Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis

Yanpei Chen, Kiran Srinivasan*, Garth Goodson*, Randy Katz
University of California, Berkeley, *NetApp Inc.
{ychen2, randy}@eecs.berkeley.edu, *{skiran, goodson}@netapp.com

ABSTRACT

Enterprise storage systems are facing enormous challenges due to increasing growth and heterogeneity of the data stored. Designing future storage systems requires comprehensive insights that existing trace analysis methods are ill-equipped to supply. In this paper, we seek to provide such insights by using a new methodology that leverages an objective, multi-dimensional statistical technique to extract data access patterns from network storage system traces. We apply our method on two large-scale real-world production network storage system traces to obtain comprehensive access patterns and design insights at user, application, file, and directory levels. We derive simple, easily implementable, threshold-based design optimizations that enable efficient data placement and capacity optimization strategies for servers, consolidation policies for clients, and improved caching performance for both.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*

1. INTRODUCTION

Enterprise storage systems are designed around a set of data access patterns. The storage system can be specialized by designing to a specific data access pattern; e.g., a storage system for streaming video supports different access patterns than a document repository. The better the access pattern is understood, the better the storage system design. Insights into access patterns have been derived from the analysis of existing file system workloads, typically through trace analysis studies [1, 3, 17, 19, 24]. While this is the correct general strategy for improving storage system design, past approaches have critical shortcomings, especially given recent changes in technology trends. In this paper, we present a new design methodology to overcome these shortcomings.

The data stored on enterprise network-attached storage systems is undergoing changes due to a fundamental shift in the underlying technology trends. We have observed three such trends, including:

- *Scale*: Data size grows at an alarming rate [12], due to new types of social, business and scientific applications [20], and the desire to "never delete" data.
- *Heterogeneity*: The mix of data types stored on these storage systems is becoming increasingly complex, each having its own requirements and access patterns [22].
- *Consolidation*: Virtualization has enabled the consolidation of multiple applications and their data onto fewer storage servers [6, 23]. These virtual machines (VMs) also present aggregate data access patterns more complex than those from individual clients.

Better design of future storage systems requires insights into the changing access patterns due to these trends. While trace studies have been used to derive data access patterns, we believe that they have the following shortcomings:

- *Unidimensional:* Although existing methods analyze many access characteristics, they do so one at a time, without revealing cross-characteristic dependencies.
- *Expertise bias:* Past analyses were performed by storage system designers looking for specific patterns based on prior workload expectations. This introduces a bias that needs to be revisited based on the new technology trends.
- *Storage server centric:* Past file system studies focused primarily on storage servers. This creates a critical knowledge gap regarding client behavior.

To overcome these shortcomings, we propose a new design methodology backed by the analysis of storage system traces. We present a *method that simultaneously analyzes multiple characteristics and their cross dependencies.* We use a multi-dimensional, statistical correlation technique, called k-means [2], that is completely agnostic to the characteristics of each access pattern and their dependencies. The K-means algorithm can analyze hundreds of dimensions simultaneously, providing added objectivity to our analysis. To further reduce expertise bias, we involve as many relevant characteristics as possible for each access pattern. In addition, we analyze patterns at different granularities (e.g., at the user session, application, file level) on the storage server as well as the client, thus addressing the need for understanding client patterns. The resulting design insights enable policies for building new storage systems.

Client side observations and design implications	Server side observations and design implications
1. Client sessions with IO sizes >128KB are read only or write only. ⇒ Clients can consolidate sessions based on only the read-write ratio. 2. Client sessions with duration >8 hours do ≈10MB of IO. ⇒ Client caches can already fit an entire day's IO. 3. Number of client sessions drops off linearly by 20% from Monday to Friday. ⇒ Servers can get an extra "day" for background tasks by running at appropriate times during week days. 4. Applications with <4KB of IO per file open and many opens of a few files do only random IO. ⇒ Clients should always cache the first few KB of IO per file per application. 5. Applications with >50% sequential read or write access entire files at a time. ⇒ Clients can request file prefetch (read) or delegation (write) based on only the IO sequentiality. 6. Engineering applications with >50% sequential read and sequential write are doing code compile tasks, based on file extensions. ⇒ Servers can identify compile tasks; server has to cache the output of these tasks.	7. Files with >70% sequential read or write have no repeated reads or overwrites. ⇒ Servers should delegate sequentially accessed files to clients to improve IO performance. 8. Engineering files with repeated reads have random accesses. ⇒ Servers should delegate repeatedly read files to clients; clients need to store them in flash or memory. 9. All files are active (have opens, IO, and metadata access) for only 1-2 hours in a few months. ⇒ Servers can use file idle time to compress or deduplicate to increase storage capacity. 10. All files have either all random access or >70% sequential access. (Seen in past studies too) ⇒ Servers can select the best storage medium for each file based on only access sequentiality. 11. Directories with sequentially accessed files almost always contain randomly accessed files as well. ⇒ Servers can change from per-directory placement policy (default) to per-file policy upon seeing any sequential IO to any files in a directory. 12. Some directories aggregate only files with repeated reads and overwrites. ⇒ Servers can delegate these directories entirely to clients, tradeoffs permitting.

Table 1: Summary of design insights, separated into insights derived from client access patterns and server access patterns.

We analyze two recent, network-attached storage file system traces from a production enterprise datacenter. Table 1 summarizes our key observations and design implications, they will be detailed later in the paper. Our methodology leads to observations that would be difficult to extract using past methods. We illustrate two such access patterns, one showing the value of multi-granular analysis (Observation 1 in Table 1) and another showing the value of multi-feature analysis (Observation 8).

First, we observe (Observation 1) that *sessions with more than 128KB of data reads or writes are either read-only or write-only.* This observation affects shared caching and consolidation policies across sessions. Specifically, client OSs can detect and co-locate cache sensitive sessions (read-only) with cache insensitive sessions (write-only) using just one parameter (read-write ratio). This improves cache utilization and consolidation (increased density of sessions per server).

Similarly, we observe (Observation 8) that *files with >70% sequential read or sequential write have no repeated reads or overwrites.* This access pattern involves four characteristics: read sequentiality, write sequentiality, repeated read behavior, and overwrite behavior. The observation leads to a useful policy: sequentially accessed files do not need to be cached at the server (no repeated reads), which leads to an efficient buffer cache.

These observations illustrate that our methodology can derive unique design implications that leverage the correlation between different characteristics. To summarize, our contributions are:

- Identify storage system access patterns using a multi-dimensional, statistical analysis technique.
- Build a framework for analyzing traces at different granularity levels at both server and client.
- Analyze our specific traces and present the access patterns identified.
- Derive design implications for various storage system components from the access patterns.

In the rest of the paper, we motivate and describe our analysis methodology (Sections 2 and 3), present the access patterns we found and the design insights (Section 4), provide the implications on storage system architecture (Section 5), and suggest future work (Section 6).

2. MOTIVATION AND BACKGROUND

Past trace-based studies have examined a range of storage system protocols and use cases, delivering valuable insights for designing storage servers. Table 2 summarizes the contributions of past studies. Many studies predate current technology trends. Analysis of real-world, corporate workloads or traces have been sparse, with only three studies among the ones listed [13, 15, 18]. A number of studies have focused on NFS trace analysis only [8, 10]. This focus somewhat neglects systems using the Common Internet File System (CIFS) protocol [5], with only a single CIFS study [15]. CIFS systems are important since CIFS is the network storage protocol for Windows, the dominant OS on commodity platforms. Our work uses the same traces as [15], but we perform analysis using a methodology that extracts multi-dimensional insights at different layers. This methodology is sufficiently different from prior work as to make the analysis findings not comparable. The following discusses the need for this methodology.

2.1 Need for Insights at Different Layers

We divide our view of the storage system into behavior at clients and servers. Storage *clients* interface directly with users, who create and view content via applications. Separately, *servers* store the content in a durable and efficient fashion over the network. Past network storage system trace studies focus mostly on storage servers (Table 2). Storage client behavior is underrepresented primarily due to the reliance on stateless NFS traces. This leaves a knowledge gap about access patterns at storage clients. Specifically, these questions are unanswered:

- Do applications exhibit clear access patterns?
- What are the user-level access patterns?
- Any correlation between users and applications?
- Do all applications interact with files the same way?

