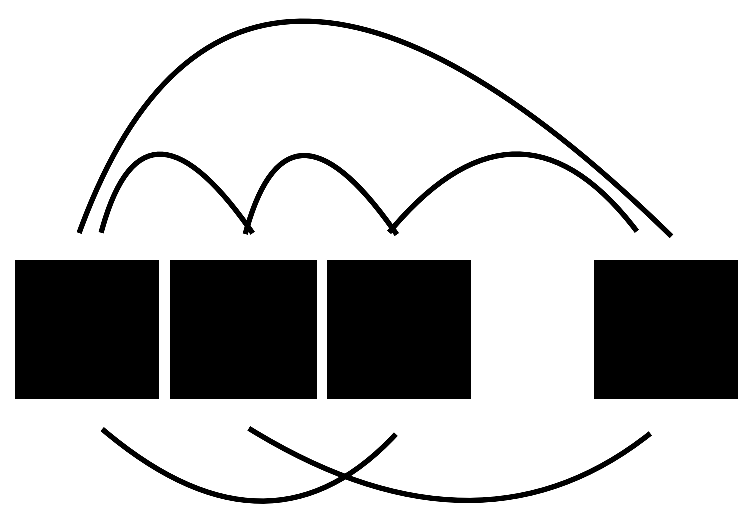
We use

d-span



Alternative 1

Average block distance



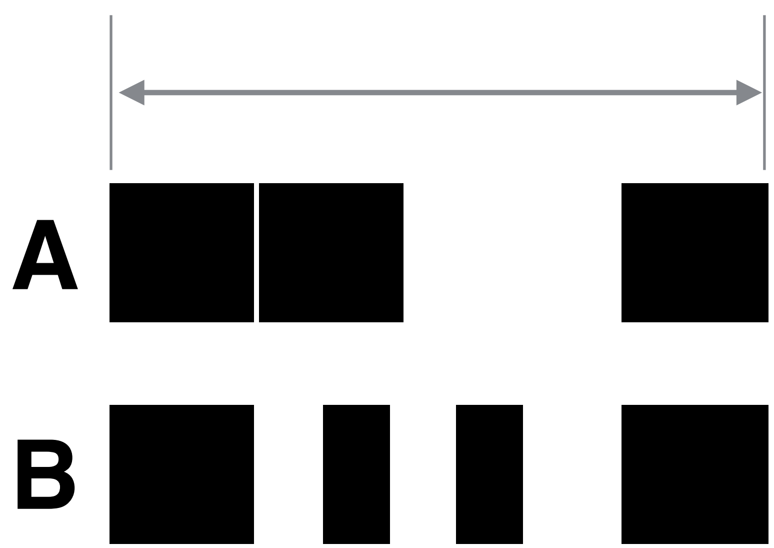
d-span is a signal of block allocation problems

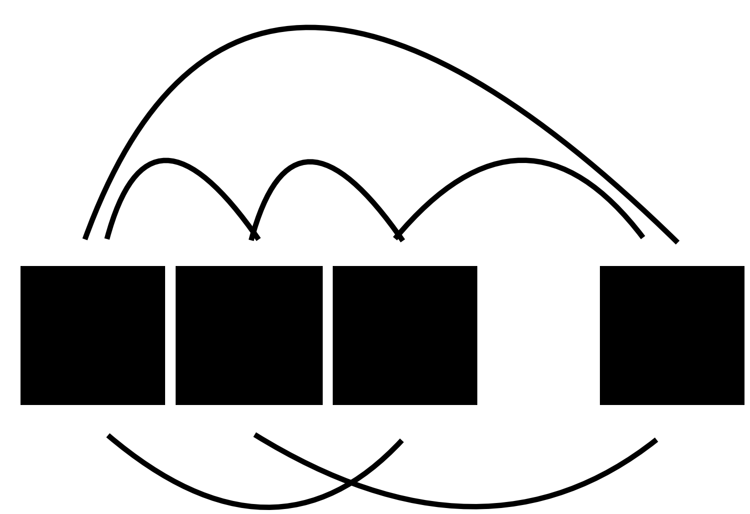
We use

d-span

Alternative 1

Average block distance

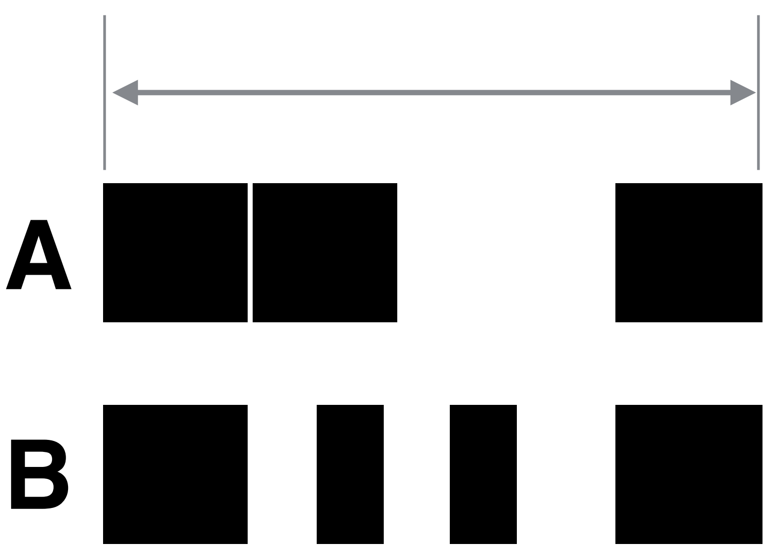




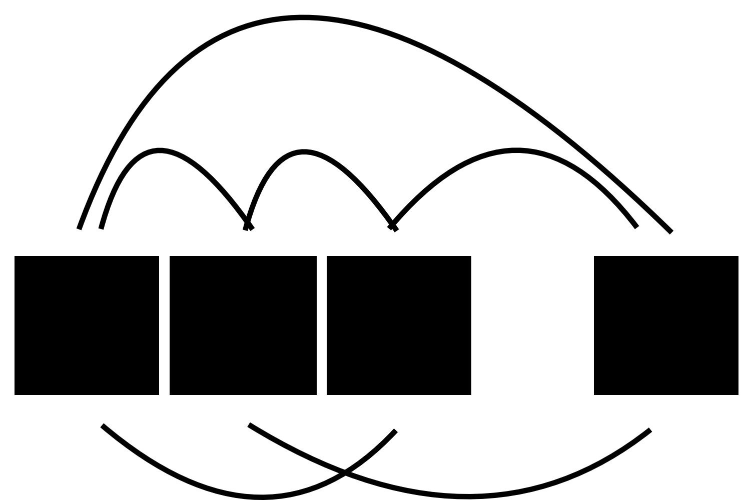
Complex

d-span is a signal of block allocation problems

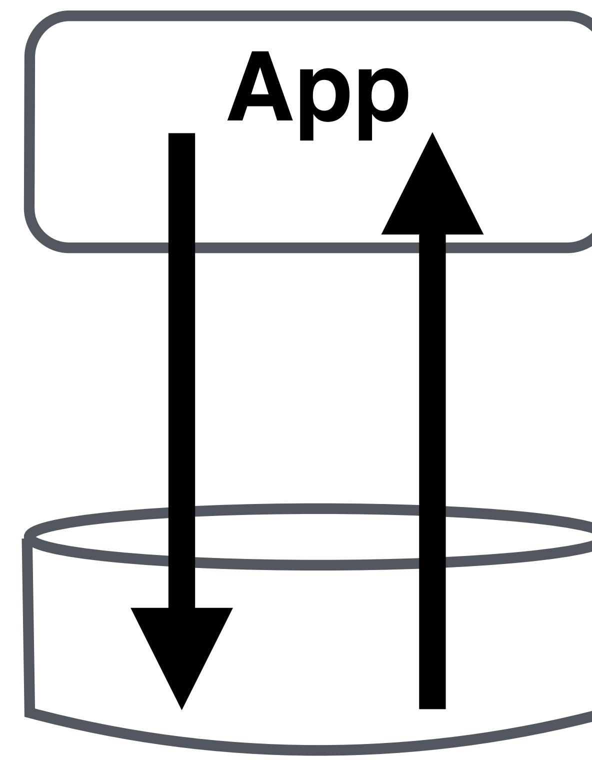
We use
d-span



Alternative 1
Average block distance



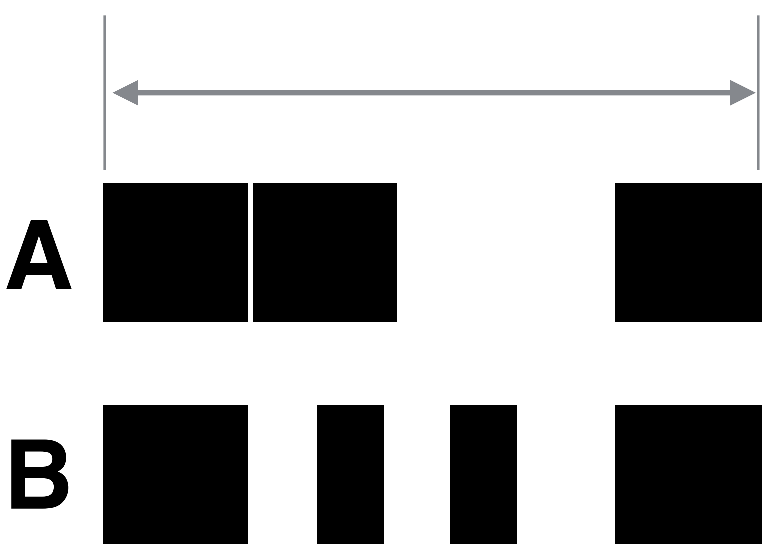
Alternative 2
End-to-end performance

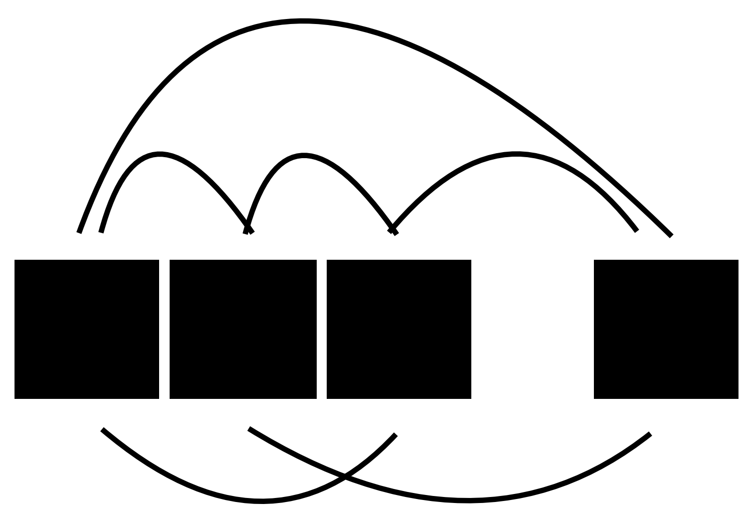


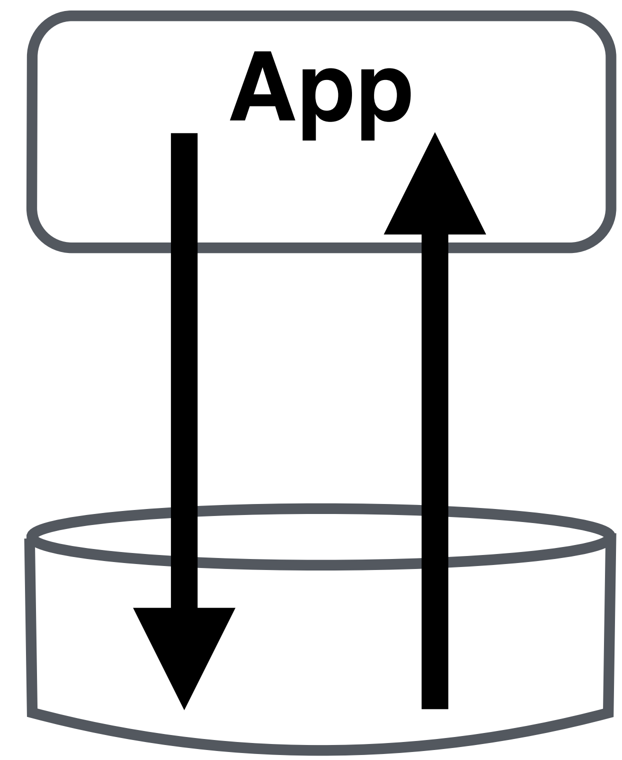
Complex

d-span is a signal of block allocation problems

We use	Alternative 1	Alternative 2
d-span	**Average block distance**	**End-to-end performance**





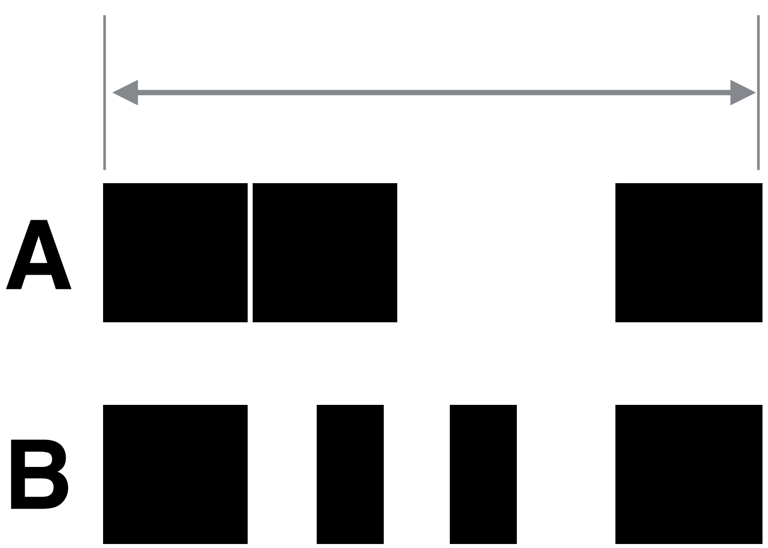


Complex **Confounded**

d-span is a signal of block allocation problems

We use

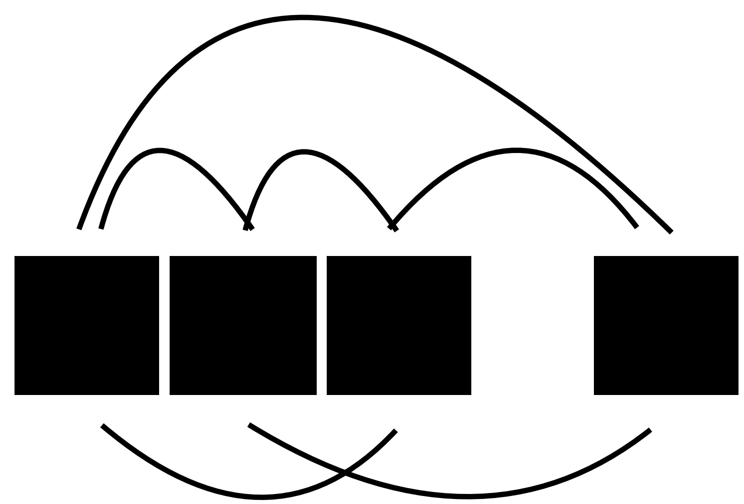
d-span



Simple & Informative

Alternative 1

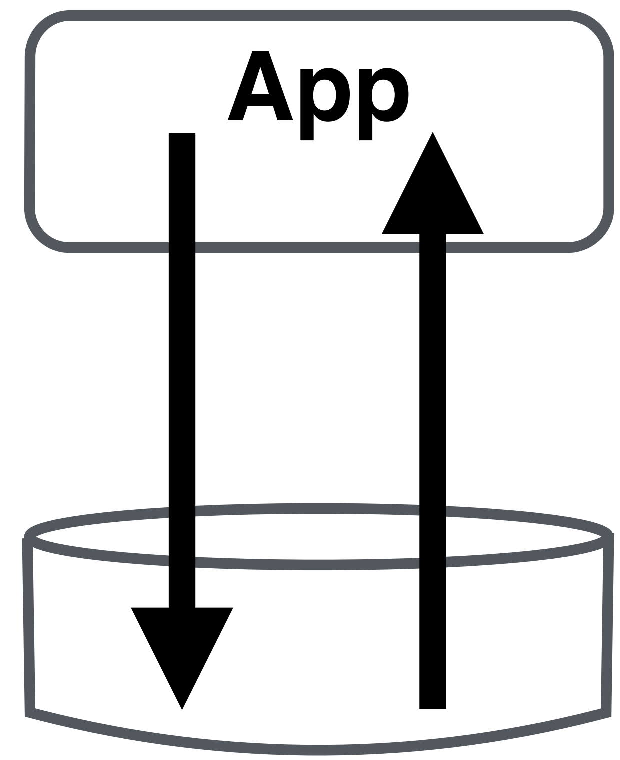
Average block distance



Complex

Alternative 2

End-to-end performance



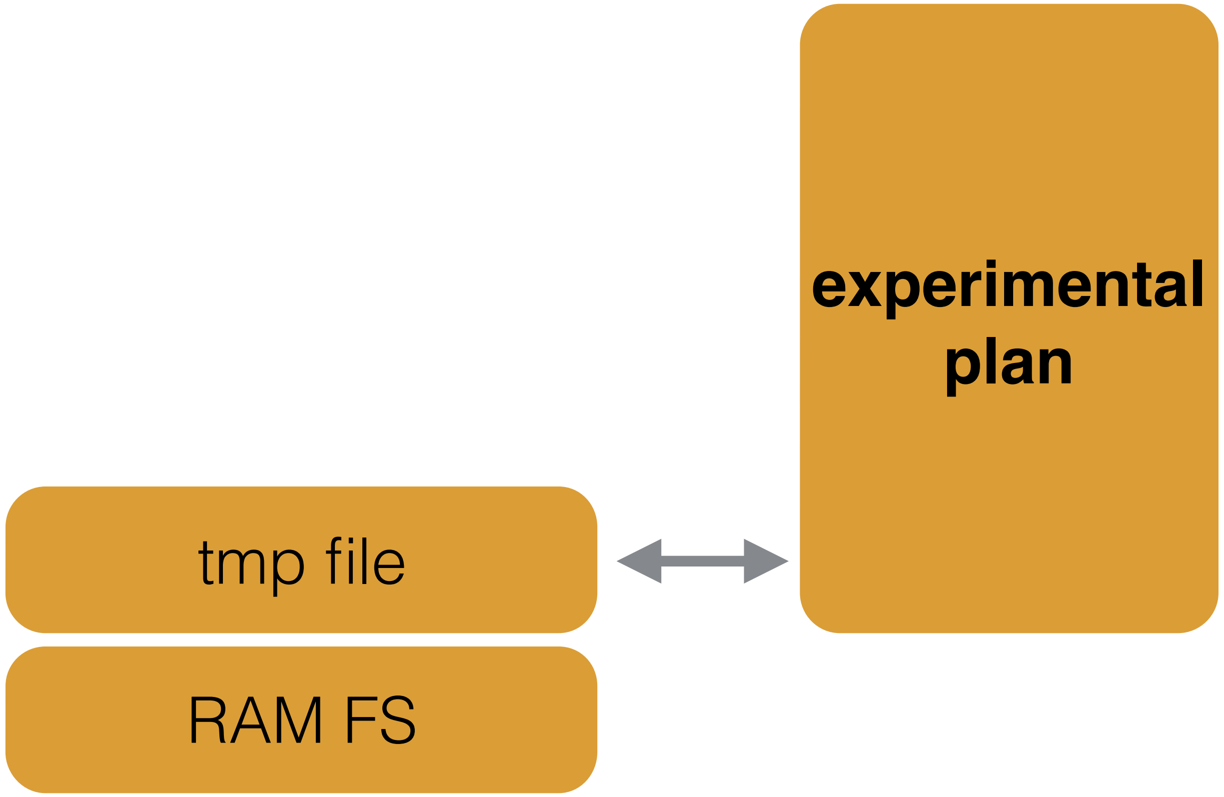
Confounded

How *Chopper* works?

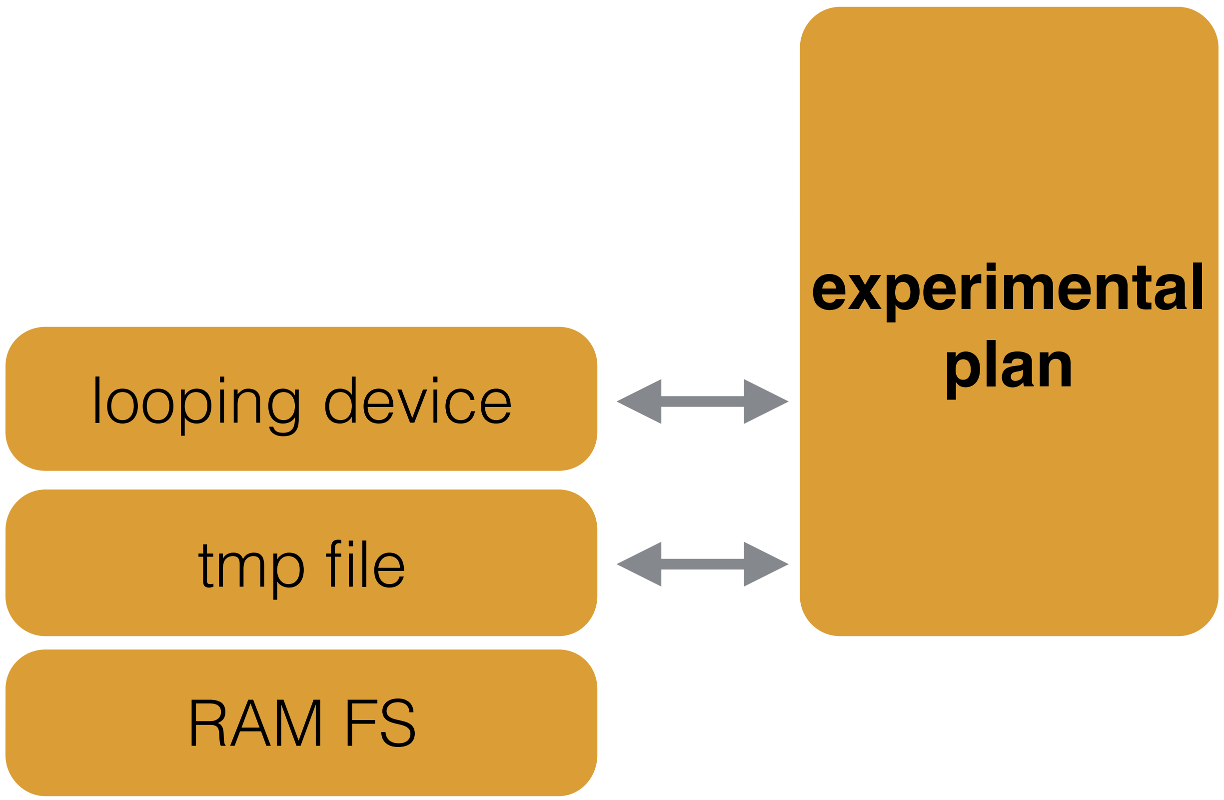
experimental
plan

How *Chopper* works?

