Designing a Fast File System Crawler with Incremental Differencing

Tim Bisson
NetApp Inc.

Yuvraj Patel
NetApp Inc.

Shankar Pasupathy
NetApp Inc.

ABSTRACT

Search engines for storage systems rely on crawlers to gather the list of files that need to be indexed. The recency of an index is determined by the speed at which this list can be gathered. While there has been a substantial amount of literature on building efficient web crawlers, there is very little literature on file system crawlers. In this paper we discuss the challenges in building a file system crawler. We then present the design of two file system crawlers: the first uses the standard POSIX file system API but carefully controls the amount of memory and CPU that it uses. The second leverages modifications to the file systems's internals, and a new API called SnapDiff, to detect modified files rapidly. For both crawlers we describe the incremental differencing design; the method to produce a list of changes between a previous crawl and the current point in time.

Categories and Subject Descriptors

D.4.3 [**Software**]: Operating Systems—*File organization*

Keywords

NetApp, indexing, search, crawling

1. INTRODUCTION

A file system crawler is a piece of software that walks down a file system tree and gathers information about each subdirectory and file within that tree. This information is typically the full path name for each file and subdirectory, and optionally file metadata (the `stat` call). Any application that enables users to search the contents of a file system depends on a file system crawler to gather the list of files to be indexed. Besides file system search, storage resource management (SRM) applications and source code cross-reference tools such as cscope, depend on efficient file system crawlers to generate the list of files that need to be indexed. A fast, and resource-efficient file system crawler is critical for the performance of such applications.

While there already exists a large body of research on web crawlers [2, 3, 6, 12], we are not aware of any prior published work that focuses on building a fast file system crawler. Web crawling usually focuses on exploiting the link structure of web pages across geographically dispersed sites; FS crawling is focused on quickly traversing deep trees on a single physical machine.

Enterprise search engines such as the Google Search Appliance [11] and Kazeon [1] need to build an index of file system contents to search against. A file system crawler provides the data for such indexes by crawling the contents of a network-based file system.

The speed with which a crawl completes, dictates how soon data or metadata can be indexed for search. Because network file systems' contents change constantly—users add and delete files, admins back-up files, and so on—any index of the data must be periodically updated to handle file additions, modifications, and deletions. Therefore, a file system crawler must be able to produce an incremental list of changes between two points in time (*incremental crawler*) as well as a base list of all files within a file system (*baseline crawler*). The speed of the baseline crawler determines how soon one can index a file system's contents for the first time. The speed of the incremental crawler determines how quickly an index can be updated, thereby providing search results that are less stale.

Despite operating at high speed, a file system crawler should not hog the CPU or memory on its host. Very often the file system crawler runs on the same system on which the indexing infrastructure runs. The latter is often CPU and memory intensive, and one cannot afford to hurt search performance with a resource-intensive file system crawler.

The typical speed for file system crawlers we've observed with commercial search engines is in the range of 600 to 1,500 files/sec. The hardware for such search engines is typically a dual-processor Intel® or Opteron blade, with 4GB of memory. This translates to 18 to 36 hours of crawling for a 100 million file data set. A petabyte scale storage system can often contain a 1/2 billion files, which implies crawl time on the order of weeks.

One of the problems associated with periodically crawling the entire file system is that the cost is often disproportionate to the file system's rate of change. Even if there are only a few files modified between two points in time, a crawl of the entire file system at the new point in time is needed to determine what has changed.

The goal of this paper is to present the design of two file system crawlers: one that uses the standard file system APIs, and a faster one that leverages modifications to the underlying file system.

