A File By Any Other Name: Managing File Names with Metadata

Name1
Affiliation1
Email1

Name2
Affiliation2
Email2

Name3 Name4
Affiliation3
Email3/4

Abstract

File names are an ancient abstraction, a string of characters to uniquely identify a file and help users remember the contents of a file when they look for it later. However, they are not an ideal solution for these tasks. Names often contain useful semantic metadata provided by the user, but this metadata is opaque to the rest of the system. The formatting of the file name is up to the user, and as a result, metadata in file names is often error-prone, inconsistently formatted, and hard to search for later. File names need to be more meaningful and reliable, while simplifying application design and encouraging users and applications to provide more metadata for search.

We describe our prototype file system, TrueNames, a POSIX compliant file system which demonstrates an alternate approach to naming and metadata, by providing metadata-aware naming. Our approach is intended to be complementary to non-hierarchical systems, while remaining effective within a hierarchical context. TrueNames separates the task of uniquely identifying a file from the task of helping the user remember its contents. It captures metadata in a more useable format, and uses it to generate file names which are *correct*, *regenerable*, and *structured* by design. TrueNames simplifies application design by providing a consistent interface for metadata aware naming, incurs minimal overhead of approximately 15% under realistic workloads, and can simplify a wide variety of data management tasks for both applications and users.

1. Introduction

File names have existed since the earliest file systems, and serve two important functions. First, they serve to uniquely identify a file over time. Second, they serve to jog our memory, describing the contents of a file, and helping us to find it or recognize it when we look at it later. In order to help users to find and remember files, they often contain a bounty of useful metadata about the file.

However current approaches to file names have flaws:

- They are unstructured semantic metadata which is opaque to the system
- Formatting is up to the user, and is error prone and inconsistent, making it hard to find files later
- Changing a file name destroys information

Consider the following file name drawn from the experiments for this paper:
`createfiles_HDD_truenames_100000files_1threads.data`
Looked at in one light, this is a long, arbitrary, error prone string of characters. In another light, it is a rich source of semantic metadata about the contents of the file that could be used for search and analysis. The challenge is to extract that information.

To resolve these flaws, we propose to disassociate the two functions of names, separating the task of uniquely identifying a file for applications from helping users to identify a file. As a proof of concept, we describe our new prototype file system, TrueNames, a FUSE-based file system which provides a durable unique identifier for a file which can be used by applications, captures rich metadata in a structured format, and uses it to dynamically generate user-friendly file names using *templates*. These file names have many advantages over conventional file names. They are *correct*, because they are continuously synchronized with the file's metadata. They are *regenerable*, allowing us to compress and recreate names at will without loss of information. File names can be *disambiguated* using all available metadata, which reduces accidental data over-writes. The metadata which we capture is structured, making it readily available for search and data management. We describe these new file names as *metadata aware* names.

TrueNames is designed to demonstrate the feasibility of several goals. Metadata aware file naming can make naming more reliable and less error prone. It can also prevent accidental overwriting of existing files by detecting collisions between names, and adding additional metadata to disambiguate. Many applications already offer some form of auto-

[Copyright notice will appear here once 'preprint' option is removed.]

matic file naming, and by making that functionality part of the file system, we can speed application development, and prevent code duplication and the resulting proliferation of bugs. Being able to regenerate file names allows us to port files between file systems with differing constraints, generating a meaningful file name in each location, and then reconstructing the original file name whenever needed. We can easily move metadata between file names and directory names, or store it for later use. And finally, by gently encouraging users and applications to share more metadata in a structured fashion, we can make metadata more readily available to improve the quality of file system search or support a non-hierarchical file system.

While TrueNames is a user space prototype file system, it demonstrates the feasibility of doing metadata aware naming. It offers extensive new functionality, and incurs very low overheads, less than 15% on realistic workloads. Much of the additional cost is incurred by added kernel crossings, suggesting that an in-kernel implementation would have negligible performance impact, while significantly improving file system search and data management.

2. Use Cases

Metadata aware naming can be used as a broadly applicable framework for solving cross-cutting concerns. It can be used by applications to simplify common tasks, and by scripts and end-users to better manage files. It can even help to prevent data loss caused by overwrites, and allow multiple applications to cooperatively name files. We describe a variety of use cases for TrueNames, and explain how it can benefit users and applications in each case.

2.1 Managing experimental data

One of the common problems a scientist faces is that of managing experimental results. A scientist might run a series of experiments, varying some parameters while holding others constant, and outputting the results to a file, then decide based on the results to vary other parameters. This is commonly managed by creating file names programmatically. However, this approach has some drawbacks. For instance, if the user wants to search the results by parameter, they will need to use string matching or a regular expression, which relies on consistent formatting, such as using the same field separator and the same field order. Likewise, a metadata field which contains the same character as the chosen field separator (such as an underscore in a library name) can throw a regular expression off. Finally, common queries such as range searches require complex regular expressions to perform.

Another common problem occurs when rerunning experiments while setting additional parameters. The user also needs to remember to change the file name output to reflect the new parameters, and older file names will not have the new parameters, even if they were applicable. If the user forgets to change the code which generates file names for experimental results, they may overwrite existing results files, wasting hours or days of compute time.

TrueNames simplifies creating file names and searching for metadata. Rather than programmatically generating file names, experiments can simply add metadata for all the experiment parameters. File names can be generated using a simple template, and then the template can be expanded by the user as new parameters become relevant, which results in both old and new files using the new template. File name collision detection can help prevent overwriting by disambiguating files on the fly. The results files are easily searchable by metadata field, regardless of the order fields occur in the file name, or what characters occur in the field. Using structured data allows both exact match and range queries, and file names are an accurate reflection of their contents. The authors used TrueNames to manage experimental results for this paper, and found it to be extremely helpful and simple to use, allowing them to easily manage multiple experiments and parameters, generate accurate and meaningful file names, and then search their results later.

2.2 Managing research papers

Applications such as Mendeley [6] are designed to manage a collection of research papers, making them easy to search. Mendeley indexes all of the metadata fields of each publication, does full text indexing, and can manage directory structures and file names. Mendeley does not use a fixed format for file names. Rather, they allow the user to pick the metadata fields and the order in which they should be used to generate a file name, much like TrueNames does. Applications such as Mendeley can work symbiotically with TrueNames, serving as an interface for extracting metadata and helping the user generate templates, while allowing TrueNames to do the work of keeping file names up to date.