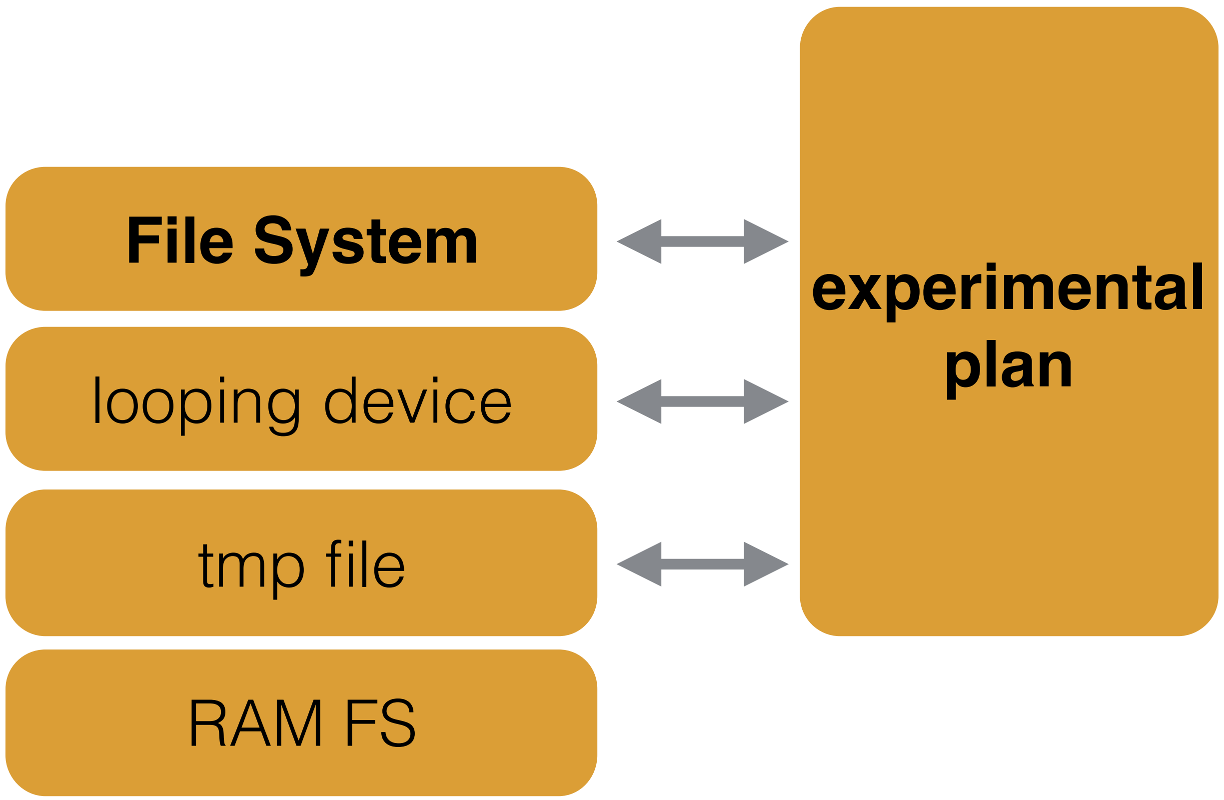
How *Chopper* works?



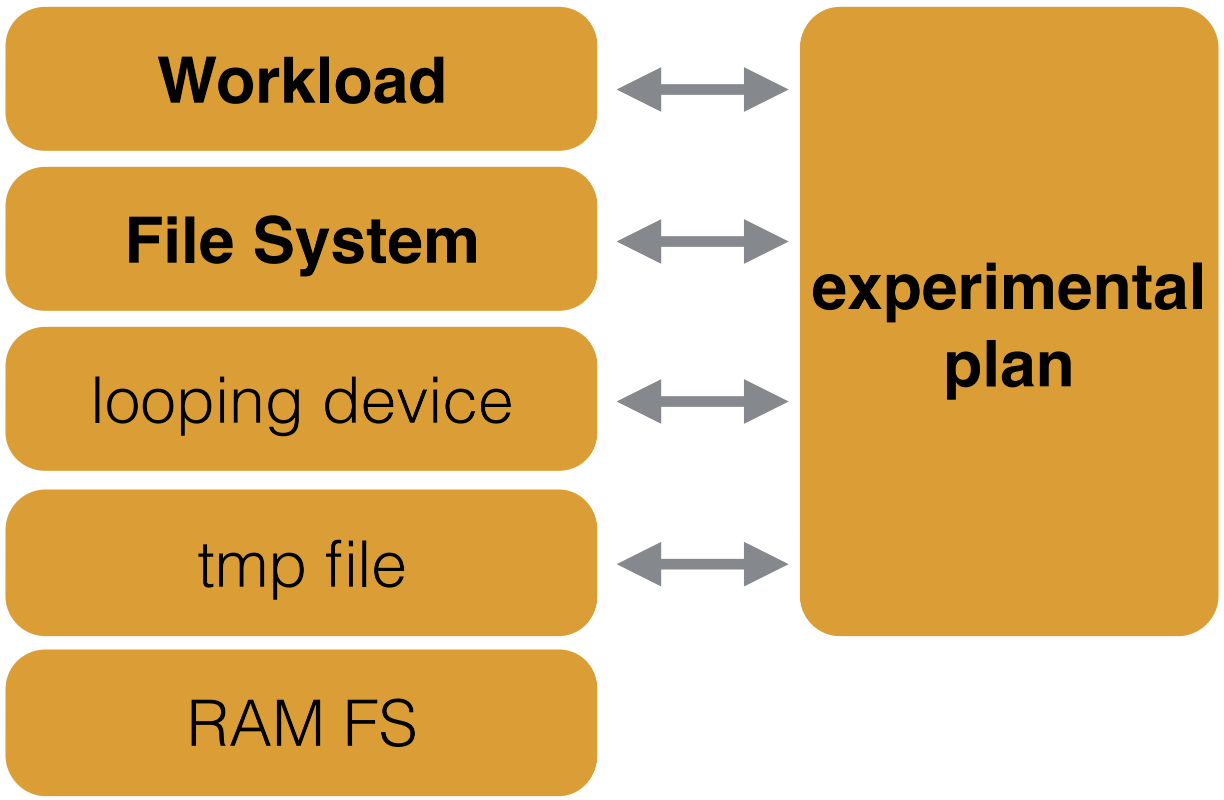
How *Chopper* works?



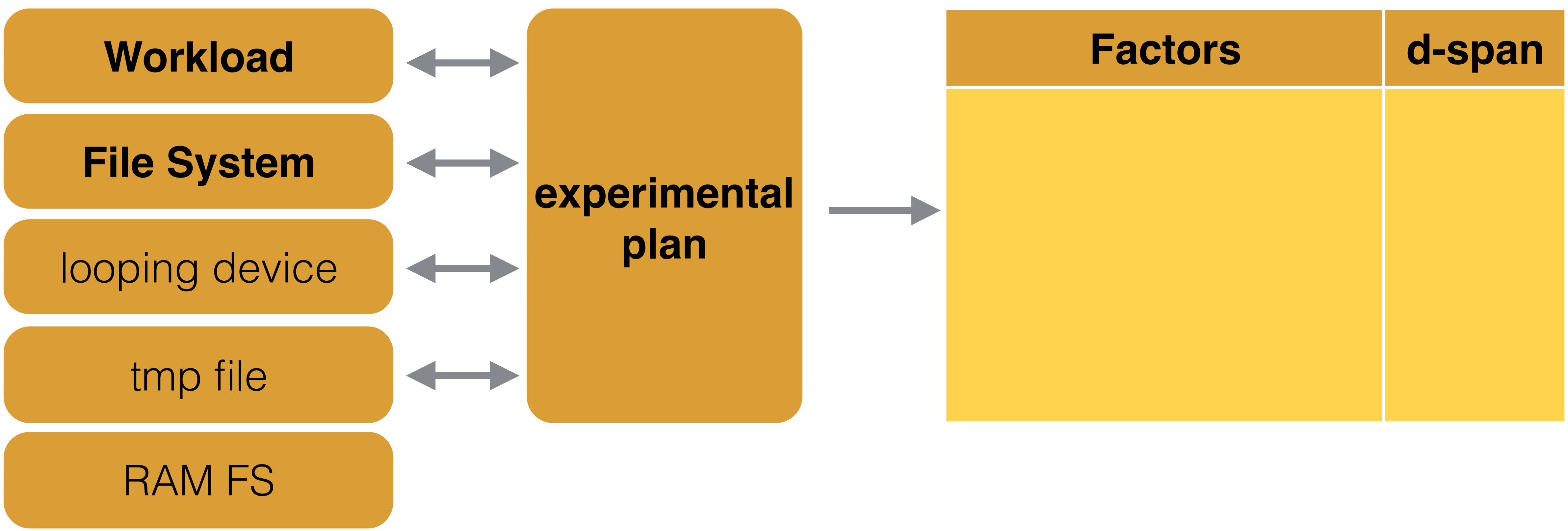
How *Chopper* works?



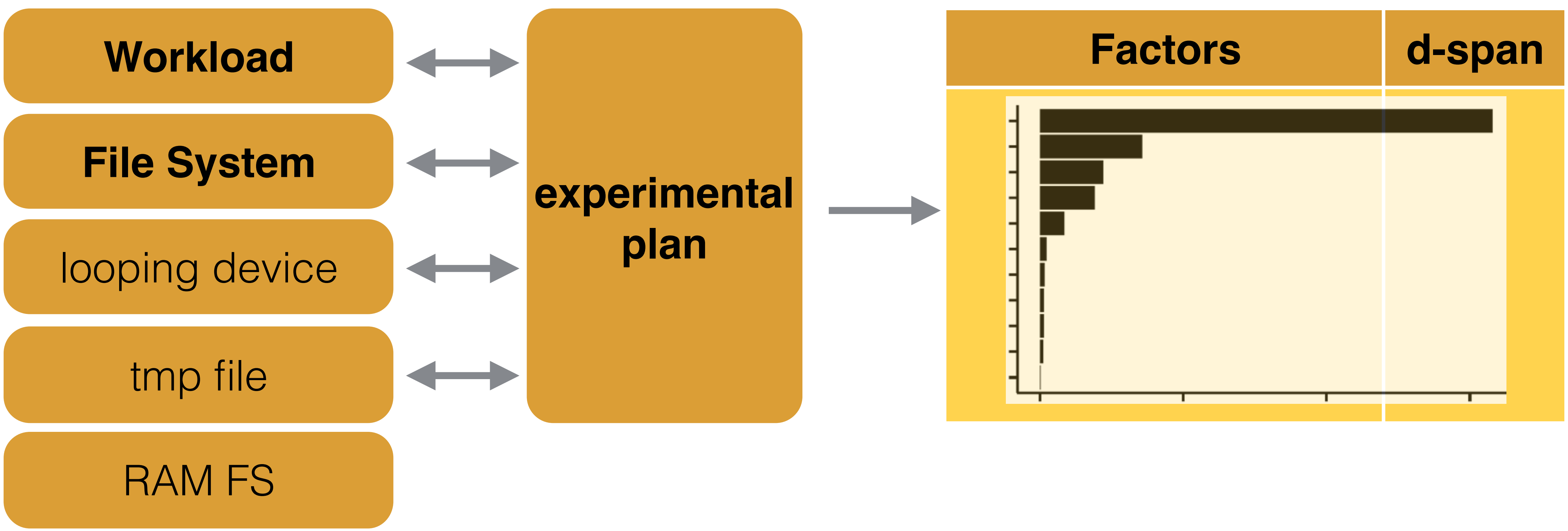
How *Chopper* works?



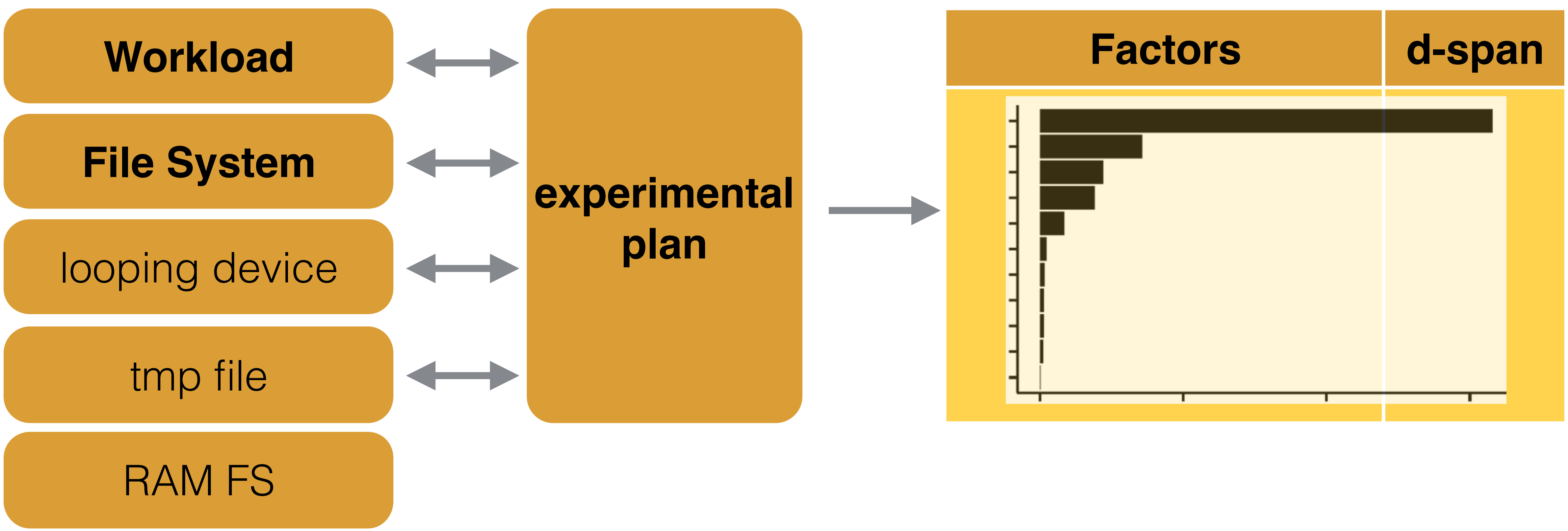
How *Chopper* works?



How *Chopper* works?



How *Chopper* works?