Study	Date of Traces	File System	N/w FS	Multi-Dim.	Multi-Layer	Data Set	Trace Info	Insights/ Contributions
Ousterhout, *et al.* [17]	1985	BSD				Eng	Live	Seminal patterns analysis: Large, sequential read access; limited read-write; bursty I/O; short file lifetimes, etc.
Ramakrishnan, *et al.* [18]	1988-89	VAX/ VMS	✓			Eng, HPC, Corp	Live	Relationship between files and processes - on usage patterns, sharing, etc.
Baker, *et al.* [3]	1991	Sprite	✓			Eng	Live	Analysis of distributed file system; comparison to [17], caching effects.
Gribble, *et al.* [10]	1991-97	Sprite, NFS, VxFS	✓			Eng, Backup	Live, Snap	Workload self-similarity
Douceur, *et al.* [7]	1998	FAT, FAT32, NTFS				Eng	Snap	Analysis of file and directory attributes: size, age, lifetime, directory depth
Vogels [24]	1998	FAT, NTFS				Eng, HPC	Live, Snap	Supported past observations and trends in NTFS
Zhou *et al.* [25]	1999	VFAT				PC	Live	Analysis of personal computer workloads
Roselli, *et al.* [19]	1997-00	VxFS, NTFS				Eng, Server	Live	Increased block lifetimes, caching strategies
Ellard, *et al.* [8]	2001	NFS	✓			Eng, Email	Live	NFS peculiarities, pathnames can aid file layout
Agrawal, *et al.* [1]	2000-04	FAT, FAT32, NTFS				Eng	Snap	Distribution of file size and type in namespace, change in file contents over time
Leung, *et al.* [15]	2007	CIFS	✓			Corp, Eng	Live	File re-open, sharing, activity characteristics; changes compared to previous studies
Kavalanekar, *et al.* [13]	2007	NTFS				Web, Corp	Live	Study of web (live maps, web content, etc.) workloads in servers via events tracing.
This paper	2007	CIFS	✓	✓	✓	Corp, Eng	Live	Section 4

Table 2: Past studies of storage system traces. "Corp" stands for corporate use cases. "Eng" stands for engineering use cases. "Live" implies live requests or events in traces were studied, "Snap" implies snapshots of file systems were studied.

Insights on these access patterns lead to better design of *both* clients and servers. They enable server capabilities such as per session quality of service (QoS), or per application service level objectives (SLOs). They also inform various consolidation, caching, and prefetching decisions at clients.

Each of these access patterns is visible only at a particular semantic layer within the client: users or applications. We define each such layer as an *access unit*, with the observed behaviors at each access unit being an *access pattern*. The analysis of client side access units represents an improvement on prior work.

On the server side, we extend the previous focus on files. We need to also understand how files are grouped within a directory, as well as cross-file dependencies and directory organization. Thus, we perform multi-layer and cross-layer dependency analysis on the server also. This is another improvement on past work.

2.2 Need for Multi-Dimensional Insights

Each access unit has certain inherent characteristics. Characteristics that can be quantified are *features* of that access unit. For example, for an application, the read size in bytes is a feature; the number of unique files accessed is another. Each feature represents an independent mathematical dimension that describes an access unit. We use the terms dimension, feature, and characteristic interchangeably. The global set of features for an access unit is limitless. Picking a good feature set requires domain knowledge.

Many recent studies analyze access patterns *only one feature at a time*. This represents a key limitation. The resulting insights, although valuable, lead to *uniform policies* around a single design point. For example, study [15] revealed that most bytes are transferred from larger files. Although this is an useful observation, it does not reveal other characteristics of such large files: Do they have repeated reads? Do they have overwrites? Do they have many metadata requests? And so on. Adding these dimensions breaks up the predominant access pattern into smaller, *minority* access patterns, each may require a specific storage policy.

Understanding minority access patterns is increasingly important, because the trend toward data heterogeneity implies that no "common case" will dominate storage system behavior. Minority access patterns become visible only upon analyzing multiple features simultaneously, hence the need for multi-dimensional insights. We also need to select a reasonable number of features. Doing so allows us to fully describe the access patterns and reduce the bias in picking any one feature.

Manually identifying multi-feature dependencies is difficult, and can lead to an untenable analysis. Therefore, we need techniques that analyze a large number of features, scale to a high number of analysis data points, and do not require a priori knowledge of any cross-feature dependencies. Multi-dimensional statistics techniques have solved similar problems in other domains [4, 9, 21]. We can apply similar techniques and combine them with domain specific knowledge of the storage systems being analyzed.

In short, the need for multi-layered and multi-dimensional insights motivates our methodology.

3. METHODOLOGY

In this section, we describe our analysis method in detail. We start with a description of the traces we analyzed, followed by a description of the access units selected for our study. Next, we describe key steps in our analysis process, including selecting the right features for each access unit, using the k-means data clustering algorithm to identify access patterns, and additional information needed to interpret and generalize the results.

3.1 Traces Analyzed

We collected CIFS traces from two large-scale, enterprise-class file servers deployed at our corporate datacenters. One server covers roughly 1000 employees in marketing, sales, finance, and other corporate roles. We call this the *corporate trace*. The other server covers roughly 500 employees in various engineering roles. We call this the *engineering trace*. We described the trace collecting infrastructure in [15].

The corporate trace reflects activities on 3TB of active storage from 09/20/2007 to 11/21/2007. It contains activity from many Windows applications. The engineering trace reflects activities on 19TB of active storage from 08/10/2007 to 11/14/2007. It interleaves activity from both Windows and Linux applications. In both traces, many clients use virtualization technologies. Thus, we believe we have representative traces with regards to the technology trends in scale, heterogeneity, and consolidation. Also, since protocol-independent users, applications, and stored data remain the primary factors affecting storage system behavior, we believe our analysis is relevant beyond CIFS.

3.2 Access Units

As mentioned in Section 2.1, we analyze access patterns at multiple access units at the server and the client. Selecting access units is subjective. We chose access units that form clear semantic design boundaries. On the client side, we analyze two access units:

- *Sessions*: Sessions reflect aggregate behavior of an user. A CIFS session is bounded by matching session connect and logoff requests. CIFS identifies it by a tuple - {client IP address, session ID}.
- *Application instance*: Analysis at this level leads to application specific optimizations in client VMs. CIFS identifies each application instance by the tuple - {client IP address, session ID, and process ID}.

We also analyzed file open-closes, but obtained no useful insights. Hence we omit that access unit from the paper.

We also examined two server side access units:

- *File*: Analyzing file level access patterns facilitates per-file policies and optimization techniques. Each file is uniquely identified by its full path name.
- *Deepest subtree*: This access unit is identified by the directory path immediately containing the file. Analysis at this level enables per-directory policies.

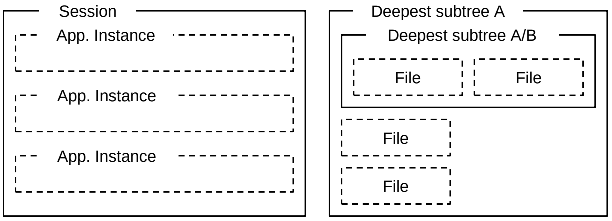

Figure 1: Access units analyzed. At clients, each session contains many application instances. At servers, each subtree contains many files.

Figure 1 shows the semantic hierarchy among different access units. At clients, each session contains many application instances. At servers, each subtree contains many files.

3.3 Analysis Process

Our method (Figure 2) involves the following steps:

1. Collect network storage system traces (Section 3.1).
2. Define the descriptive features for each access unit. This step requires domain knowledge about storage systems (Section 3.3.1).
3. Extract multiple instances of each access unit, and compute from the trace the corresponding numerical feature values of each instance.
4. Input those values into k-means, a multi-dimensional statistical data clustering technique (Section 3.3.2).
5. Interpret the k-means output and derive access patterns by looking at only the relevant subset of features. This step requires knowledge of both storage systems and statistics. We also need to extract considerable additional information to support our interpretations (Section 3.3.3).
6. Translate access patterns to design insights.

We give more details about Steps 2, 4, and 5 below.