2.3 Managing a music collection

Most modern MP3 players, in addition to playing music, also perform metadata and file management. They can download new music, organize music into folders, and assign names to music based on its metadata (stored in ID3 tags.) For instance, iTunes will place MP3s into folders based on artist and album, and then generate a file name based on a fixed template using the disc number, track number, and song title. However, this means that music can only be managed through the application. New music downloaded from outside the application is unknown, and changes to music's metadata are not reflected in the file name unless done through the application. Utilities such as MusicBrainz Picard [7] can repair missing metadata and names, but require the user to recreate their iTunes database to reimport the new file names.

TrueNames, like iTunes, can manage file names based on metadata, using whatever fields are desired. However, file names will always be kept in sync with the latest metadata,

regardless of the source of that metadata. New music can automatically be given a name in the desired format, no matter what application downloaded it. If the user prefers a different name, they can choose a different format, and all files of that type will automatically be renamed. By using unique identifiers rather than file names, databases don't have to be recreated when file names change. If multiple files are imported with the same name, rather than overwriting the existing files, TrueNames can check the metadata to prevent a collision which would result in losing one or more files.

2.4 Managing a photo collection

File names have lagged behind UIs in the photography field, making it challenging to find and manage photo files. While most applications offer sophisticated GUI photo management, many use the default file name generated by the camera, which contains only a per-camera sequence number, such as `IMG_655.jpg`, or perhaps a manufacturer and sequence number, such as `DSC_1967.jpg`. File name collisions are common. The latest version of iPhoto [4] names photos using the camera's generated name. Derivative files may be given a suffix based on size such as `IMG_655_1024.jpg`, or, in the case of a facial recognition thumbnail, an index corresponding to order of face discovery, such as `IMG_655_face1.jpg`. Photo names cannot be managed by the user, and offer very little information about their contents. Higher end applications such as Aperture [1] or Adobe Lightroom [5] allow the user to bulk rename files during import and export, using metadata such as EXIF fields and creation dates. However, these applications cannot keep file names in sync if metadata changes, making it difficult to manage photos in more than one application. For instance, a user might want to use facial recognition from iPhoto, while touching up photos in Lightroom. If the user wants to find a retouched photo of a certain person, they cannot search for it using metadata, and if they wish to put the information in the file name, they must painstakingly name the files manually.

TrueNames can significantly improve photo naming, by taking metadata extracted by the application and automatically constructing file names for photos based on metadata such as where they were taken, who was in them, and what size they are. If the metadata changes (such as a recognized face, or the addition of geo-tagging data), TrueNames will automatically update the file name to reflect the correct information. In addition, TrueNames allows multiple applications which can operate on the same files, such as iPhoto and Lightroom, to share responsibility for generating a meaningful name. Both applications can export metadata which can be used for naming and search, and TrueNames can manage the formatting and creation of file names, merging metadata from both applications into a single meaningful description of the file, something which is currently very difficult.

2.5 Recreating file names

One problem found in computing is that of file name portability. For instance, some file systems support longer names and paths than others. When moving files from one system to another, such as from primary to archival storage, or between two servers, file names can be truncated, losing important information, and making it difficult to find a file, even if it is brought back from archival. TrueNames can help with this, by allowing a file name to be regenerated based on the metadata. For instance, files being moved to archival storage can be updated to use a different name template with fewer fields, which fits within name constraints without truncating the file name at an arbitrary point. If the file is retrieved from archival storage back to primary storage, the original file name can be recreated without loss of data. Since structured metadata was stored and used to generate the original name, it can also be retrieved to search the archive, or help create meaningful directory names, in essence pivoting metadata between file names and directory names as needed.

3. Architecture

Having file names which can change outside the control of users and user applications poses a number of unique and interesting challenges. One must choose a storage mechanism for the metadata used for naming. There must be a way to define the structure of names, and what metadata they will use. Applications require a durable way to reference files, in order to maintain internal databases and repeatedly reference files. There must be a way to prevent accidental data loss through file name collisions. We discuss these challenges, and the necessary modifications to support metadata-aware naming throughout the file system and applications. The architecture of our prototype is shown in Figure 1.

3.1 Storing metadata

Our goal was to store rich metadata for file names in a way that was portable, did not significantly change the semantics of the file system, and was easy to understand and manage. To that end, we selected the extended attribute interface as being the most compatible with our goals. Extended attributes provide a simple key-value interface, and are associated with the inode (either by a reference to a metadata block or resource fork, or in the case of small metadata on ext4, directly stored in the inode), which means that files with hard links, which contain the same data, will also share metadata and names. (Soft links continue to offer the ability to have a mix of automatic and manual names for a single file.) Many applications already use extended attributes, they require no additional libraries, and are supported by most modern file systems, improving portability.

3.2 Managing names

Our file system assumes that file names have *schemas*, a set of rich metadata which is broadly applicable to many differ-