- All operations are in user space
- No kernel modification needed

Outline

Part 1

Collect Data

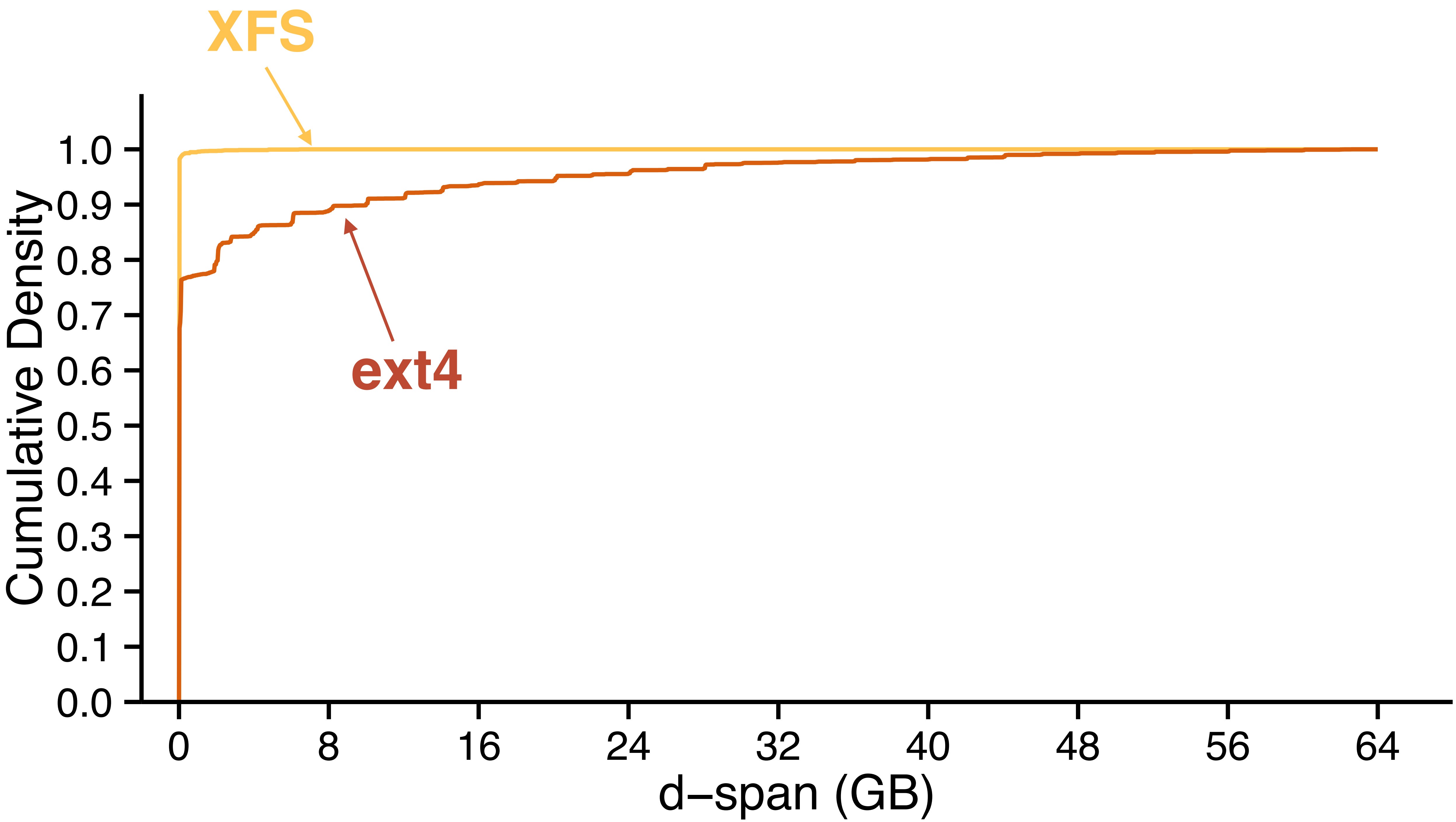
Part 2

Analyze Data

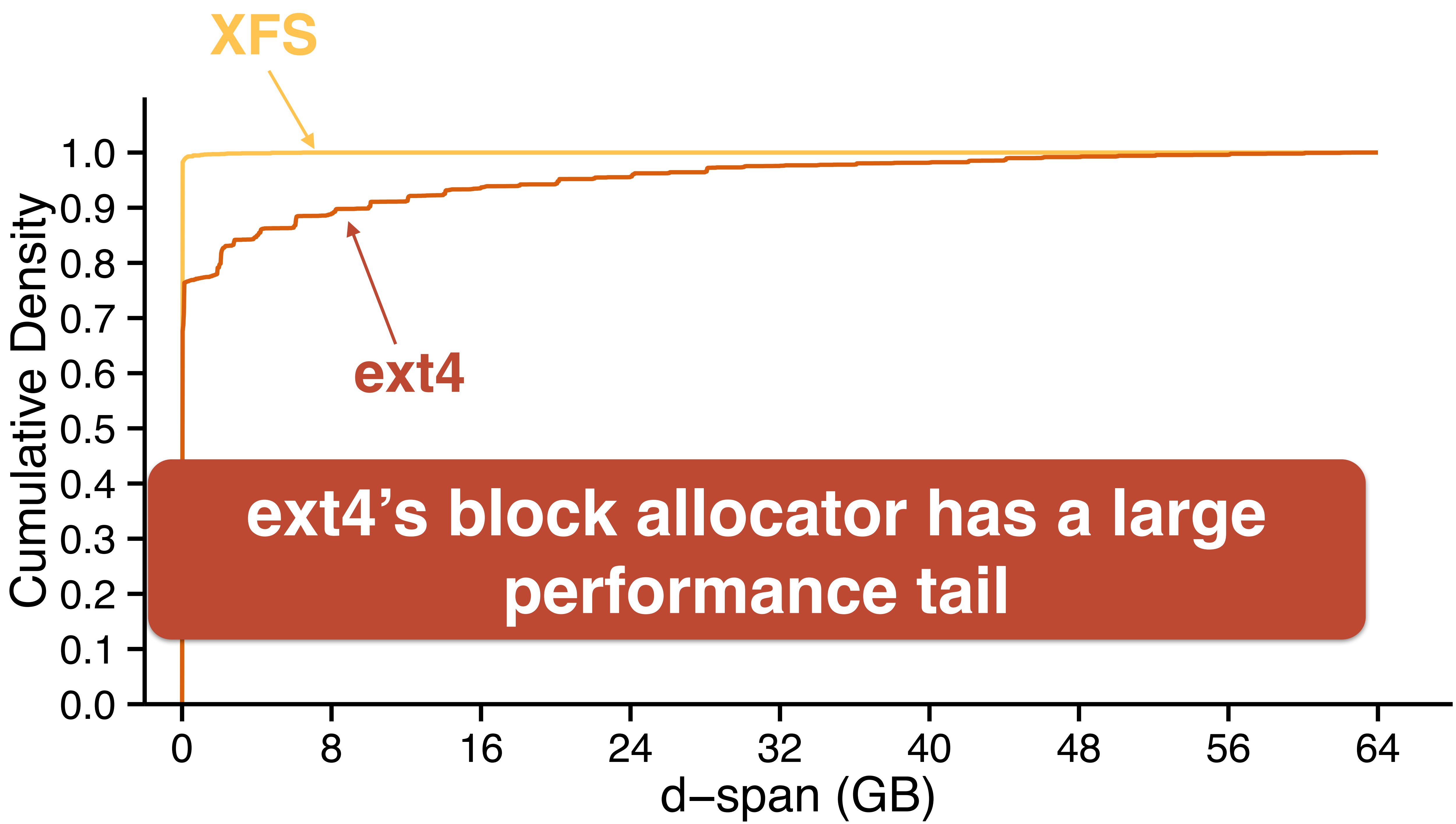
Part 3

Understand File System

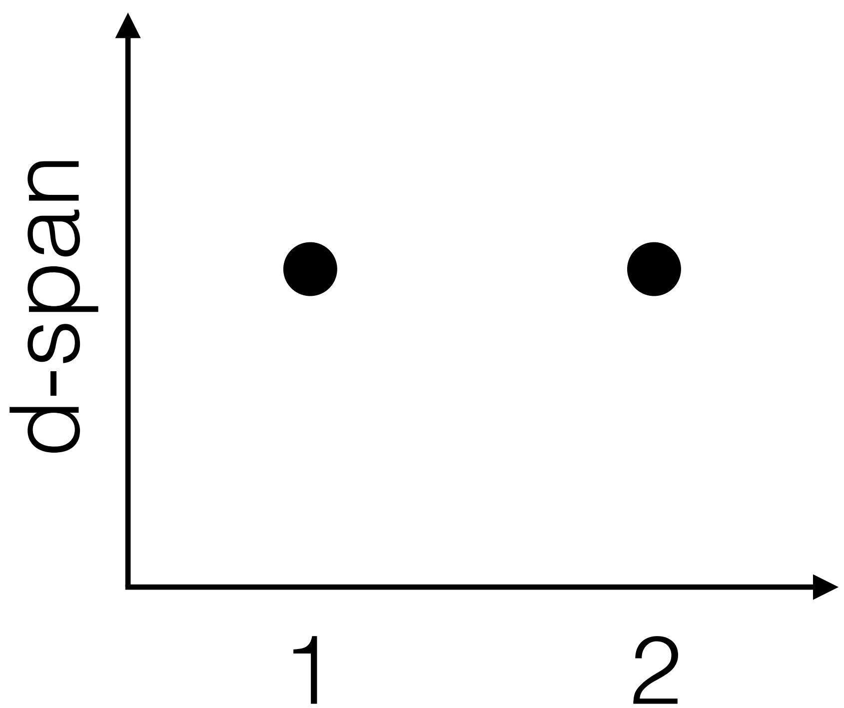
10% of tests on ext4 have d-span > 10GB



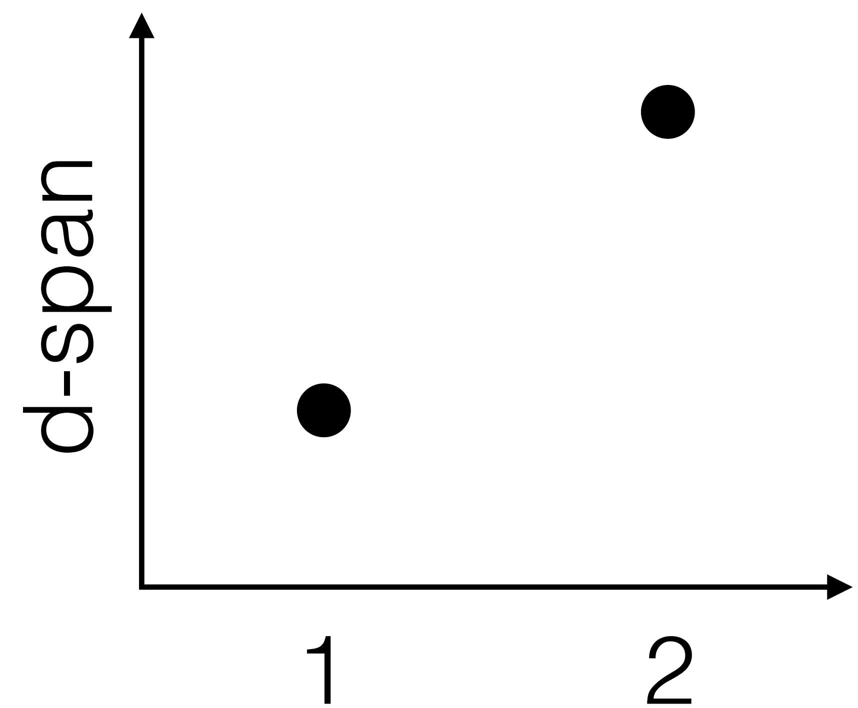
10% of tests on ext4 have d-span > 10GB



Factor prioritization shows which factors influence layout the most



Factor A

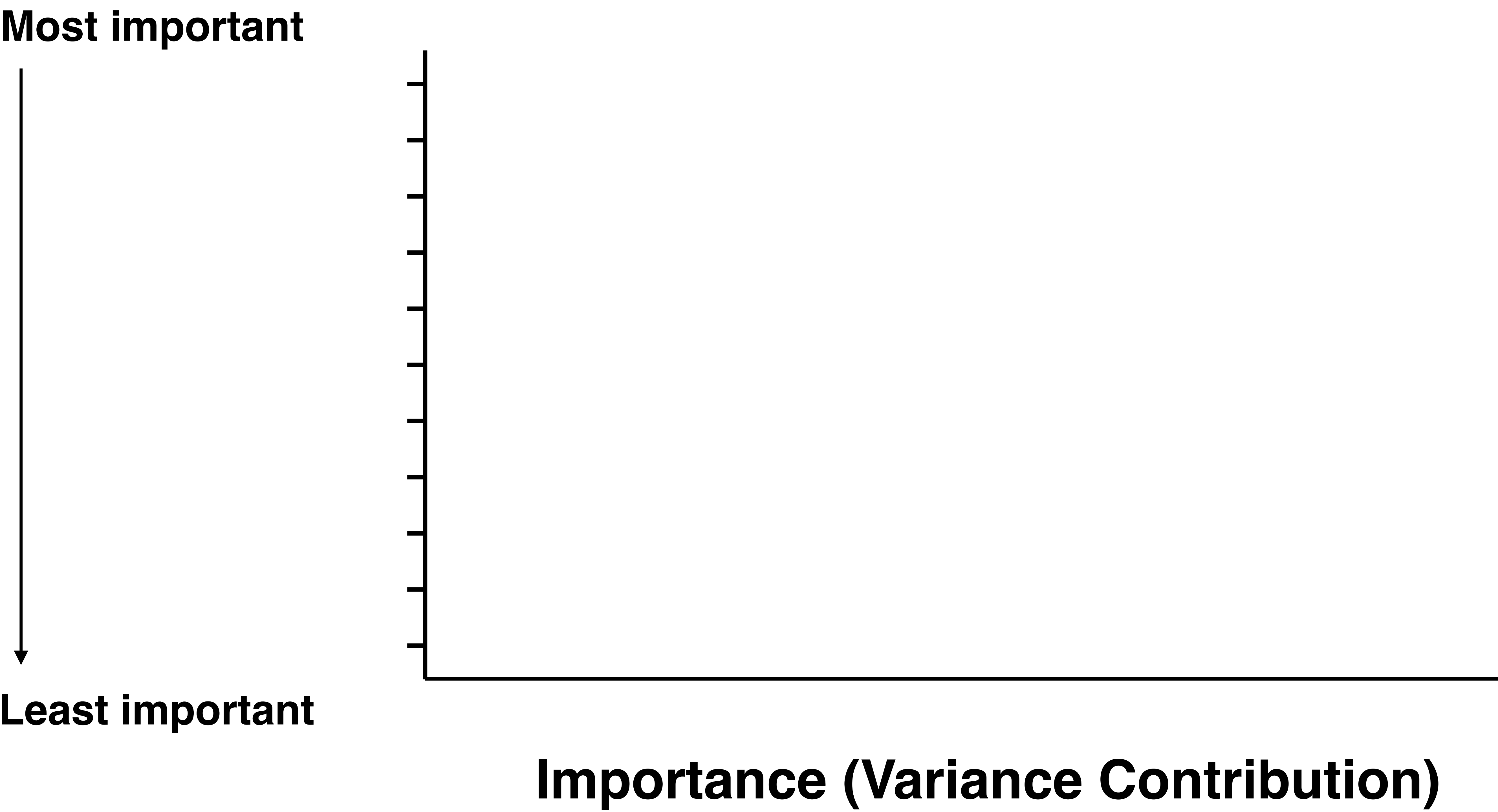


Factor B

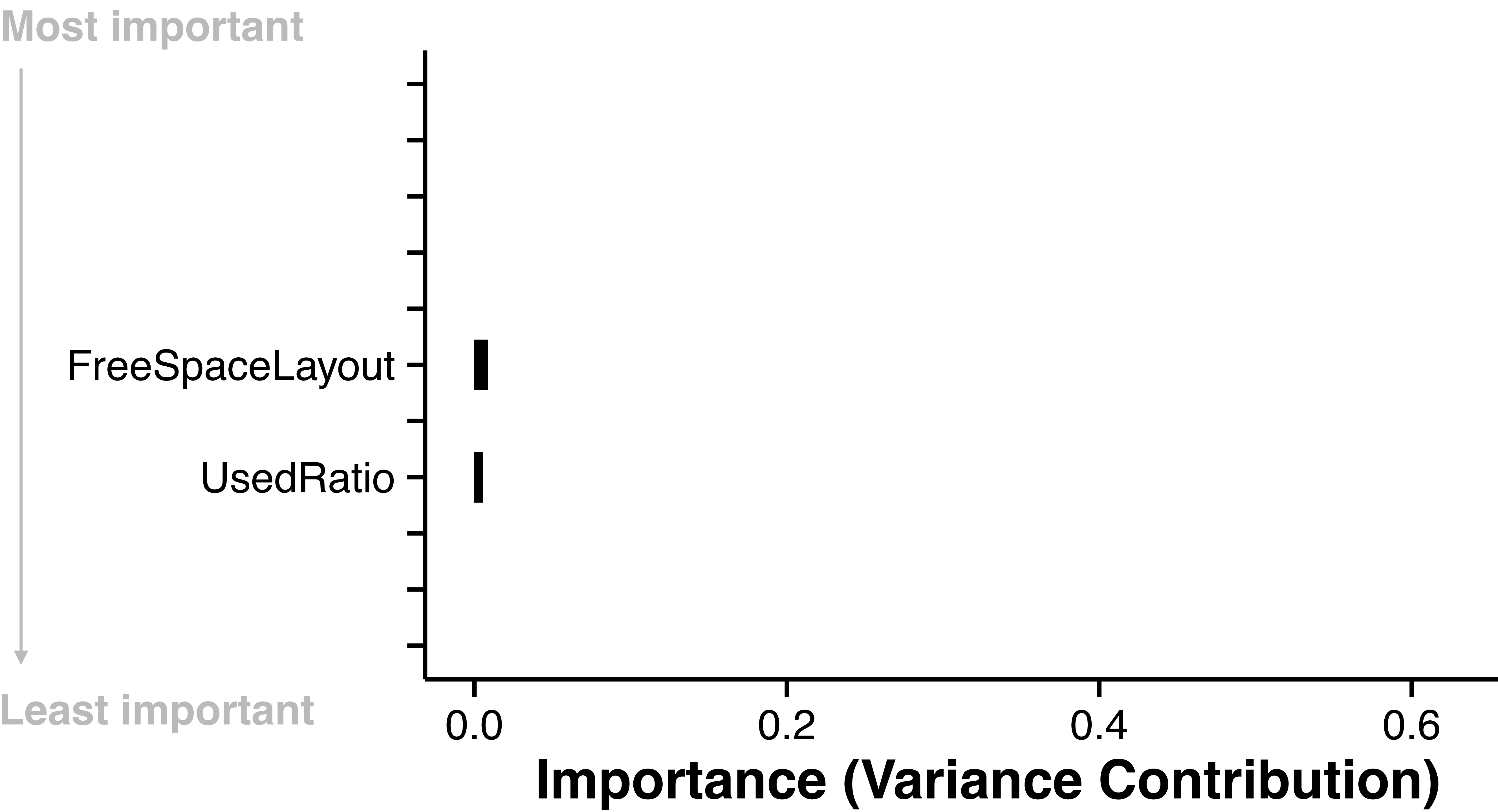
Less important

More important

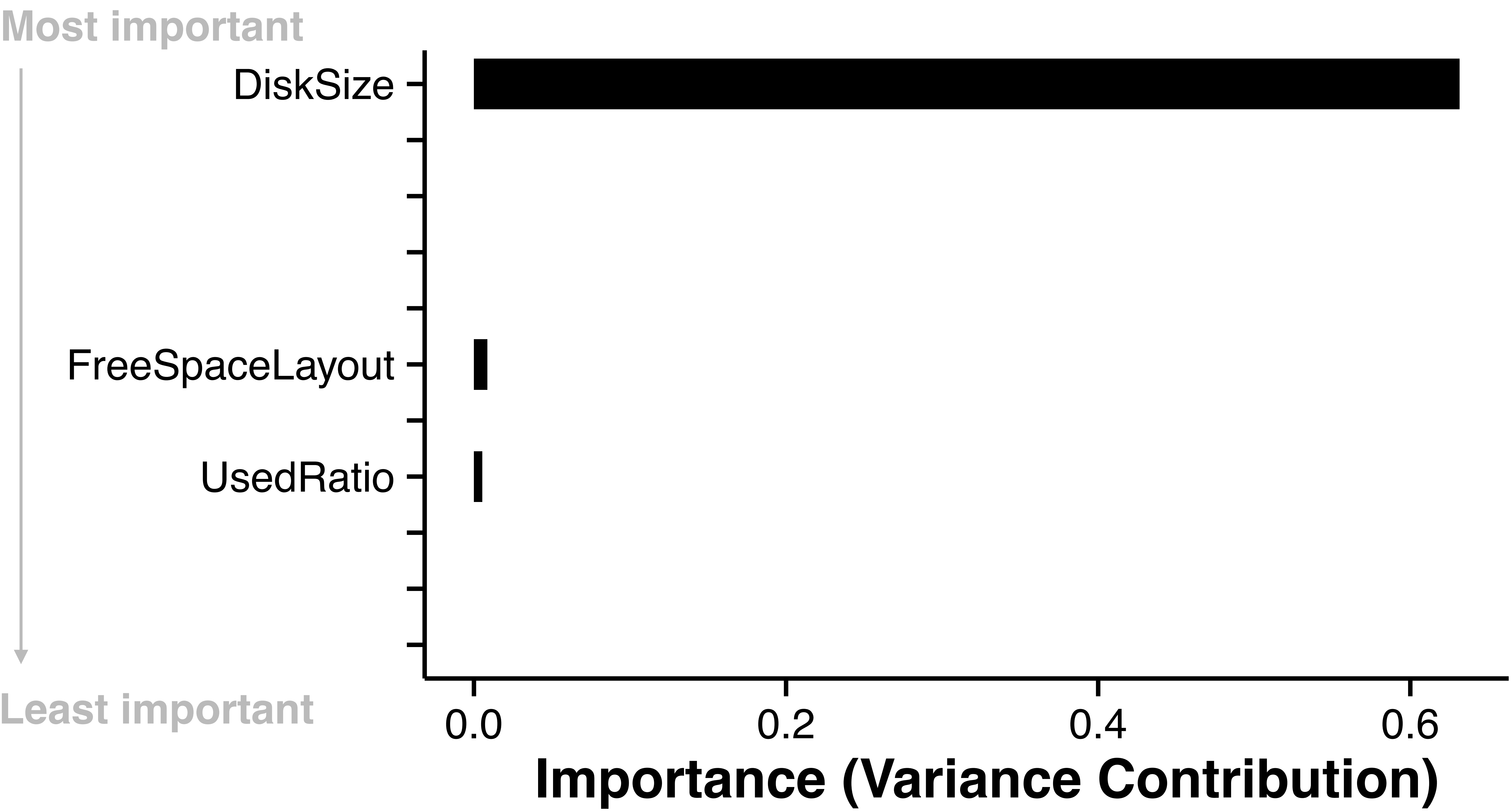
Factor prioritization shows which factors influence layout the most in ext4



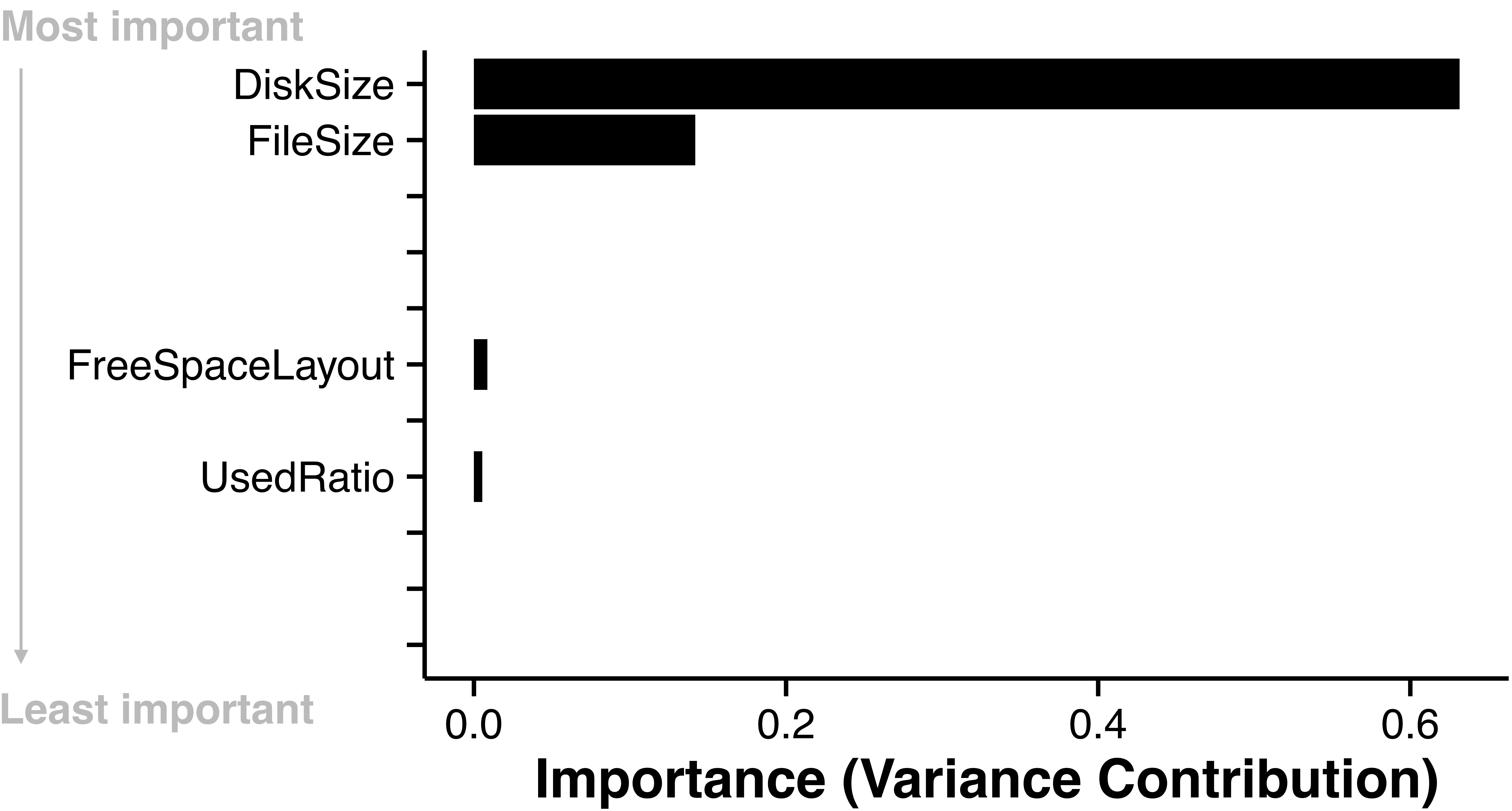
Factor prioritization shows which factors influence layout the most in ext4