3.3.1 Selecting features for each access unit

Selecting the set of descriptive features for each access unit requires domain knowledge about storage systems (Step 2 in Figure 2). It also introduces some subjectivity, since the choice of features limits on how one access pattern can differ from another. The human designer needs to select some basic features initially, e.g., total IO size and read-write ratio for a file. We will not know whether we have a good set of features until we have completed the entire analysis process. If the analysis results leave some design choice ambiguities, we need to add new features to clarify those ambiguities, again using domain knowledge. For example, for the deepest subtrees, we compute various percentiles (25th, 50th, and 75th) of certain features like read-write ratio because the average value for those features did not clearly separate the access patterns. We then repeat the analysis process using the new feature set. This iterative process leads to a long feature set for all access units, somewhat reducing the subjective bias of a small feature set. We list in Section 4 the chosen features for each access unit.

Most of the features used in our analysis (Section 4) are self-explanatory; some ambiguous or complex features require precise definitions, such as:

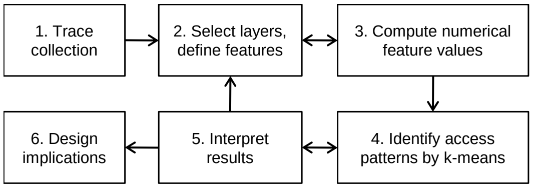
1. Trace collection
2. Select layers, define features
3. Compute numerical feature values
4. Identify access patterns by k-means
5. Interpret results
6. Design implications

Figure 2: Methodology overview. The two-way arrows and the loop from Step 2 through Step 5 indicate our many iterations between the steps.

IO: We use "IO" as a substitute for "read and write".

Sequential reads or writes: We consider two read or writes requests to be sequential if they are consecutive in time, and the file offset + request size of the first request equals the file offset of the second request. A single read or write request is by definition not sequential.

Repeated reads or overwrites: We track accesses at 4KB block boundaries within a file, with the offset of the first block being zero. A read is considered repeated if it accesses a block that has been read in the past half hour. We use an equivalent definition for overwrites.

3.3.2 Identifying access patterns via k-means

A key part of our methodology is the k-means multi-dimensional correlation algorithm. We use it to identify access patterns simultaneously across many features (Step 4 in Figure 2). K-means is a well-known, statistical correlation algorithm. It identifies sets of data points that congregate around a region in n-dimensional space. These congregations are called *clusters*. Given data points in an n-dimensional space, k-means picks k points at random as initial cluster centers, assigns data points to their nearest cluster centers, and recomputes new cluster centers via arithmetic means across points in the cluster. K-means iterates the assignment-recompute process until the cluster centers become stationary. K-means can run with multiple sets of initial cluster centers and return the best result [2].

For each access unit, we extract different instances of it from the trace, i.e., all session instances, application instances, etc. For each instance, we compute the numerical values of all its features. This gives us a data array in which each row correspond to an instance, i.e., a data point, and each column correspond to a feature, i.e., a dimension. We input the array into k-means, and the algorithm finds the natural clusters across all data points. *We consider all data points in a cluster as belonging to a single equivalence class, i.e., a single access pattern.* The numerical values of the cluster centers indicate the characteristics of each access pattern.

We choose k-means for two reasons. First, k-means is algorithmically simple. This allows rapid processing on large data sets. We used a modified version of the k-means C library [14], in which we made some improvements to limits the memory footprint when processing large data sizes. Second, k-means leads to intuitive labels of the cluster centers. This helps us translate the statistical behavior extracted from the traces into tangible insights. Thus, we prefer k-means to other clustering algorithms such as hierarchical clustering and k-means derivatives [2].

K-means requires us to specify k, the number of clusters. This is a difficult task since we do not know a priori the number of "natural" clusters in the data. We compute the intra-cluster "residual" variance from the k-means results - the sum of squared distances from each data point to its assigned cluster center. This is a standard metric for cluster quality, and gives us a lower bound on k. We cannot set k so small that the residual variance forms a large fraction of the total variance, i.e., residual variance ≈ the sum of squared distances from each data point to the global average of all data points. We optionally increase k beyond the lower bound until some key access patterns can be separated. Concurrently, we take care not to increase k too high, to prevent having an unwieldy number of access patterns and design targets. We applied this reasoning to set k at each client and server access unit.

3.3.3 Interpreting and generalizing the results

The k-means algorithm gives us a set of access patterns with various characteristics. We need additional information to understand the significance of the results. This information comes from computing various secondary data outside of k-means analysis (Step 5 in Figure 2:

- We gathered the start and end times of each session instance, aggregated by times of the day and days of the week. This gave us insight into how users launch and end sessions.
- We examine filename extensions of files associated with every access pattern belonging to these access units: application instances, files, and deepest subtrees. This information connects the access patterns to more easily recognizable file extensions.
- We perform correlation analysis between the file and deepest subtrees access units. Specifically, we compute the number of files of each file access pattern that is located within directories in each deepest subtree access pattern. This information captures the organizations of files in directories.

Such information gives us a detailed picture about the semantics of the access patterns, resulting in human understandable labels to the access patterns. Such labels help us translate observations to design implications.

Furthermore, after identifying the design implications, we explore if the design insights can be extrapolated to other trace periods and other storage system use cases. We accomplish this by repeating our exact analysis over multiple subsets of the traces, for example, a week's worth of traces at a time. This allow us to examine how our analysis would be different had we obtained only a week's trace. Access patterns that are consistent, stable across different weeks would indicate that they are likely to be more general than just our tracing period or our use cases.

4. ANALYSIS RESULTS & IMPLICATIONS

This section presents the access patterns we identified and the accompanying design insights. We discuss client and

(a). Descriptive features for each session			
Duration	Total metadata requests	Overwrite ratio	Directories accessed
Total IO size	Avg. time between IO requests	Tree connects	Application instances seen
Read:write ratio by bytes	Read sequentiality	Unique trees accessed	
Total IO requests	Write sequentiality	File opens	
Read:write ratio by requests	Repeated read ratio	Unique files opened	

(b). Corporate session access patterns	Full day work	Half day content viewing	Short content viewing	Short content generation	Supporting metadata	Supporting read-write
% of all sessions	0.5%	0.7%	1.2%	0.2%	96%	1.4%
Duration	8 hrs	4 hrs	10 min	70 min	7 sec	10 sec
Total IO size	11 MB	3 MB	128 KB	3 MB	0	420 B
Read:write ratio by bytes	3:2	1:0	1:0	0:1	0:0	1:1
Metadata requests	3000	700	230	550	1	20
Read sequentiality	70%	80%	0%	-	-	0%
Write sequentiality	80%	-	-	90%	-	0%
File opens:files	200:40	80:15	30:7	50:15	0:0	6:3
Tree connect:Trees	5:2	3:2	2:2	2:2	1:1	2:2
Directories accessed	10	7	4	6	0	2
Application instances	4	3	2	2	0	1

(c). Engineering session access patterns	Full day work	Human edit small files	Application generated backup/copy	Short content generation	Supporting metadata	Machine generated update
% of all sessions	0.4%	1.0%	4.4%	0.4%	90%	3.6%
Duration	1 day	2 hrs	1 min	1 hr	10 sec	10 sec
Total IO size	5 MB	5 KB	2 MB	2 MB	0	36 B
Read:write ratio	7:4	1:1	1:0	0:1	0:0	1:0
Metadata requests	1700	130	40	200	1	0
Read sequentiality	60%	0%	90%	-	-	0%
Write sequentiality	70%	0%	-	90%	-	-
File opens:files	130:20	9:2	6:5	15:6	0:0	1:1
Tree connect:Trees	1:1	1:1	1:1	1:1	1:1	1:1
Directories accessed	7	2	1	3	0	1
Application instances	4	2	1	1	0	1

Table 3: Session access patterns. (a): Full list of descriptive features. (b) and (c): Short names and descriptions of sessions in each access pattern; listing only the features that help separate the access patterns.

serve side access patterns (Section 4.1, 4.2). We also check if these patterns persist across time (Section 4.3).

For each access unit, we list the descriptive features (only some of which help separate access patterns), outline how we derived the high-level name (label) for each access pattern, and discuss relevant design insights.

4.1 Client Side Access Patterns

As mentioned in Section 3.2, we analyze sessions and application instances at clients.