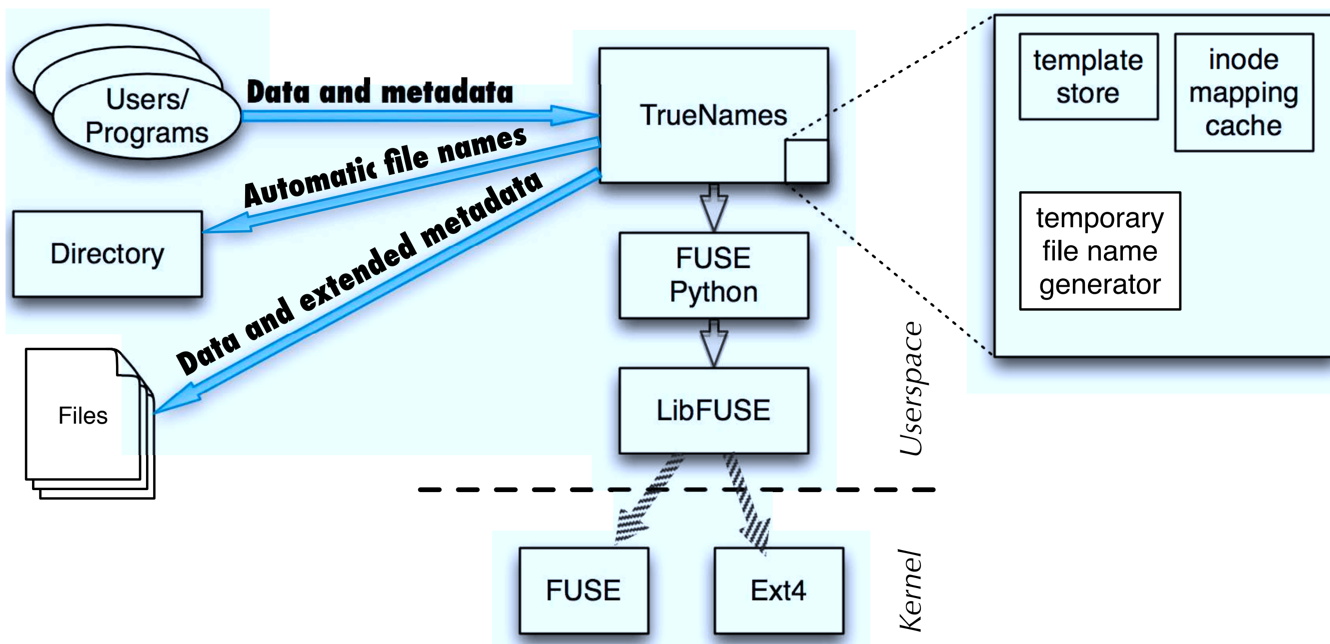


Figure 1: Architecture of TrueNames

ent files. However, not all files of the same type are assumed to have the same schema. For instance, an PDF file may be a scientific paper, a graph of experimental results, or an e-book. A text file might be a configuration file, or a letter to a relative. These will have different schemas that are appropriate. Likewise, two files of different types, such as `.jpg` and `.gif` files, may have a shared schema that is appropriate for both. Thus, we allow a *template* to be assigned to a file. A template defines the structure of a file name that is appropriate for that file, as shown in Example 1. Templates can contain file extensions which serve as the default extension. However, if the user supplies a file extension, it will override the template extension, allowing different types of files to share templates. For instance, `.jpg` and `.gif` files can share a `photo` template.

Example 1 Template file

```
music {$artist}-{$album}-{$track}.mp3
photo {$seq}_{$date}_{$camera}_{$location}.jpg
paper {$author}_{$conference}${year}_${tags}.pdf
exp {$wkload}_{$files}files_{$threads}threads.data
manual_name {$user.file_name}
```

In our prototype, templates are stored on a global basis, in a text configuration file, and loaded on file system startup, but more sophisticated implementations are possible. This file contains a list of template names and templates. Templates are associated with a file using an extended metadata field called `user.naming.type`, which references the name of a template. This field is not mandatory, making it possible to mix manually and automatically named files throughout the file system. If a user or application wishes to rename the file, they can change the value of the attribute to the name of a different template, adding additional metadata as needed. If a user wishes to manually manage a name, they can remove the current template from the file's metadata, or not choose a template during file creation.

As metadata is added or updated by users and applications, the file name is updated to reflect the current state of the file. This entails storing the extended metadata, looking up the template, re-calculating the file name, and then renaming the file. It may also entail updating the inode cache, and handling file name collisions caused by the rename.

If not all of the metadata is available at any point in time, TrueNames makes a best effort to update the file name, populating all of the fields that are known, and marking fields that are missing with a default value (in our prototype, unknown fields are marked with a `??`). In our photo example, that might result in a file name of `DSC1967_7-21-13_NikonD50_{$location}??.jpg` for a photo that has not yet been geotagged.

3.3 Programming with changing file names

One complication created by automatically named files is that the file name can change between accesses or modifications. For instance, when metadata is being added to the file, each new metadata item which is used by the template will trigger a rename. In these cases, the user or application needs a durable way to reference the file they are updating. TrueNames allows you to reference files either by name, or using a unique identifier which serves as a durable reference.

In our prototype, we use the inode to guarantee uniqueness of references. To guarantee that the inode has not been freed and reused, we will add the inode's generation number in the next version, similarly to how NFS handles stale file handles [17]. Unfortunately, most file systems do not provide a convenient way to look up and reference a file by its inode number. By contrast, we allow files to be referenced either by file name or directly by inode number, as if the inode number were a file name.

To reference a file in a directory by its inode number rather than by name, it can be opened using a reference to `<dirname>/.inode/<inodenumber>`, as shown in Example 2. This allows programs and scripts to create an automatically named file, call `fstat` on the file handle to acquire a durable reference, and then add metadata, all without needing to know the file's name.

Example 2 Reference by inode

```
$ curl http://indyband.org/1.mp3 > template=music
$ ls -i
13146 {$artist}??-{$album}??-{$track}??nozwZ8.mp3
$ setfattr -n user.artist -v"Indy Band" .inode/13146
$ setfattr -n user.track -v"So Obscure" .inode/13146
$ setfattr -n user.tracknum -v"1" .inode/13146
$ ls
Indy Band-{$album}??-So Obscure.mp3
```

Under a typical file system such as ext4, accessing a file by its inode number requires searching the system for a file which has a matching inode number and then resolving it to a name, which can be prohibitively expensive. While we already reduce the cost by reducing the search space to a single directory, this still requires an linear scan over the directory inode. To get acceptable performance, we further reduced this cost by adding an inode cache map to TrueNames, which allows us to look up a file name in constant time given an inode number. This is similar to the name cache used by the Linux VFS, although the lookups occur from inode to name, rather than vice versa. The inode map is populated from the directory inode on first access, and then caches all inodes in that directory up to some threshold (ten thousand files in our experiments.)

By encouraging applications to use a static reference, while allowing the name to vary, users can modify file names at will to create more user-friendly names, without breaking existing application references. In addition, references by inode will work for all files, not just automatically named ones, so even manually named files can benefit from this feature.

