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

Abstract

We present *Chopper*, a tool that efficiently explores the vast input space of file system policies to find behaviors that lead to costly performance problems. We focus specifically on block allocation, as unexpected poor layouts can lead to high tail latencies. Our approach utilizes sophisticated statistical methodologies, based on Latin Hypercube Sampling (LHS) and sensitivity analysis, to explore the search space efficiently and diagnose intricate design problems. We apply *Chopper* to study the overall behavior of two file systems, and to study Linux ext4 in depth. We identify four internal design issues in the block allocator of ext4 which form a large tail in the distribution of layout quality. By removing the underlying problems in the code, we cut the size of the tail by an order of magnitude, producing consistent and satisfactory file layouts that reduce data access latencies.

1 Introduction

As the distributed systems that power the cloud have matured, a new performance focus has come into play: *tail latency*. As Dean and Barroso describe, long tails can dramatically harm interactive performance and thus limit the applications that can be effectively deployed at scale in modern cloud-based services [17]. As a result, a great deal of recent research effort has attacked tail latency directly [6,49,51]; for example, Alizadeh et al. show how to reduce the network latency of the 99th percentile by a factor of ten through a combination of novel techniques [6].

The fundamental reason that reducing such tail latency is challenging is that rare, corner-case behaviors, which have little impact on a single system, can dominate when running a system at scale [17]. Thus, while the well-tested and frequently-exercised portions of a system perform well, the unusual behaviors that are readily ignored on one machine become the common case upon one thousand (or more) machines.

To build the next generation of robust, predictably performing systems, we need an approach that can readily discover corner-case behaviors, thus enabling a developer to find and fix intrinsic tail-latency problems before deployment. Unfortunately, finding unusual behavior is hard: just like exploring an infinite state space for correctness bugs remains an issue for today's model checkers [10, 19], discovering the poorly-performing tail-influencing behaviors presents a significant challenge.

One critical contributor to tail latency is the local file system [8]. Found at the heart of most distributed file systems [20, 47], local file systems such as Linux ext4, XFS, and btrfs serve as the building block for modern scalable storage. Thus, if rare-case performance of the local file system is poor, the performance of the distributed file system built on top of it will suffer.

In this paper, we present *Chopper*, a tool that enables developers to discover (and subsequently repair) high-latency operations within local file systems. *Chopper* currently focuses on a critical contributor to unusual behavior in modern systems: block allocation, which can reduce file system performance by one or more orders of magnitude on both hard disk and solid state drives [1, 11, 13, 30, 36]. With *Chopper*, we show how to find such poor behaviors, and then how to fix them (usually through simple file-system repairs).

The key and most novel aspect of *Chopper* is its usage of advanced statistical techniques to search and investigate an infinite performance space systematically. Specifically, we use Latin hypercube sampling [29] and sensitivity analysis [40], which has been proven efficient in the investigation of many-factor systems in other applications [24,31,39]. We show how to apply such advanced techniques to the domain of file-system performance analysis, and in doing so make finding tail behavior tractable.

We use *Chopper* to analyze the allocation performance of Linux ext4 and XFS, and then delve into a detailed analysis of ext4 as its behavior is more complex and varied. We find four subtle flaws in ext4, including behaviors that spread sequentially-written files over the entire disk volume, greatly increasing fragmentation and inducing large latency when the data is later accessed. We also show how simple fixes can remedy these problems, resulting in an order-of-magnitude improvement in the tail layout quality of the block allocator. *Chopper* and the ext4 patches are publicly available at:

`research.cs.wisc.edu/adsl/Software/chopper`

The rest of the paper is organized as follows. Section 2 introduces the experimental methodology and implementation of *Chopper*. In Section 3, we evaluate ext4 and XFS as black boxes and then go further to explore ext4 as a white box since ext4 has a much larger tail than XFS. We present detailed analysis and fixes for internal allocator design issues of ext4. Section 4 introduces related work. Section 5 concludes this paper.

2 Diagnosis Methodology

We now describe our methodology for discovering interesting tail behaviors in file system performance, particularly as related to block allocation. The file system input space is vast, and thus cannot be explored exhaustively; we thus treat each file system experiment as a simulation, and apply a sophisticated sampling technique to ensure that the large input space is explored carefully.

In this section, we first describe our general experimental approach, the inputs we use, and the output metric of choice. We conclude by presenting our implementation.

2.1 Experimental Framework

The Monte Carlo method is a process of exploring simulation by obtaining numeric results through repeated random sampling of inputs [38,40,43]. Here, we treat the file system itself as a simulator, thus placing it into the Monte Carlo framework. Each run of the file system, given a set of inputs, produces a single output, and we use this framework to explore the file system as a black box.

Each input factor X_i $(i = 1, 2, ..., K)$ (described further in Section 2.2) is estimated to follow a distribution. For example, if small files are of particular interest, one can utilize a distribution that skews toward small file sizes. In the experiments of this paper, we use a uniform distribution for fair searching. For each factor X_i, we draw a sample from its distribution and get a vector of values $(X_i^1, X_i^2, X_i^3, .., X_i^N)$. Collecting samples of all the factors, we obtain a matrix M.

$$M = \begin{bmatrix} X_1^1 & X_2^1 & ... & X_K^1 \\ X_1^2 & X_2^2 & ... & X_K^2 \\ ... \\ X_1^N & X_2^N & ... & X_K^N \end{bmatrix} \qquad Y = \begin{bmatrix} Y^1 \\ Y^2 \\ ... \\ Y^N \end{bmatrix}$$

Each row in M, i.e., a *treatment*, is a vector to be used as input of one run, which produces one row in vector Y. In our experiment, M consists of columns such as the size of the file system and how much of it is currently in use. Y is a vector of the output metric; as described below, we use a metric that captures how much a file is spread out over the disk called *d-span*. M and Y are used for exploratory data analysis.

The framework described above allows us to explore file systems over different combinations of values for uncertain inputs. This is valuable for file system studies where the access patterns are uncertain. With the framework, block allocator designers can explore the consequences of design decisions and users can examine the allocator for their workload.

In the experiment framework, M is a set of treatments we would like to test, which is called an *experimental plan* (or *experimental design*). With a large input space, it is essential to pick input values of each factor and organize them in a way to efficiently explore the space in a limited number of runs. For example, even with our refined space in Table 1 (introduced in detail later), there are about 8×10^9 combinations to explore. With an overly optimistic speed of one treatment per second, it still would take 250 compute-years to finish just one such exploration.

Latin Hypercube Sampling (LHS) is a sampling method that efficiently explores many-factor systems with a large input space and helps discover surprising behaviors [25, 29, 40]. A *Latin hypercube* is a generalization of a *Latin square*, which is a square grid with only one sample point in each row and each column, to an arbitrary number of dimensions [12]. LHS is very effective in examining the influence of each factor when the number of runs in the experiment is much larger than the number of factors. It aids visual analysis as it exercises the system over the entire range of each input factor and ensures all levels of it are explored evenly [38]. LHS can effectively discover which factors and which combinations of factors have a large influence on the response. A poor sampling method, such as a completely random one, could have input points clustered in the input space, leaving large unexplored gaps inbetween [38]. Our experimental plan, based on LHS, contains 16384 runs, large enough to discover subtle behaviors but not so large as to require an impractical amount of time.

2.2 Factors to Explore

File systems are complex. It is virtually impossible to study all possible factors influencing performance. For example, the various file system formatting and mounting options alone yield a large number of combinations. In addition, the run-time environment is complex; for example, file system data is often buffered in OS page caches in memory, and differences in memory size can dramatically change file system behavior.

In this study, we choose to focus on a subset of factors that we believe are most relevant to allocation behavior. As we will see, these factors are broad enough to discover interesting performance oddities; they are also not so broad as to make a thorough exploration intractable.