4.1.1 Sessions

Sessions reflect aggregate behavior of human users. We used 17 features to describe sessions (Table 3). The corporate trace has 509,076 sessions, and the engineering trace has 232,033.

In Table 3, we provide quantitative descriptions and short names for all the session access patterns. We derive the names from examining the significant features: duration, read-write ratio, and IO size.

We also looked at the aggregate session start and end times to get additional semantic knowledge about each access pattern. Figure 3 shows the start and end times for selected session access patterns. The start times of corporate full-day work sessions correspond exactly to the U.S. work day – 9am start, 12pm lunch, 5pm end. Corporate content generation sessions show slight increase in the evening and towards Friday, indicating rushes to meet daily or weekly deadlines. In the engineering trace, the application generated backup and machine generated update sessions depart significantly from human workday and work week patterns, leading us to label them as application and machine (client OS) generated.

One surprise was that the 'supporting metadata' sessions account for >90% of all sessions in both traces. We believe these sessions are not humanly generated. They last roughly 10 seconds, leaving little time for human mediated interactions. Also, the session start rate averages to roughly one per employee per minute. We are certain that our colleagues are not connecting and logging off every minute of the entire day. However, the shape of the start time graphs have a strong correlation with the human work day and work week. We call these supporting metadata sessions – machine generated in support of human user activities. These metadata sessions form a sort of "background noise" to the storage system. We observe the same background noise at other layers both at clients and servers.

Observation 1: The sessions with IO sizes greater than 128KB are either read-only or write-only, except for the full-day work sessions. Among these sessions, only read-only sessions utilize buffer cache for repeated reads and prefetches. Write-only sessions only use the cache to buffer writes. Thus, if we have a cache eviction policy that recognizes their write-only nature and releases the buffers immediately on flushing dirty data, we can satisfy many write-only sessions with relatively little buffer cache space. We can attain better consolidation and buffer cache utilization by managing the ratio of co-located read-only and write-only sessions. This insight can be used by virtualization managers and client operating systems to manage a shared buffer cache between sessions. Recognizing such read-only and write-only sessions

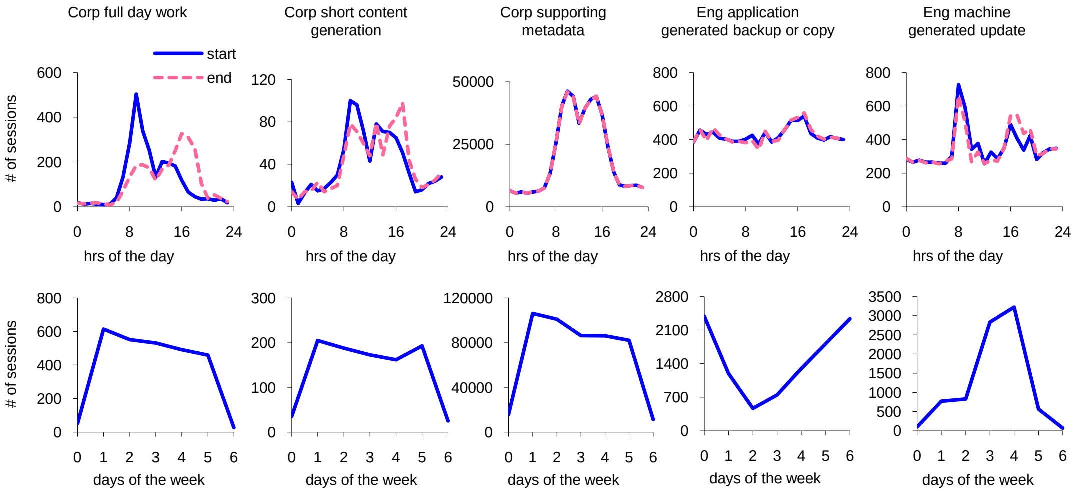

Figure 3: Number of sessions that start or ends at a particular time. Number of session starts and ends in times of the day (top) and session starts in days of the week (bottom). Showing only selected access patterns.

is easy. Examining a session's total read size and write size reveals their read-only or write-only nature. *Implication 1: Clients can consolidate sessions efficiently based only on the read-write ratio.*

Observation 2: The full-day work, content-viewing, and content-generating sessions all do ≈10MB of IO. This means that a client cache of 10s of MB can fit the working set of a day for most sessions. Given the growth of flash devices on clients for caching, despite large-scale consolidation, clients should easily cache a day's worth of data for all users. In such a scenario, most IO requests would be absorbed by the cache, reducing network latency and bandwidth utilization, and load on the server. Moreover, complex cache eviction algorithms are unnecessary. *Implication 2: Clients caches can already fit an entire day's IO.*

Observation 3: The number of human-generated sessions and supporting sessions peaks on Monday and decreases steadily to 80% of the peak on Friday (Figure 3). This is true for all human generated sessions, including the ones not shown in Figure 3. There is considerable "slack" in the server load during evenings, lunch times, and even during working hours. This implies that the server can perform background tasks such as consistency checks, maintenance, or compression/deduplication, at appropriate times during the week. A simple count of active sessions can serve as an effective start and stop signal. By computing the area under the curve for session start times by days of the week, we estimate that background tasks can squeeze out roughly one extra day's worth of processing without altering the peak demand on the system. This is a 50% improvement over a setup which performs background tasks only during weekends. In the engineering trace, the application generated backup or copy sessions seem to have been already designed this way. *Implication 3: Servers get an extra "day" for background tasks by running them at appropriate times during week-days.*

4.1.2 Application instances

Application instance access patterns reflects application behavior, facilitating application specific optimizations. We used 16 features to describe application instances (Table 4). The corporate trace has 138,723 application instances, and the engineering trace has 741,319.

Table 4 provides quantitative descriptions and short names for all the application instance access patterns. We derive the names from examining the read-write ratio, IO size, and file extensions accessed (Figures 4 and 5).

We see again the metadata background noise. The supporting metadata application instances account for the largest fraction, and often do not even open a file.

There are many files without a file extension, a phenomenon also observed in recent storage system snapshot studies [16]. We notice that file extensions turn out to be poor indicators of application instance access patterns. This is not surprising because we separate access patterns based on read/write properties. A user could either view a `.doc` or create a `.doc`. The same application software has different read/write patterns. This speaks to the strength of our multi-layer framework. Aggregating IO by application instances gives clean separation of patterns; while aggregating just by application software or file extensions will not.

We also find it interesting that most file extensions are immediately recognizable. This means that *what* people use network storage systems for, i.e., the file extensions, remains easily recognizable, even though *how* people use network storage systems, i.e., the access patterns, is ever changing and becoming more complex.

Observation 4: The small content viewing application and content update application instances have <4KB total reads per file open and access a few unique files many times. The small read size and multiple reads from the same files means that clients should prefetch and place the files in a cache optimized for random access (flash/SSD/memory). The trend towards flash caches on clients should enable this transfer.

Application instances have bi-modal total IO size - either very small or large. Thus, a simple cache management algorithm suffices; we always keep the first 2 blocks of 4KB in cache. If the application instance does more IO, it is likely to have IO size in the 100KB-1MB range, so we evict it from the cache. We should note that such a policy makes sense even though we proposed earlier to cache all 11MB of a typical day's working set - 11MB of cache becomes a concern when we have many consolidated clients. *Implication 4: Clients should always cache the first few KB of IO per file per application.*

Observation 5: We see >50% sequential read and write ratio for the content update applications instances (corporate) and the content viewing applications instances for human-generated content (both corporate and engineering). Dividing the total IO size by the number of file opens suggest that these application instances are sequentially reading and writing entire files for office productivity (`.xls`, `.doc`, `.ppt`, `.pdf`, etc.) and multimedia applications.

This implies that the files associated with these applications should be prefetched and delegated to the client. Prefetching means delivering the whole file to the client before the whole file is requested. Delegation means giving a client temporary, exclusive access to a file, with the client periodically synchronizing to server to ensure data durability. CIFS does delegation using opportunistic locks, while NFSv4 has a dedicated operation for delegation. Prefetching and delegation of such files will improve read and write performance, lower network traffic, and lighten server load.

