TagFS: A Fast and Efficient Tag-Based File System

6.033 Design Project 1
Yanping Chen
yanpingc@mit.edu
Dan Ports (TR11)
3/17/2011

1. Overview

A typical problem in directory-based file-systems is searching for groups of related files. Often times, target files are spread across multiple directories or have no common filename pattern. To address such issues, this document details the design of Tag File-System (TagFS or TFS), a non-hierarchical tag-based file-system: files contain descriptive tags and can be filtered based on such tags into scopes.
In order to make the design perform at a usable level, we focused on simplicity and speed:

1. Scopes are intersections of tags. By removing unions, scopes are made lightweight and fast, allowing applications multiple scopes with little cost.
2. Tag data is stored in a bidirectional mapping between tags and files. This increases storage but allows quick listing of a file's tag-list or a tag's file-list.
3. Tag data is fully stored in memory to facilitate very fast searches.

The goal of this design is to provide high performance through fast searches, allowing users and applications to quickly locate files needed.

2. Design Description

2.1 TFS Description

TFS consists of four main components: devices, files, tags, and scopes. Tags are tuples attached to files for organizational or descriptive purposes, while scopes represent group of files with a common tag-set.

2.1.1 Devices

A "device" in TFS is any data storage object. This could be physical, such as a CD, or virtual, such as a partition. A 64-bit device-id uniquely identifies each device.

2.1.2 Files

Files in TFS are arbitrary sequences of bits coupled with metadata and identified by 64-bit file-ids unique to each device – a file is uniquely referenced by a (device-id,file-id) file-tuple. File-metadata entries are summarized in Table1: while much of this metadata is unnecessary for TFS, they are ubiquitous to file-systems and hence included for application support.

Table 1: File-Metadata

Name	Datatype	Description
file-id	Long	Unique 64-bit file-id
created-time	Long	Time of creation
accessed-time	Long	Time of last access
modified-time	Long	Time of last modification
permissions	Long	Bits indicate file permissions
file-name	String(256)	Name of file
file-extension	String(32)	Extension of file
file-size	Long	Size of file, in bytes
tag-count	Long	Number of normal tags on file
tag-list	String list	List of tags on file

2.1.3 Tags

A tag is a string-tuple of the form (tag-name,tag-content) where tag-name and tag-content are UTF-8 strings of lengths 32 and 128 respectively. Two tags are "different" if they differ in tag-name or tag-content. Tags are like mini-file-descriptions: each file can have multiple different tags and the same tag can be applied to multiple files.

Table 2: System-tags

Tag-name	Tag-content	Default
file-name	Name of file	file-metadata:file-name
file-extension	Extension of file	file-metadata:file-extension

Some restrictions on special tag-names are listed in Table 2. Such system-tags can be implied from file-metadata, so every file has at least two tags. These system-tags allow TFS to interact with untagged read-only devices.

2.1.4 Scopes

A scope represents a set of files sharing a common set of tags (tag-set) and is identified by a unique 64-bit scope-id. Scopes are the main way applications access files, and all scopes are stored in memory but can be persisted to disk (see Section 2.2.3-4). For different usage cases, there are two types of scopes: dynamic and static.

2.1.5 Static-Scopes

A static-scope is simply a static array of file-tuples along with a tag-set. Their preferred use-case is fast random-access to an unchanging or "snapshot" file-list.

2.1.6 Dynamic-Scopes

A dynamic-scope is a tag-set along with a "lazy" iterator. Instead of keeping an updated list of file-tuples, a dynamic-scope utilizes an iterator: when the next file is requested, the iterator searches for the next file with a larger file-id satisfying the tag-set. As such, their preferred use-case is sequential access to large file-sets without the overhead of listing all files-tuples.

In addition to user-created scopes, there are also device-scopes and global-scopes, default dynamic-scopes to be used as source-scopes of searches: a device-scope contains all files on a device, and a global-scope contains all files in the file-system.

2.2 On-Disk-Storage Description

On-disk storage consists of the file-table, tag-tables, file-objects, and data-blocks. The file-table maps file-ids to file-objects, and the tag-tables is a two level map, mapping tags to corresponding lists of file-ids (file-lists). Their on-disk layout is demonstrated in Figure 1.

Figure 1: On-disk storage

2.2.1 File-Table and File-Objects

The file-table is a B+ tree where the keys are file-ids and values are pointers to corresponding file-objects. A file-object stores all the file-metadata and tag-list as described in (2.1.2) as well as a data-block-list.