3.4 File creation with automatic names

In order to create a file, we need a way to signal two things to the operating system. First, we must signal what directory we are creating a file in. Second, we need to signal what template we would like to use. Additionally, we must make sure that the file name we are creating is unique, to prevent overwriting an existing file. Finally, none of the extended attributes for the file are yet available, so we cannot use them to generate the initial file name. This is similar to the problems faced by `mkstmp()` and related functions. However, our goal was to maintain existing semantics as much as possible, and require minimal rewrite of existing code, which ruled out adding an additional system call. We therefore overrode the semantics of `open()`, `creat()`, and `mknod()`, such that if a file is created with a name which ends in `template=X`, where `X` matches the name of a known template, the name is managed by the file system, and a new file is generated with an automatically generated unique file name. The initial file name is composed of the template contents, followed by a unique alphanumeric suffix such as `GzyH07`, and finally, any file extension, as shown in Example 2. We use atomic file creation with `O_EXCL` internally, to prevent race conditions, as well random back-off retries if we receive `EEXIST` during initial file creation. Once the file is created, metadata can be added to populate the file name. Since metadata must be added one field at a time, there is a possibility of a temporary name collision during metadata addition, which we attempt to detect and handle.

Alternatively, the application can create a file name using any name it wishes (including one given by the user), and then set the template and metadata fields after file creation. The original file name can be stored in metadata, and then reapplied later, or used for resolving collisions.

3.5 Handling collisions

One problem that can arise with automatic naming is that two files in the same directory may be different in content and metadata, but share all the fields that are currently referenced in the template. During rename, we check to see if the new metadata results in two files with the same name. If so, we check to see if any metadata differs. If disambiguating metadata is available, we add the first available metadata to the file name which will disambiguate the files, along with its extended metadata key. If not, then and only then, do we overwrite the existing file. In the future, we plan to explore more user-friendly strategies for disambiguation.

3.6 Application level support

A file system which requires extensive changes to applications is unlikely to see adoption. By maintaining POSIX compliance, TrueNames makes adoption easier. Existing applications which don't wish to take advantage of the added functionality can run on TrueNames without any modifications, and see very little change in performance. In order to take advantage of the new functionality, applications simply need to create a template, and begin exporting metadata for each new file. Optionally, they can begin referencing files using an inode reference. We describe below the changes to semantics in our prototype, and how they affect applications.

open()/creat()/mknod() As noted above, if these calls are invoked using a directory path followed by a template name, they will create an automatically named file using the template as a name, followed by a unique suffix and and an extension, and set `user.naming.type` to the template name. If the path supplied does not end in a template, they will create a file in the normal fashion.

setxattr()/removexattr() In addition to setting and removing extended attributes, these calls now additionally trigger a recalculation of the file name. If the attribute set or removed is one present in the file template, then the file will be renamed. Additionally, these can be used to change the template, or even remove the file from the set of dynamically named files and freeze the current name, by setting or removing the `user.naming.type` extended attribute.

rename() This operation can be used in a variety of ways.

- If the supplied target path contains a different directory, but the same file name, the file is moved to the new directory using its current file name.
- If the target path contains a different file name which is the name of an existing template, we update the file to use the new template.
- If the target path contains a different file name which is not the name of an existing template, we assume the user wishes to manually control the file name. We rename the file to the new file name, and remove `user.naming.type` from the file's metadata.
- In all cases, if a new inode is created during rename (for instance, if the file is migrated between file systems to a file system which supports extended attributes), all the extended metadata is copied to the new inode.

link()/Calls which use hard links Under our prototype, a hard link shares an inode, and therefore all extended metadata, with the path it links to. Therefore, files which are hard links to an automatically named file will share a name with the file they link to. If a file is dynamically named, a hard link in in the same directory is not feasible, since it has the same name as the target. An attempt to create a hard link in the same directory will fail with `EEXIST`. Otherwise, hard links function as expected. This is a limitation of our prototype. In future work, we plan to create a semantically complete method for automatically managing both hard and soft links.

symlink()/Calls which use soft links Soft links work as before, and can target either a file name, or an inode path, depending on the desired behavior. Soft links can be used to supply multiple names in the same directory, such as an automatic name and a manual name. Due to restrictions on extended attributes, soft links cannot be automatically named in our prototype.

Every file now has at least two names: its human readable name, either auto-generated or assigned by a human, and its inode number preceded by its directory path. It may have additional names via hard and soft links. Human readable names and inode references are interchangeable in all file system calls. A reference by inode can be opened, linked, have metadata set or gotten, and so on. However, `.inode/` itself is not a real directory on disk, and cannot be opened or have its directory entries iterated over.

If multiple applications manage the same files, then there is the possibility of both applications attempting to manage the template and metadata. Adding additional metadata to file names and templates allows richer search, and can improve generated names, but applications may wish to prompt the user before changing the template or removing fields from it.

4. Experimental Design

We have demonstrated how new functionality can be added to the file system to make it more searchable, to make file names more correct and structured, and how this functionality can easily integrate into existing file systems. However, a file system's functionality must also be balanced against its performance. We describe how we evaluated the performance of TrueNames. In order to effectively benchmark such a file system, we need to answer questions on how it affects basic file system operations, such as file creation, deletion, and renaming. We also need to describe the effect on extended metadata operations. In order to evaluate our file system, we compared it against two other file systems, one comparable Python FUSE file system, as well as a raw ext4 file system in order to characterize the overhead of FUSE and Python versus the overhead of our file system.

- `xmp` is the example file system which ships with fuse-python. We added support for extended attributes, using the same library, `py-xattr`, as used for TrueNames. Otherwise, it is a vanilla FUSE file system with no additional functionality.
- As a point of reference, we also include file system performance on a raw ext4 file system.

In the case of both TrueNames and `xmp`, we ran in single threaded mode, since `xmp` does not support multiple threads by default, and we wanted to modify it as little as possible. TrueNames will support multiple threads in the future. We disabled the inode cache, and set the `entry_timeout` to 0 in order to prevent stale file name entries. The only other modification to `xmp` was a fix for a bug which caused all writes to occur in append mode.

We ran two batches of experiments, one using an SSD, and one using a hard disk drive. Our SSD experiments were run on a 100 GB Intel 330 Series SSD, in an 8 core Intel Xeon CPU E3-1230 V2 @ 3.30GHz with 16 GB of RAM. Our HDD experiments were run on a 7200 RPM 500 GB

Seagate Constellation drive, in an 8 core Intel Xeon CPU E5620 @ 2.40GHz and 24 GB of RAM. In both cases, we ran on Fedora with a 3.9.10-200.fc18.x86_64 kernel, and an ext4 file system as our backing store.