1.1 Terminology

This paper refers to *baseline* and *incremental* crawls, which are defined as follows:

- **Baseline crawl**: A baseline crawl of a file system is a recursive traversal of a given file system tree, which produces as output metadata about all files and directories in that file system.
- **Incremental crawl**: An incremental crawl produces as output, metadata about all files and directories that have been modified, added, or deleted since some previous point in time. Therefore, an incremental crawl is always with respect to a previous baseline or incremental crawl.
- **Metadata**: In this paper, the output of a baseline crawl or incremental crawl is file metadata, which can contain the following:
    - ○ Inode number: unique id for the file within the file system
    - ○ M/C/D: This inode was modified, created, or deleted since the previous crawl
    - ○ Full path name
    - ○ File extension, File Size
    - ○ Soft or hard link
    - ○ File or directory
    - ○ user/group id
    - ○ hard link count
    - ○ Modified time, Access time, and Change time

2. A FAST FILE SYSTEM CRAWLER

2.1 Generating Baseline File System Crawls

In this section we present the design of a fast file system crawler to generate a baseline crawl, followed by a performance evaluation. While designing the crawler, we used the following design principles as guidelines:

- The crawler should be able to traverse any file system accessible via the standard POSIX interface, NFS or CIFS.
- The crawler should be capable of producing both a baseline crawl and an incremental crawl.
- The crawler should efficiently manage resources on the host on which it runs. In particular, the crawler should be able to control the amount of CPU and memory it uses on the host.
- The crawler should not perturb the performance of the storage system it is crawling.
- The crawler should be able to gather metadata (`stat`) for all files and directories when run as *root* or as the appropriate privileged user.

2.1.1 Basic Single-Threaded Crawler

Before describing our file system crawler, we discuss how existing file system crawlers work. A good example is python's file system crawler, `os.walk()`. It is a single threaded function that starts at the root of a file system hierarchy. The crawler recursively gathers and reports metadata information about its subdirectories and files. By default it uses a depth-first approach, listing all the directory entries for the current directory being processed before proceeding to the next directory.

Every traversal of a directory involves a call to `readdir()`, which lists its directory entries. For each entry, a `lstat()` call is issued, which returns the metadata associated with that file or directory. Each of these system operations translates into network operations when the file system being crawled is mounted over a network protocol, such as CIFS or NFS. These network operations can incur a significant latency, reducing a file system crawler's performance.

Our baseline crawler follows the same principle, traversing down the file system tree collecting metadata. However, our crawler is multithreaded and attempts to reduce memory consumption when necessary.

2.1.2 A Fast File System Crawler

Our hypothesis for a fast file system crawler is simple: by parallelizing the crawl, we can improve performance.

The work of gathering all metadata for the file system tree is offloaded to multiple threads. There is a global queue that contains a full path of directories to crawl. Each thread does work by pulling a directory off the queue, performing a `readdir()` call on the directory, followed by an `lstat()` for each directory entry. Each entry's metadata is appended to a buffer associated with the thread. When the thread encounters a directory entry that is also a directory, the thread adds the directory's full path to the global queue. Note that traversing a soft link (that is, a symbolic link) pointing to a directory can lead to a cycle. Therefore, we don't add directory entries that point to a directory through a soft link to the global work queue. After a thread finishes processing the current directory, it appends the buffer contents to an output file associated with the thread and clears the buffer. The thread then pulls another directory off the global queue and repeats the above process.

When the the global queue is empty and all threads are idle, the file system tree has been completely crawled. The master process then appends each thread's output file to a final output file, which contains the results of a complete baseline crawl.

2.1.3 Controlling Memory and CPU Usage on the Host

A key design point for the file system crawler is resource utilization, in particular the CPU and memory utilization of its host. If I/O is the performance bottleneck, we can coarsely control CPU utilization by adjusting the number of threads used to crawl the file system.

The major memory consumption in the crawler belongs to the global queue, which contains the full path names of directories that need to be crawled. The queue's default size is 1GB, but a user can adjust the queue size when starting the crawler. Memory allocated by the queue occurs at runtime and depends on two factors: the file system tree layout and the number of threads. For example, if the file system is many levels deep but has only a few entries per subdirectory, the queue does not consume very much space if multiple threads perform the crawl—the threads can pull directory paths from the global queue as fast as they are being added to the queue.

There are two water levels in the queue: a minimum and maximum threshold. When the maximum threshold is reached, a portion (50% by default) of the queue is swapped out to a file, reducing the queue's memory consumption by a corresponding amount. Once the queue size reaches the minimum threshold, queued directory entries from the swap file are reinserted into the queue. The minimum threshold ensures that there will always remain enough entries in the queue for threads to perform their metadata-gathering operations without having to block while queued entries are swapped back into memory.