2.2.2 Tag-Tables

There are two levels of tag-tables: the first level is the tag-name-table, a B+ tree with tag-names as keys tag-content-tables as values. The second level, the tag-content-table, is another B+ tree with tag-contents as keys and file-lists as values. Essentially, both levels together form a single map from a tag to a list of file-ids with that tag. Such a splitting speeds key comparisons and provides structure for future functionality.

2.2.3 Static-Scope

A static-scope is persisted by writing the file-tuple list to a file, as shown in Figure 2.

2.2.4 Dynamic-Scope

A dynamic-scope is persisted as a file containing the tag-set and current iterator position.

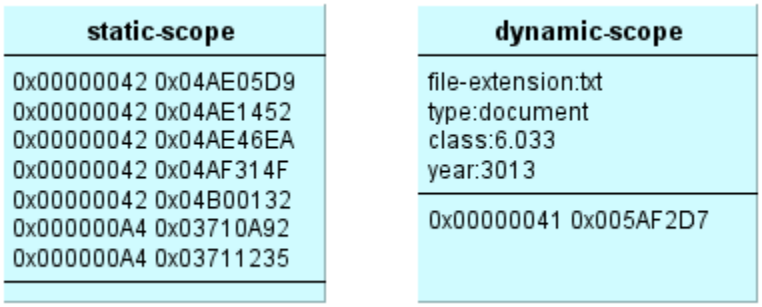

Figure 2: Example file formats for static and dynamic scopes

2.3 In-Memory-Storage Description

While most components of TFS are stored to disk, for performance reasons a lot of tag and file data is read into memory. In this section we detail caches used and scope storage in memory.

2.3.1 File-Cache
Since certain files might be accessed constantly, we can save disk reads using a file-cache hash-table mapping file-tuples to file-objects in memory. This allows direct access to a file's data-block-list or tag-list through memory-accesses.

2.3.2 Tag-Cache
The tag-cache is a two level hash-table in memory that consolidates the two level tag-tables of all devices. First, hash-tables map tag-names and tag-contents to unique 64-bit ids to conserve memory. Then, the tag-cache maps each (tag-name-id,tag-content-id) pair to a list of file-tuples in sorted order. In implementation, we store all tag-tables in memory, and whenever the OS detects a new device is added or removed, we update the in-memory tag-table appropriately: addition updates the cache eagerly, while deletion updates lazily – when files of disconnected devices are encountered, they are skipped and removed from the tag-cache.

2.3.3 Scopes
Scopes are stored in memory as previous described: a dynamic-scope is represented as a list of file-tuples in memory along with a tag-set, while a static-scope is represented as a tag-set and iterator. A hash-table, the scope-table, maps scope-ids to corresponding scope-objects.

2.4 API Description

2.4.1 Search
Search(source-scope-id,list-of-tags,destination-scope-id,[force-static])

This method searches from source-scope to destination-scope all files satisfying the given tag-list. Note that this is a replace-operation, not an append-operation, since scopes represent intersections of tags. This restriction provides performance and simplicity, and is futher discussed in Section 4.3.1.

Implementation wise, searches only rely on memory-accesses. First, the tag-set is copied from source to destination and updated with new tags. Then, for static-scopes we concurrently iterate through the file-lists of the scope and each tag, similar to a multi-list merge, copying all common file-tuples to the new scope (Figure 3). While for dynamic-scopes, we copy the iterator state, updating the current-file pointer if it is no longer in the scope by calling *get-next* on the iterator.