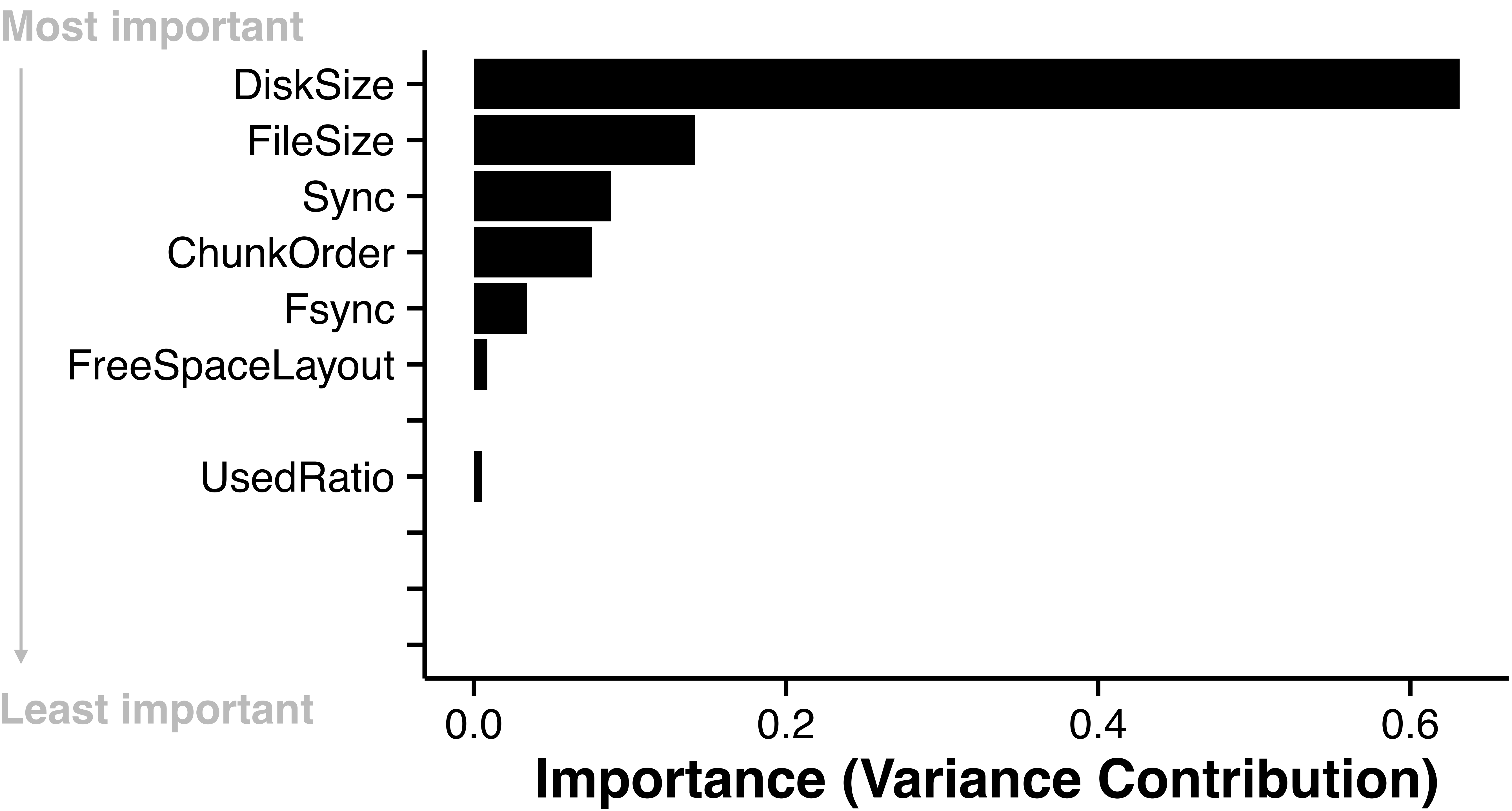
Factor prioritization shows which factors influence layout the most in ext4



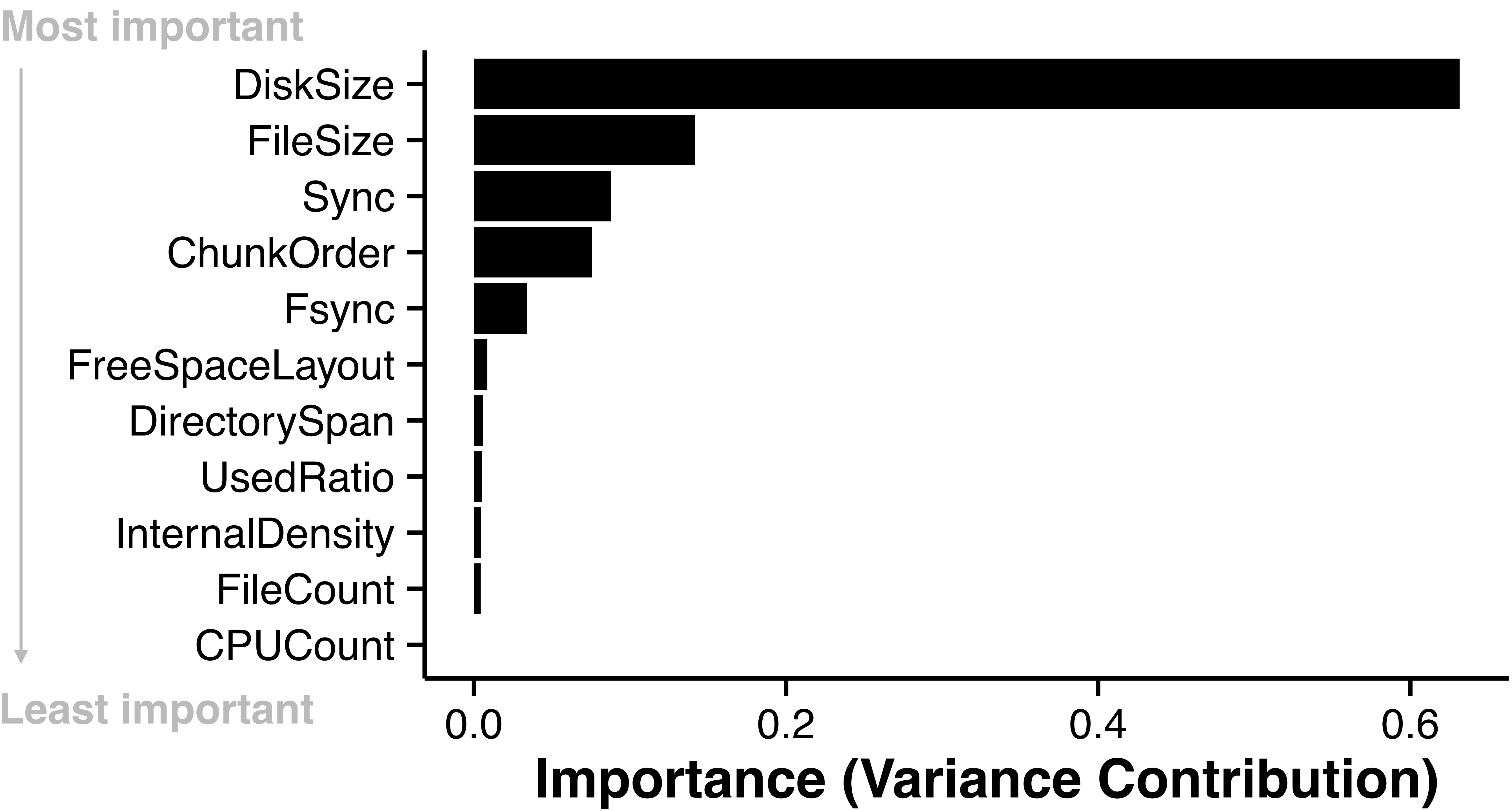
Factor prioritization shows which factors influence layout the most in ext4



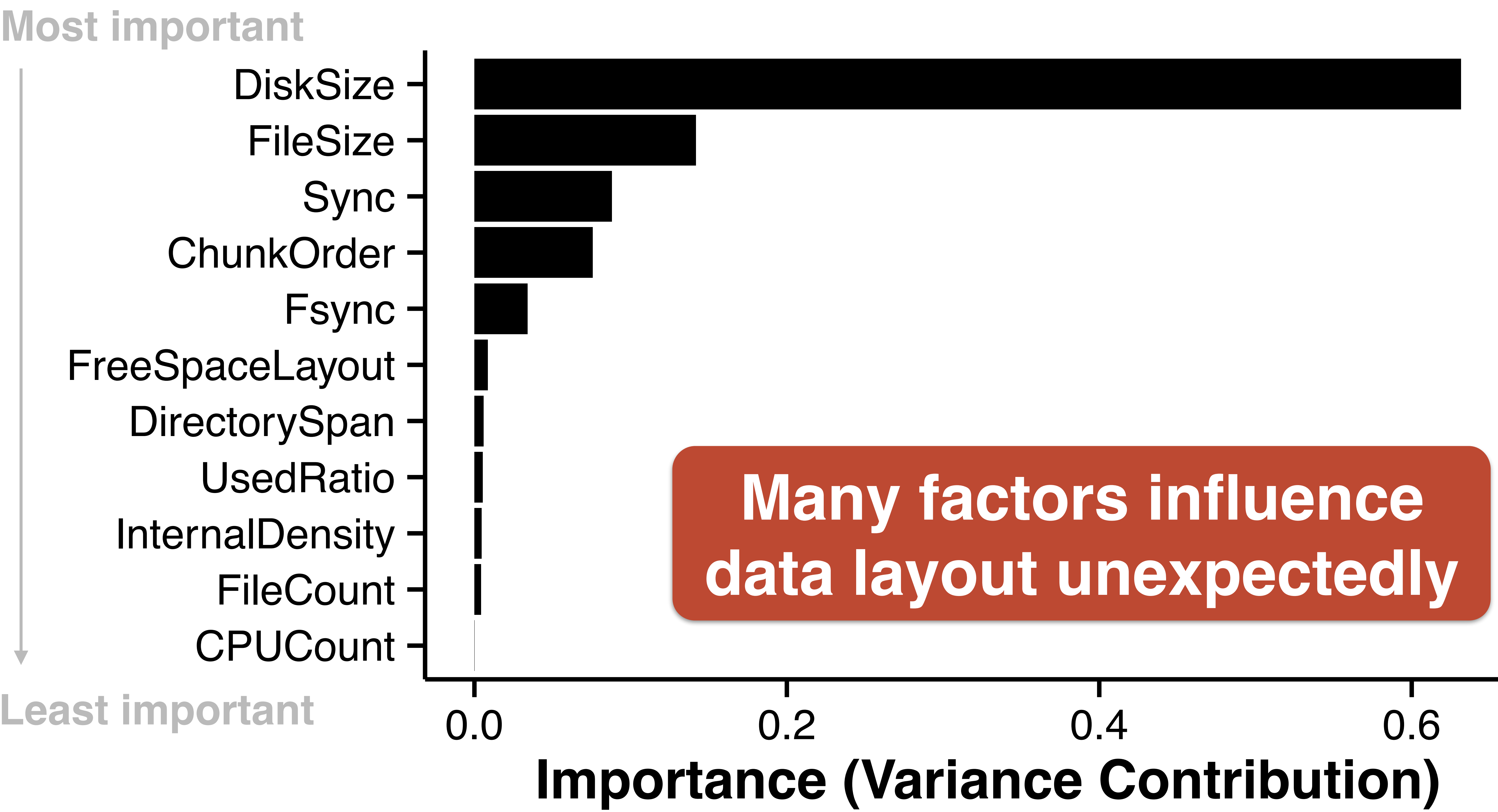
Factor prioritization shows which factors influence layout the most in ext4



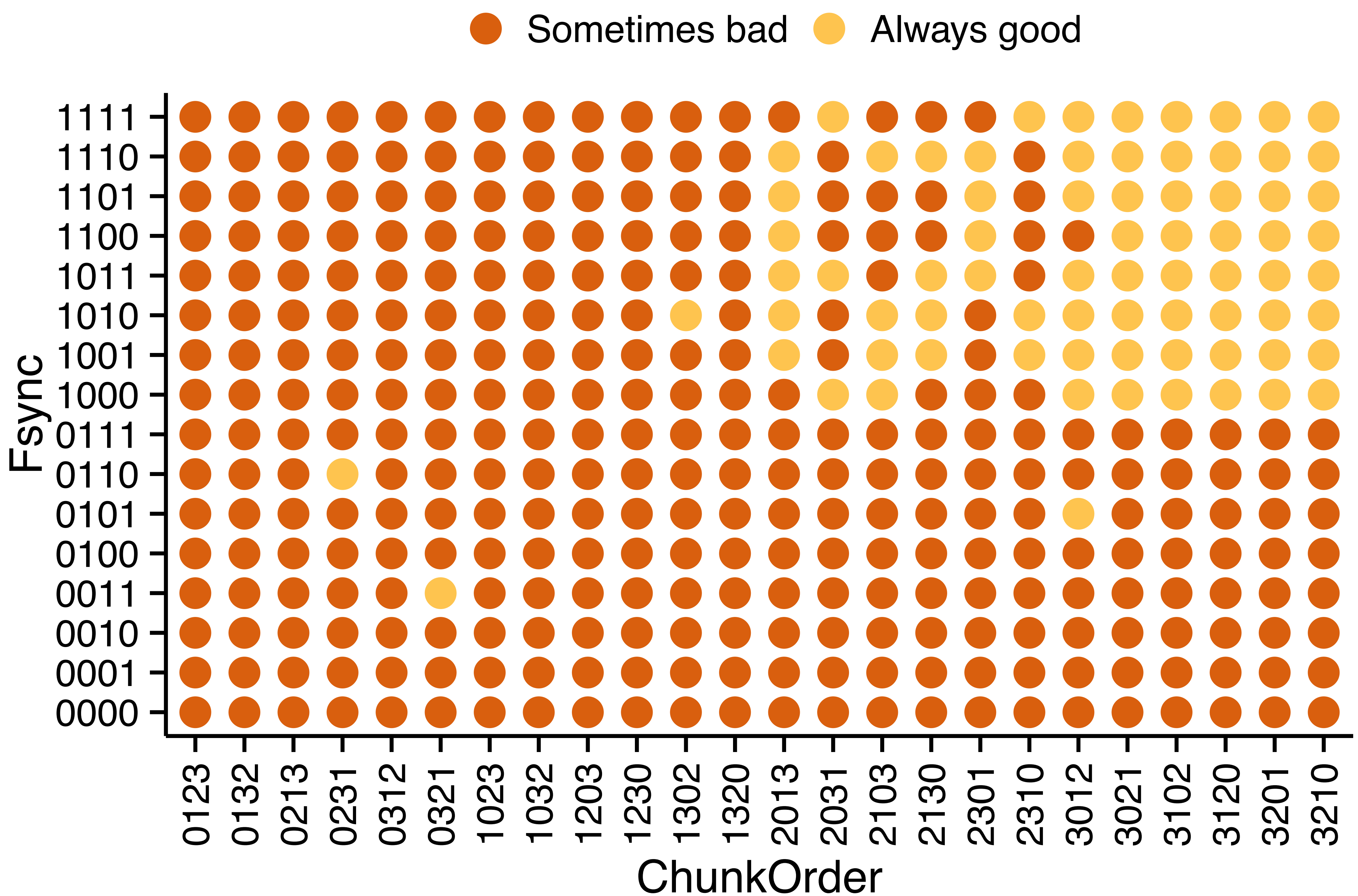
Factor prioritization shows which factors influence layout the most in ext4



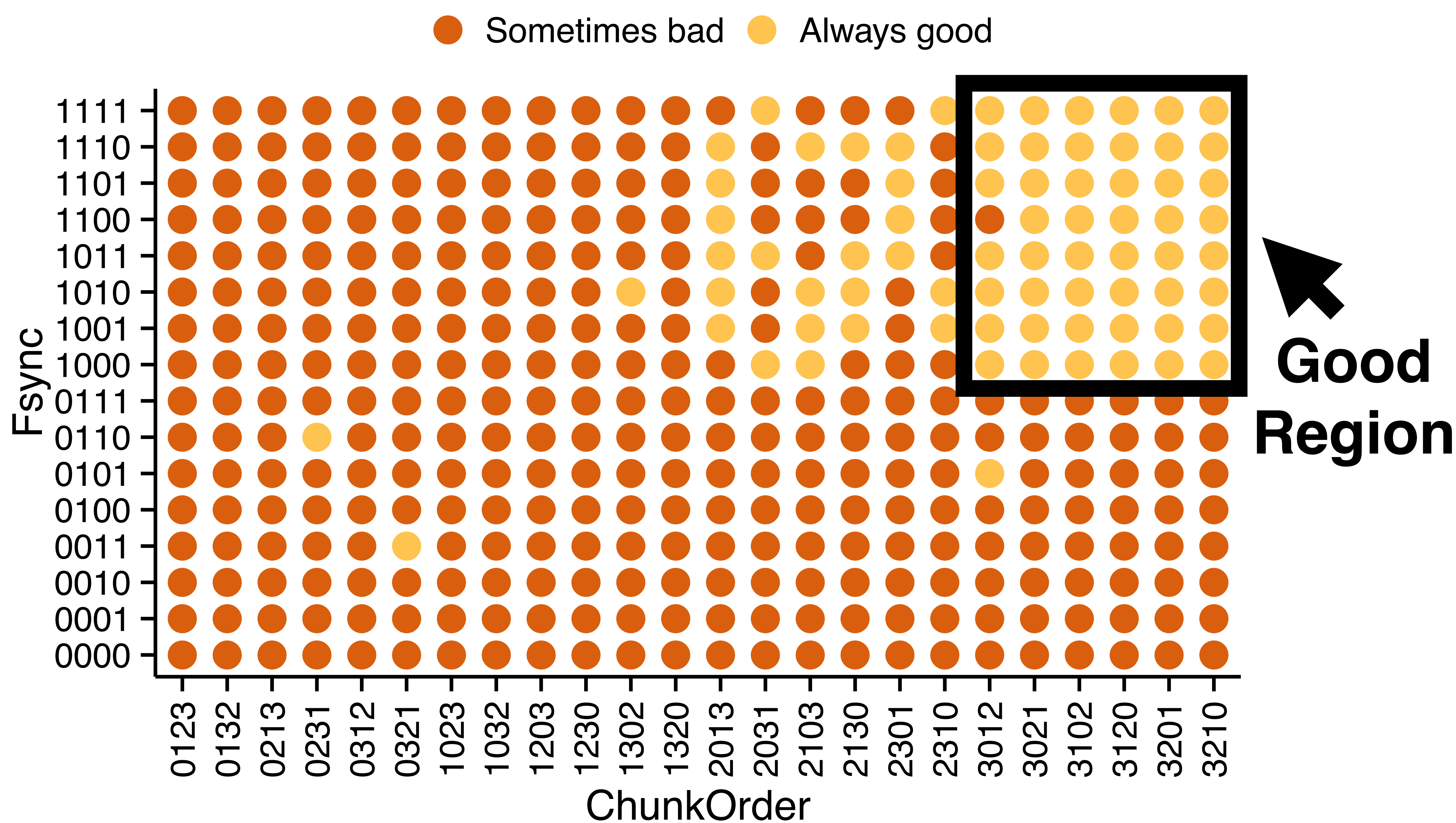
Factor prioritization shows which factors influence layout the most in ext4



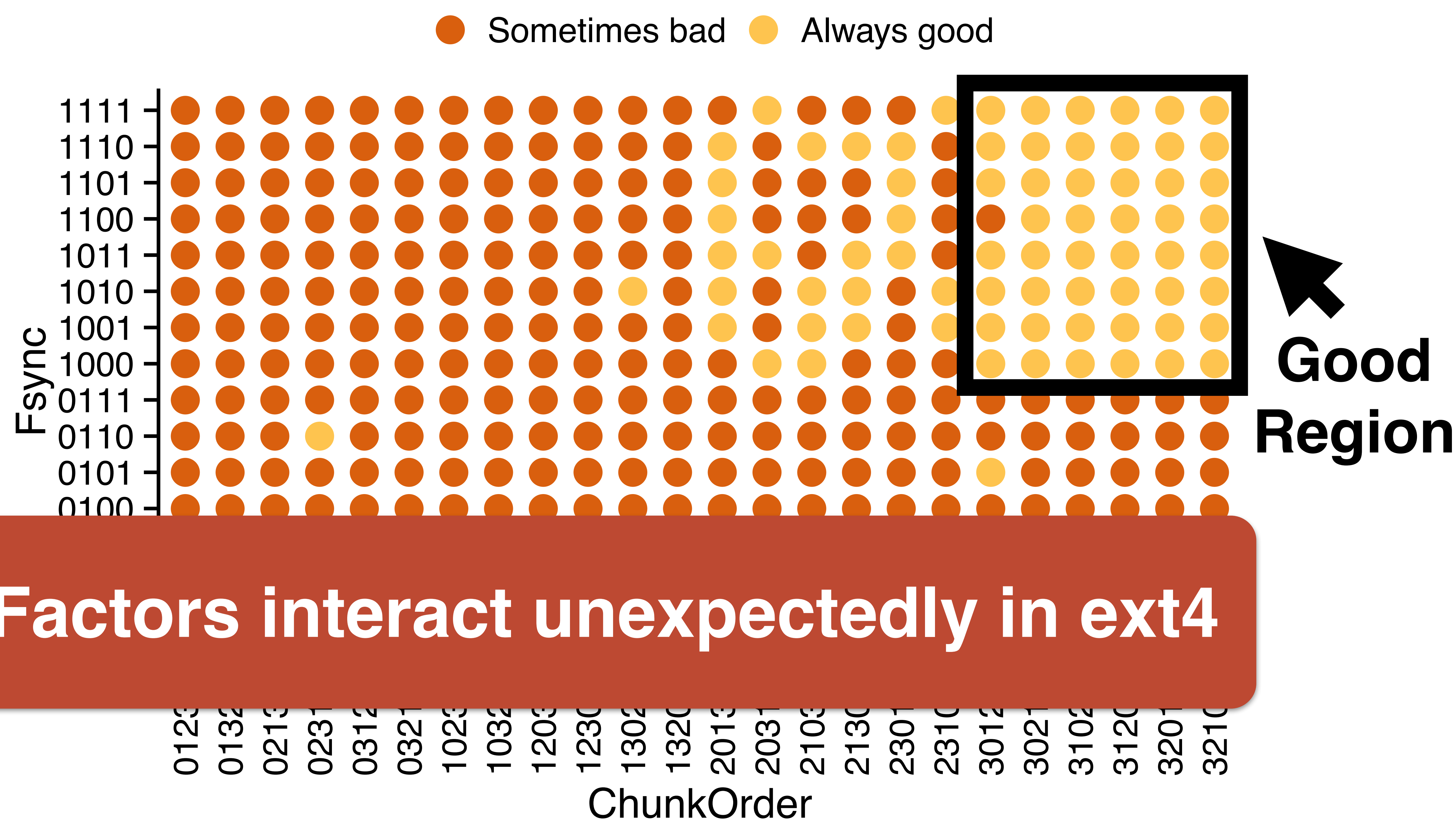
Some combinations always produce good layouts



Some combinations always produce good layouts



Some combinations always produce good layouts



Factors interact unexpectedly in ext4

Summary of unexpected behaviors in ext4

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Sequential writes can be harmful

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Sequential writes can be harmful

Patterns of `sync()` and `fsync()` change layout

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Sequential writes can be harmful

Patterns of `sync()` and `fsync()` change layout

Fragmentations and used ratio of disk don’t matter

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Sequential writes can be harmful