The access patterns again offer a simple, threshold-based decision algorithm. If an application instance does more than 10s of KB of sequential IO, and has no overwrite, then it is likely to be a content viewing or update application instance; such files are prefetched and delegated to the clients. *Implication 5: Clients can request file prefetch (read) and delegation (write) based on only IO sequentiality.*

Observation 6: Engineering applications with >50% sequential reads and >50% sequential writes are doing code compile tasks. We know this from looking at the file extensions in Figure 5. These compile processes show read sequentiality, write sequentiality, a significant overwrite ratio and large number of metadata requests. They rely on the server heavily for data accesses. We need more detailed client side information to understand why client caches are ineffective in this case. However, it is clear that the server cache needs to prefetch the read files for these applications. The high percentage of sequential reads and writes gives us another threshold-based algorithm to identify these applications. *Implication 6: Servers can identify compile tasks by the presence of both sequential reads and writes; server has to cache the output of these tasks.*

4.2 Server Side Access Patterns

As mentioned in Section 3.2, we analyzed two kinds of server side access units: files and deepest subtrees.

4.2.1 Files

File access patterns help storage server designers develop per-file placement and optimization techniques. We used 25 features to describe files (Table 5). Note that some of

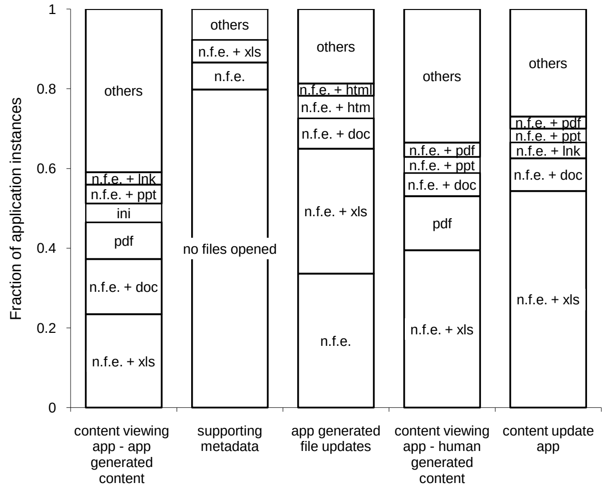

Figure 4: File extensions for corporate application instance access patterns. For each access pattern (column), showing the fraction of the two most frequent file extensions that are accessed together within a single application instance. "n.f.e." denotes files with "no file extension".

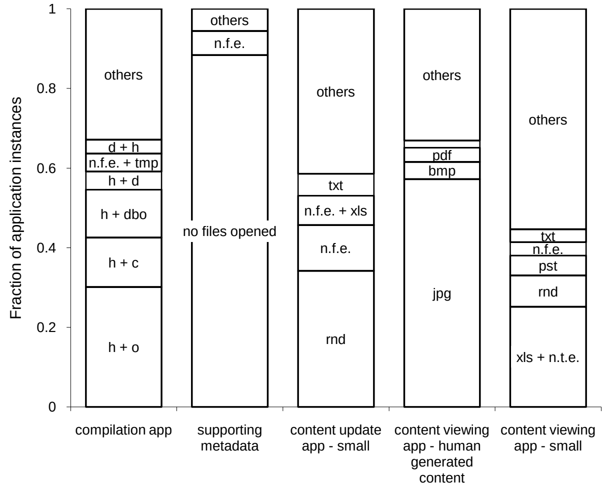

Figure 5: File extensions for engineering application instance access patterns. For each access pattern (column), showing the fraction of the two most frequent file extensions that are accessed together within a single application instance. "n.f.e." denotes files with "no file extension".

the features include different percentiles of a characteristic, e.g., read request size as percentiles of all read requests. We believe including different percentiles rather than just the average would allow better separation of access patterns. The corporate trace has 1,155,099 files, and the engineering trace has 1,809,571.

In Table 5, we quantitative descriptions and short names for all the file access patterns. Figures 6 and 7 give the most common file extensions in each. We derived the names by examining the read-write ratio and IO size. For the engineering trace, examining the file extensions also proved useful, leading to labels such as "edit code and compile output", and "read only log/backup".

(a). Descriptive features for each application instance			
Total IO size	Total metadata requests	Repeated read ratio	File opens
Read:write ratio by bytes	Avg. time between IO requests	Overwrite ratio	Unique files opened
Total IO requests by bytes	Read sequentiality	Tree connects	Directories accessed
Read:write ratio by requests	Write sequentiality	Unique trees accessed	File extensions accessed

(b). Corporate application instance access patterns	Content viewing app - app generated content	Supporting metadata	App generated file updates	Content viewing app - human generated content	Content update app
% of all app instances	16%	56%	14%	8.8%	5.1%
Total IO	100 KB	0	1 KB	800 KB	3.5 MB
Read:write ratio	1:0	0:0	1:1	1:0	2:3
Metadata requests	130	5	50	130	500
Read sequentiality	5%	-	0%	80%	50%
Write sequentiality	-	-	0%	-	80%
Overwrite ratio	-	-	0%	-	5%
File opens:files	19:4	0:0	10:4	20:4	60:11
Tree connect:Trees	2:2	0:0	2:2	2:2	2:2
Directories accessed	3	0	3	3	4
File extensions accessed	2	0	2	2	3

(c). Engineering application instance access patterns	Compilation app	Supporting metadata	Content update app - small	Content viewing app - human generated content	Content viewing app - small
% of all app instances	1.6%	93%	0.9%	2.0%	2.5%
Total IO	2 MB	0	2 KB	1 MB	3 KB
Read:write ratio	9:1	0:0	0:1	1:0	1:0
Metadata requests	400	1	14	40	15
Read sequentiality	50%	-	-	90%	0%
Write sequentiality	80%	-	0%	-	-
Overwrite ratio	20%	-	0%	-	-
File opens:files	145:75	0:0	3:1	5:4	2:1
Tree connect:Trees	1:1	0:0	1:1	1:1	1:1
Directories accessed	15	0	1	1	1
File extensions accessed	5	0	1	1	1

Table 4: Application instance access patterns. (a): Full list of descriptive features. (b) and (c): Short names and descriptions of application instances in each access pattern; listing only the features that help separate the access patterns.

We see that there are groupings of files with similar extensions. For example, in the corporate trace, the small random read access patterns include many file extensions associated with web browser caches. Also, multi-media files like `.mp3` and `.jpg` congregate in the sequential read and write access patterns. In the engineering trace, code libraries group under the sequential write files, and read only log/backup files contain file extensions `.0` to `.99`. However, the most common file extensions in each trace still spread across many access patterns, e.g., office productivity files in the corporate trace and code files in the engineering trace.

Observation 7: For files with >70% sequential reads or sequential writes, the repeated read and overwrite ratios are close to zero. This implies that there is little benefit in caching these files at the server. They should be prefetched as a whole and delegated to the client. Again, the bimodal IO sequentiality offers a simple algorithm for the server to detect which files should be prefetched and delegated – if a file has any sequential access, it is likely to have a high percentage of sequential access, therefore it should be prefetched and delegated to the client. Future storage servers can suggest such information to clients, leading to delegation requests. *Implication 7: Servers should delegate sequentially accessed files to clients to improve IO performance.*

Observation 8: In the engineering trace, only the edit code and compile output files have a high % of repeated reads. Those files should be delegated to the clients as well. The repeated reads do not show up in the engineering application instances, possibly because a compilation process launches many child processes repeatedly reading the same files. Each child process reads “fresh data,” even though the server sees repeated reads. With larger memory or flash caches at clients, we expect this behavior to drop. The working set issues that lead to this scenario need to be examined. If the repeated reads come from a single client, then the server can suggest that the client cache the appropriate files.

We can again employ a threshold-based algorithm. Detecting any repeated reads at the server signals that the file should be delegated to the client. At worst, only the first few reads will hit the server. Subsequent repeated reads are stopped at the client. *Implication 8: Servers should delegate repeatedly read files to clients.*

Observation 9: Almost all files are active (have opens, IO, and metadata access) for only 1-2 hours over the entire trace period, as indicated by the typical opens/read/write activity of all access patterns. There are some regularly accessed files, but they are so few that they do not affect the k-means analysis. The lack of regular access for most files means that there is room for the server to employ techniques to increase capacity by doing compaction on idle files.