5. Results and Analysis

TrueNames is a proof of concept user space prototype, not a production file system, but even so, it has very low overhead, demonstrating that automatic naming can be added to production file systems with minimal performance implications. We tested TrueNames under a variety of micro benchmarks and macro benchmarks, in order to analyze its performance under extreme conditions as well as realistic workloads. TrueNames is noticeably slower under our micro benchmarks, as expected, but the performance penalty can be measured in fractions of a millisecond, and TrueNames exhibits no scaling issues under high load. Under normal file system loads, such as in our macro benchmarks, TrueNames performs with only a minimal overhead, much of it due to being Python and single-threaded. In addition, TrueNames shows no impact on operations other than file creation and extended metadata operations. In aggregate, these numbers suggest that an production implementation of these ideas can serve as an replacement for many other file systems, large and small alike, adding useful new functionality at little to no performance cost.

5.1 Microbenchmarks

There are a number of benchmarks designed to exercise metadata. However, these are generally aimed towards file system metadata, such as performing high-speed updates to modification times. Tools such as Filebench [2], while quite flexible, do not offer the ability to modify extended attributes out of the box. By contrast, we needed to evaluate our system's impact on extended metadata performance. In order to do this, we wrote a benchmark designed to add, update, and delete extended attributes continuously. In effect, if the file is of an automatically named type, this results in TrueNames continuously renaming the file.

In order to focus as much as possible on the metadata speed, we pre-created a file set of size n. We then iterated over the file set until all n files had been touched, setting a single extended attribute on each file, calculating the latency of each operation and collecting statistics. Ext4 stores small metadata attributes in the inode, so files with a very large number of extended attributes will show lower performance than our benchmarks. One potential optimization is to ensure that metadata attributes relevant to the name are kept in the inode, since they are likely to be small, and not numerous.

Once the add benchmark completed, we ran similar benchmarks to update an attribute, and finally, to delete an attribute. This allowed us to exercise the new code paths continuously, highlighting any performance differences from our baseline file systems.

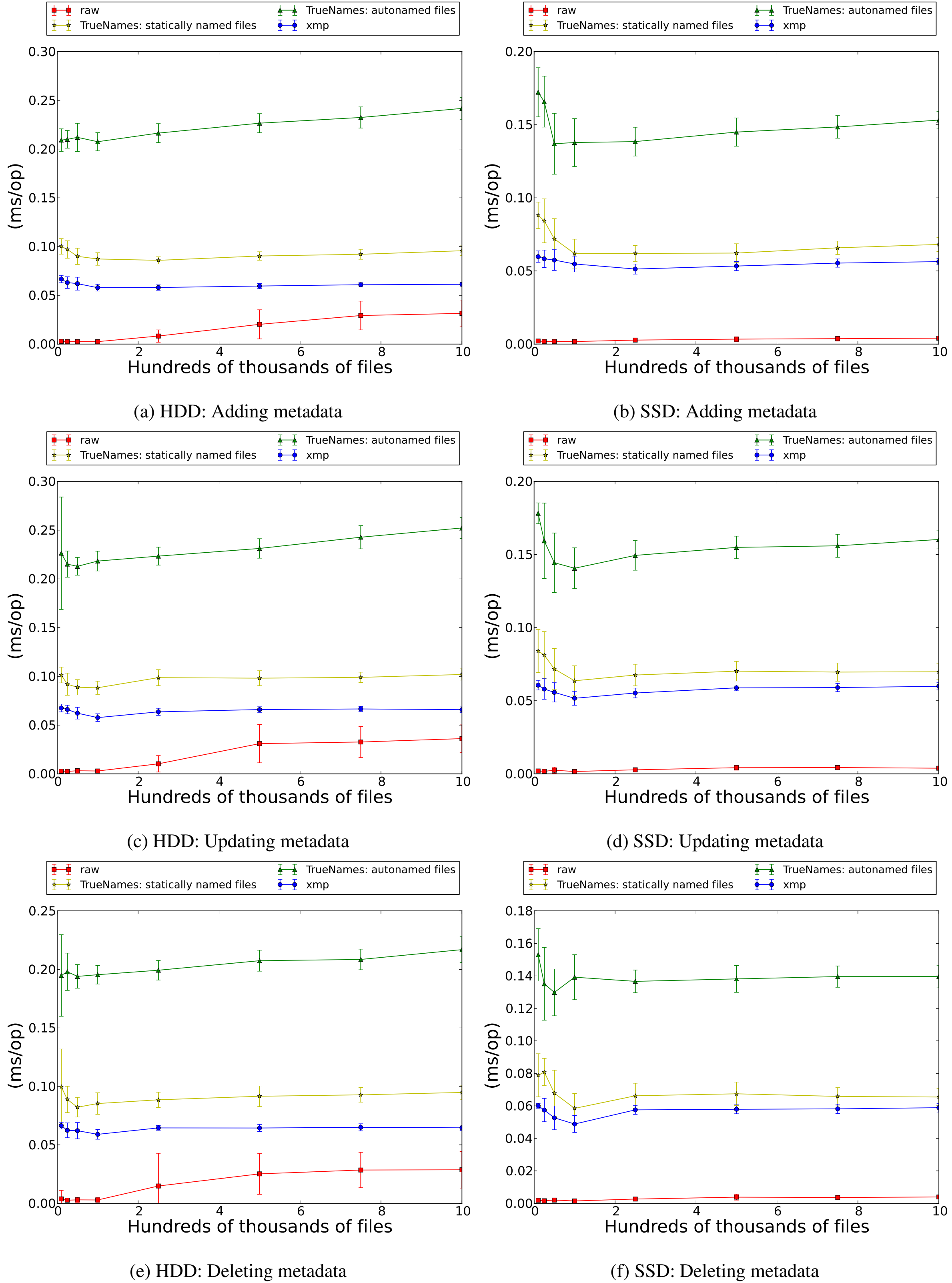
We ran this benchmark for both automatically named file types, and files which were not automatically named, in order to quantify the overhead incurred both with and without using the new features. We ran the metadata add, update and delete benchmarks for file sets from 10,000 to 1,000,000 files, which is comparable to modifying every file on a modern laptop at once. We then repeated each test forty times, in order to smooth noise and calculate a standard deviation.