At the onset of a crawl, the traversal algorithm is a breadth-first walk; each thread inserts the full path name of discovered directories to the end of the queue. This implies that top-level directories are processed before subdirectories. However, because threads can pull and insert entries from and to the queue at different rates, it is not a pure breadth-first approach.

There is also a pseudo depth-first traversal, which occurs when queued entries are swapped out of memory. In this approach, all threads insert the full paths of discovered directories to the head of the queue. It is not a pure depth-first traversal because the current directory is processed before it is added to the global queue (that is, it doesn't have the recursive nature of a pure depth-first traversal). Note that we transition back to a breadth-first traversal when the minimum threshold is reached and queued entries are swapped back into memory.

There is a potential trade-off between the two algorithms. A breadth-first traversal can run faster if a file system groups directory entries together because the breadth-first approach attempt to process entries in the same subdirectory together more often than does a depth-first approach. However, a breadth-first approach consumes more queue space because it processes deeper queue entries (longer full paths) paths before shallower queue entries (shorter full paths).

For example, Figure 1 shows a simple directory tree consisting of five directories and the corresponding entries in the queue for both a breadth-first and depth-first traversal as a given directory is being processed. In the breadth-first traversal, the longer full-path directory entries build up in the queue, resulting in more memory consumption.

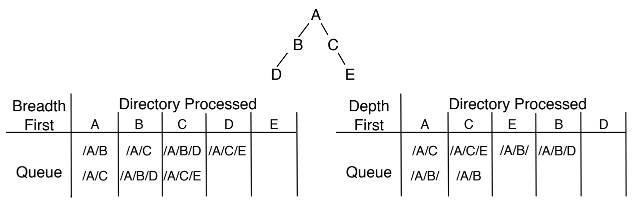


Figure 1: Queue size example: breadth-first versus depth-first.

2.1.4 Experimental Setup

Table 1: Datasets

Dataset Name	Num files	Mean Tree height	Mean Subdirectories
Dataset1	3,990,603	10	10
Dataset2	4,021,967	6	20
Dataset3	3,692,594	10	20
Dataset4	3,455,304	20	5

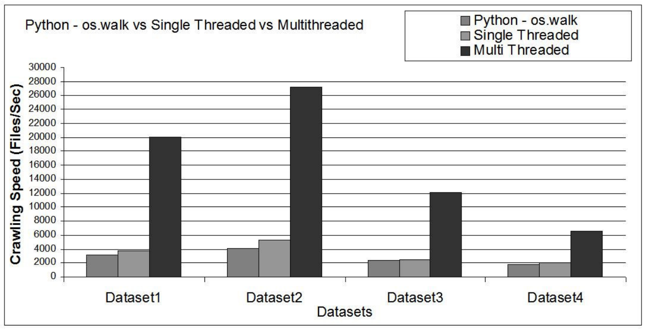


Figure 2: Comparison of multithreaded versus single-threaded crawler.

We measured the performance of our file system crawler on different datasets to asses the impact of our design decisions. The four datasets we used were artificially constructed by using DataManip [14], which generates pseudo-random datasets. DataManip provides several configurable parameters to control the created dataset. In particular, we varied the height of the tree and the number of entries per level to produce both *deep* and *wide* file system hierarchies. DataManip creates the files and directories in a depth-first manner, but it creates all files and directories in a directory before proceeding down the tree. Table 1 shows the number of files, mean tree height, and mean number of subdirectories for each dataset.

The host we used for our experiments was an AMD Opteron 248 (2 processor) with 4GB of memory. The file systems reside on a FAS3050 filer running the NetApp ® Data ONTAP ® 8 operating system. The aggregate hosting each file system was 1TB in size and comprised 11 SATA drives. The file server and the host machine reside in the same subnet.

In order to measure CPU and memory utilization during a file system crawl, we periodically polled memory and CPU utilization (every three seconds), then took an average of the polled values.