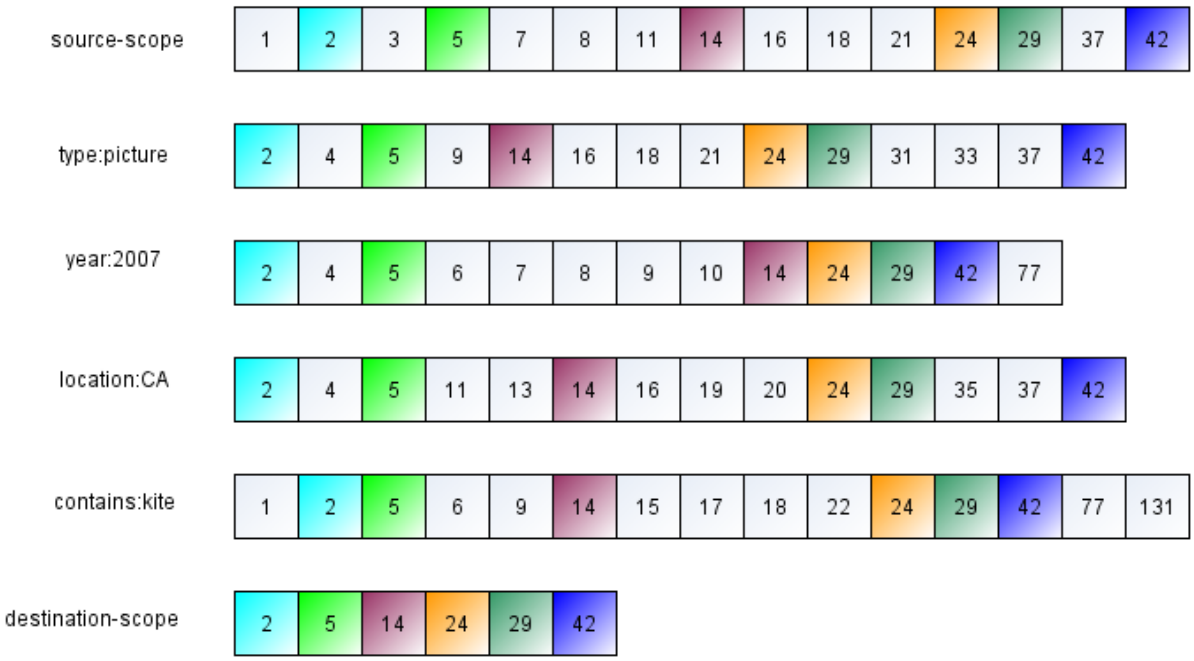

Figure 3: Example static scope search with 4 tags using file-lists only

The force-static option allows conversion of dynamic-scopes to static-scopes. This is achieved by a static-scope search from the global-scope using the entire tag-set.

2.4.2 File I/O

create(device-id)->file-id
delete(device-id,file-id)
open(device-id,file-id,flags)->file-descriptor-id
read/write(file-descriptor-id,buffer,bytes)->bytes-read
seek(file-descriptor-id,offset,reference)->bytes-seeked

Methods for file operations: *Create* creates a new file and returns its file-id while *open, read, write*, and *seek* act like file I/O methods from Unix, using file-descriptors. This provides a simpler and more standard interface for file I/O than arbitrary reads and writes.

2.4.3 Scope Operations

mkscope(type)->scope-id
deletescope(scope-id)
save/loadscope(scope-id,device-id,file-id)
list(scope-id)->file-tuple-list
get_file(static-scope-id,index)->file-tuple
get_next/previous/first/last/current(dynamic-scope-id)->file-tuple

Methods for manipulating scopes: *savescope* and *loadscope* are implemented by using file I/O to write or read scopes from files. *List* returns a list of the file-tuples: for static-scopes, this is a simple copy of the file-list in the static-scope-object; for dynamic-scopes, a new iterator is created to list all files currently in the scope. In addition to *list*, static-scopes also support *get_file*, which returns the i^{th} file in the scope, and dynamic-scopes support iterator commands. The

iteration method is implemented like the search operation of a static-scope, but stops on the first match (Figure 4).