Microbenchmarks are the most intensive tests, so it is unsurprising that they show the largest difference between the file systems. If we examine the difference between automatically named files and manually named files, as seen in Figure 2, we can see that automatic naming incurs a 100% penalty over metadata operations which do not affect the name, across all operations and both disk types. However, even at a 100% penalty, the additional cost can be measured in fractions of a millisecond. Looking at the difference between the baseline fuse system `xmp`, and TrueNames without name templates, we can see that there is approximately a 30% overhead, primarily due to checking every time if the file has a template, which requires retrieving extended metadata, and therefore both an additional kernel crossing and potentially a disk access. In total, TrueNames adds four extra kernel crossings per one file creation. The additional kernel crossings are a performance issue specific to FUSE, and would not occur in an in-kernel file system.

Even at this high operation rate, and for a million files, we can see from the latencies that the disk cache is rarely saturated. This performance will occur in a small fraction of operations, usually during file creation, and the additional latency will be masked under most normal workloads, as we discuss in the next section. TrueNames shows a fixed overhead without scaling bottlenecks up to a million files, making it suitable for fairly large workloads.

5.2 Macrobenchmarks

While micro benchmarks can be useful for setting an upper bound on performance, they are often an unrealistic assessment of how a file system will perform in practice. In the real world, file systems are experiencing a variety of operations from many different sources. In order to simulate the performance in a realistic environment, we chose a standard benchmark, Filebench [2]. This benchmark does not exercise the extended metadata functionality, and is therefore comparable to the statically named files experiments from section 5.1. We ran two different benchmark suites, the `fileserver` suite, which is designed to simulate the behavior of a typical file server, by performing a series of creates, deletes, appends, reads, writes and attribute operations on a directory tree. Mean directory size is 20 files, and the mean file size is 128kB. The workload generated is somewhat similar to SPECsfs [3]. We also ran the `createfiles` benchmark, which creates a specified number of files in a directory tree, with an average directory size of 100 files. File sizes are cho-



(a) HDD: Adding metadata

(b) SSD: Adding metadata

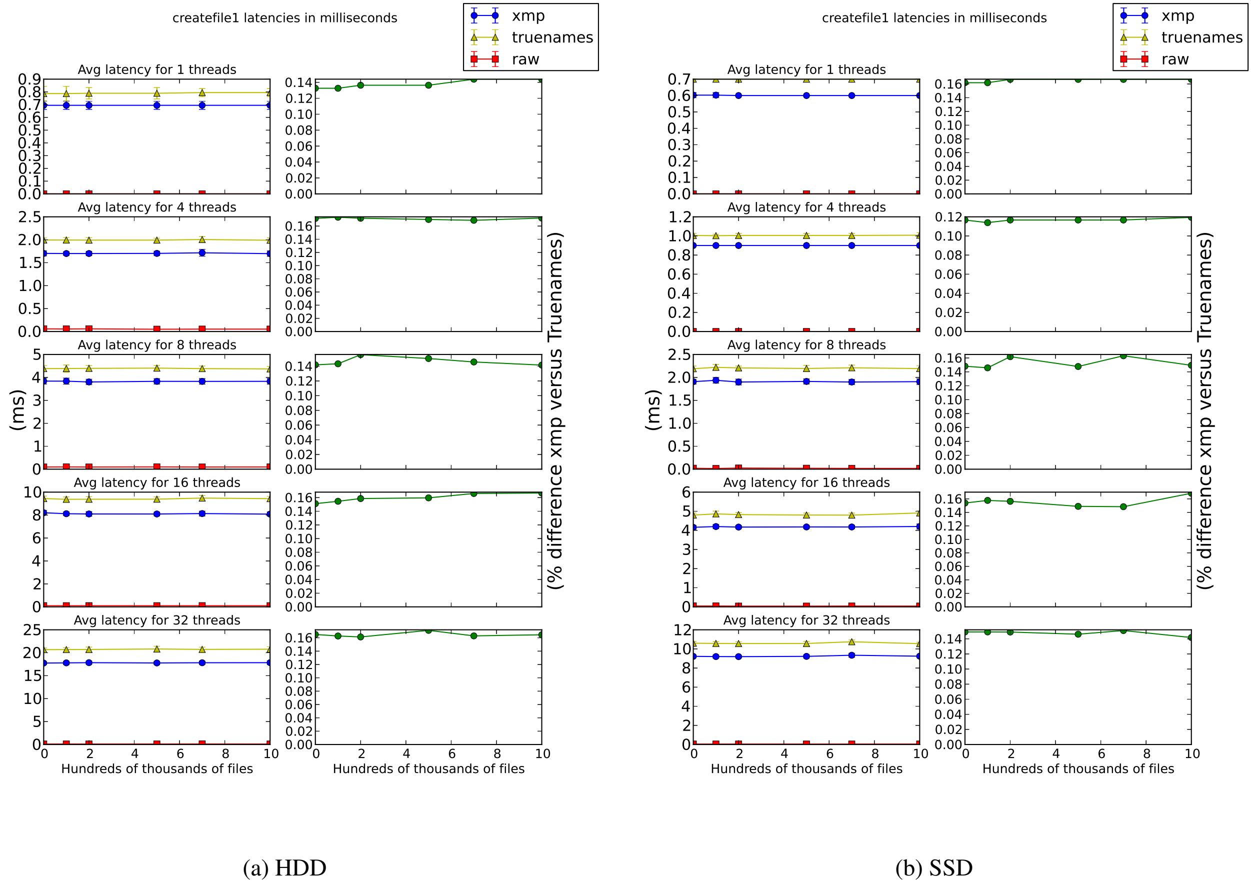
(c) HDD: Updating metadata

(d) SSD: Updating metadata

(e) HDD: Deleting metadata

(f) SSD: Deleting metadata

Figure 2: Latency and standard deviation per operation, for 40 runs of the extended metadata operations benchmarks. We tested TrueNames where the file either does or does not have an automatically named type, and on two baseline file systems, xmp and a raw ext4 system.