Patterns of `sync()` and `fsync()` change layout

Fragmentations and used ratio of disk don’t matter

Factors interact when determining data layout

Summary of unexpected behaviors in ext4

Linux ext4 may spread files wider on larger disks

File size influences d-span more than expected

Sequential writes can be harmful

Patterns of `sync()` and `fsync()` change layout

Fragmentations and used ratio of disk don’t matter

Factors interact when determining data layout

More in the paper

Outline

Part 1

Collect Data

Part 2

Analyze Data

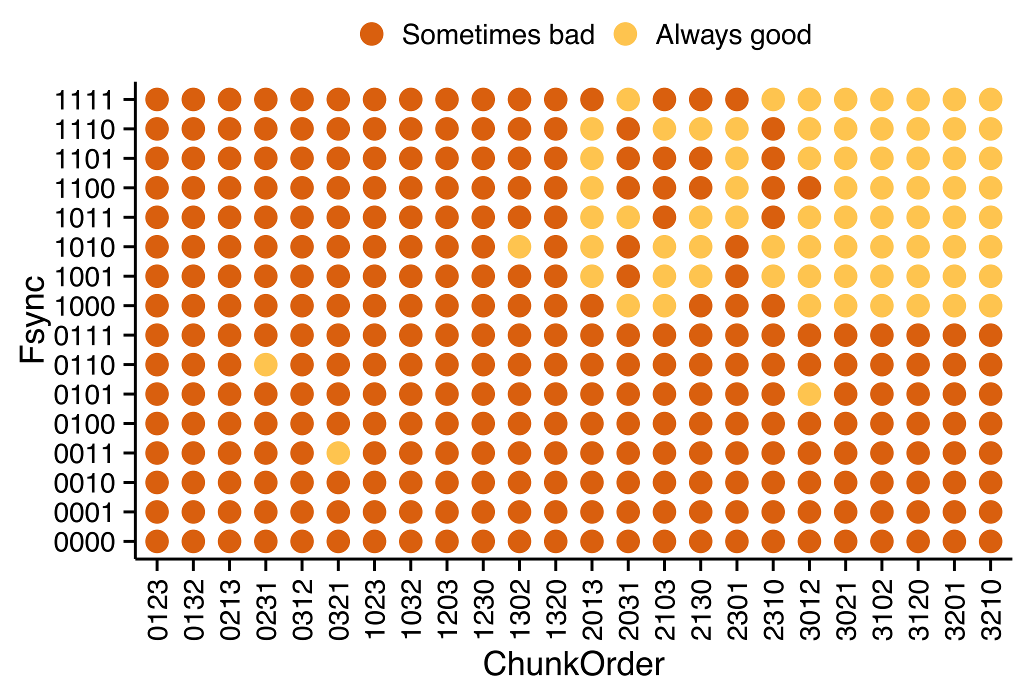
Part 3

Understand File System

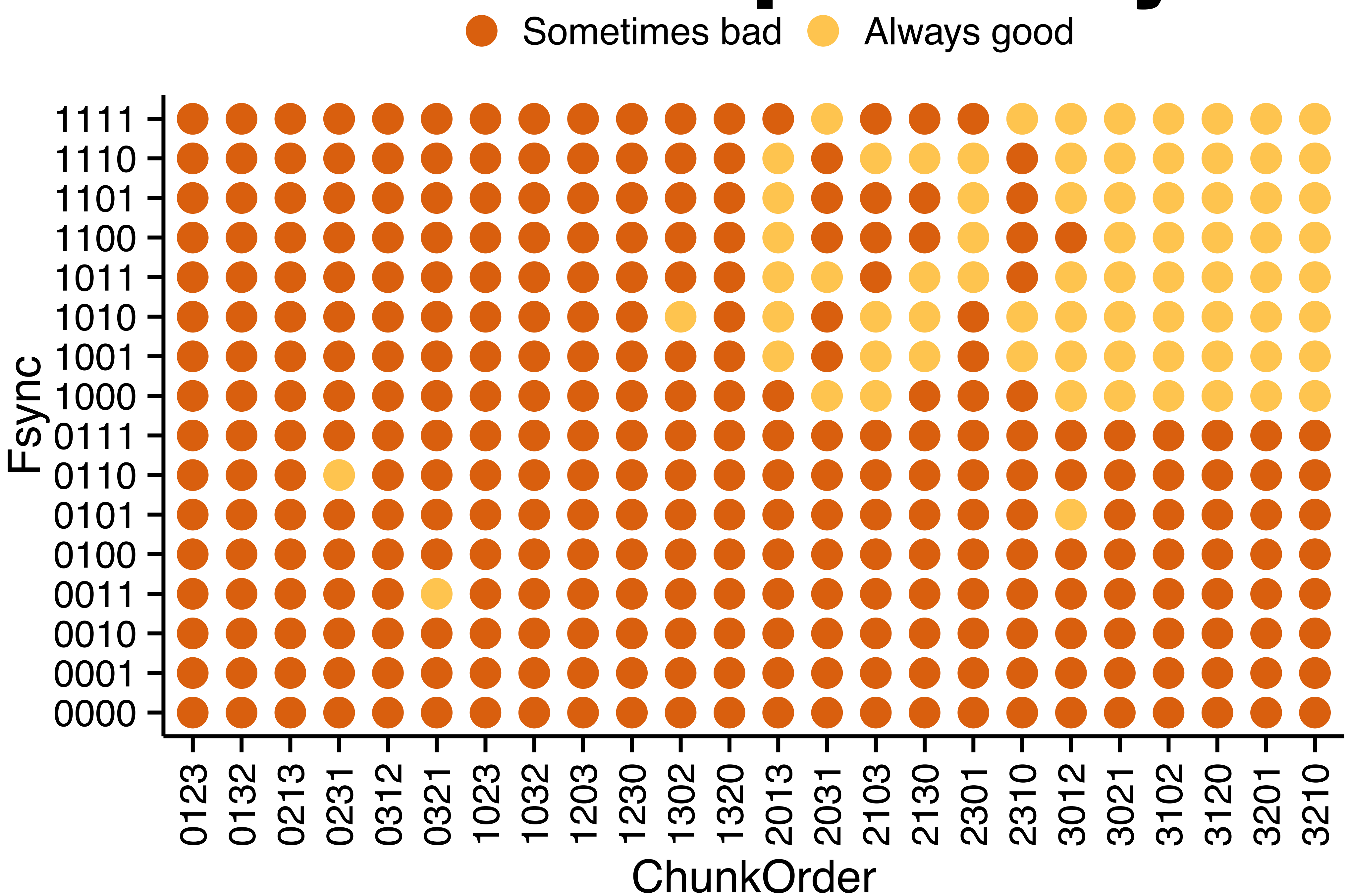
The unexpected behaviors in ext4 can be explained by four design issues

Special End Policy	in this talk
Scheduler Dependency	in this talk
File Size Dependency	in paper
Normalization Bug	in paper

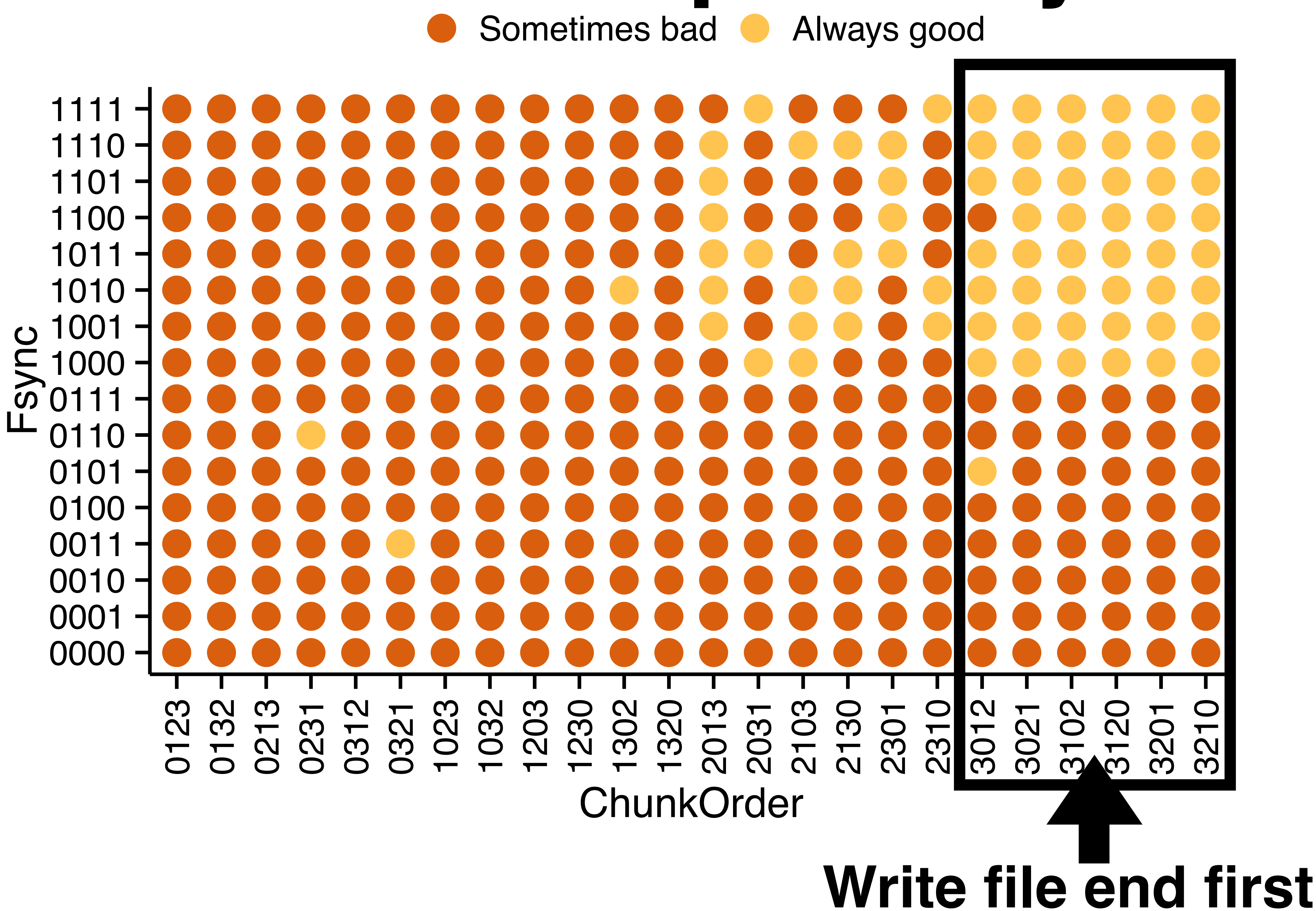
Special End Policy



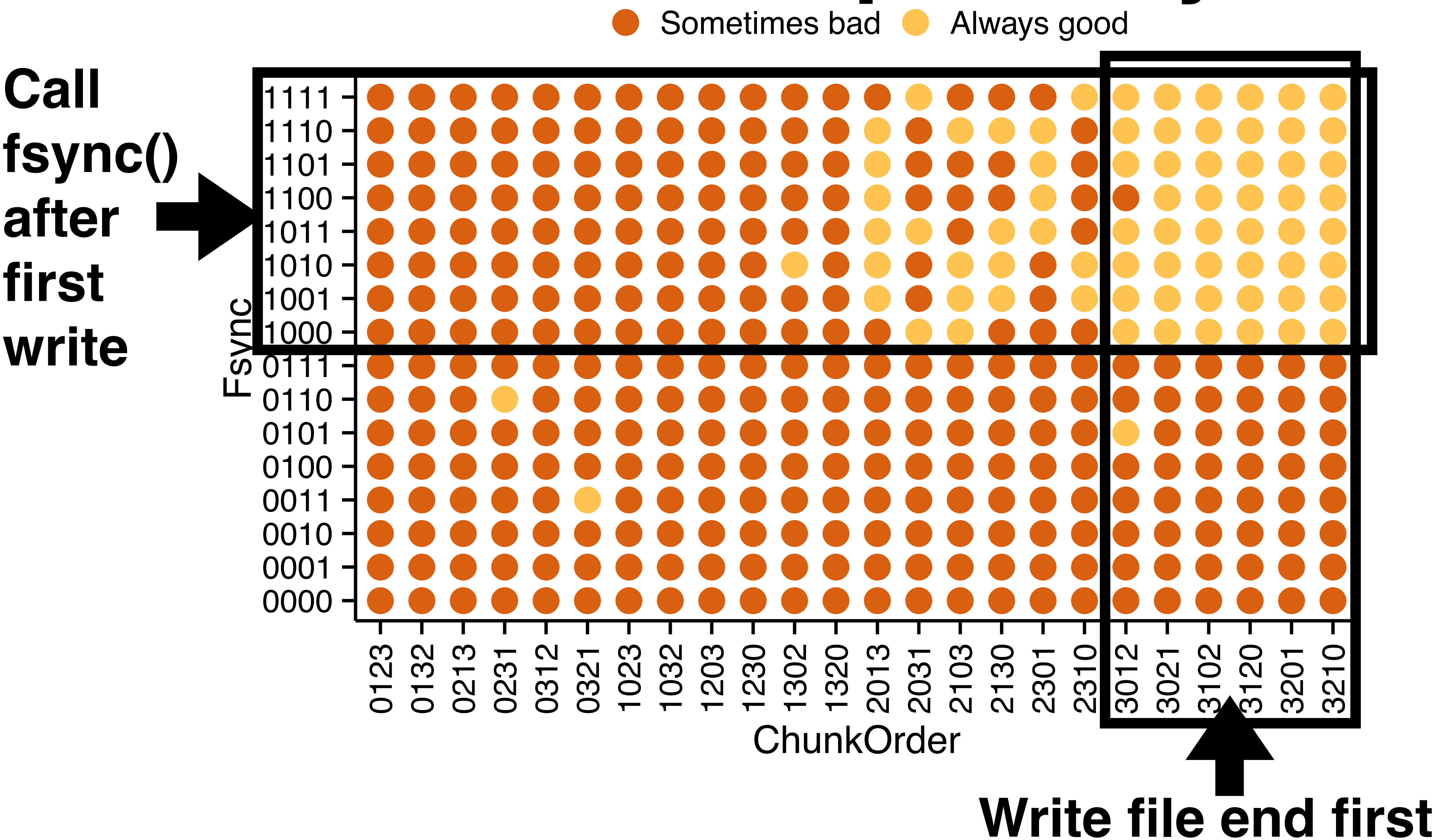
Writing and syncing file end first could avoid poor layout



Writing and syncing file end first could avoid poor layout

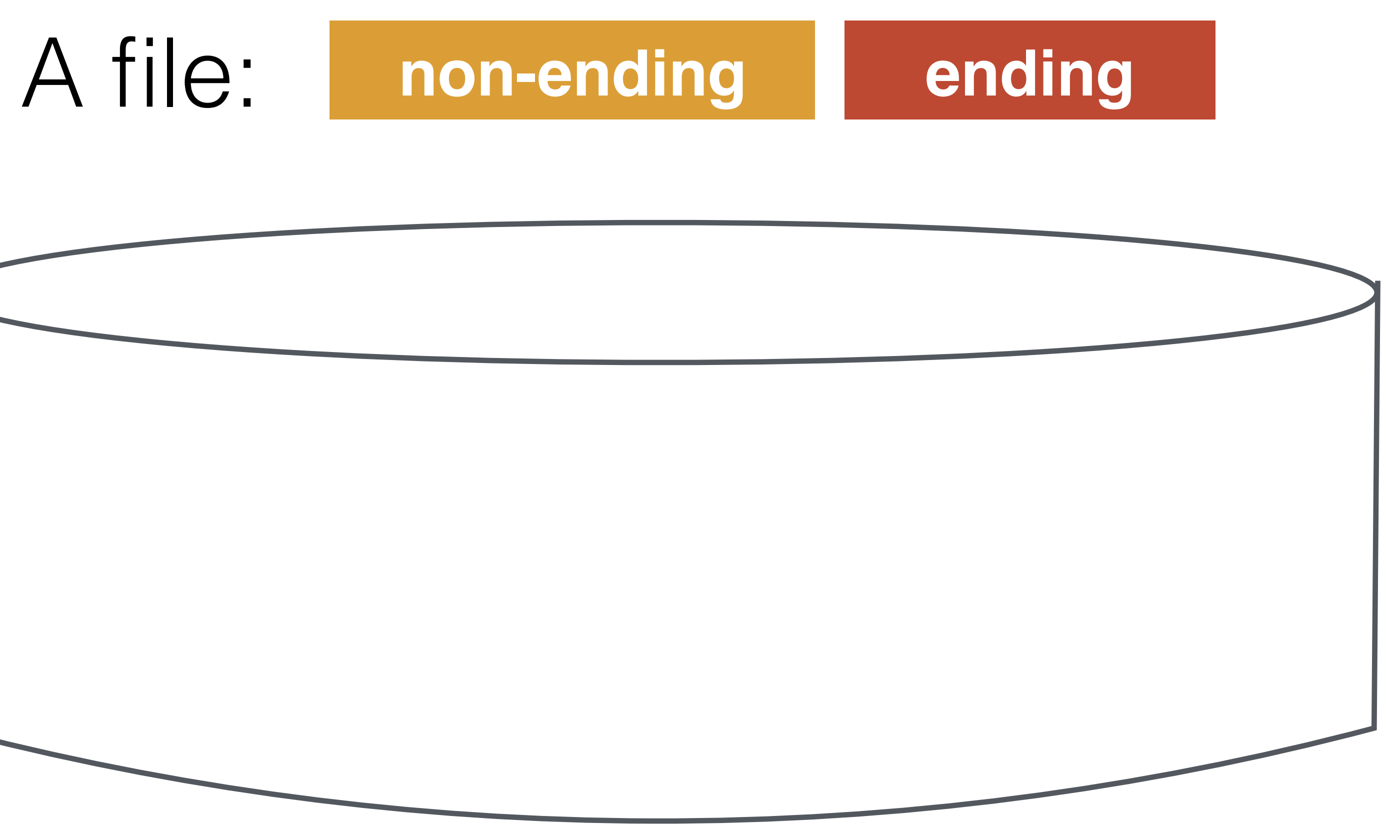


Writing and syncing file end first could avoid poor layout



Why layout is bad? The allocator treats the ending data extent of a file differently

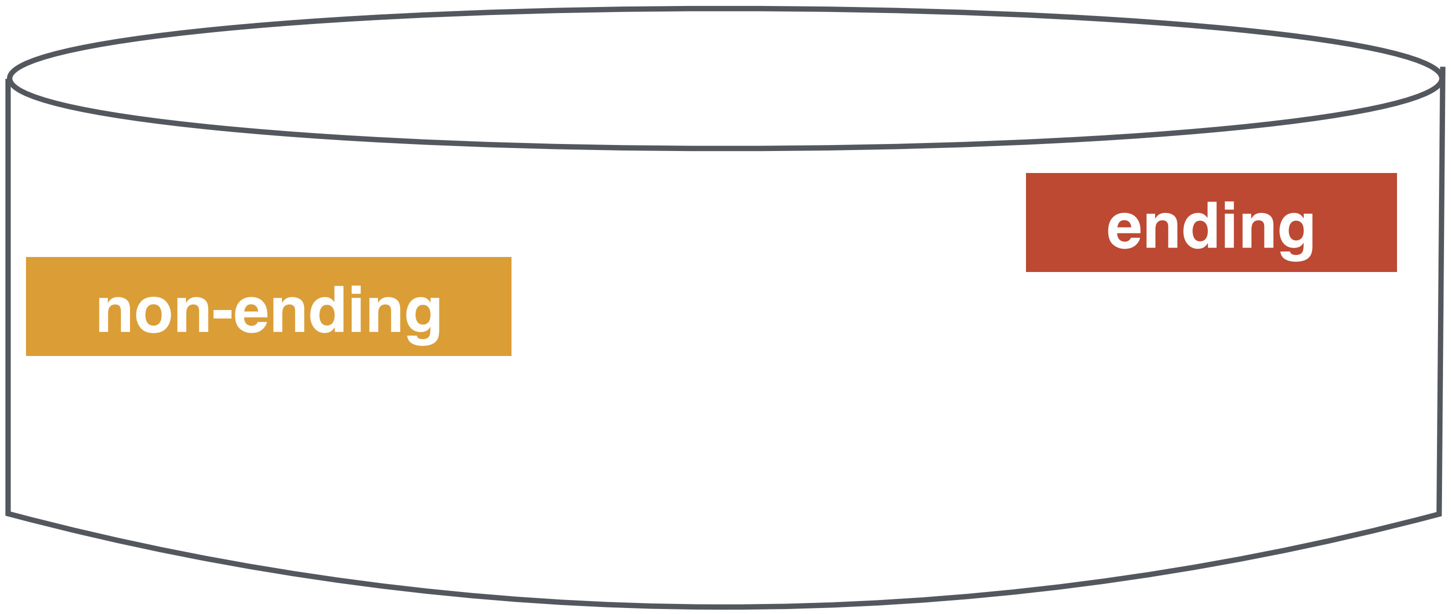
Condition 1: the extent is at the end of the file
Condition 2: the file is closed