2.1.5 Performance

Our first performance results compare the standard python-based crawler (`os.walk`) to a single-threaded and multithreaded (16 threads) version of our C-based crawler. These experiments are shown in Figure 2. For all datasets, the python crawler consumed 30 to 33% of CPU and roughly 3MB of main memory, and our single-threaded crawler consumed 10 to 13% of CPU and 12MB of main memory for all datasets. The multithreaded crawler used 60 to 65% of CPU and 10 to 15MB for dataset1, dataset2, and dataset3. CPU utilization dropped to 25 to 30%, and the memory use increased to 30MB for dataset4 because of its thin and deep hierarchy.

In Figure 3 we show how varying CPU and memory resources affect performance of the baseline crawler. Figure 3(a) shows crawl performance as a function of the number of threads. We see that beyond 16 threads, crawl performance did not improve. The average crawl speed with 16 threads varies from 10,000 to 25,000 files/second depending on the dataset. Crawl performance of dataset4 is poor because there is very little parallelism to exploit, again due its file system hierarchy. We repeated these experiments with a 4-core AMD Opteron machine with 8GB of memory and observed a maximum performance of 30,000 files/second when the number of threads reached 20.

We also analyzed the trade-off of switching between depth-first traversal when we swap out (labeled as Mixed in figure's legend) and continuing to use breadth-first traversal when we swap out (labeled as BFS-Swap in the figure's legend). Figure 3(b) shows crawling performance as a function of the queue's maximum memory allocation size for both traversal schemes. We can see that even in an memory constrained environment, performance does not become degraded.

Figure 4 shows the performance of BFS-Swap, DFS, BFS, and Mixed with respect to crawling performance, host CPU utilization, host memory utilization, and filer CPU utilization. BFS and DFS have unlimited queue space and correspond to a breadth-first traversal and depth-first traversal, respectively. These figures show that the BFS strategy has the best performance, with the lowest CPU utilization but with the most memory consumption; BFS is able to exploit locality of reference because inodes in a directory are located adjacently on disk. Figure 4(d) shows that file server CPU utilization varies very little with respect to the host-side traversal algorithm.

2.2 Generating Incremental File System Crawls

The goal of an incremental file system crawler is to generate the list of files and associated metadata that has changed between two points in time. These two points in time might be represented by snapshots, as in the NetApp ® Data ONTAP WAFL ® filesystem.

The baseline file system crawler described in the previous sections is very fast. However, the order in which results are returned is nondeterministic because the crawler is multithreaded and its partial results are not sorted. To be clear, the results returned are always the same for a given point in time —it is only the order of entries that changes across runs.

Because our incremental crawler relies on sorted file lists (sorted by inode number) to generate the differences, we start by describing the modifications we made to the baseline crawler to produce sorted output across each of its runs.

2.2.1 Generating a Sorted Crawler Output

We made four fundamental changes to the baseline crawler. First, a thread's output buffer is always written out to a unique file; a thread in the original baseline crawler always appended the output buffer to the same file. Second, each entry in the in-memory output buffer is sorted by inode number before being written out to a unique file. Third, each output buffer is written out to a new file only when it is completely full. Naturally, larger buffers mean fewer files are generated. Finally, the unique files are then mergesorted into a single file sorted by inode number.

To ensure uniqueness, a thread's id and iteration number make up each file's name, using the format `threadnum_iteration`.txt. For example, when Thread2 writes its output buffer to a unique file for the third time, it will write to a file with the name `2_3`.txt. Writing each sorted output buffer out to a unique file provides the foundation for creating a deterministically sorted baseline crawl.

Once these sorted files are generated, we merge them into a single file, which is done in parallel. For example, eight files are sorted by four threads. The resulting four files are merged by two threads into two files, and so on, down to one file. Note that this is a customized version of mergesort.

Unfortunately, we cannot always completely read a merged file into memory before merging it with another file; the size of a merged file is the sum of the two files it was merged from, so the size of merged files increases exponentially after each merge. Therefore, each file merged with another file has a read buffer associated with it, which is used to temporarily store the file's entries as the two files are merged.