Common techniques include deduplication and compression. The activity on these files indicate that the IO performance impact should be small. Even if run constantly, compaction has a low probability of affecting an active file. Since common libraries like `gzip` optimize for decompression [11], decompressing files at read time should have only slight performance impact. *Implication 9: Servers can use file idle time to compress or deduplicate data to increase storage capacity.*

(a). Descriptive features for each file	
Number of hours with 1, 2-3, or 4 file opens	Read sequentiality
Number of hours with 1-100KB, 100KB-1MB, or >1MB reads	Write sequentiality
Number of hours with 1-100KB, 100KB-1MB, or >1MB writes	Read:write ratio by bytes
Number of hours with 1, 2-3, or 4 metadata requests	Repeated read ratio
Read request size - 25th, 50th, and 75th percentile of all requests	Overwrite ratio
Write request size - 25th, 50th, and 75th percentile of all requests	
Avg. time between IO requests - 25th, 50th, and 75th percentile of all request pairs	

(b). Corporate file access patterns	Metadata only	Sequential write	Sequential read	Small random write	Smallest random read	Small random read
% of all files	59%	4.0%	4.1%	4.7%	19%	9.2%
Opens activity	2hrs, 1 open	1hr, 2-3 opens	1hr, 2-3 opens	1hr, 2-3 opens	1hr, 1 open	1hr, 1 open
Read activity	0	0	1hr, 100KB-1MB	0	1hr, 1-100KB	1hr, 1-100KB
Write activity	0	1hr, 100KB-1MB	0	1hr, 1-100KB	0	0
Read request size	-	-	4-32KB	-	2KB	32KB
Write request size	-	60KB	-	4-22KB	-	-
Read sequentiality	-	-	70%	-	0%	0%
Write sequentiality	-	80%	-	0%	-	-
Read:write ratio	0:0	0:1	1:0	0:1	1:0	1:0

(c). Engineering file access patterns	Metadata only	Sequential write	Small random read	Edit code & compile output	Sequential read	Read-only log/backup
% of all files	42%	1.9%	32%	7.3%	8.3%	8.1%
Opens activity	1hr, 1 open	1hr, 2-3 opens	1hr, 2-3 opens	1hr, 2-3 opens	1hr, 2-3 opens	2hrs, 2-3 opens
Read activity	0	0	1hr, 1-100KB	1hr, 1-100KB	1hr, 1-100KB	2hrs, 1-100KB
Write activity	0	1hr, >1MB	0	0	0	0
Read request size	-	-	3-4KB	4KB	8-16KB	1KB
Write request size	-	64KB	-	-	-	-
Read sequentiality	-	-	0%	0%	70%	0%
Write sequentiality	-	90%	-	-	-	-
Repeated read ratio	-	-	0%	50%	0%	0%
Read:write ratio	0:0	0:1	1:0	1:0	1:0	1:0

Table 5: File access patterns. (a): Full list of descriptive features. (b) and (c): Short names and descriptions of files in each access pattern; listing only the features that help separate the access patterns.

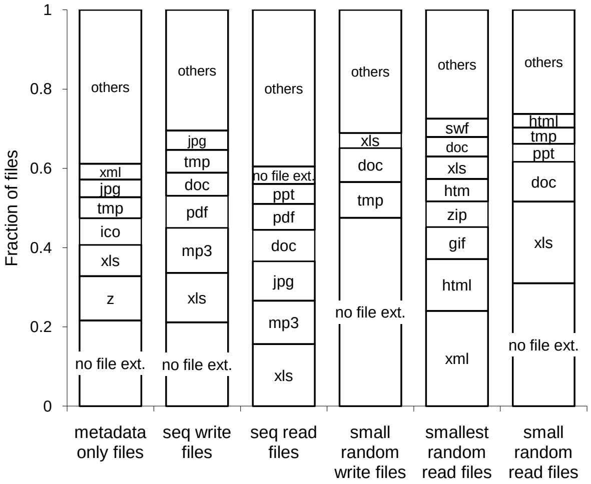

Figure 6: File extensions for corporate files. Fraction of file extensions in each file access pattern.

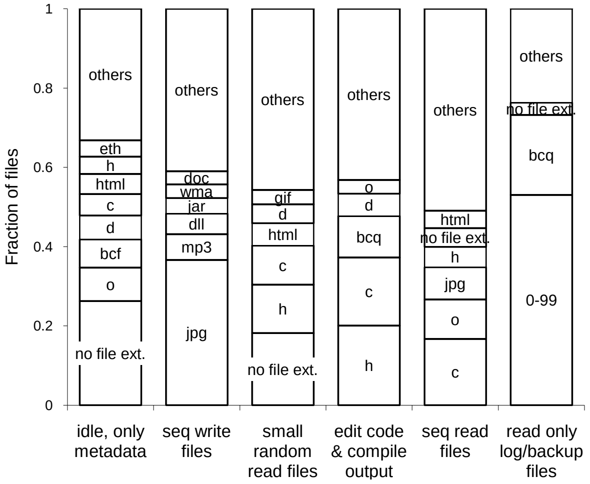

Figure 7: File extensions for engineering files. Fraction of file extensions in each file access pattern.

Observation 10: All files have either all random access or >70% sequential access. The small random read and write files in both traces can benefit from being placed on media with high random access performance, such as solid state drives (SSDs). Files with a high percentage of sequential access can reside on traditional hard disk drives (HDDs), which already optimize for sequential access. The bimodal IO sequentiality offers yet another threshold-based placement algorithm – if a file has any sequential access, it is likely to have a high percentage of sequential access; therefore place it on HDDs. Otherwise, place it on SSDs. We note that there are more randomly accessed files than sequentially accessed files. Even though sequential files tend to be larger, we still need to do a working set analysis to determine the right size of server SSDs for each use case. *Implication 10: Servers can select the best storage medium for each file based only on access sequentiality.*

4.2.2 Deepest subtrees

Deepest subtree access patterns help storage server designers develop per-directory policies. We used 40 features to describe deepest subtrees (Table 6). Some of the features

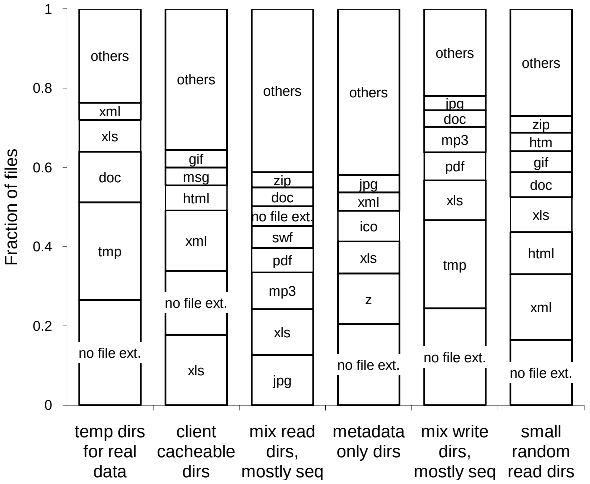

Figure 8: File extensions for corporate deepest subtrees. Fraction of file extensions in deepest subtree access patterns.

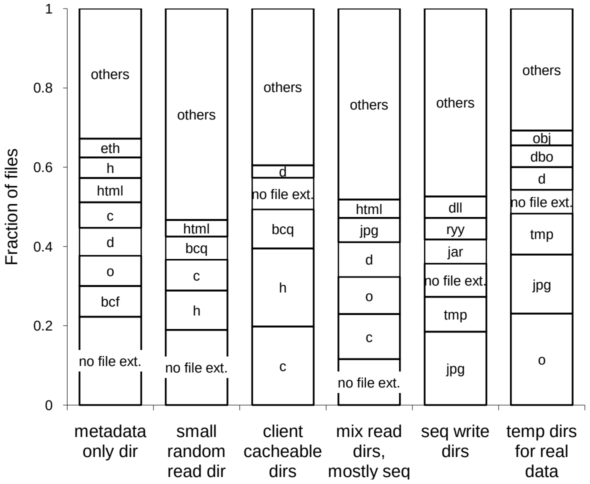

Figure 9: File extensions for engineering deepest subtrees. Fraction of file extensions in deepest subtree access patterns.

include different percentiles of a characteristic, e.g. per file read sequentiality as percentiles of all files in a directory. Including different percentiles rather than just the average allows better separation of access patterns. The corporate trace has 117,640 deepest subtrees, and the engineering trace has 161,858. We use "directories" and "deepest subtrees" interchangeably.