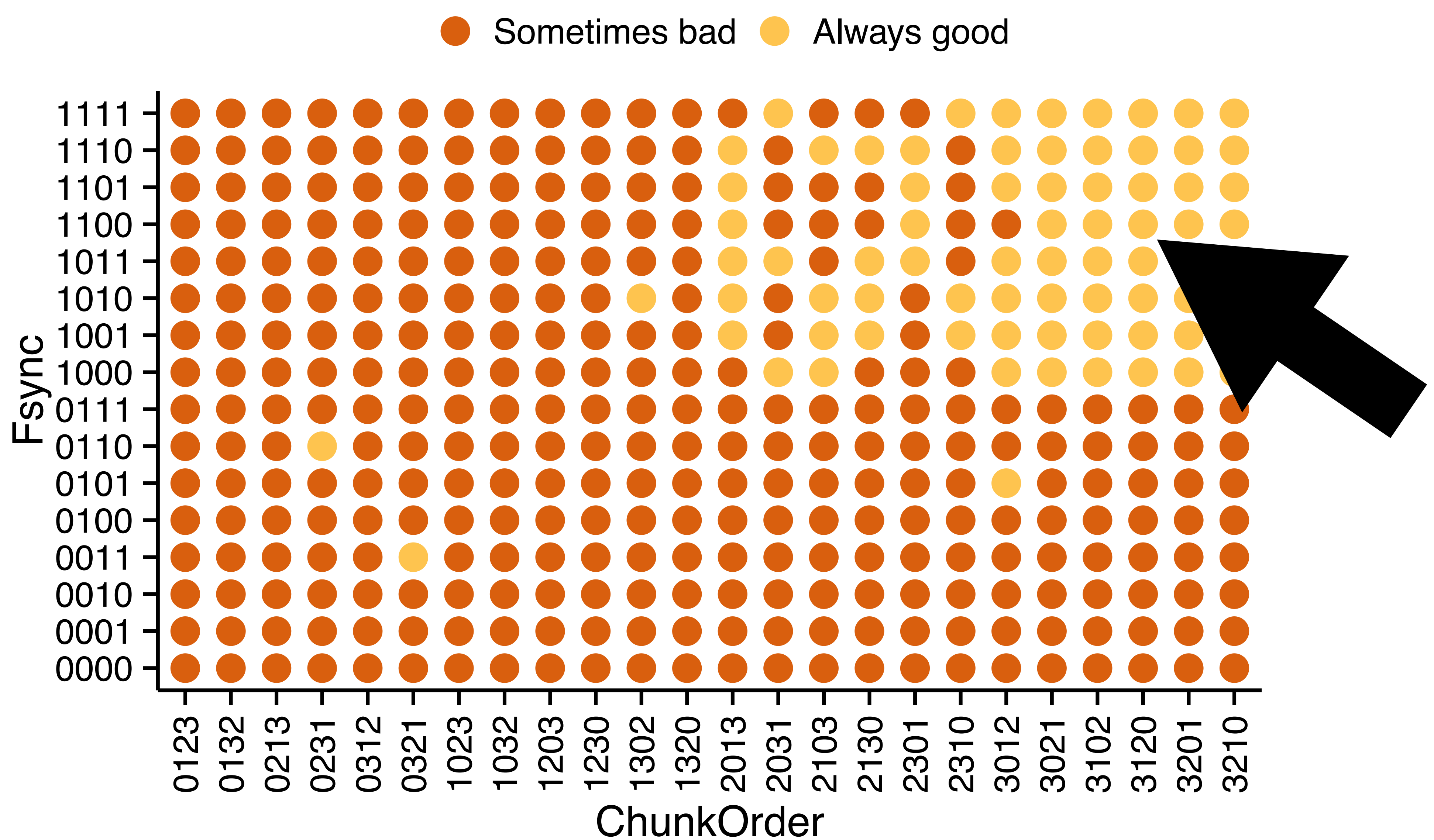


Why layout is bad? The allocator treats the ending data extent of a file differently

Condition 1: the extent is at the end of the file
Condition 2: the file is closed

A file:





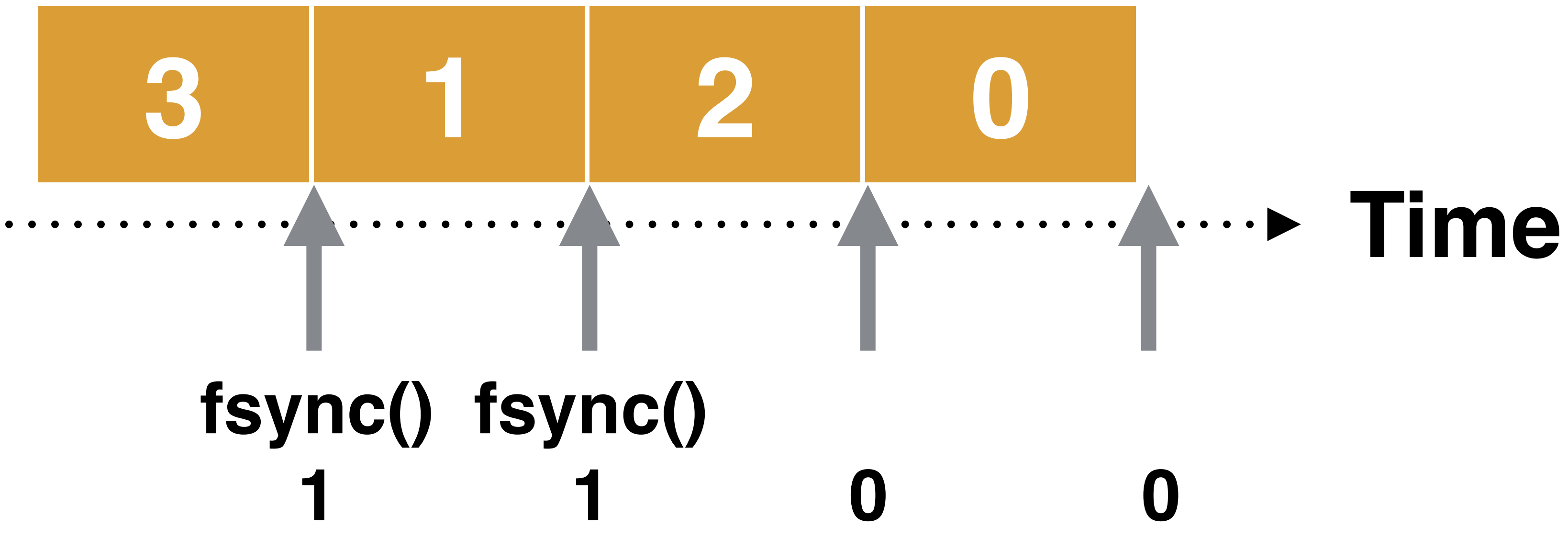
Sometimes bad
Always good
Fsync
1111
1110
1101
1100
1011
1010
1001
1000
0111
0110
0101
0100
0011
0010
0001
0000
ChunkOrder
0123
0132
0213
0231
0312
0321
1023
1032
1203
1230
1302
1320
2013
2031
2103
2130
2301
2310
3012
3021
3102
3120
3201
3210

Why ChunkOrder=3120 and Fsync=1100 always have good layout?

Cond 1 (is end?):

Cond 2 (is closed?):

Special End?

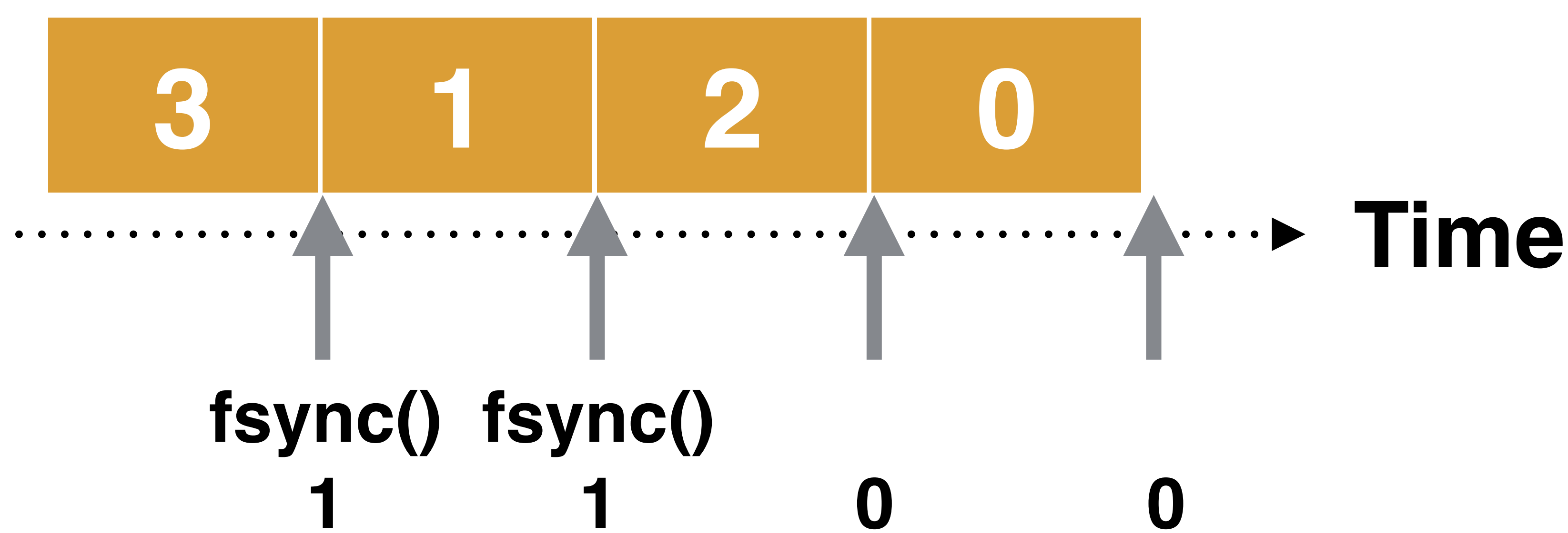


Why ChunkOrder=3120 and Fsync=1100 always have good layout?

Cond 1 (is end?): ✔

Cond 2 (is closed?):

Special End?

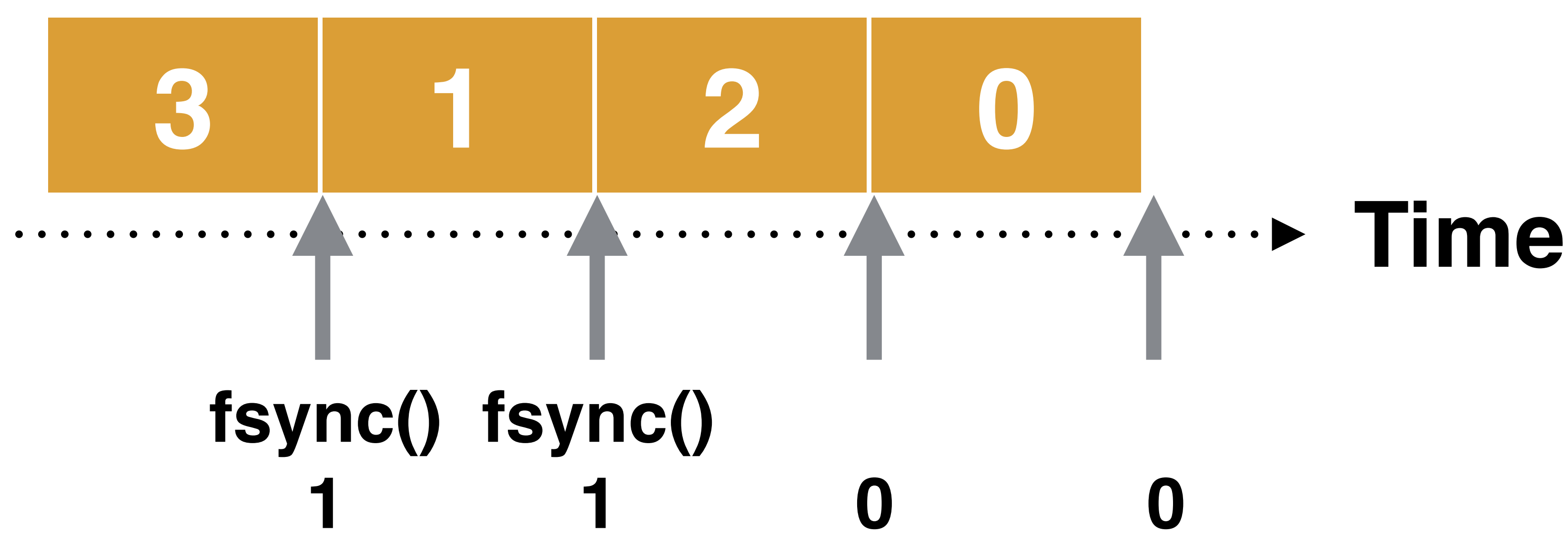


Why ChunkOrder=3120 and Fsync=1100 always have good layout?

Cond 1 (is end?): ✔

Cond 2 (is closed?): ✖

Special End?

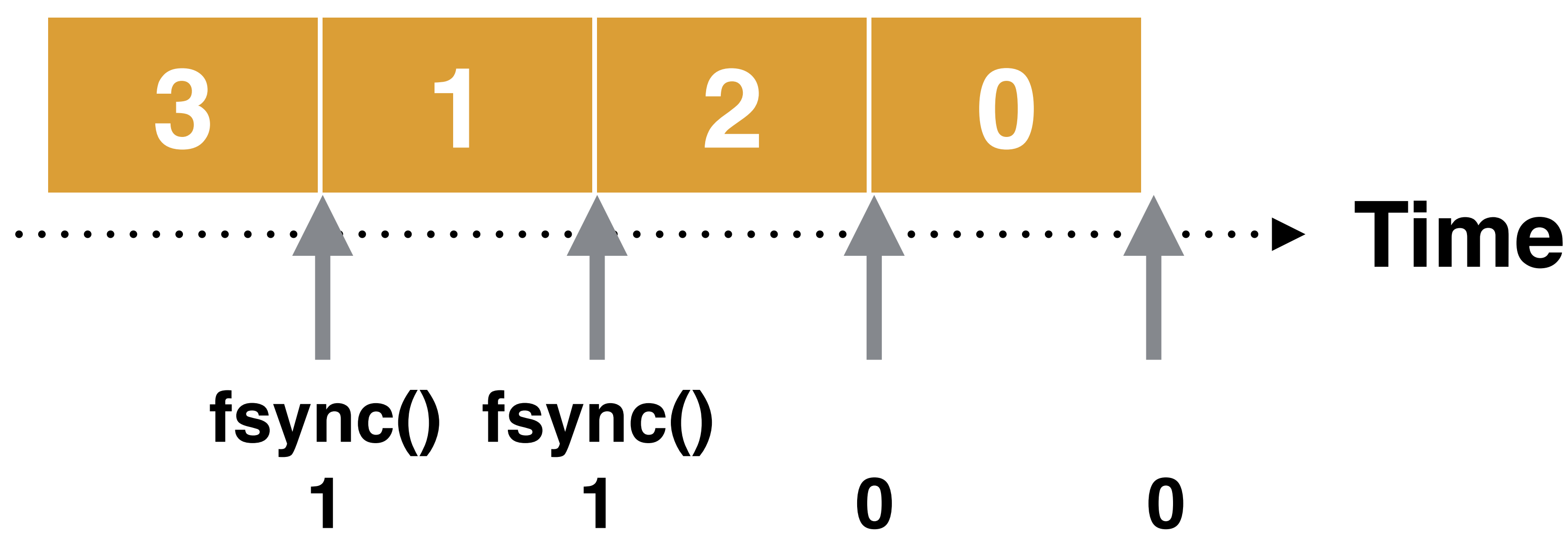


Why ChunkOrder=3120 and Fsync=1100 always have good layout?

Cond 1 (is end?): ✔

Cond 2 (is closed?): ✖

Special End? ✖

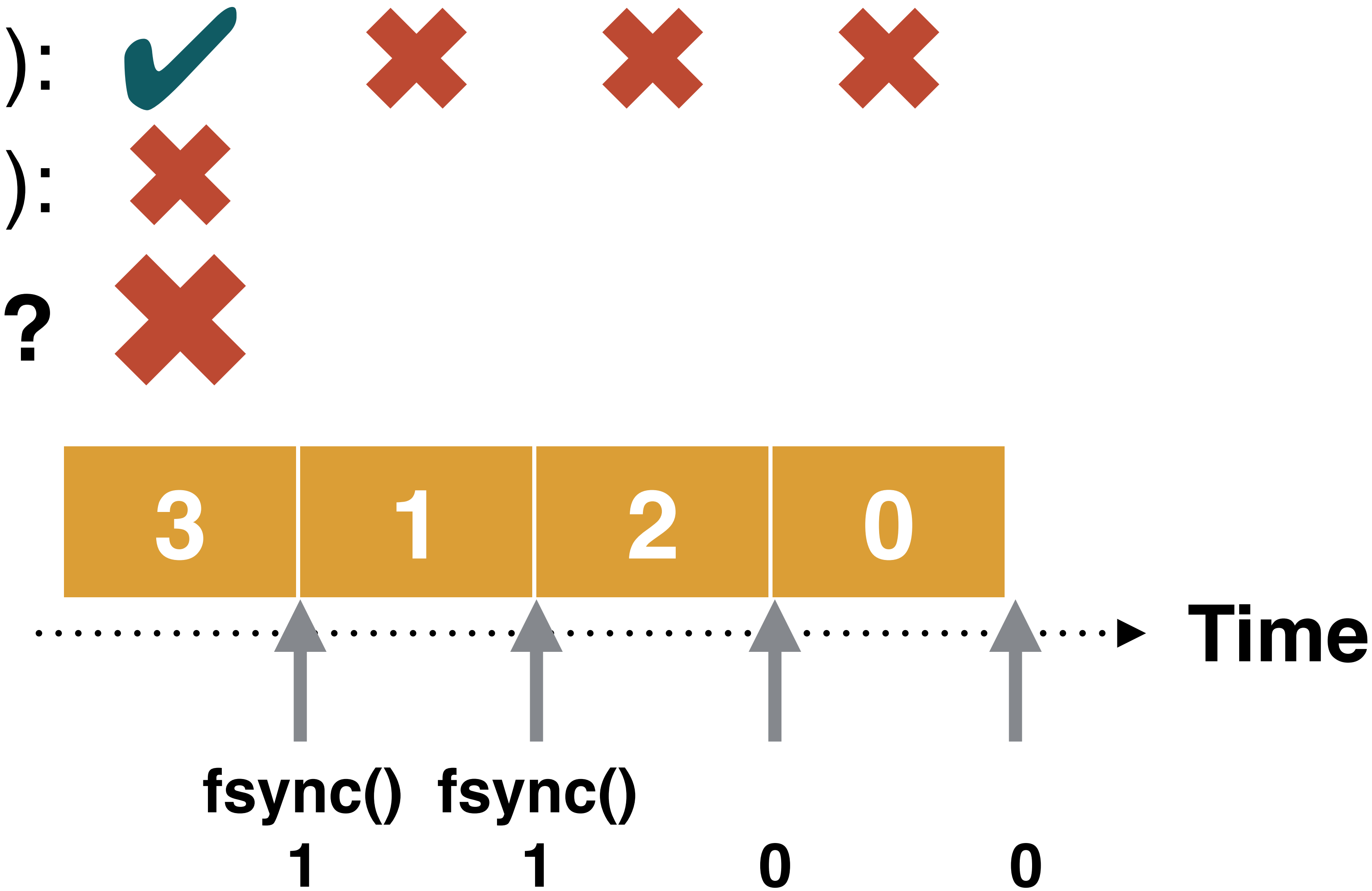


Why ChunkOrder=3120 and Fsync=1100 always have good layout?