There are three categories of input factors in *Chopper*. The first category of factors describes the initial state of the file system. The second category includes a relevant OS state. The third category includes factors describing the workload itself. All factors are picked to reveal potentially interesting design issues. In the rest of this paper, a value picked for a factor is called a *level*. A set of levels, each of which is selected for a factor, is called a *treatment*. One execution of a treatment is called a *run*. We picked twelve factors, which are summarized in Table 1 and introduced as follows.

We create a virtual disk of **DiskSize** bytes, because block allocators may have different space management policies for disks of different sizes.

The **UsedRatio** factor describes the ratio of disk that

	Factor	Description	Presented Space
FS	DiskSize	Size of disk the file system is mounted on.	1,2,4,...,64GB
	UsedRatio	Ratio of used disk.	0, 0.2, 0.4, 0.6
	FreeSpaceLayout	Small number indicates high fragmentation.	1,2,...,6
OS	CPUCount	Number of CPUs available.	1,2
Workload	FileSize	Size of file.	8,16,24,...,256KB
	ChunkCount	Number of chunks each file is evenly divided into.	4
	InternalDensity	Degree of sparseness or overwriting.	0.2,0.4,...,2.0
	ChunkOrder	Order of writing the chunks.	permutation(0,1,2,3)
	Fsync	Pattern of `fsync()`.	****, *=0 or 1
	Sync	Pattern of `close()`, `sync()`, and `open()`.	***1, *=0 or 1
	FileCount	Number of files to be written.	1,2
	DirectorySpan	Distance of files in the directory tree.	1,2,3,...,12

Table 1: Factors in Experiment.

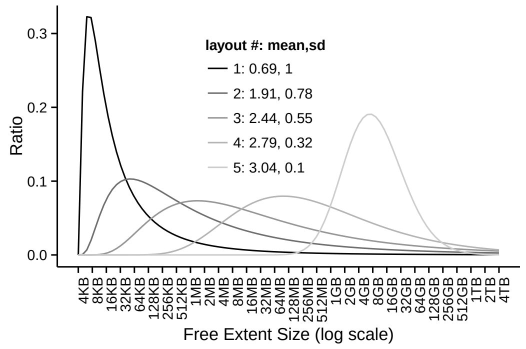


Figure 1: LayoutNumber. Degree of fragmentation represented as lognormal distribution.

has been used. *Chopper* includes it because block allocators may allocate blocks differently when the availability of free space is different.

The **FreeSpaceLayout** factor describes the contiguity of free space on disk. Obtaining satisfactory layouts despite a paucity of free space, which often arises when file systems are aged, is an important task for block allocators. Because enumerating all fragmentation states is impossible, we use six numbers to represent degrees from extremely fragmented to generally contiguous. We use the distribution of free extent sizes to describe the degree of fragmentations; the extent sizes follow lognormal distributions. Distributions of layout 1 to 5 are shown in Figure 1. For example, if layout is number 2, about $0.1 \times DiskSize \times (1 - UsedRatio)$ bytes will consist of 32KB extents, which are placed randomly in the free space. Layout 6 is not manually fragmented, in order to have the most contiguous free extents possible.

The **CPUCount** factor controls the number of CPUs the OS runs on. It can be used to discover scalability issues of block allocators.

The **FileSize** factor represents the size of the file to be written, as allocators may behave differently when different sized files are allocated. For simplicity, if there is more than one file in a treatment, all of them have the same size.

A chunk is the data written by a `write()` call. A file is often not written by only one call, but a series of writes. Thus, it is interesting to see how block allocators act with different numbers of chunks, which **ChunkCount** factor captures. In our experiments, a file is divided into multiple chunks of equal sizes. They are named by their positions in file, e.g., if there are four chunks, chunk-0 is at the head of the file and chunk-3 is at the end.

Sparse files, such as virtual machine images [26], are commonly-used and important. Files written non-sequentially are sparse at some point in their life, although the final state is not. On the other hand, overwriting is also common and can have effect if any copy-on-write strategy is adopted [34]. The **InternalDensity** factor describes the degree of coverage (e.g. sparseness or overwriting) of a file. For example, if InternalDensity is 0.2 and chunk size is 10KB, only the 2KB at the end of each chunk will be written. If InternalDensity is 1.2, there will be two writes for each chunk; the first write of this chunk will be 10KB and the second one will be 2KB at the end of the chunk.

The **ChunkOrder** factor defines the order in which the chunks are written. It explores sequential and random write patterns, but with more control. For example, if a file has four chunks, ChunkOrder=0123 specifies that the file is written from the beginning to the end; ChunkOrder=3210 specifies that the file is written backwards.

The **Fsync** factor is defined as a bitmap describing whether *Chopper* performs an `fsync()` call after each chunk is written. Applications, such as databases, often use `fsync()` to force data durability immediately [15, 23]. This factor explores how `fsync()` may interplay with allocator features (e.g., delayed allocation in Linux ext4 [28]). In the experiment, if ChunkOrder=1230 and Fsync=1100, *Chopper* will perform an `fsync()` after chunk-1 and chunk-2 are written, but not otherwise.

The **Sync** factor defines how we open, close, or sync the file system with each write. For example, if ChunkOrder=1230 and Sync=0011, *Chopper* will perform the three calls after chunk-3 and perform `close()` and `sync()` after chunk-0; `open()` is not called after the last chunk is written. All Sync bitmaps end with 1, in order to place data on disk before we inquire about layout information. *Chopper* performs `fsync()` before `sync()` if they both are requested for a chunk.

The **FileCount** factor describes the number of files written, which is used to explore how block allocators preserve spatial locality for one file and for multiple files. In the experiment, if there is more than one file, the chunks of each file will be written in an interleaved fashion. The ChunkOrder, Fsync, and Sync for all the files in a single treatment are identical.

Chopper places files in different nodes of a directory tree to study how parent directories can affect the data layouts. The **DirectorySpan** factor describes the distance between parent directories of the first and last files in a breadth-first traversal of the tree. If FileCount=1, DirectorySpan is the index of the parent directory in the breadth-first traversal sequence. If FileCount=2, the first file will be placed in the first directory, and the second one will be at the *DirectorySpan*-th position of the traversal sequence.

In summary, the input space of the experiments presented in this paper is described in Table 1. The choice is based on efficiency and simplicity. For example, we study relatively small file sizes because past studies of file systems indicates most files are relatively small [5, 9, 35]. Specifically, Agrawal et. al. found that over 90% of the files are below 256 KB across a wide range of systems [5]. Our results reveal many interesting behaviors, many of which also apply to larger files. In addition, we study relatively small disk sizes as large ones slow down experiments and prevent broad explorations in limited time. The file system problems we found with small disk sizes are also present with large disks.

Simplicity is also critical. For example, we use at most two files in these experiments. Writing to just two files, we have found, can reveal interesting nuances in block allocation. Exploring more files make the results more challenging to interpret. We leave further exploration of the file system input space to future work.

2.3 Layout Diagnosis Response

To diagnose block allocators, which aim to place data compactly to avoid time-consuming seeking on HDDs [7, 36] and garbage collections on SSDs [11, 30], we need an intuitive metric reflecting data layout quality. To this end, we define *d-span*, the distance in bytes between the first and last physical block of a file. In other words, *d-span* measures the worst allocation decision the allocator makes in terms of spreading data. As desired, *d-span* is an *indirect* performance metric, and, more importantly, an intuitive diagnostic signal that helps us find unexpected file-system behaviors. These behaviors may produce poor layouts that eventually induce long data access latencies. *d-span* captures subtle problematic behaviors which would be hidden if end-to-end performance metrics were used. Ideally, *d-span* should be the same size as the file.

d-span is not intended to be an one-size-fits-all metric. Being simple, it has its weaknesses. For example, it cannot distinguish cases that have the same span but different internal layouts. An alternative of *d-span* that we have investigated is to model data blocks as vertices in a graph and use *average path length* [18] as the metric. The minimum distance between two vertices in the graph is their corresponding distance on disk. Although this metric is able to distinguish between various internal layouts, we have found that it is often confusing. In contrast, *d-span* contains less information but is much easier to interpret.