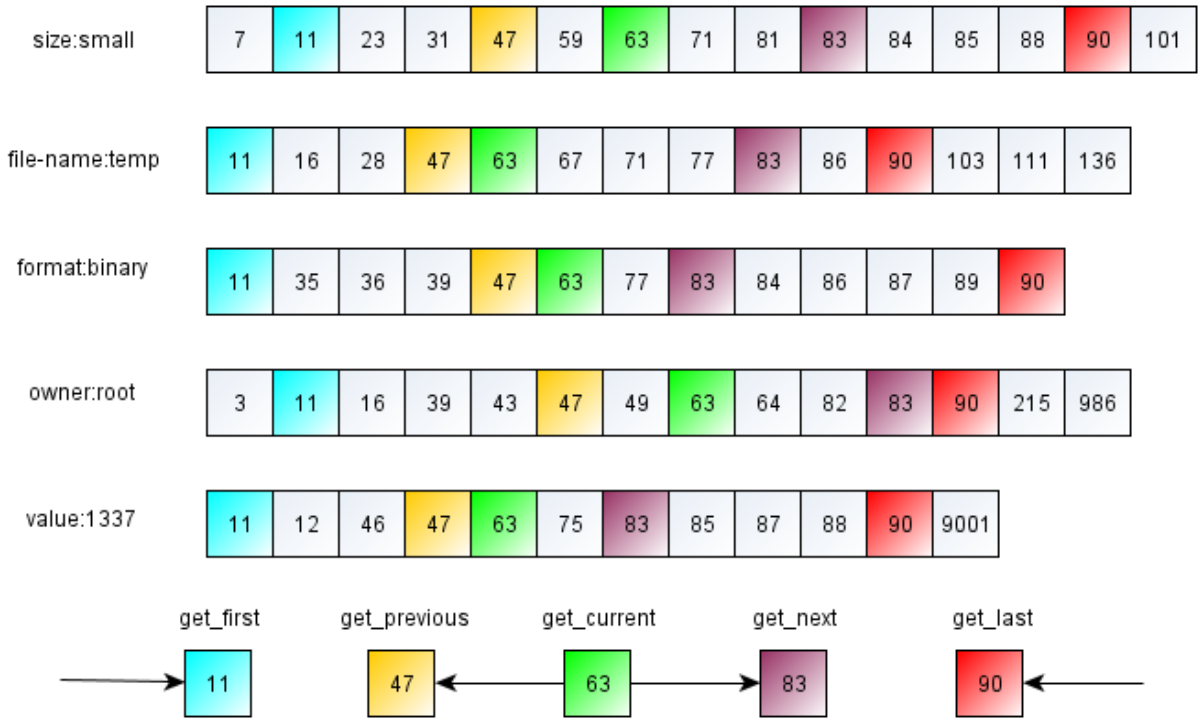

Figure 4: Dynamic-scope iteration, arrows indicate origin of iteration for each case

Here, care must be taken for two special-cases. First, *get_current* must check that the current file is still in the scope, since tag operations may have occurred. If not, use *get_next* to update the iterator. Second, special returns should be implemented to indicate the beginning or end of a dynamic-scope.

2.4.4 Tag Operations

tag_add/remove(device-id,file-id,tag-name,tag-content)

tag_get(device-id,file-id)

Methods to manipulate tags: adding and removing tags involves modifying the file's tag-list, the tag-tables, and the tag-cache; in addition, they must ensure system-tags are maintained. *Tag_get* returns a copy of a file's tag-list.

2.4.5 Other

device_list()

This method uses system functions to return a list of device-ids of connected devices.

3. Analysis

3.1 Usage Scenarios

Here we describe how two example applications, a photo-viewer and a shell, interact with TFS, demonstrating how we expect TFS to be used.

3.1.1 Photo-Viewer

The first application is a photo-viewer that displays pictures from all devices. Using a tag such as (“type”,“photo”), the photo-viewer creates a dynamic-scope from the global-scope. Then, it iterates through this scope, using file-descriptors to read each file. Upon reaching the end of the scope, application can restart with *get_first* and repeat. Similarly, to see a specific set of photos, the user can enter filter tags into the photo-viewer, which then creates the appropriate dynamic-scope to iterate over. This implementation is fully dynamic and has no slowdowns: during the display of one picture, the application can start searching for the next file – avoiding the overhead of a full list. Meanwhile, if tags are added or devices connected, they will automatically be part of the iteration loop.

To view photos on a read-only CD with no tags, system-tags can be used: when a device is connected, the OS reads the device and inserts tag information into the tag-cache, inferring system-tags from file-metadata. After this is done, the application can find photos by using a static-scope searching from the CD’s device-scope using tags based on extension, e.x. (“extension”,”jpeg”). If multiple extensions are requested, the photo-viewer can utilize multiple scopes, displaying them in order, since such read-only files only have one extension each.

Finally, to copy photos from the CD to a USB device, the application will first create static-scopes of various photo-file-extensions on the CD. Then, it will iterate over each list, reading each file and writing a corresponding file to the USB device. In addition, the user can specify tags such as (“Type”,”Photo”), to be added to copied files in order to aid future searches.

3.1.2 Shell

The second scenario is a user running python in a shell. The shell takes the user command, making a static-scope search for (“executable”,“python”) on device-scopes of system devices (the OS provides a list of system devices). If none are found, the search can be extended to other devices. If multiple programs are returned, there are three solutions: first, the shell can just run the first on the list, or use tags such as “version” to differentiate. Second, the shell can keep a configuration file detailing which version to run for each program. Finally, in the same configuration file, the shell can store a file-tuple pointing to the program that should be run for that command. We assume applications installers set appropriate “executable” and “version” tags.

Similarly, the python interpreter can find libraries through a static-scope search, for example: {(“type”,“library”),(“application”,“python”),(“version”,“2.6”)}. This should return a static-scope of python2.6 library files, which can then be further refined to find specific modules or pyc files.