We chose to restrict the total memory consumed by the merging process to 1GB and use a read buffer size of 1MB. Therefore, from the following equation, we can have at most 500 simultaneous merge threads running, $1GB \div (2 files\ per\ thread \times 1MB) = 500$ threads. Our merge algorithm is described by the following pseudocode:

```
file_count = get_files(files);
file_count = min(file_count, 500);
while(file_count > 1){
    for (i=1; i<file_count; i+=2){
        start_merge_thread(file[i-1],file[i])
    }
    file_count = get_files(files);
    file_count = min(file_count, 500);
}
```

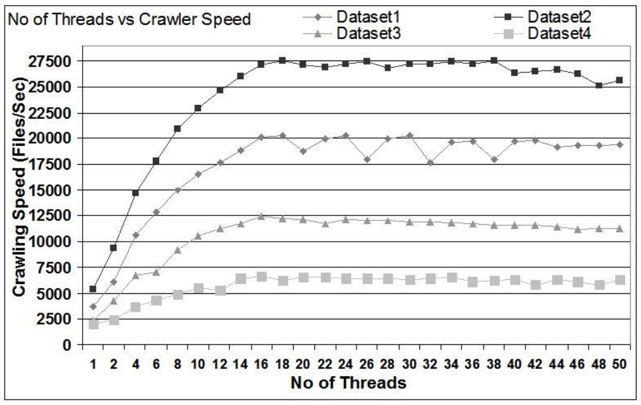
To reiterate, we have introduced two new parameters that affect incremental file system crawling performance:

- **Output Buffer**: This is the memory buffer per thread dedicated to storing the metadata for each crawled entry. The bigger this buffer is, the more entries that can be sorted in memory before being written out to a new file, resulting in fewer files that need to be merged when the crawl is complete.
- **Read Buffer**: When an output file is read into memory during the merging phase, we store the entries read from the file in this buffer. As many entries are read into this buffer from the file as can fit into it. As the read buffer size increases, fewer read I/O operations are needed.

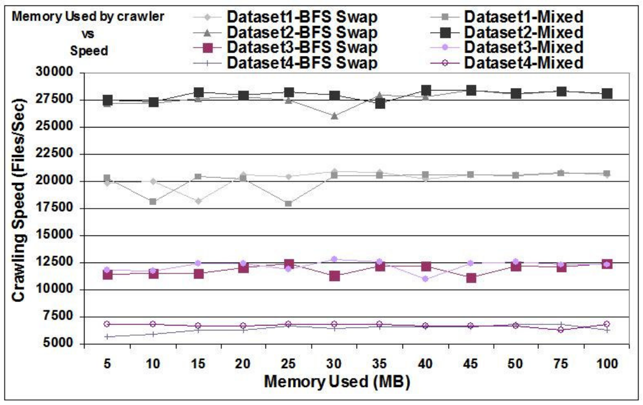
2.2.2 Generating Differences between two Sorted Crawls

Once the two sorted baseline files have been generated, each representing a view of the file system from some particular time, we can generate the incremental differences between the two versions by iterating through the two baselines and comparing the corresponding entries.

There are three possible cases for each comparison: if an entry is in the older baseline but not the the newer baseline, the corresponding file was deleted; if an entry is in the newer baseline but not the



(a) **Thread Impact**



(b) **Memory Impact (Queue memory size)**

Figure 3: Resource Impact

older baseline, the corresponding file was created; if an entry exists in both baselines and if the metadata is different, the corresponding file has possibly changed. The algorithm for generating the `diff` is shown in the pseudocode below:

```
file_base_ind = 0;
file_delta_ind = 0;
while ((file_base_ind < file_base_entries) ||
       (file_delta_inode < file_delta_entries) {
    if (file_base_ind == file_base_entries) {
        process_create(file_delta_ind);
        file_delta_ind++;
    } else if (file_delta_ind == file_delta_entries) {
        process_delete(file_base_ind);
        file_base_ind++;
    } else if (inode_num(file_base_ind) == inode_num(file_delta_ind)) {
        process_potential_mod(file_base_ind, file_delta_ind);
        file_base_ind++;
        file_delta_ind++;
    } else if (inode_num(file_base_ind) < inode_num(file_delta)) {
        process_delete(file_base_ind);
        file_base_ind++;
    } else {
        process_create(file_delta_ind);
        file_delta_ind++;
    }
}
```

`File_base_ind` and `file_delta_ind`, which represent an index into each of the baseline files. One is referred to as the *base file*, and the other is referred to as the *delta file*. The algorithm traverses the two baseline files by incrementing their respective file index variable. When it reaches the end of both baseline files, the algorithm has completed a `diff` of the two baseline files.