In addition to the metrics above, we have also explored

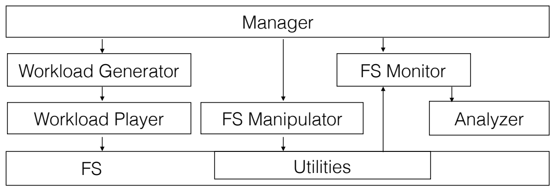


Figure 2: *Chopper* components.

metrics such as number of data extents, layout score (fraction of contiguous blocks) [42], and normalized versions of each metric (e.g. $d\text{-}span/ideal\ d\text{-}span$). One can even create a metric by plugging in a disk model to measure quality. Our diagnostic framework works with all of these metrics, each of which allows us to view the system from a different angle. However, *d-span* has the best trade-off between information gain and simplicity.

2.4 Implementation

The components of *Chopper* are presented in Figure 2. The **Manager** builds an experimental plan and conducts the plan using the other components. The **FS Manipulator** prepares the file system for subsequent workloads. In order to speed up the experiments, the file system is mounted on an in-memory virtual disk, which is implemented as a loop-back device backed by a file in a RAM file system. The initial disk images are re-used whenever needed, thus speeding up experimentation and providing reproducibility. After the image is ready, the **Workload Generator** produces a workload description, which is then fed into the **Workload Player** for running. After playing the workload, the Manager informs the **FS Monitor**, which invokes existing system utilities, such as *debugfs* and *xfs_db*, to collect layout information. No kernel changes are needed. Finally, layout information is merged with workload and system information and fed into the **Analyzer**. The experiment runs can be executed in parallel to significantly reduce time.

3 The Tale of Tail

We use *Chopper* to help understand the policies of file system block allocators, to achieve more predictable and consistent data layouts, and to reduce the chances of performance fluctuations. In this paper, we focus on Linux ext4 [28] and XFS [41], which are among the most popular local file systems [2–4, 33].

For each file system, we begin in Section 3.1 by asking whether or not it provides robust file layout in the presence of uncertain workloads. If the file system is robust (i.e., XFS), then we claim success; however, if it is not (i.e., ext4), then we delve further into understanding the workload and environment factors that cause the unpredictable layouts. Once we understand the combination of factors that are problematic, in Section 3.2, we search for the responsible policies in the file system source code and improve those policies.

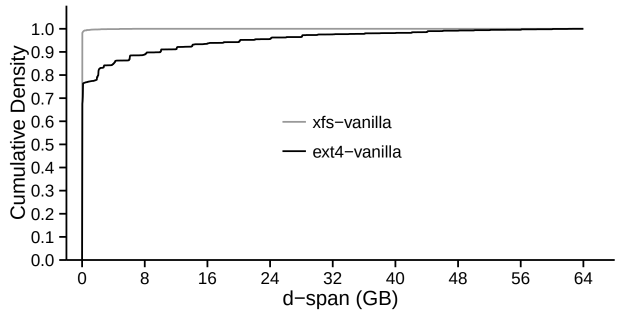


Figure 3: *d-span* CDFs of ext4 and XFS. The 90th%, 95th%, and max *d-span*s of ext4 are 10GB, 20GB, and 63GB, respectively. The 90th%, 95th%, and max *d-span*s of XFS are 2MB, 4MB, and 6GB, respectively.

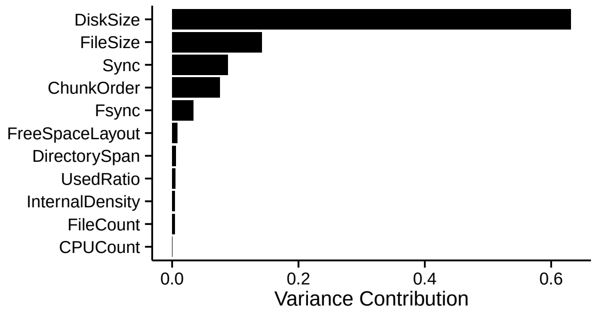


Figure 4: Contribution to *d-span* variance. It shows contributions calculated by factor prioritization of sensitivity analysis.

3.1 File System as a Black Box

3.1.1 Does a Tail Exist?

The first question we ask is whether or not the file allocation policies in Linux ext4 and XFS are robust to the input space introduced in Table 1.

To find out if there are tails in the resulting allocations, we conducted experiments with 16384 runs using *Chopper*. The experiments were conducted on a cluster of nodes with 16 GB RAM and two Opteron-242 CPUs [21]. The nodes ran Linux v3.12.5. Exploiting *Chopper*'s parallelism and optimizations, one full experiment on each file system took about 30 minutes with 32 nodes.

Figure 3 presents the empirical CDF of the resulting *d-span*s for each file system over all the runs; in runs with multiple files, the reported *d-span* is the maximum *d-span* of the allocated files. A large *d-span* value indicates a file with poor locality. Note that the file sizes are never larger than 256KB, so *d-span* with optimal allocation would be only 256KB as well.

The figure shows that the CDF line for XFS is nearly vertical; thus, XFS allocates files with relatively little variation in the *d-span* metric, even with widely differing workloads and environmental factors. While XFS may not be ideal, this CDF (as well as further experiments not shown due to space constraints) indicates that its block allocation policy is relatively robust.

In contrast, the CDF for ext4 has a significant tail. Specifically, 10% of the runs in ext4 have at least one file spreading over 10GB. This tail indicates instability in the ext4 block allocation policy that could produce poor layouts inducing long access latencies.

3.1.2 Which factors contribute to the tail?

We next investigate which workload and environment factors contribute most to the variation seen in ext4 layout. Understanding these factors is important for two reasons. First, it can help file system users to see which workloads run best on a given file system and to avoid those which do not run well; second, it can help file system developers track down the source of internal policy problems.

The contribution of a factor to variation can be calculated by variance-based *factor prioritization*, a technique in sensitivity analysis [38]. Specifically, the contribution of factor X_i is calculated by:

$$S_i = \frac{V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))}{V(Y)}$$

S_i is always smaller than 1 and reports the ratio of the contribution by factor X_i to the overall variation. In more detail, if factor X_i is fixed at a particular level x_i^*, then $E_{X_{\sim i}}(Y|X_i = x_i^*)$ is the resulting mean of response values for that level, $V_{X_i}(E_{X_{\sim i}}(Y|X_i = x_i^*))$ is the variance among level means of X_i, and $V(Y)$ is the variance of all response values for an experiment. Intuitively, S_i indicates how much changing a factor can affect the response.

Figure 4 presents the contribution of each factor for ext4; again, the metric indicates the contribution of each factor to the variation of *d-span* in the experiment. The figure shows that the most significant factors are DiskSize, FileSize, Sync, ChunkOrder, and Fsync; that is, changing any one of those factors may significantly affect *d-span* and layout quality. DiskSize is the most sensitive factor, indicating that ext4 does not have stable layout quality with different disk sizes. It is not surprising that FileSize affects *d-span* considering that the definition *d-span* depends on the size of the file; however, the variance contributed by FileSize ($0.14 \times V(dspan_{real}) = 3 \times 10^{18}$) is much larger than ideally expected ($V(dspan_{ideal}) = 6 \times 10^{10}, dspan_{ideal} = FileSize$). The significance of Sync, ChunkOrder, and Fsync imply that certain write patterns are much worse than others for ext4 allocator.