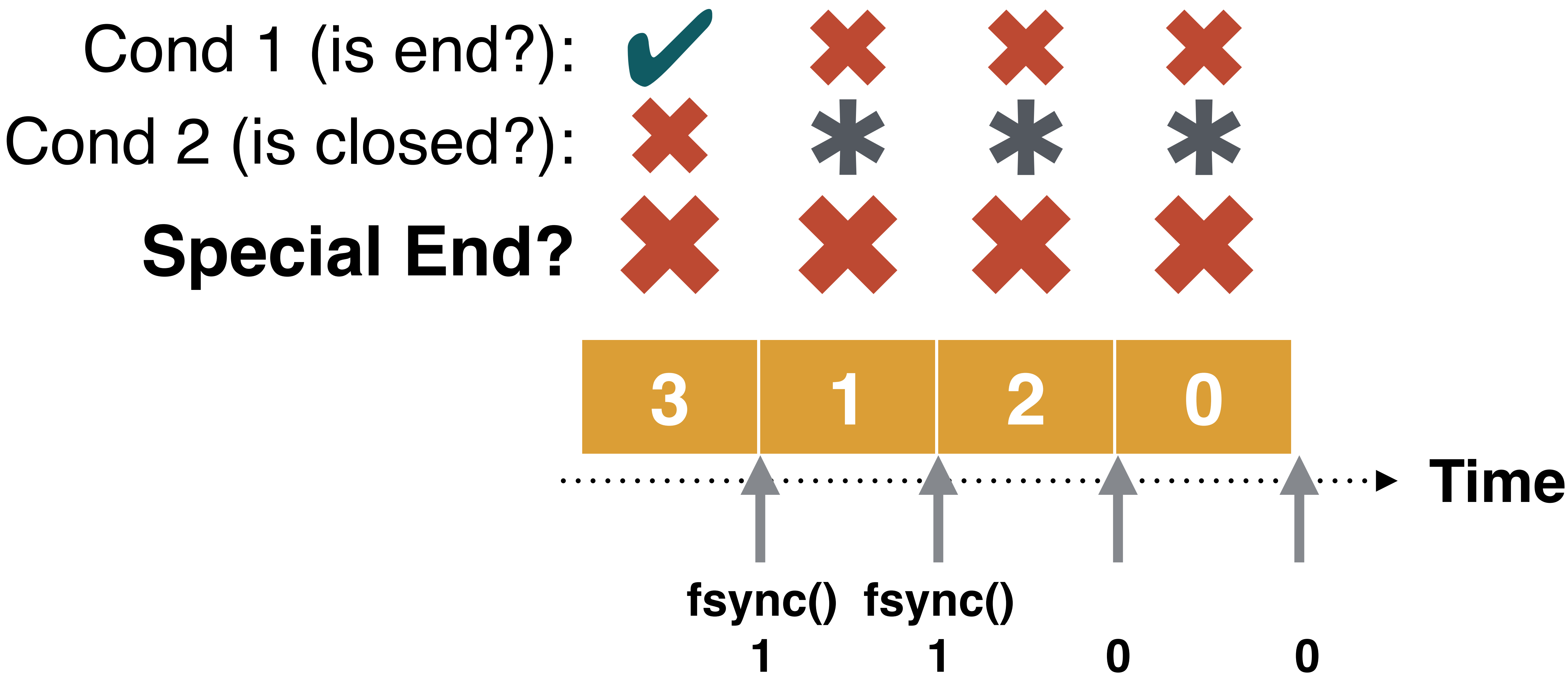
Cond 1 (is end?):

Cond 2 (is closed?):

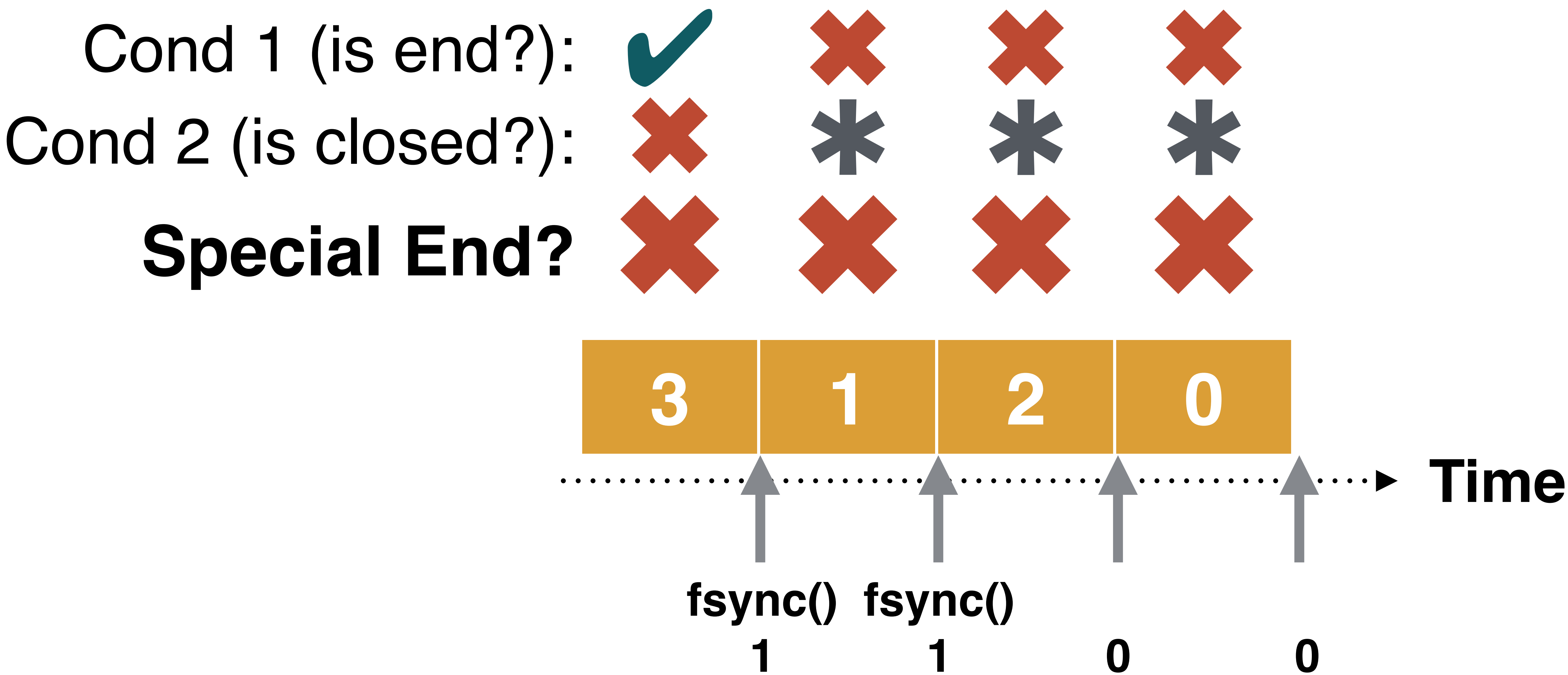
Special End?



Why ChunkOrder=3120 and Fsync=1100 always have good layout?



Why ChunkOrder=3120 and Fsync=1100 always have good layout?



Special End Policy is never triggered

Special End Policy

Fix

Treat non-ending and ending extents equally

Lesson Learned

Policies for different circumstances should be harmonious with one another

Scheduler Dependency

6% of d-spans are different in two repeated experiments

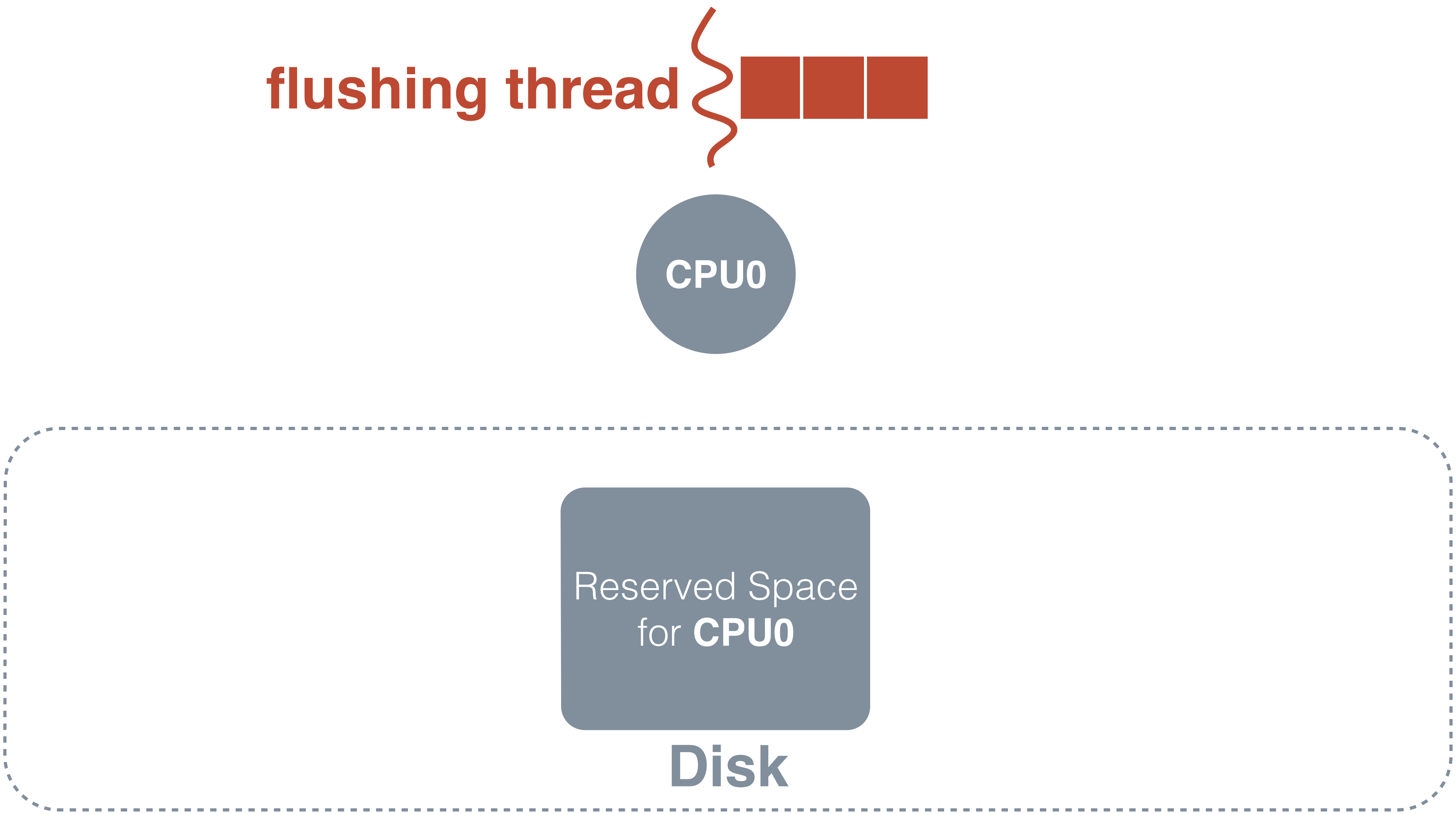
Up to 44GB difference on a 64GB disk

6% of d-spans are different in two repeated experiments

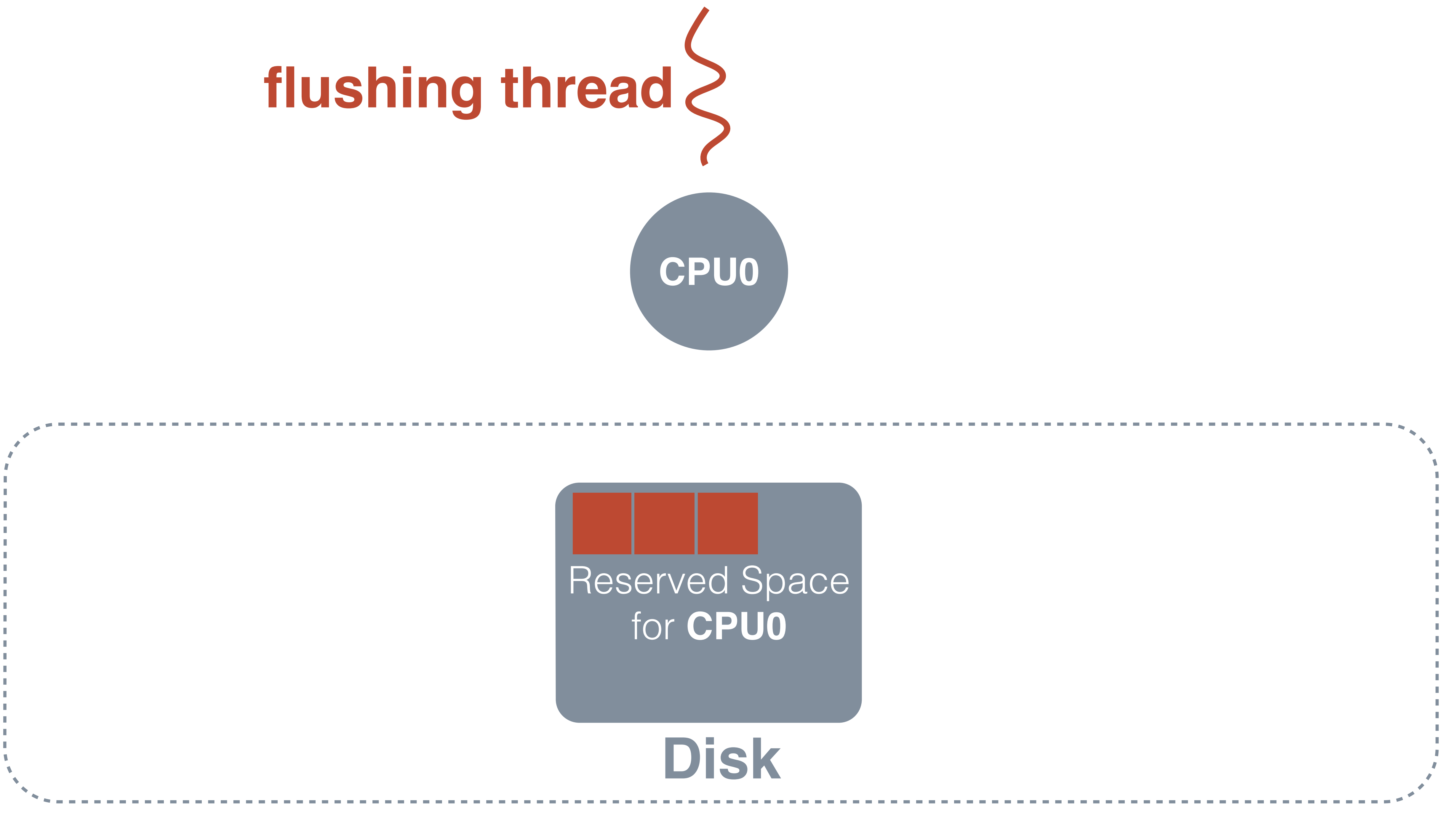
Up to 44GB difference on a 64GB disk

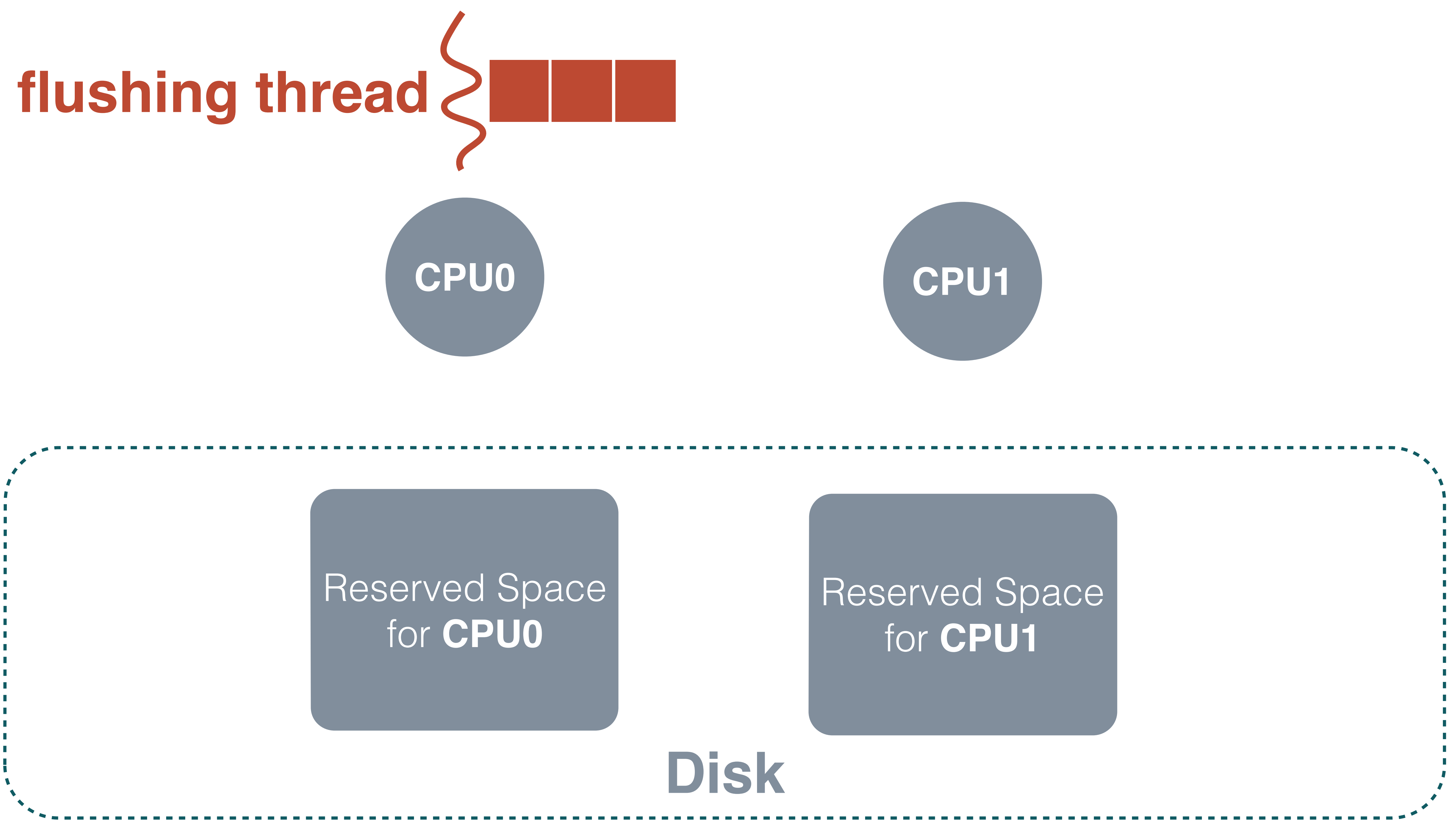
Data layout can be random

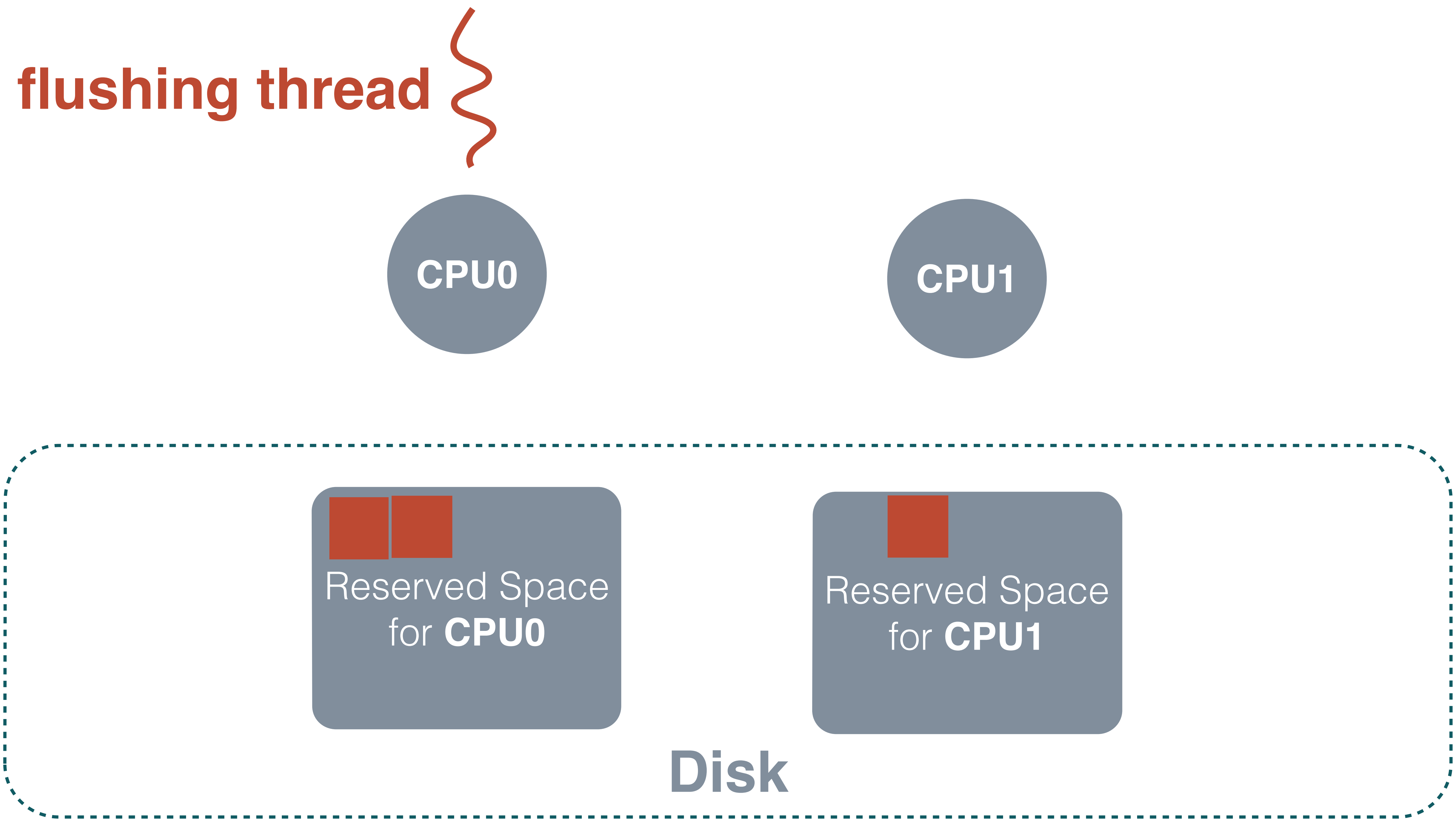
Data layouts of small files depend on OS scheduler



Data layouts of small files depend on OS scheduler




flushing thread
CPU0
CPU1
Reserved Space for **CPU0**
Reserved Space for **CPU1**
Disk


flushing thread
CPU0
CPU1
Reserved Space
for CPU0
Reserved Space
for CPU1
Disk

Scheduler Dependency

Fix

Choose locations based on inode number, instead of CPU id

Lesson Learned

Policies should not depend on environmental factors that may change and are outside the control of the file system