The first conditional case is encountered when the base file has been completely traversed, which means that any existing entries in the delta file represent creations. Alternatively, the second conditional is encountered when the delta file has been completely traversed, which means any existing entries in the base file represent deletions. If the two inode numbers for their relative indexes are the same, we check to see if the entry has been modified between the two versions by comparing their respective attributes. If any values change other than atime, we report that a modify has occurred. If the inode number for an entry in the base file is less than the inode number for an entry in the delta file, we have a deletion (because that entry doesn't exist in the delta file; otherwise the inode numbers would match). The `file_base_ind` variable is incremented because we have only processed an entry from the base file. If the inode number of the delta file was greater than the base index's inode number, we have a creation.

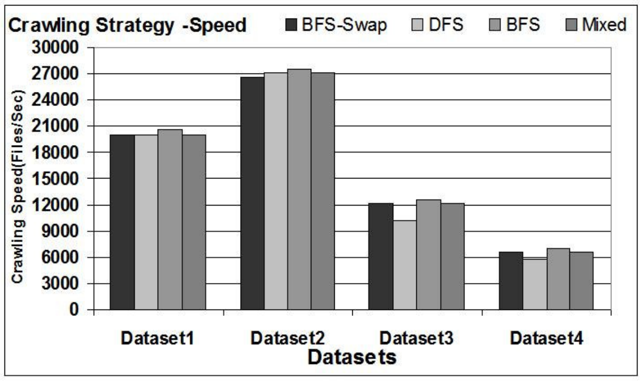
For each creation, deletion, or modification, we report the `stat` information for the corresponding entry, including the file's full path and a field stating whether the operation was a created, modified, or deleted. For modifications and creates, we report that `stat` info from the delta file. For deletions, we are limited to reporting the `stat` info from the base file.

2.2.3 *Handling Hard Links*

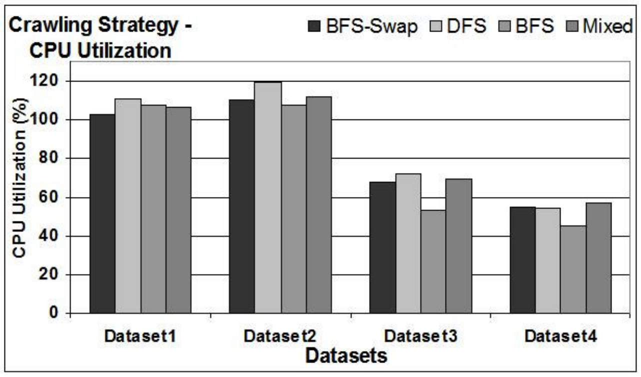
In addition to regular files, we also process files with hard links. Processing hard links requires additional work because the differencing algorithm sequentially scans two baseline files. The functionality to check for hard links is built into the `check for potential modify` phase.

As soon as the function to `check for a modify` is entered, we build up a structure with a name for each entry in the baseline file that has the same inode number we are trying to process. This is done by scanning through the next set of entries in the baseline file until an entry's inode number doesn't match the inode number we are processing. The structure contains the `stat` information for that entry, followed by a list of full paths. This is done for both baseline files. If the number of paths was just one in both structures, we perform a normal modification check, checking the appropriate attribute fields.

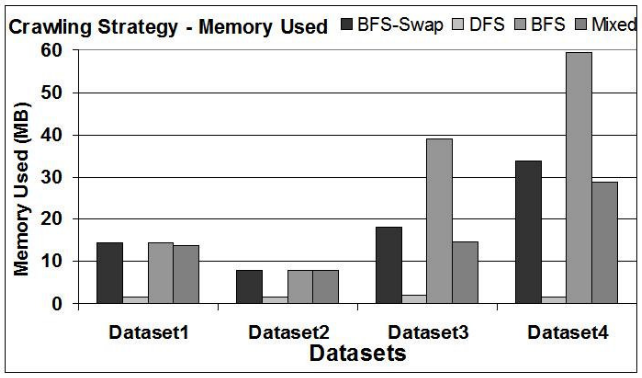
If the number of full paths is greater than one for either the base or delta file, we mark full paths that appear in both structures as unchanged. Those entries not marked in the base structure represent