Factor prioritization gives us an overview of the importance of each factor and guides further exploration. We would also like to know which factors and which levels of a factor are most responsible for the tail. This can be determined with *factor mapping* [38]; factor mapping uses a threshold value to group responses (i.e., *d-span* values) into tail and non-tail categories and finds the input space of factors that drive the system into each category. We define the threshold value as the 90th% (10GB in this case)

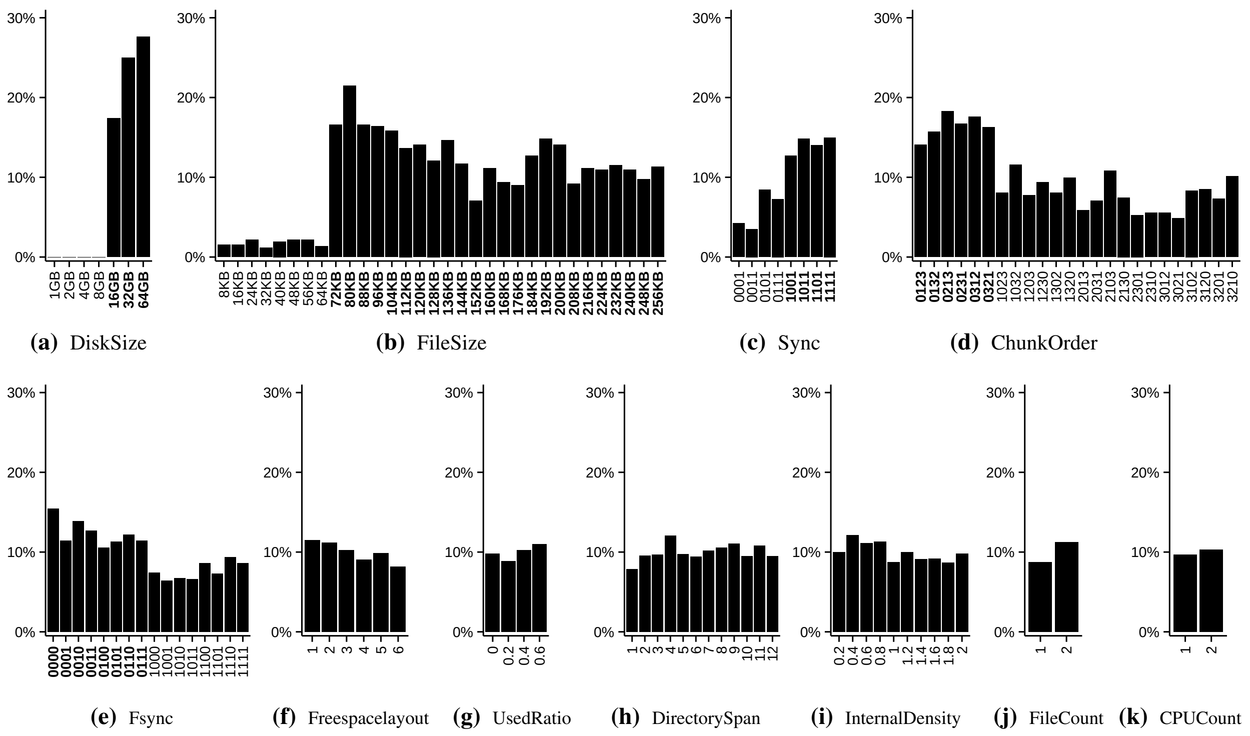


Figure 5: Tail Distribution of 11 Factors. In the figure, we can find what levels of each factor have tail runs and percentage of tail runs in each level. Regions with significantly more tail runs are marked bold. Note that the number of total runs of each level is identical for each factor. Therefore, the percentages between levels of a factor are comparable. For example, (a) shows all tail runs in the experiment have disk sizes $\geq$ 16GB. In addition, when DiskSize=16GB, 17% of runs are in the tail (*d-span*$\geq$10GB) which is less than DiskSize=32GB.

of all *d-span*s in the experiment. We say that a run is a *tail run* if its response is in the tail category.

Factor mapping visualization in Figure 5 shows how the tails are distributed to the levels of each factor. Thanks to the balanced Latin hypercube design with large sample size, the difference between any two levels of a factor is likely to be attributed to the level change of this factor and not due to chance.

Figure 5a shows that all tail runs lay on disk sizes over 8GB because the threshold *d-span* (10GB) is only possible when the disk size exceeds that size. This result implies that blocks are spread farther as the capacity of the disk increases, possibly due to poor allocation polices in ext4. Figure 5b shows a surprising result: there are significantly more tail runs when the file size is larger than 64KB. This reveals that ext4 uses very different block allocation polices for files below and above 64KB.

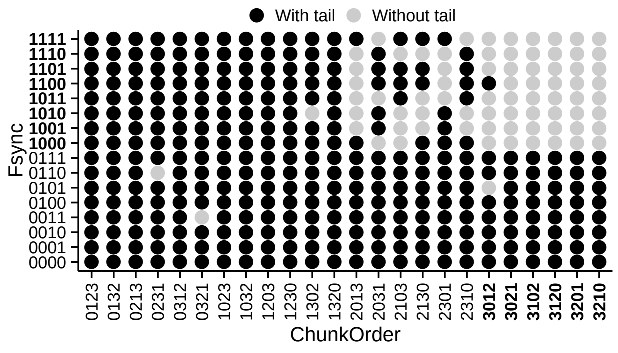
Sync, ChunkOrder, and Fsync also present interesting behaviors, in which the first written chunk plays an important role in deciding the tail. Figure 5c shows that closing and sync-ing after the first written chunk (coded 1***) causes more tail runs than otherwise. Figure 5d shows that writing chunk-0 of a file first (coded 0***), including sequential writes (coded 0123) which are usually preferred, leads to more tail runs. Figure 5e shows that, on average, not fsync-ing the first written chunk (coded 0***) leads to more tail runs than otherwise.

The rest of the factors are less significant, but still reveal interesting observations. Figure 5f and Figure 5g show that tail runs are always present and not strongly correlated with free space layout or the amount of free space, even given the small file sizes in our workloads (below 256KB). Even with layout number 6 (not manually fragmented), there are still many tail runs. Similarly, having more free spaces does not reduce tail cases. These facts indicate that many tail runs do not depend on the disk state and instead it is the ext4 block allocation policy itself causing these tail runs. After we fix the ext4 allocation polices in the next section, the DiskUsed and FreespaceLayout factors will have a much stronger impact.

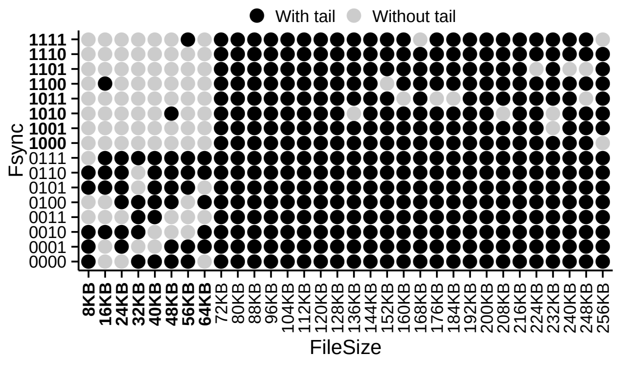
Finally, Figure 5h and Figure 5i show that tail runs are generally not affected by DirectorySpan and InternalDensity. Figure 5j shows that having more files leads to 29% more tail cases, indicating potential layout problems in production systems where multi-file operations are common. Figure 5k shows that there are 6% more tail cases when there are two CPUs.

3.1.3 Which factors interact in the tail?

In a complex system such as ext4 block allocator, performance may depend on more than one factor. We have inspected all two-factor interactions and select two cases in Figure 6 that present clear patterns. The figures show how pairwise interactions may lead to tail runs, reveal-



(a) ChunkOrder and Fsync.



(b) FileSize and Fsync.

Figure 6: Tail Runs in the Interactions of Factors Note that each interaction data point corresponds to multiple runs with other factors varying. A black dot means that there is at least one tail case in that interaction. Low-danger zones are marked with bold labels.

ing both dangerous and low-danger zones in the workload space; these zones give us hints about the causes of the tail, which will be investigated in Section 3.2. Figure 6a shows that, writing and fsync-ing chunk-3 first significantly reduces tail cases. In Figure 6b, we see that, for files not larger than 64KB, fsync-ing the first written chunk significantly reduces the possibility of producing tail runs. These two figures do not conflict with each other; in fact, they indicate a low-danger zone in a three-dimension space.