(a) HDD

(b) SSD

Figure 3: File creation times and standard deviation for 40 runs of the createfiles benchmark, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

sen according to a gamma distribution with a mean size of 16kB. We varied each of these from 1 to 32 threads, and from 100,000 to 1,000,000 files. We then repeated each test forty times, in order to smooth noise and generate a standard deviation.

In both Figures 3 and 4, which show file creation performance, TrueNames has performance that is highly comparable with that of `xmp`, at about 15% overhead. Other typical operations, such as writing a file, deleting a file, or calling `stat` on a file, had insignificant overhead, meaning that TrueNames is suitable for all but the most create-intensive workloads. This demonstrates that most of the fixed overhead of TrueNames is masked by normal operation latencies.

6. Related Work

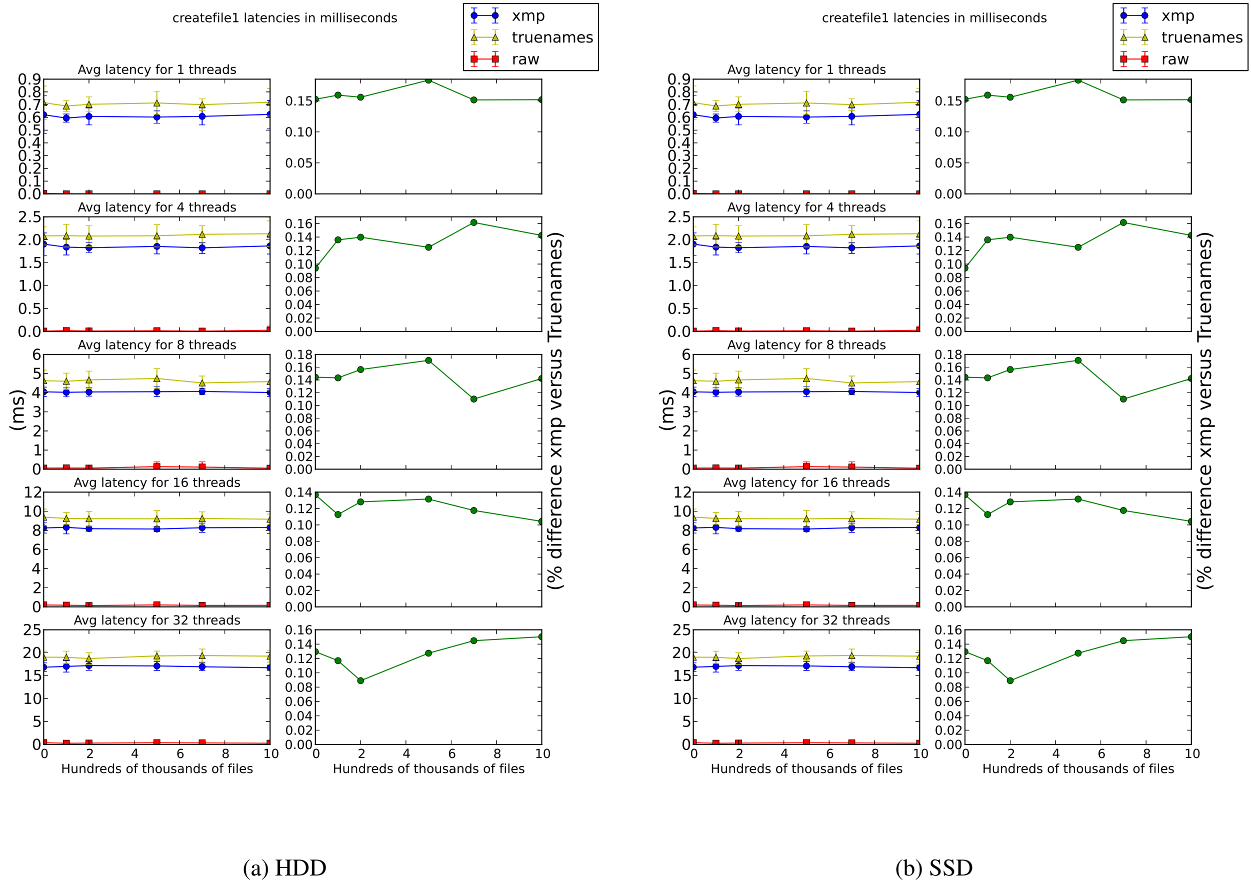
Automatically naming files is an under-explored area. The most similar areas of research are those of application-generated names, web search snippets, and non-hierarchical file systems, each of which we discuss.

6.1 Application-generated names

As discussed in Section 2, many applications such as iTunes [9], iPhoto [4], and Mendeley [6] currently generate file names, either for their own use or that of the user. However, none of them offers a generalized framework for file naming, and do not reflect outside changes to metadata. By contrast, TrueNames offers automatic naming as a service which any application can use, simplifying application development, keeping names synchronized with metadata regardless of source, and encouraging developers to export structured metadata.

6.2 Naming on the web

File naming can be thought of as analogous to the problem of disambiguating search results on the web. Web search, like file system search, has a huge number of files, (many with the same name, such as `index.html`), and when returning results the search engine must help the user choose between them. On the web, the historical assumption is that the user is retrieving textual information, and the common approach is to reveal a snippet from the page containing the search terms in context. Newer search types, such as video, rely



(a) HDD

(b) SSD

Figure 4: File creation times and standard deviation for 40 runs of the fileserver benchmark, on TrueNames and on two baseline file systems, xmp and a raw ext4 system. We also show the percentage difference between xmp and TrueNames.

on a title and key frame. However, in the file system arena, these approaches are not feasible. Many files are in opaque data formats, and no text snippet is available. Command line interfaces do not lend themselves to key frames, and many files do not offer them. Thus, we rely on file type-specific metadata to help the user identify their files.

6.3 Non-hierarchical and semantic file systems

The problem of naming directories has been a subject of interest for some time. In particular, semantic file systems [8, 11, 12, 14, 15, 20] present directory names based on the metadata of files, allowing the user to navigate and select files using their metadata. For instance, the original Semantic File System (SFS) [11] treats all directory names as queries. If the user enters a query containing an unbound field name (such as `user:`), SFS will return `/jones, /root, /smith` and so on as subdirectories, automatically generating names based on lists of attributes and values, or, in the case of the Linking File System [8], links.