Issues found and fixed

Special End Policy just covered

Scheduler Dependency just covered

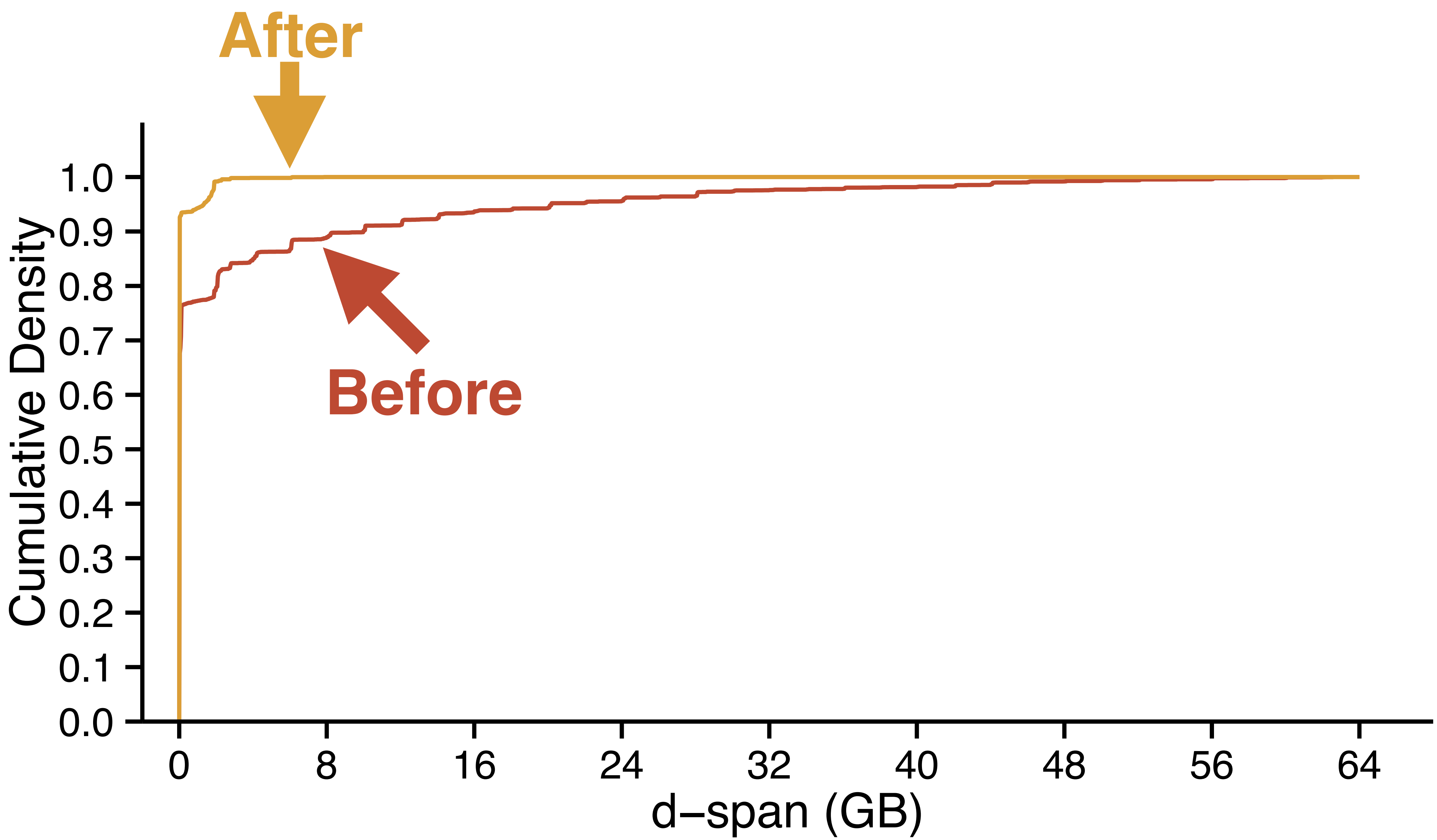
File Size Dependency in paper

Target locations depend on dynamic file size

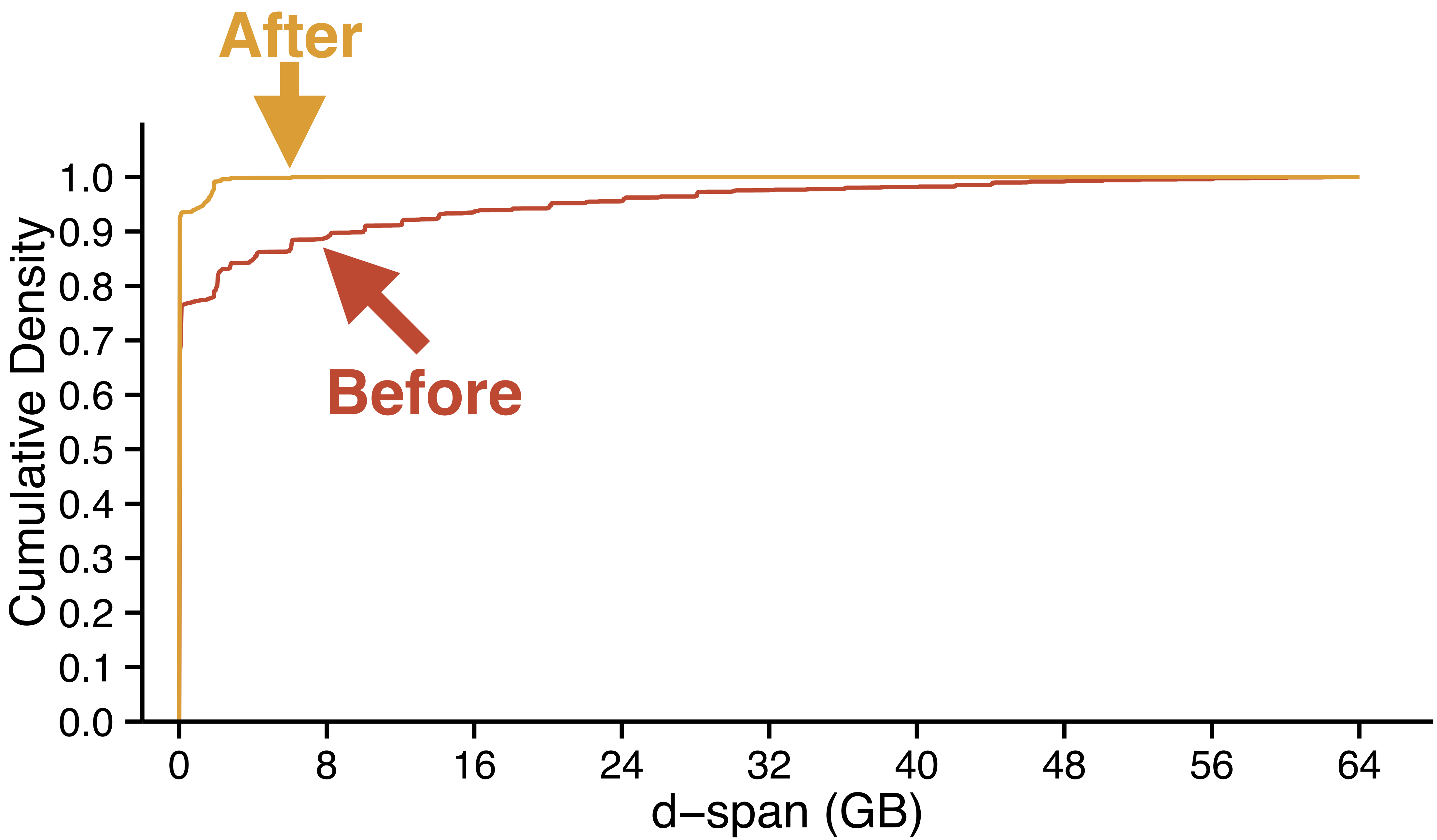
Normalization Bug in paper

Block allocation request are not correctly adjusted

Removing the issues significantly cuts tail size of d-span distribution

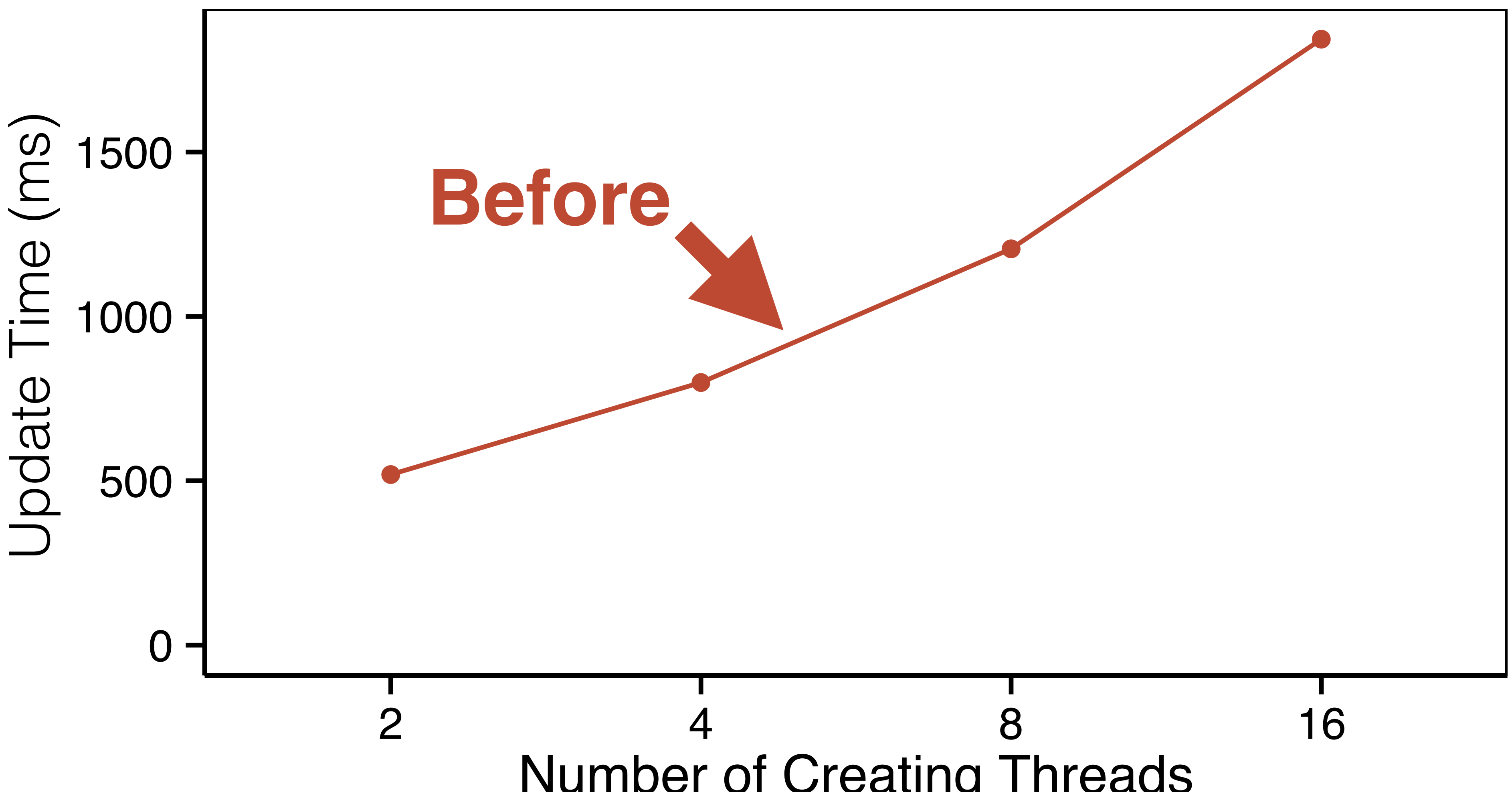


Removing the issues significantly cuts tail size of d-span distribution

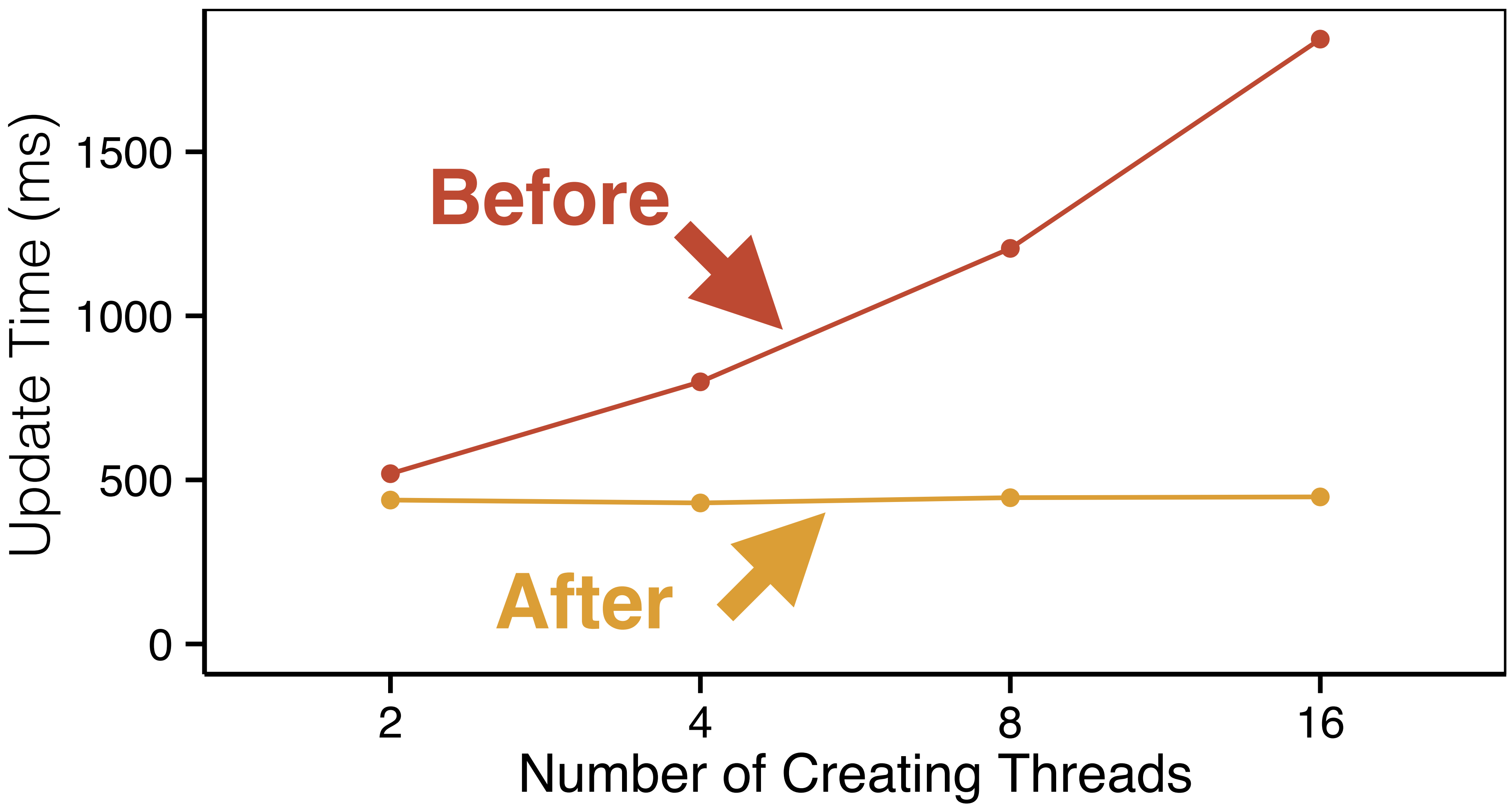


Our fixes improve ext4's data layout

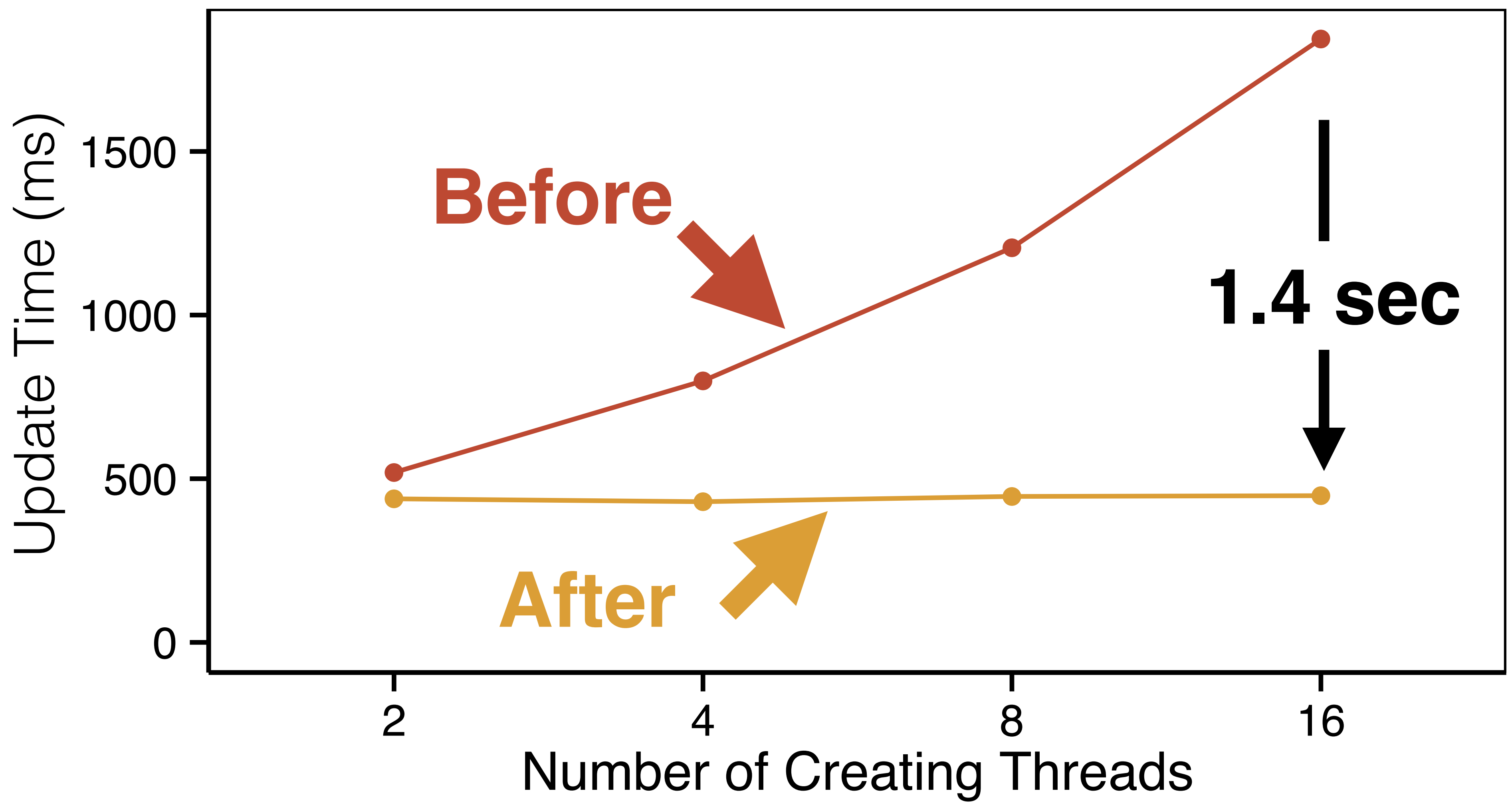
But, do our fixes reduce latencies?



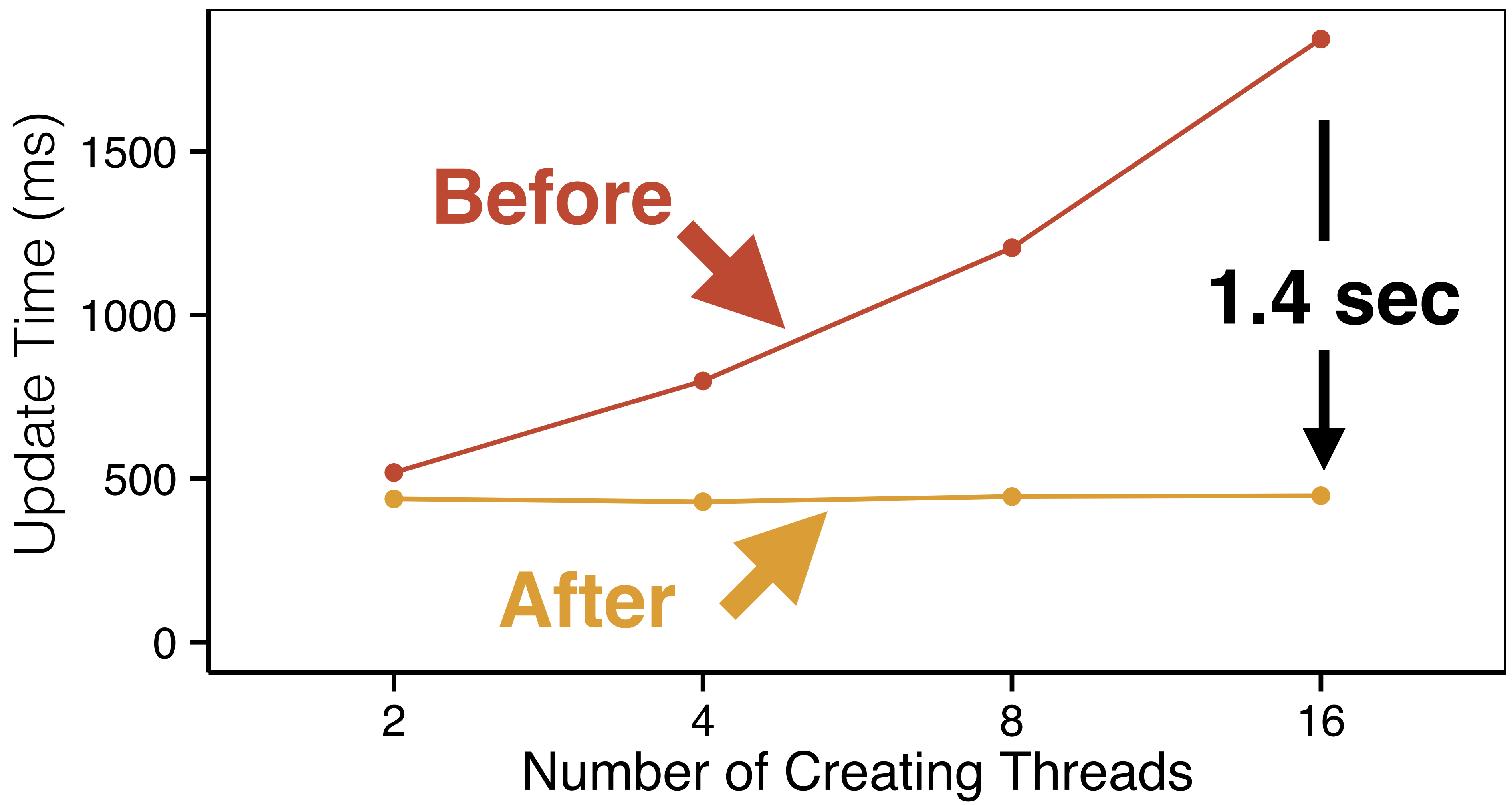
But, do our fixes reduce latencies?



But, do our fixes reduce latencies?



But, do our fixes reduce latencies?



Our fixes reduce tail latencies

Conclusions

- Statistical techniques are practical
- Found and fixed four allocation issues in ext4
- Our fixes ▶ better layouts ▶ lower latency at a node
  ▶ lower latency at scale
- Lessons learned
  - Policies should be harmonious
  - Policies should not depend on environmental factors

Conclusions

- Statistical techniques are practical
- Found and fixed four allocation issues in ext4
- Our fixes ▶ better layouts ▶ lower latency at a node
▶ lower latency at scale
- Lessons learned
  - Policies should be harmonious
  - Policies should not depend on environmental factors

Rigorous statistics will help to reduce <u>unexpected</u> issues caused by <u>intuitive</u> but <u>unreliable</u> design decisions

Source code and data

http://research.cs.wisc.edu/adsl/Software/chopper/

Thanks!