In Table 6, we provide quantitative descriptions and short names for all the deepest subtree access patterns. We derive the names using two types of information. First, we analyze the file extensions in each subtree access pattern (Figures 8 and 9). Second, we examine how many files of each file access patterns are within each subtree pattern (Figures 10). For brevity, we show only the graph for corporate deepest subtrees. The graph for the engineering deepest subtrees conveys the same information with regard to our design insights.

For example, the "random read" and "client cacheable" labels come from looking at the IO patterns. "Temporary directories" accounted for the `.tmp` files in those directories. "Mix read" and "mix write" directories considered the presence of both sequential and randomly accessed files in those directories.

The metadata background noise remains visible at the subtree layer. The spread of file extensions is similar to that for file access patterns – some file extensions congregate and others spread evenly. Interestingly, some subtrees have a large fraction of metadata-only files that do not affect the descriptions of those subtrees.

Some subtrees contain only files of a single access pattern (e.g., small random read subtrees in Figures 10). There, we can apply the design insights from the file access patterns to the entire subtree. For example, the small random read subtrees can reside on SSDs. Since there are more files than subtrees, per-subtree policies can lower the amount of policy information kept at the server.

In contrast, the mix read and mix write directories contain both sequential and randomly accessed files. Those subtrees need per-file policies: Place the sequentially accessed files on HDDs and the randomly accessed files on SSDs. Soft links to files can preserve the user-facing directory organization, while allowing the server optimize per-file placement. The server should automatically decide when to apply per-file or per-subtree policies.

Observation 11: Directories with sequentially accessed files almost always contain randomly accessed files also. Conversely, some directories with randomly access files will not contain sequentially accessed files. Thus, we can default all subtrees to per-subtree policies. Concurrently, we track the IO sequentiality per subtree. If the sequentiality is above some threshold, then the subtree switches to per-file policies. *Implication 11: Servers can change from per-directory placement policy (default) to per-file policy upon seeing any sequential IO to any files in a directory.*

Observation 12: The client cacheable subtrees and temporary subtrees aggregate files with repeated reads or overwrites. Additional computation showed that the repeated reads and overwrites almost always come from a single client. Thus, it is possible for the entire directory to be prefetched and delegated to the client. Delegating entire directories can preempt all accesses that are local to a directory, but consumes client cache space. We need to understand the tradeoffs through a more in-depth working set and temporal locality analysis at both the file and deepest subtree levels. *Implication 12: Servers can delegate repeated read and overwrite directories entirely to clients, tradeoffs permitting.*

4.3 Access Pattern Evolutions Over Time

We want to know if the access patterns are restricted to our particular tracing period or if they persist across time. Only if the design insights remain relevant across time can we rationalize their existence in similar use cases.

We do not have enough traces to generalize beyond our monitoring period. We investigate the reverse problem - if we

(a). Descriptive features for each subtree	
Number of hours with 1, 2-3, or 4 file opens	Read:write ratio - 25th, 50th, and 75th percentile of files
Number of hours with 1-100KB, 100KB-1MB, or >1MB reads	Repeated read ratio - 25th, 50th, and 75th percentile of files
Number of hours with 1-100KB, 100KB-1MB, or >1MB writes	Overwrite ratio - 25th, 50th, and 75th percentile of files
Number of hours with 1, 2-3, or 4 metadata requests	Read sequentiality - aggregated across all files
Read request size - 25th, 50th, and 75th percentile of all requests	Write sequentiality - aggregated across all files
Write request size - 25th, 50th, and 75th percentile of all requests	Read:write ratio - aggregated across all files
Avg. time between IO requests - 25th, 50th, and 75th percentile of all request pairs	Repeated read ratio - aggregated across all files
Read sequentiality - 25th, 50th, and 75th percentile of files in the subtree	Overwrite ratio - aggregated across all files
Write sequentiality - 25th, 50th, and 75th percentile of files in the subtree	

(b). Corp. subtree access patterns	Temp dirs for real data	Client cacheable dirs	Mix read dirs, mostly sequential	Metadata only dirs	Mix write dirs, mostly sequential	Small random read dirs
% of all subtrees	2.3%	4.1%	5.6%	64%	3.5%	21%
Opens activity	3hrs, >4 opens	3hr, 1 open	2hr, 1 open	2hr, 1 open	1hr, >4 opens	1hr, >4 opens
Read activity	3hrs, 1-100KB	2hrs, 1-100KB	1hr, 1-100KB	0	0	1hr, 1-100KB
Write activity	2hrs, 0.1-1MB	0	0	0	1hr, >1MB	0
Read request size	4KB	4-10KB	4-32KB	-	-	1-8KB
Write request size	4KB	-	-	-	64KB	-
Read sequentiality	10-30%	0%	50-70%	-	-	0%
Write sequentiality	50-70%	-	-	-	70-80%	-
Repeat read ratio	20-50%	50%	0%	-	-	0%
Overwrite ratio	30-70%	-	-	-	0%	-
Read:write ratio	1:0 to 0:1	1:0	1:0	0:0	0:1	1:0

(c). Eng. subtree access patterns	Metadata only dirs	Small random read dirs	Client cacheable dirs	Mixed read dirs, mostly sequential	Sequential write dirs	Temp dirs for real data
% of all subtrees	59%	25%	6.1%	7.1%	1.9%	1.3%
Opens activity	1hr, 2-3 opens	1hr, >4 opens	1hr, >4 opens	1hr, >4 opens	1hr, >4 opens	3hrs, >4 opens
Read activity	0	1hr, 1-100KB	1hr, 1-100KB	1hr, 0.1-1MB	0	3hrs, 1-100KB
Write activity	0	0	0	0	1hr, 0.1-1MB	1hr, 1-100KB
Read request size	-	1-4KB	2-4KB	8-10KB	-	4-32KB
Write request size	-	-	-	-	32-60KB	4-60KB
Read sequentiality	-	0%	0%	40-70%	-	10-65%
Write sequentiality	-	-	-	-	70-90%	60-80%
Repeat read ratio	-	0%	50-60%	0%	-	0-40%
Overwrite ratio	-	-	-	-	0%	0-30%
Read:write ratio	0:0	1:0	1:0	1:0	0:1	1:0 to 0:1

Table 6: Deepest subtree access patterns. (a): Full list of descriptive features. (b) and (c): Short names and descriptions of subtrees in each access pattern; listing only the features that help separate access patterns.

had to analyze traces from only a subset of our tracing period, how would our results differ? We divided our traces into weeks and repeated the analysis for each week. For brevity, we present only the results for weekly analysis of corporate application instances and files. These two layers have yielded the most interesting design insights and they highlight separate considerations at the client and server.

Figure 11 shows the result for files. All the large access patterns remain steady across the weeks. However, the access pattern corresponding to the smallest number of files, the small random write files, comes and goes week to week. There are exactly two, temporary, previously unseen access patterns that are very similar to the small random files. The peaks in the metadata only files correspond to weeks that contain U.S. federal holidays or weeks immediately preceding a holiday long weekend. Furthermore, the numerical values of the descriptive features for each access pattern vary in a moderate range. For example, the write sequentiality of the sequentiality write files ranges from 50% to 90%.

Figure 12 shows the result for application instances. We see no new access patterns, and the fractional weight of each access pattern remains nearly constant, despite holidays. Furthermore, the numerical values of descriptive features also remain nearly constant. For example, the write sequentiality of the content update applications varies in a narrow range from 80% to 85%.

Thus, if we had done our analysis on just a week's traces, we would have gotten nearly identical results for application instances, and qualitatively similar result for files. We believe that the difference comes from the limited duration of client sessions and application instances, versus the long-term persistence of files and subtrees.

Based on our results, we are confident that the access patterns are not restricted just to our particular trace period. Future storage systems should continuously monitor the access patterns at all levels, automatically adjusting policies as needed, and notify designers of previously unseen access patterns.

We should always be cautious when generalizing access patterns from one use case to another. For use cases with the same applications running on the same OS file API, we expect to see the same application instance access patterns. Session access patterns such as daily work sessions are also likely to be general. For the server side access patterns, we expect the files and subtrees with large fractional weights to appear in other use cases.