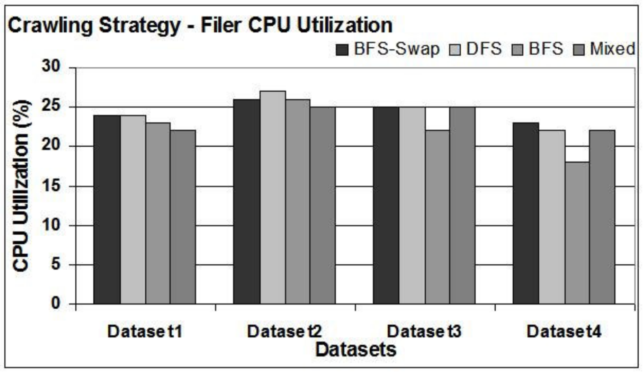
(a) **Throughput**



(b) **Host CPU Utilization**



(c) **Host Memory Utilization**



(d) **Filer CPU Utilization**

Figure 4: Baseline crawler performance.

deleted hard links, and those not marked in the delta structure represent created hard links. If there are no unmarked entries in the base or delta structure, and none of the `stat` info changed, we continue on to the next index in the respective baseline files.

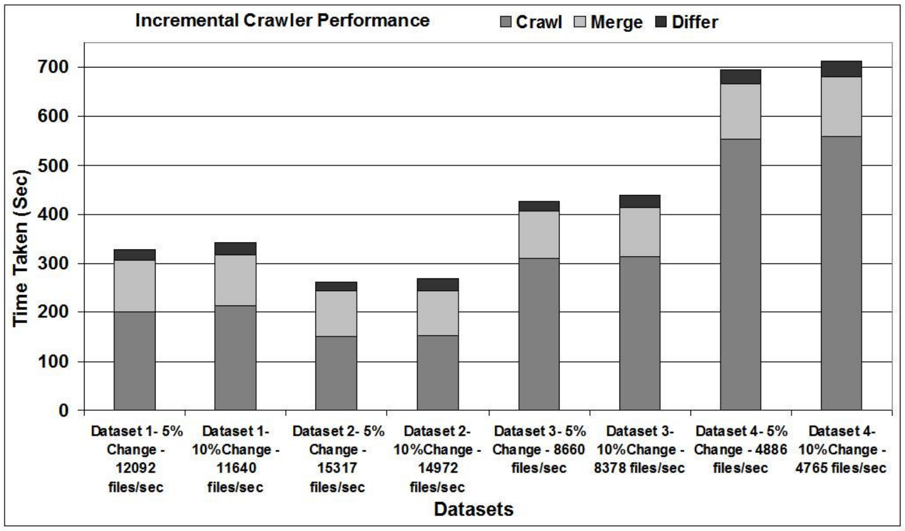
2.2.4 *Results*

We examined the performance of the incremental crawler using the same host, file server, and datasets from the baseline crawler performance analysis. We made changes to the datasets by modifying each dataset by 5% and 10%. Figure 5 shows the results with respect to crawling performance, host CPU, and memory utilization. In Figure 5(a), we have broken the time down into three phases: crawling, merging, and diffing. Note that the crawl time represents crawling one version of the file system because we retain a previous merged copy of the file system with which we perform the diff. The merging phase corresponds to merging the uniquely sorted files into a single sorted baseline file. The diffing phase refers to reading each baseline file sequentially while differencing the corresponding entries. This graph shows that merging and diffing take the least amount of time, which is because they result in the least amount of I/O—the crawler always has to go to disk to `stat` a file or read a directory's contents (unless it's cached). Although merging is O(n log n), we have reduced the recursion element because an output buffer size of 1MB can hold roughly 20,000 file entries—the base case isn't two entries as with a pure mergesort algorithm.