Then, suppose the user wants to run a specific version: one solution is for applications to have redundant “executable” tags: python2.5 can have both the (“executable”,”python) tag and the

(“excutable”,”python2.5”) tag. Another method is to provide special shell syntax for specifying additional tags to filter by. For example, to run the default python program, the user types:
>python

While to run the second installation of python2.5 he types:
>python[“version”:“2.5”, “Install”:“2”]

Then, whichever version of python will make the appropriate static-scope(s) and proceed from there. This can be very powerful, since it allows multiple installs of the same version of python, separating their files using tags.

3.3 Storage Analysis

We analyze the amount of disk-space and memory required using the assumptions in Table 3:

Table 3: Storage assumptions

Harddrive capacity	500GB
Block size	32KB
Number of files	1,000,000
Tags per file	10
Unique tag-names	1,000
Unique tag-contents	5,000
Unique tags	100,000
Memory size	8GB
Number of static-scopes	1,000
Files per static-scope	10,000
Number of dynamic-scopes	1,000
Size of typical scope rule-set	10

3.3.1 Disk-Space Usage

Overall, the design uses less than 0.5% of the total disk-space, even with the high estimates for number of files and tags, as detailed in Table 4. A note of clarification: assuming a block-size of 32KB, 14,000,000 blocks references approximately 450GB of disk-space

Table 4: Disk-Space Usage

Category	Item	Size	Number	Cost
Files	Metadata	336 bytes/file	1,000,000	336MB
	Tag-list	160 bytes/tag	10,000,000	1,600MB
	Data-block-list	8 bytes/block	14,000,000	112MB
Tags	B+ trees overhead	200 bytes/unique-tag	100,000	20MB
	File-list	8 bytes/file	10,000,000	80MB
			Total:	2,148MB

3.3.2 Memory Usage

Shown in Table 5, a 1MB file requires 0.75KB of memory. Since average file size is approximately 500KB and users are not likely to be manipulating a large number of very large files, we reasonably assume file-objects use 1KB/file in the file-cache. So, a 50MB file-cache stores a sizable 50,000 file-objects.

Table 5: Cost of 1MB file-object in file-cache

Item	Size	Number	Cost
File-tuple key	16 bytes/file	1	16 bytes
Metadata	336 bytes/file	1	336 bytes
Tag-set	16 bytes/tag	10	160 bytes
Data-block-list	8 bytes/block	32	256 bytes
		Total:	768 bytes

Next, the entire tag/cache requires about 160MB of memory (Table 6).

Table 6: Tag-cache memory usage

Item	Size	Number	Cost
Tag-content-id table	40 bytes/entry	1,000	.04MB
Tag-name-id table	136 bytes/entry	5,000	.68MB
Tag-cache overhead	16 bytes/unique tag	100,000	1.60MB
File-list	16 bytes/file/tag	10,000,000	160.00MB
		Total:	162.28MB

Finally, consider scopes: we ignore the trivial overhead of the scope-table and instead focus on the memory cost of scopes. As the tables below dictate, even large static-scopes and complicated dynamic-scopes are quite lightweight, so applications can have over a hundred scopes without using much memory – 50 static-scopes of 1,000 files each and 50 dynamic-scopes only use about 1MB in total.

Table 7: Cost of Large Static-scope

Item	Size	Number	Cost
Tag-set	16 bytes/tag	10	~0MB
File-list	16 bytes/file	100,000	1.60MB
		Total:	~1.6MB

Table 8: Cost of Large Dynamic-scope

Item	Size	Number	Cost
Tag-set	16 bytes/tag	20	0.320KB
Iterator overhead	256 bytes	1	0.256KB
		Total:	~0.5KB

Overall, with a 50MB file-cache and full tag-cache, TFS takes a reasonable 200MB of memory and applications use less than 1MB of memory for scopes.

3.2 Time Analysis

For time analysis, consider the following timings [1], [2]:

Table 9: Timing of Various Operations

CPU cycle	0.1ns
Memory access	10.0ns
SSD access	100,000.0ns
HDD seek time	10,000,000.0ns

As shown, the CPU is never a bottleneck when accessing TFS. Instead, it falls down to the number of memory-accesses and disk-accesses. While the original problem specifies SSDs (Solid-Sate-Drives), HDDs (hard-drives) are included to demonstrate the versatility of our design.