5. ARCHITECTURAL IMPLICATIONS

Section 4 offered many specific optimizations for placement, caching, delegation, and consolidation decisions. We combine the insights here to speculate on the architecture of future enterprise storage systems.

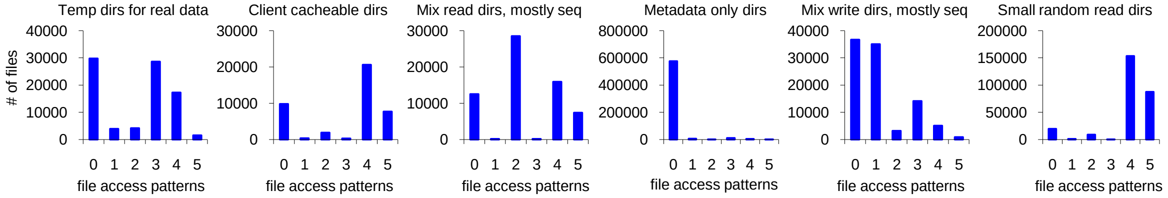

Figure 10: Corporate file access patterns within each deepest subtree. For each deepest subtree access pattern (i.e., each graph), showing the number of files belonging to each file access pattern that belongs to subtrees in the subtree access pattern. Corporate file access pattern indices: 0. metadata only files; 1. sequential write files; 2. sequential read files; 3. small random write files; 4. small random read files; 5. less small random read files.

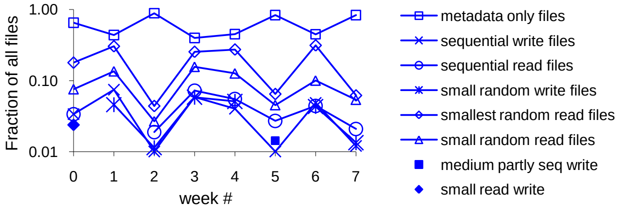

Figure 11: Corporate file access patterns over 8 weeks. All patterns remain (hollow markers), but the fractional weight of each changes greatly between weeks. Some small patterns temporarily appear and disappear (solid markers).

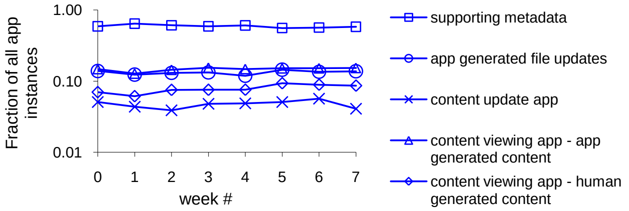

Figure 12: Corporate application instance access patterns over 8 weeks. All patterns remain with near constant fractional weight. No new patterns appear.

We see a clear separation of roles for clients and servers. The client design can target high IO performance by a combination of efficient delegation, prefetching and caching of the appropriate data. The servers should focus on increasing their aggregated efficiency across clients: collaboration with clients (on caching, delegation, etc.) and exploiting user patterns to schedule background tasks. Automating background tasks such as offline data deduplication delivers capacity savings in a timely and hassle-free fashion, i.e., without system downtime or explicit scheduling. Regarding caching at the server, we observe that very few access patterns actually leverage the server's buffer cache for data accesses. Design insights 4-6, 8 and 12 indicate a heavy role for the client cache and Design insight 7 suggests how *not* to use the server buffer cache - caching metadata only and acting as a warm/backup cache for clients would result in lower latencies for many access patterns.

We also see simple ways to take advantage of new storage media such as SSDs. The clear identification of sequential and random access file patterns enables efficient device-specific data placement algorithms (Design insights 10 and 11). Also, the background metadata noise seen at all levels suggests that storage servers should both optimize for metadata accesses and redesign client-server interactions to decrease the metadata chatter. Depending on the growth of metadata and the performance requirements, we also need to consider placing metadata on low latency, non-volatile media like flash or SSDs.

Furthermore, we believe that storage systems should introduce many monitoring points to dynamically adjust the decision thresholds of placement, caching, or consolidation policies. We need to monitor both clients and servers. For example, when repeated read and overwrite files have been properly delegated to clients, the server would no longer see files with such access patterns. Without monitoring points at the clients, we would not be able to quantify the file delegation benefits. Storage systems should make extensible tracing APIs to expedite the collection of long-term future traces. This will facilitate future work similar to ours.

6. CONCLUSIONS AND FUTURE WORK

We must address the storage technology trends toward ever-increasing scale, heterogeneity, and consolidation. Current storage design paradigms that rely on existing trace analysis methods are ill equipped to meet the emerging challenges because they are unidimensional, focus only on the storage server, and are subject to designer bias. We showed that a multi-dimensional, multi-layered trace-driven design methodology leads to more objective design points with highly targeted optimizations at both storage clients and servers. Using our corporate and engineering use cases, we present a number of insights that informs future designs. We described in some detail the access patterns we observed, and we encourage fellow storage system designers to extract further insights from our observations.

Future work includes exploring the dynamics of changing working sets and access sequences, with the goal of anticipating data accesses before they happen. Another worthwhile analysis is to look for optimization opportunities *across clients*; this requires collecting traces at different clients, instead of only at the server. Also, we would like to explore opportunities for deduplication, compression, or data placement. Doing so requires extending our analysis from *data movement* patterns to also include *data content* patterns. Furthermore, we would like to perform on-line analysis in live storage systems to enable dynamic feedback on placement and optimization decisions. In addition, it would be useful to build tools to synthesize the access patterns, to enable designers to evaluate the optimizations we proposed here.

We believe that storage system designers face an increasing challenge to anticipate access patterns. Our paper builds the case that system designers can longer accurately anticipate access patterns using intuition only. We believe that the corporate and engineering traces from our corporate headquarters would have similar use cases at other traditional and high-tech businesses. Other use cases would require us to perform the same trace collection and analysis process to extract the same kind of "ground truth". We also need similar studies at regular intervals to track the evolving use of storage system. We hope that this paper contributes to an objective and principled design approach targeting rapidly changing data access patterns.

NetApp, the NetApp logo, and Go further, faster are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.

7. REFERENCES

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *FAST 2007*.

[2] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, Cambridge, Massachusetts, 2004.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *SOSP 1991*.

[4] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys 2010*.

[5] Common Internet File System Technical Reference. Storage Network Industry Association, 2002.

[6] IDC Whitepaper: The economics of Virtualization. `www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf`.

[7] J. R. Douceur and W. J. Bolosky. A Large-Scale Study of File-System Contents. In *SIGMETRICS 1999*.

[8] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *FAST 2003*.

[9] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE 2009*.

[10] S. Gribble, G. S. Manku, E. Brewer, T. J. Gibson, and E. L. Miller. Self-Similarity in File Systems: Measurement and Applications. In *SIGMETRICS 1998*.

[11] The gzip algorithm. `http://www.gzip.org/algorithm.txt`.

[12] IDC Report: Worldwide File-Based Storage 2010-2014 Forecast Update. `http://www.idc.com/getdoc.jsp?containerId=226267`.

[13] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production Windows Servers. In *IISWC 2008*.

[14] Open Source Clustering Software - C Clustering Library. `http://bonsai.hgc.jp/~mdehoon/software/cluster/software.htm`, 2010.

[15] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC 2008*.

[16] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *FAST 2010*.

[17] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *SOSP 1985*.

[18] K. K. Ramakrishnan, P. Biswas, and R. Karedla. Analysis of file I/O traces in commercial computing environments. In *SIGMETRICS 1992*.

[19] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *USENIX 2000*.

[20] I. Stoica. A Berkeley View of Big Data: Algorithms, Machines and People. UC Berkeley EECS Annual Research Symposium, 2011.

[21] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy 2011*.

[22] R. Villars. The Migration to Converged IT: What it Means for Infrastructure, Applications, and the IT Organization. IDC Directions Conference 2011.

[23] VMware Whitepaper: Server Consolidation and Containment. `www.vmware.com/pdf/server_consolidation.pdf`.

[24] W. Vogels. File system usage in Windows NT 4.0. In *SOSP 1999*.

[25] M. Zhou and A. J. Smith. Analysis of Personal Computer Workloads. In *MASCOTS 1999*.