Evaluating ext4 as black box, we have shown that ext4 does not consistently provide good layouts given diverse inputs. Our results show that unstable performance with ext4 is not due to the external state of the disk (e.g., fragmentation or utilization), but to the internal policies of ext4. To understand and fix the problems with ext4 allocation, we use detailed results from *Chopper* to guide our search through ext4 documentation and source code.

3.2 File System as a White Box

Our previous analysis uncovered a number of problems with the layout policies of ext4, but it did not pinpoint the location of those policies within the ext4 source code. We now use the hints provided by our previous data analysis to narrow down the sources of problems and to perform detailed source code tracing given the set of workloads suggested by *Chopper*. In this manner, we are able to fix a series of problems in the ext4 layout policies and show that each fix reduces the tail cases in ext4 layout.

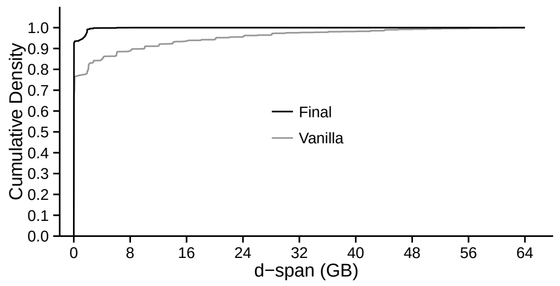
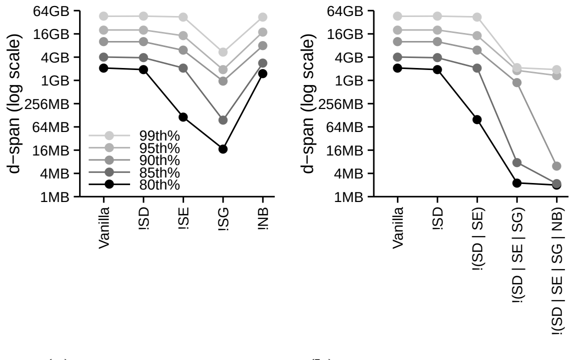


Figure 7: *d-span* CDF of vanilla and final versions of ext4. The final version reduces the 80th, 90th, and 99th percentiles by 1.0×10^3, 1.6×10^3, and 24 times, respectively.



(a) Effect of Single Fix **(b)** Cumulative Effect of Fixes

Figure 8: Effect of fixing issues. Vanilla: Linux v3.12.5. "!" means "without". *SD*: Scheduler Dependency; *SE*: Special End; *SG*: Shared Goal; *NB*: Normalization Bug. !(X | Y) means X and Y are both removed in this version.

Figure 7 compares the original version of ext4 and our final version that has four sources of layout variation removed. We can see that the fixes significantly reduce the size of the tail, providing better and more consistent layout quality. We now connect the symptoms of problems shown by *Chopper* to their root causes in the code.

3.2.1 Randomness → Scheduler Dependency

Our first step is to remove non-determinism for experiments with the same treatment. Our previous experiments corresponded to a single run for each treatment; this approach was acceptable for summarizing from a large sample space, but cannot show intra-treatment variation. After we identify and remove this intra-treatment variation, it will be more straightforward to remove other tail effects.

We conducted two repeated experiments with the same input space as in Table 1 and found that 6% of the runs have different *d-span*s for the same treatment; thus, ext4 can produce different layouts for the same controlled input. Figure 9a shows the distribution of the *d-span* differences for those 6% of runs. The graph indicates that the physical data layout can differ by as much as 46GB for the same workload.

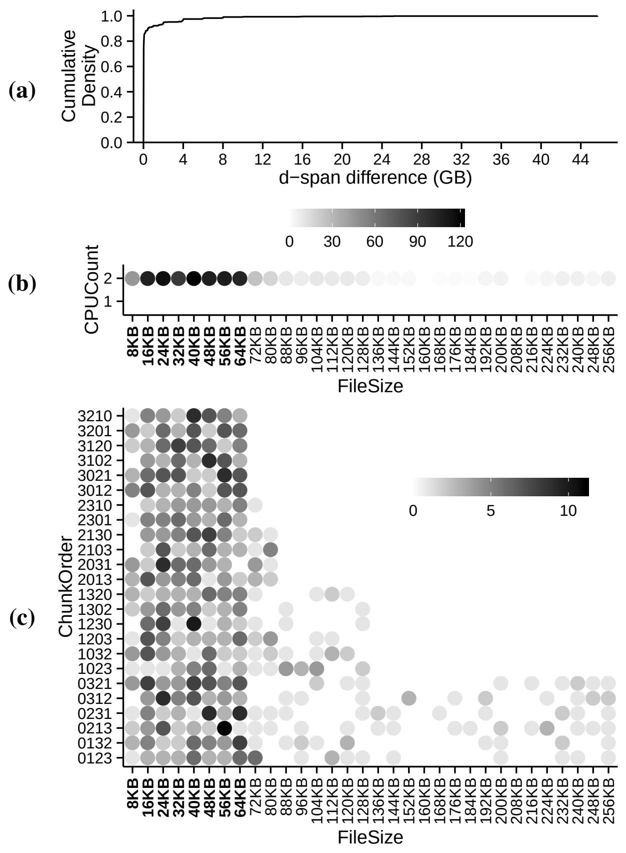


Figure 9: Symptoms of Randomness. (a): CDF of *d-span* variations between two experiments. The median is 1.9MB. The max is 46GB. (b): Number of runs with changed *d-span*, shown as the interaction of FileSize and CPUCount. (c): Number of runs with changed *d-span*, shown as the interaction of FileSize and ChunkOrder. Regions with considerable tail runs are marked with bold labels.

Examining the full set of factors responsible for this variation, we found interesting interactions between FileSize, CPUCount, and ChunkOrder. Figure 9b shows the count of runs in which *d-span* changed between identical treatments as a function of CPUCount and FileSize. This figure gives us the hint that *small* files in multiple-CPU systems may suffer from unpredictable layouts. Figure 9c shows the number of runs with changed *d-span* as a function of ChunkOrder and FileSize. This figure indicates that most small files and those large files written with more sequential patterns are affected.

Root Cause: With these symptoms as hints we focused on the interaction between small files and the CPU scheduler. Linux ext4 has an allocation policy such that files not larger than 64KB (*small* files) are allocated from *locality group (LG) preallocations*; further, the block allocator associates each LG preallocation with a CPU, in order to avoid contention. Thus, for *small* files, the layout location is based solely on which CPU the flusher thread is running. Since the flusher thread can be scheduled on different CPUs, the same small file can use different LG preallocations spread across the entire disk.

This policy is also the cause of the variation seen by some large files written sequentially: large files written sequentially begin as small files and are subject to LG preallocation; large files written backwards have large sizes from the beginning and never trigger this scheduling dependency[1]. In production systems with heavy loads, more cores, and more files, we expect more unexpected poor layouts due to this effect.

Fix: We remove the problem of random layout by choosing the locality group for a *small* file based on its i-number range instead of the CPU. Using the i-number not only removes the dependency on the scheduler, but also ensures that *small* files with close i-numbers are likely to be placed close together. We refer to the ext4 version with this new policy as *!SD*, for no Scheduler Dependency.

Figure 8a compares vanilla ext4 and *!SD*. The graph shows that the new version slightly reduces the size of the tail. Further analysis shows that in total *d-span* is reduced by 1.4 TB in 7% of the runs but is increased by 0.8 TB in 3% of runs. These mixed results occur because this first fix unmasks other problems which can lead to larger *d-span*s. In complex systems such as ext4, performance problems interact in surprising ways; we will progressively work to remove three more problems.