Similarly, The Logic File System (LISFS) [15] establishes a *taxonomy* of attributes, such that some attributes subsume others. If the results to a query contain one or more attributes which are subsumed, only the higher level attribute will be displayed as a directory name, and only attributes which distinguish between the query results are shown. This is more similar to our disambiguation method. However, we rely on the template for a name, and add metadata only when required by a name collision.

However, none of these focus on the file names themselves, instead focusing on naming directories as a way to create queries over documents. Our approach is designed to complement non-hierarchical systems, allowing files to be easily recognized regardless of context, and allowing non-hierarchical systems to disambiguate file names if needed, by adding additional metadata.

The most similar work is to what we propose is that of Jones et al. [13], who proposed a non-hierarchical HPC file system with automatically generated file names, chosen by examining the distribution of metadata fields. By contrast, our work uses a more robust and less complex scheme which puts the user and application in control of which metadata is used, and allows them to select attributes which are most appropriate for the file's semantic type, rather than relying on statistical techniques.

Our work on collision detection and disambiguation also has implications for systems with very large directories, making it similar to systems such as Giga+ [16]. One of the challenges for scalable directories is being able to create a large number of unique names, and in future work, we will examine ways to make TrueNames scale to very large distributed directories.

7. Future Work

TrueNames was designed with non-hierarchical file systems in mind. In the future, we intend to extend TrueNames to a non-hierarchical context, where metadata is used to find files and display them. In a search context, TrueNames can help disambiguate search results, provide additional metadata, and potentially expand or collapse file names dynamically, based on a list of useful fields provided by the user. We are particularly interested in the potentials of object storage. We note that some minor modifications to the POSIX specification, allowing a user or application to simultaneously create a file and add metadata, and get a file name and file handle back as the return value, would be significantly simpler and more robust, reducing the possibility of collisions and improving performance. Bulk metadata addition would also improve performance and stability.

We would like to see wide spread adoption of a system like TrueNames. In the future, we plan to make it more robust, adding support for multi-threading and other performance and stability enhancements, as well as porting it to additional platforms. We will add support for templates on a per-user basis in addition to global templates. This allows users and applications to define more personalized templates, as well as providing a higher degree of privacy around metadata fields. Templates will be stored and accessed based on the `uid` of the calling application, and user templates will override global templates of the same name. We will also investigate ways of providing more flexibility for naming hard-linked files, such as associating templates with directories. We are considering ways of setting a default template for an entire directory, such as an images folder.

Additionally, we intend to explore the implications of using TrueNames in distributed file systems such as Lustre [10] and Ceph [19], where file names and extended metadata are stored on multiple servers, adding interesting new challenges around scalability and metadata management. Large scale systems for science can benefit greatly from better metadata and file name management.

8. Conclusions

Metadata aware file naming provides a set of abstractions for easily managing file names and metadata. It is designed to ease file management by automating the process of file naming, and giving files truthful, structured, and automatically updated file names. It also gently encourages users and applications to supply more searchable metadata. Using metadata aware file naming can not only benefit users and application developers in the short term, it can ease the migration path to non-hierarchical file systems and improve search. By separating unique system identifiers from human readable file names, we enable file systems and users to work together to manage data more effectively.

We have demonstrated the generality of metadata aware file naming by describing a variety of use cases in existing software which could be simplified by using our prototype filesystem, TrueNames. We have shown how it can simplify application development, reduce data loss, and make it easier for users and applications to find and manage files. Additionally, we have found TrueNames to be an extremely useful tool during the writing of this paper, and will be porting it to OS X in the near future in order to take advantage of it on more of our machines. Finally, we have shown that under a variety of workloads, TrueNames adds a minimal 15% overhead with the possibility of further optimizations, while adding useful new capabilities to the file system.

References

[1] Aperture. http://www.apple.com/aperture/what-is.html.

[2] Filebench. http://sourceforge.net/projects/filebench/.

[3] Filebench wiki: Pre-defined personalities. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Pre-defined_personalities.

[4] iPhoto. http://www.apple.com/ilife/iphoto/.

[5] Lightroom Help: The Filename Template Editor and Text Template Editor. http://helpx.adobe.com/lightroom/help/filename-template-editor-text-template.html.

[6] Mendeley add & organize. http://www.mendeley.com/features/add-and-organize/.

[7] Musicbrainz picard. https://musicbrainz.org/doc/MusicBrainz_Picard/Documentation.

[8] Ames, S., Bobb, N., Greenan, K. M., Hofmann, O. S., Storer, M. W., Maltzahn, C., Miller, E. L., and Brandt, S. A. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, May 2006), IEEE.

[9] Apple Inc. iTunes. http://www.apple.com/itunes/overview/, Jan 2010.

[10] Braam, P. J. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[11] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, Jr., J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (Oct. 1991), ACM, pp. 16–25.

[12] Gopal, B., and Manber, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proceed-*

ings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI) (Feb. 1999), pp. 265–278.

[13] JONES, S., STRONG, C., PARKER-WOOD, A., HOLLOWAY, A., AND LONG, D. D. E. Easing the Burdens of HPC File Management. In *Proceedings of the 6th Parallel Data Storage Workshop (PDSW '11)* (Nov. 2011).

[14] OLSON, M. A. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (San Diego, California, USA, Jan. 1993), pp. 205–217.

[15] PADIOLEAU, Y., AND RIDOUX, O. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 99–112.

[16] PATIL, S., AND GIBSON, G. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2011).

[17] SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., SUN MICROSYSTEMS, I., BEAME, C., LTD., H., EISLER, M., NOVECK, D., AND NETWORK APPLIANCE, I. Network file system (nfs) version 4 protocol. `http://www.citi.umich.edu/projects/nfsv4/rfc/rfc3530.txt`, April 2003.

[18] VINGE, V. *True Names*, vol. 5 of *Binary Star*. Dell, 1981.

[19] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2006).

[20] XU, Z., KARLSSON, M., TANG, C., AND KARAMANOLIS, C. Towards a semantic-aware file store. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)* (May 2003).