3.2.1 Searches

As described in Section 2.4.1, searches only require memory-accesses. We will analyze the following worst-case scenario:

- 20 tag intersection
- Global-scope as source
- Each tag contains 50,000 files
- Matching rate of 0.00001 (10 files)

For static-scopes: we iterate through all tags' file-lists concurrently to take the intersection, requiring a read of 20*50,000*16 bytes = 160MB of memory. Using DDR3 peak transfer speeds of 12,800MB/s, this requires 12.5ms.

For dynamic-scopes, scope construction takes negligible constant time. Instead, we focus on iteration: taking an average over the 10 files in the scope, we see that we will need to read 16MB of memory per *get-next*. Using the same time analysis, this requires 1.25ms per *get_next*.

Overall, this worst-case is handled very well. In more general usage cases of smaller source-scopes, smaller tag-sets, and higher matching rates, we expect search times to be at the 0.1ms range, and *get_next* times to be near the 0.001ms range.

3.2.2 Loading Time

Finally we consider the time it takes to load the TFS data-structures into memory. This requires reading the tag-table and file-objects (to infer system-tags) of each device, evaluating to a disk-read of about 500MB, a reasonable one-time overhead for connecting a 500GB device. Since the tag-tables and file-objects will be stored in some contiguous chunk of disk-space, we see that hard-drives do not suffer too much in comparison to SSDs in terms of overall loading times.

3.4 Tradeoffs

3.4.1 Performance and Simplicity vs Additional Functionality

A major tradeoff of design is the removal of unions from scopes. However, this is very much justifiable based on the End-to-End principle: eliminating unions makes scopes very lightweight and fast, removes complexities of rule-sets, and simplifies implementation. As such, applications can have numerous scopes and run many searches at low cost. Not only that, but implementation is simplified greatly without the need for complicated rule-sets that require reduction for efficient evaluation.

Furthermore, it makes sense for the application to implement union since it knows more about scope content and can more efficiently create unions, if at all: some may not care about duplicates, while others work with mutually exclusive scopes, and hence requires no union. Finally, applications can process unions just as fast as TFS by accessing file-lists in memory.

3.4.2 Performance vs Storage

The bi-direction mapping between tags and files provides a huge increase in performance with a minor increase in storage. A bi-direction mapping is really quite required: one needs both fast access each tag's file-list and each file's tag-list, and there is no want to accomplish this with only file-lists or only tag-lists.

Similarly, the cost of storing the entire tag-cache in memory is easily offset by the tremendous speedups from being able to search with only memory-accesses: the 160MB of memory used is well worth the several orders of magnitude speed up in searches.

4. Conclusions and Future Work

Overall, this design efficiently implements a tag-based, non-hierarchical file-system using a bidirectional mapping between files and tags. It achieves great performance in searches by storing a tag-cache in memory, requiring only a minor loading overhead on system-boots or device-attachments. Additionally, the lightweight nature of scopes means applications are not limited in number of scopes used. Finally, the system scales well to larger disk-space, assuming similar increases in memory-capacity, since a minute proportion of storage is used (0.5%~2%).

4.1 Possible Problems

The slowest component of this design is iteration over dynamic-scopes with extremely large tag-sets. While such operations are still relatively fast at 0.1ms~1.0ms, this may be slow for applications requiring fast iterations over multiple complicated dynamic-scopes. However, this is a border usage case with no good solution: while an update model, where dynamic-scopes are evaluated eagerly and updated, somewhat fixes this issue, it incurs large overheads in file and tag operations, each of which must update all dynamic-scopes.

4.2 Future Work

One major area of future study is the incorporation of separate union methods inside TFS through system libraries. In reality, unions of static-scopes are not very costly, but were not included in order to provide a consistent interface and eliminate complications of rule-sets. Efficient unions of dynamic-scopes is a harder problem, but iteration should be achievable in time proportional to rule-set size through rule-set reduction and smart concurrent iteration.

Another possible improvement is to increase scalability with additional hash-tables in memory to map ids to smaller than 64-bits. Any reduction in id-size results in a proportional decrease in memory required, allowing addressing of more disk-space.

5. Acknowledgements

I would like to thank Joe Colosimo for the simplifying idea of limiting scopes to intersections.

6. References

[1] (2011, Mar.). Solid-state drive [Online]. Available: http://en.wikipedia.org/wiki/Solid-state_drive#Comparison_of_SSD_with_hard_disk_drives

[2] (2011, Mar.). DDR3 SDRAM [Online]. Available: http://en.wikipedia.org/wiki/DDR3_SDRAM#JEDEC_standard_modules

Word Count Total: 3315 (includes 494 from tables, captions, and headings)