3.2.2 Allocating Last Chunk $\rightarrow$ Special End

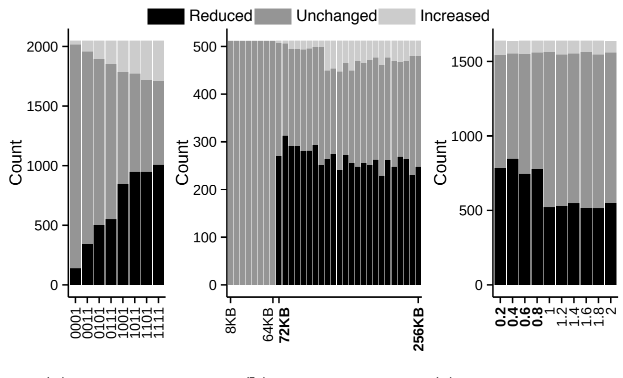
We now return to the interesting behaviors originally shown in Figure 6a, which showed that allocating chunk-3 first (Fsync=1*** and ChunkOrder=3***) helps to avoid tail runs. To determine the cause of poor allocations, we compared traces from selected workloads in which a tail occurs to similar workloads in which tails do not occur.

Root Cause: Linux ext4 uses a *Special End* policy to allocate the last extent of a file when the file is no longer open; specifically, the last extent does not trigger preallocation. The Special End policy is implemented by checking three conditions - *Condition 1*: the extent is at the end of the file; *Condition 2*: the file system is not busy; *Condition 3*: the file is not open. If all conditions are satisfied, this request is marked with the hint "do not preallocate", which is different from other parts of the file[2].

The motivation is that, since the status of a file is final (i.e., no process can change the file until the next open), there is no need to reserve additional space. While this motivation is valid, the implementation causes an inconsistent allocation for the last extent of the file compared to the rest; the consequence is that blocks can be spread

[1]Note that file size in ext4 is calculated by the ending logical block number of the file, not the sum of physical blocks occupied.

[2]In fact, this hint is vague. It means: 1. if there is a preallocation solely for this file (i.e., *i-node preallocation*), use it; 2. do not use LG preallocations, even they are available 3. do not create any new preallocations.



(a) Sync **(b)** FileSize **(c)** InternalDensity

Figure 10: Effects of removing problematic policies. The *d-span*s could be 'Reduced', 'Unchanged' or 'Increased' due to the removal. (a): removing Special End; (b) and (c): removing Shared Global.

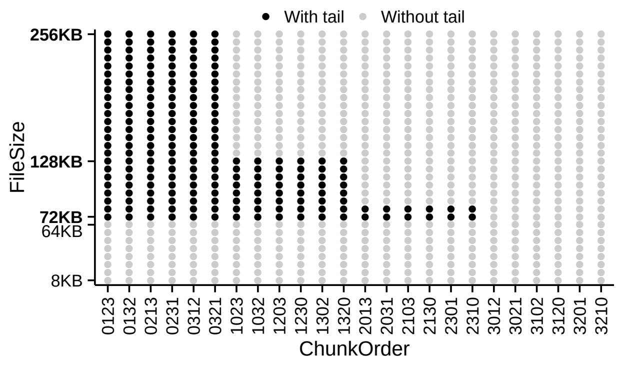


Figure 11: Tail Runs in *!(SD|SE)*. The figure shows tail runs in the interaction of ChunkOrder and FileSize, after removing Scheduler Dependency and Special End.

far apart. For example, a small file may be inadvertently split because non-ending extents are allocated with LG preallocations while the ending extent is not; thus, these conflicting policies drag the extents of the file apart.

This policy explains the tail-free zone (Fsync=1*** and ChunkOrder=3***) in Figure 6a. In these tail-free zones, the three conditions cannot be simultaneously satisfied since fsync-ing chunk-3 causes the last extent to be allocated, while the file is still open; thus, the Special End policy is not triggered.

Fix: To reduce the layout variability, we have removed the Special End policy from ext4; in this version named *!SE*, the ending extent is treated like all other parts of the file. Figure 8 shows that *!SE* reduces the size of the tail. Further analysis of the results show that removing Special End policy reduces *d-span*s for 32% of the runs by a total of 21TB, but increases *d-span*s for 14% of the runs by a total of 9TB. The increasing of *d-span* is primarily because removing this policy unmasks inconsistent policies in File Size Dependency, which we will discuss next.

Figure 10a examines the benefits of the *!SE* policy compared to vanilla ext4 in more detail; to compare only deterministic results, we set CPUCount=1. The graph shows that the *!SE* policy significantly reduces tail runs when the workload begins with sync operations (combination of `close()`, `sync()`, and `open()`); this is because the Special End policy is more likely to be triggered when the file is temporarily closed.

3.2.3 File Size Dependency $\rightarrow$ Shared Global

After removing the Scheduler Dependency and Special End policies, ext4 layout still presents a significant tail. Experimenting with these two fixes, we observe a new symptom that occurs due to the interaction of FileSize and ChunkOrder, as shown in Figure 11. The stair shape of the tail runs across workloads indicates that this policy only affects *large* files and it depends upon the first written chunk.

Root Cause: Traces of several representative data points reveal the source of the 'stair' symptom, which we call *File Size Dependency*. In ext4, one of the design goals is to place small files (less than 64KB, which is tunable) close and big files apart [7]. Blocks for *small* files are allocated from *LG preallocations*, which are shared by all *small* files; blocks in *large* files are allocated from per-file *inode preallocations* (except for the ending extent of a closed file, due to the Special End policy).

This file-size-dependant policy ignores the activeness of files, since the dynamically changing size of a file may trigger inconsistent allocation policies for the same file. In other words, blocks of a file larger than 64KB can be allocated with two distinct policies as the file grows from *small* to *large*. This changing policy explains why FileSize is the most significant workload factor, as seen in Figure 4, and why Figure 5b shows such a dramatic change at 64KB.

Sequential writes are likely to trigger this problem. For example, the first 36KB extent of a 72KB file will be allocated from the LG preallocation; the next 36KB extent will be allocated from a new i-node preallocation (since the file is now classified as *large* with 72KB $>$ 64KB). The allocator will try to allocate the second extent next to the first, but the preferred location is already occupied by the LG preallocation; the next choice is to use the block group where the last big file in the whole file system was allocated (Shared Global policy, coded *SG*), which can be far away. Growing a file often triggers this problem. File Size Dependency is the reason why runs with ChunkOrder=0*** in Figure 5d and Figure 11 have relatively more tail runs than other orders. Writing Chunk-0 first makes the file grow from a *small* size and increases the chance of triggering two distinct policies.

Fix: Placing extents of *large* files together with a shared global policy violates the initial design goal of placing big files apart and deteriorates the consequences of File Size Dependency. To mitigate the problem, we implemented a new policy (coded *!SG*) that tries to place

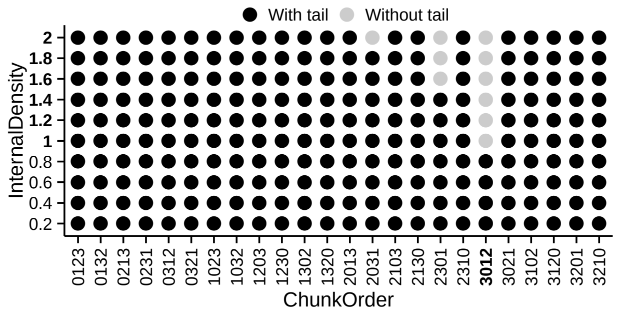


Figure 12: Tail Runs in *!(SD|SE|SG)*. This figure shows tail runs in the interaction of ChunkOrder and InternalDensity on version *!(SD|SE|SG)*.

extents of *large* files close to existing extents of that file. Figure 8a shows that *!SG* significantly reduces the size of the tail. In more detail, *!SG* reduces *d-span* in 35% of the runs by a total of 45TB.

To demonstrate the effectiveness of the *!SG* version, we compare the number of tail cases with it and vanilla ext4 for deterministic scenarios (CPUCount=1). Figure 10b shows that the layout of *large* files (>64KB) is significantly improved with this fix. Figure 10c shows that the layout of sparse files (with InternalDensity < 1) is also improved; the new policy is able to separately allocate each extent while still keeping them near one another.

3.2.4 Sparse Files → Normalization Bug

With three problems fixed in version *!(SD|SE|SG)*, we show an interesting interaction that still remains between ChunkOrder and InternalDensity. Figure 12 shows that while most of the workloads exhibit tails, several workloads do not, specifically, all "solid" (InternalDensity≥1) files with ChunkOrder=3012. To identify the root cause, we focus only on workloads with ChunkOrder=3012 and compare solid and sparse patterns.

Root Cause: Comparing solid and sparse runs with ChunkOrder=3012 shows that the source of the tail is a bug in ext4 normalization; normalization enlarges requests so that the extra space can be used for a similar extent later. The normalization function should update the request's logical starting block number, corresponding physical block number, and size; however, with the bug, the physical block number is not updated and the old value is used later for allocation[3].

Figure 13 illustrates how this bug can lead to poor layout. In this scenario, an ill-normalized request is started (incorrectly) at the original physical block number, but is of a new (correct) larger size; as a result, the request will not fit in the desired gap within this file. Therefore, ext4 may fail to allocate blocks from preferred locations

[3]This bug is present even in the currently latest version of Linux, Linux v3.17-rc6. It has been confirmed by an ext4 developer and is waiting for further tests.

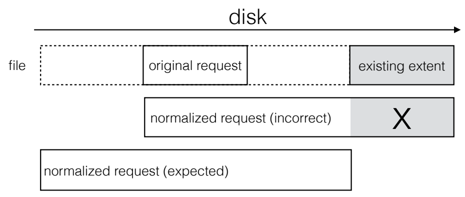


Figure 13: Ill Implementation of Request Normalization. In this case, the normalized request overlaps with the existing extent of the file, making it impossible to fulfill the request at the preferred location.

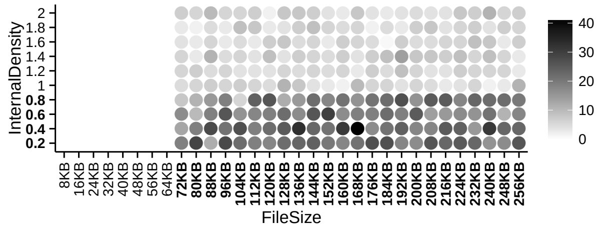


Figure 14: Impact of Normalization Bug. This figure shows the count of runs affected by Normalization Bug in the interaction of FileSize and InternalDensity. The count is obtained by comparing experimental results ran with and without the bug.

and will perform a desperate search for free space elsewhere, spreading blocks. The solid files with ChunkOrder of 3012 in Figure 12 avoid this bug because if chunks-0,1,2 are written sequentially after chunk-3 exists, then the physical block number of the request does not need to be updated.

Fix: We fix the bug by correctly updating the physical starting block of the request in version *!NB*. Figure 14 shows that *large* files were particularly susceptible to this bug, as were sparse files ($InternalDensity < 1$). Figure 8a shows that fixing this bug reduces the tail cases, as desired. In more detail, *!NB* reduces *d-span* for 19% of runs by 8.3 TB in total. Surprisingly, fixing the bug increases *d-span* for 5% of runs by 1.5 TB in total. Trace analysis reveals that, by pure luck, the mis-implemented normalization sometimes sets the request to nearby space which happened to be free, while the correct request fell in space occupied by another file; thus, with the correct request, ext4 sometimes performs a desperate search and chooses a more distant location.

Figure 8 summarizes the benefits of these four fixes. Overall, with all four fixes, the 90th-percentile for *d-span* values is dramatically reduced from well over 4GB to close to 4MB. Thus, as originally shown in Figure 7, our final version of ext4 has a much less significant tail than the original ext4.

3.3 Latencies Reduced

Chopper uses *d-span* as a diagnostic signal to find problematic block allocator designs that produce poor data layouts. The poor layouts, which incur costly disk seeks on HDDs [36], garbage collections on SSDs [11] and even

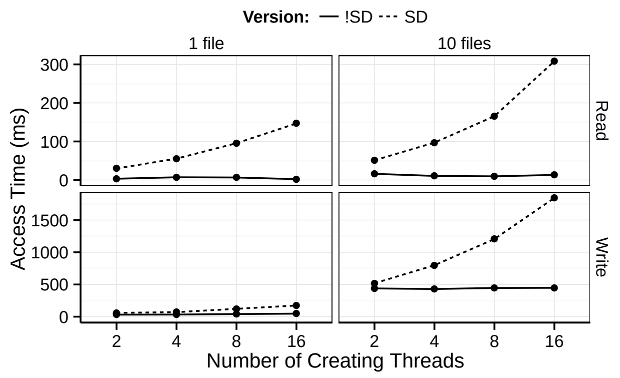


Figure 15: Latency Reduction. This figure shows that *!SD* significantly reduces average data access time comparing with *SD*. All experiments were repeated 5 times. Standard errors are small and thus hidden for clarity.

CPU spikes [1], can in turn result in long data access latencies. Our repairs based on *Chopper*'s findings reduce latencies caused by the problematic designs.

For example, Figure 15 demonstrates how Scheduler Dependency incurs long latencies and how our repaired version, *!SD*, reduces latencies on an HDD (Hitachi HUA723030ALA640: 3.0 TB, 7200 RPM). In the experiment, files were created by multiple *creating threads* residing on different CPUs; each of the threads wrote a part of a 64KB file. We then measured file access time by reading and over-writing with one thread, which avoids resource contentions and maximizes performance. To obtain application-disk data transfer performance, OS and disk cache effects were circumvented. Figure 15 shows that with the *SD* version, access time increases with more creating threads because SD splits each file into more and potentially distant physical data pieces. Our fixed version, *!SD*, reduced read and write time by up to 67 and 4 times proportionally, and by up to 300 and 1400 ms. The reductions in this experiment, as well as expected greater ones with more creating threads and files, are significant – as a comparison, a round trip between US and Europe for a network packet takes 150 ms and a round trip within the same data center takes 0.5 ms [22, 32]. The time increase caused by Scheduler Dependency, as well as other issues, may translate to long latencies in high-level data center operations [17]. *Chopper* is able to find such issues, leading to fixes reducing latencies.

3.4 Discussion

With the help of exploratory data analysis, we have found and removed four issues in ext4 that can lead to unexpected tail latencies; these issues are summarized in Table 2. We have made the patches for these issues publicly available with *Chopper*.

While these fixes do significantly reduce the tail behaviors, they have several potential limitations. First, without the Scheduler Dependency policy, flusher threads running on different CPUs may contend for the same preallocation groups. We believe that the contention degree is acceptable, since allocation within a preallocation is fast and files are distributed across many preallocations; if contention is found to be a problem, more preallocations can be added (the current ext4 creates preallocations lazily, one for each CPU). Second, removing the Shared Global policy mitigates but does not eliminate the layout problem for files with dynamically changing sizes; choosing policies based on dynamic properties such as file size is complicated and requires more fundamental policy revisions. Third, our final version, as shown in Figure 7, still contains a small tail. This tail is due to the disk state (DiskUsed and FreespaceLayout); as expected, when the file system is run on a disk that is more heavily used and is more fragmented, the layout for new files suffers.

Issue	Description
Scheduler Dependency	Choice of preallocation group for small files depends on CPU of flushing thread.
Special End	The last extent of a closed file may be rejected to allocate from preallocated spaces.
File Size Dependency	Preferred target locations depend on file size which may dynamically change.
Normalization Bug	Block allocation requests for large files are not correctly adjusted, causing the allocator to examine mis-aligned locations for free space.

Table 2: Linux ext4 Issues. This table summarizes issues we have found and fixed.

The symptoms of internal design problems revealed by *Chopper* drive us to reason about their causes. In this process, time-consuming tracing is often necessary to pinpoint a particular problematic code line as the code makes complex decisions based on environmental factors. Fortunately, analyzing and visualizing the data sets produced by *Chopper* enabled us to focus on several representative runs. In addition, we can easily reproduce and trace any runs in the controlled environmental provided by *Chopper*, without worrying about confounding noises.

With *Chopper*, we have learned several lessons from our experience with ext4 that may help build file systems that are robust to uncertain workload and environmental factors in the future. First, policies for different circumstances should be harmonious with one another. For example, ext4 tries to optimize allocation for different scenarios and as a result has a different policy for each case (e.g., the ending extent, *small* and *large* files); when multiple policies are triggered for the same file, the policies conflict and the file is dragged apart. Second, policies should not depend on environmental factors that may change and are outside the control of the file system. In contrast, data layout in ext4 depends on the OS scheduler, which makes layout quality unpredictable. By simplifying the layout policies in ext4 to avoid special cases and to be independent of environmental factors, we have shown that file layout is much more compact and predictable.

4 Related Work

Chopper is a comprehensive diagnostic tool that provides techniques to explore file system block allocation designs. It shares similarities and has notable differences with traditional benchmarks and with model checkers.

File system benchmarks have been criticized for decades [44–46]. Many file system benchmarks target many aspects of file system performance and thus include many factors that affect the results in unpredictable ways. In contrast, *Chopper* leverages well-developed statistical techniques [37,38,48] to isolate the impact of various factors and avoid noise. With its sole focus on block allocation, *Chopper* is able to isolate its behavior and reveal problems with data layout quality.

The self-scaling I/O benchmark [14] is similar to *Chopper*, but the self-scaling benchmark searches a five-dimension workload parameter space by dynamically adjusting *one parameter* at a time while keeping the rest constant; its goal is to converge all parameters to values that uniformly achieve a specific percentage of max performance, which is called a *focal point*. This approach was able to find interesting behaviors, but it is limited and has several problems. First, the experiments may never find such a focal point. Second, the approach is not feasible given a large number of parameters. Third, changing one parameter at a time may miss interesting points in the space and interactions between parameters. In contrast, *Chopper* has been designed to systematically extract the maximum amount of information from limited runs.

Model checking is a verification process that explores system state space [16]; it has also been used to diagnose latent performance bugs. For example, MacePC [27] can identify bad performance and pinpoint the causing state. One problem with this approach is that it requires a simulation which may not perfectly match the desired implementation. Implementation-level model checkers, such as FiSC [50], address this problem by checking the actual system. FiSC checks a real Linux kernel in a customized environment to find file system bugs; however, FiSC needs to run the whole OS in the model checker and intercept calls. In contrast, *Chopper* can run in an unmodified, low-overhead environment. In addition, *Chopper* explores the input space differently; model checkers consider transitions between states and often use tree search algorithms, which may have clustered exploration states and leave gaps unexplored. In *Chopper*, we precisely define a large number of factors and ensure the effects and interactions of these factors are evenly explored by statistical experimental design [29,37,38,48].

5 Conclusions

Tail behaviors have high consequences and cause unexpected system fluctuations. Removing tail behaviors will lead to a system with more consistent performance. However, identifying tails and finding their sources are challenging in complex systems because the input space can be infinite and exhaustive search is impossible. To study the tails of block allocation in XFS and ext4, we built *Chopper* to facilitate carefully designed experiments to effectively explore the input space of more than ten factors. We used Latin hypercube design and sensitivity analysis to uncover unexpected behaviors among many of those factors. Analysis with *Chopper* helped us pinpoint and remove four layout issues in ext4; our improvements significantly reduce the problematic behaviors causing tail latencies. We have made *Chopper* and ext4 patches publicly available.

We believe that the application of established statistical methodologies to system analysis can have a tremendous impact on system design and implementation. We encourage developers and researchers alike to make systems amenable to such experimentation, as experiments are essential in the analysis and construction of robust systems. Rigorous statistics will help to reduce unexpected issues caused by intuitive but unreliable design decisions.

Acknowledgments

We thank the anonymous reviewers and Angela Demke Brown (our shepherd) for their excellent feedback. This research was supported by the United States Department of Defense, NSF grants CCF-1016924, CNS-1421033, CNS-1319405, CNS-1218405, CNS-1042537, and CNS-1042543 (PRObE http://www.nmc-probe.org/), as well as generous donations from Amazon, Cisco, EMC, Facebook, Fusion-io, Google, Huawei, IBM, Los Alamos National Laboratory, Microsoft, NetApp, Samsung, Sony, Symantec, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOD, NSF, or other institutions.

References

[1] Btrfs Issues. `https://btrfs.wiki.kernel.org/index.php/Gotchas`.

[2] NASA Archival Storage System. `http://www.nas.nasa.gov/hecc/resources/storage_systems.html`.

[3] Red Hat Enterprise Linux 7 Press Release. `http://www.redhat.com/en/about/press-releases/red-hat-unveils-rhel-7`.

[4] Ubuntu. `http://www.ubuntu.com`.

[5] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the*

5th USENIX Symposium on File and Storage Technologies (FAST '07), San Jose, California, February 2007.

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[7] KV Aneesh Kumar, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, pages 263–274, 2008.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.8 edition, 2014.

[9] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, February 2010.

[11] Luc Bouganim, Björn Thór Jónsson, Philippe Bonnet, et al. uFLIP: Understanding flash IO patterns. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–12, 2009.

[12] Rob Carnell. lhs package manual. `http://cran.r-project.org/web/packages/lhs/lhs.pdf`.

[13] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[14] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, 2013.

[16] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] Nykamp DQ. Mean path length definition. `http://mathinsight.org/network_mean_path_length_definition`.

[19] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[21] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research. *USENIX ;login:*, 38(3), June 2013.

[22] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*, page 20. 2013.

[23] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 71–83. ACM, 2011.

[24] Jon C. Helton and Freddie J. Davis. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety*, 81(1):23–69, 2003.

[25] Ronald L. Iman, Jon C. Helton, and James E. Campbell. An approach to sensitivity analysis of computer models. Part I - Introduction, Input, Variable Selection and Preliminary Variable Assessment. *Journal of Quality Technology*, 13:174–183, 1981.

[26] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, 2009.

[27] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[29] Michael D. McKay, Richard J. Beckman, and William J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[30] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012.

[31] V. N. Nair, D. A. James, W. K. Ehrlich, and J. Zevallos. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica*, 8(1):165–184, 1998.

[32] Peter Norvig. Teach Yourself Programming in Ten Years. `http://norvig.com/21-days.html`.

[33] Ryan Paul. Google upgrading to Ext4. `arstechnica.com/information-technology/2010/01/google-upgrading-to-ext4-hires-former-linux-foundation-cto/`.

[34] Zachary N. J. Peterson. Data Placement for Copy-on-write Using Virtual Contiguity. Master's thesis, U.C. Santa Cruz, 2002.

[35] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[36] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[37] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. Design and analysis of computer experiments. *Statistical science*, pages 409–423, 1989.

[38] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[39] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. *Sensitivity analysis in practice: a guide to assessing scientific models*. John Wiley & Sons, 2004.

[40] Thomas J Santner, Brian J Williams, and William Notz. *The design and analysis of computer experiments*. Springer, 2003.

[41] SGI. XFS Filesystem Structure. `http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf`.

[42] Keith A. Smith and Margo I. Seltzer. File System Aging – Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pages 203–213, New York, NY, USA, 1997. ACM.

[43] IM Soboĺ. Quasi-Monte Carlo methods. *Progress in Nuclear Energy*, 24(1):55–61, 1990.

[44] Diane Tang and Margo Seltzer. Lies, damned lies, and file system benchmarks. *VINO: The 1994 Fall Harvest*, 1994.

[45] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[46] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), May 2008.

[47] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A

Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[48] CF Jeff Wu and Michael S Hamada. *Experiments: planning, analysis, and optimization*, volume 552. John Wiley & Sons, 2011.

[49] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, pages 329–341, 2013.

[50] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[51] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, August 2012.