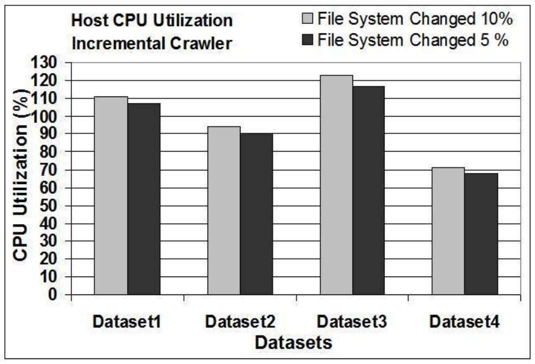
Figure 5(b) and 5(c) show the average CPU and memory utilization while the crawling, merging, and diffing. Because of the additional work needed to merge and diff the two baselines, both CPU and memory utilization have increased when the number of changes expanded from 5% to 10%.

2.3 Limitations

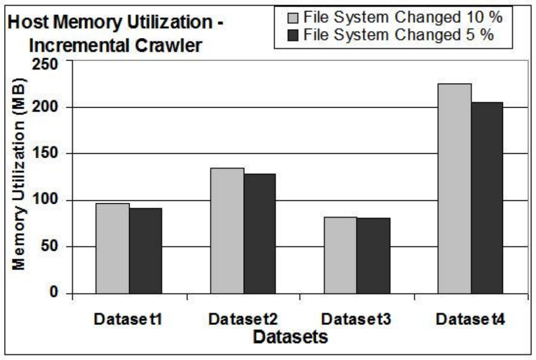
There is a key limitation to the crawler, and as a result to the incremental differences. We aren't guaranteed that the file system hasn't changed beneath us when crawling it. For instance, after we start a crawl, a file may be modified just after we `stat` it. Addition-



(a) **Throughput**



(b) **Host CPU Utilization**



(c) **Host Memory Utilization**

Figure 5: Incremental Crawler Performance

ally directories can be renamed, causing us to crawl them multiple times, or not at all.

Solving this requires holding a lock across the file system's contents, which is not practically feasible. Or we could use file system snapshots, as is done in the Data ONTAP WAFL ® filesystem. Since Snapshots are read only, we are guaranteed consistency while crawling a Snapshot copy.

3. SNAPSHOT BASED CRAWLER

The last few sections described how to implement a fast file system crawler using standard file system APIs. Modern file systems implement snapshots which are point-in-time views of an entire filesystem. In the case of the Data ONTAP WAFL ® file system, a snapshot is extremely efficient and fast to produce given the Data ONTAP WAFL ® copy-on-write file layout.

Given two snapshots, T_{n-1} and T_n, we can quickly calculate the difference between them. This difference represents all of the file metadata changes between T_{n-1} and T_n. Thus incremental indexing can be very fast.

All metadata in WAFL resides in a single file called the inode file, which is a collection of fixed length inodes. Extended attributes are included in the inodes. Performing an initial crawl of the storage system is fast because it simply involves sequentially reading the inode file. Snapshots are created by making a copy-on-write clone of the inode file. Calculating the difference between two snapshots leverages this mechanism. This is shown in Figure 6. By looking at the block numbers of the inode file's indirect and data blocks, we can determine exactly which blocks have changed. If a blocks number has not changed, then it does not need to be crawled. If this

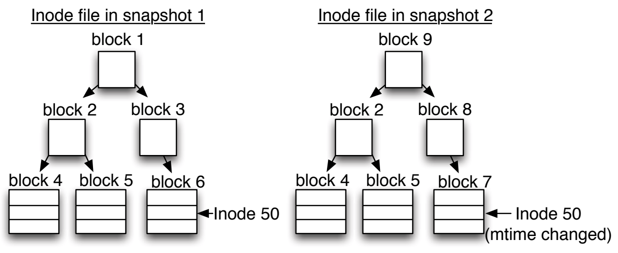


Figure 6: In snapshot 2, block 7 has changed since snapshot 1. This change is propagated up the tree. Because block 2 has not changed, we do not need to examine it or any blocks below it.

block is an indirect block, then no blocks that it points to need to be crawled either because block changes will propagate all the way back up to the inode file's root block. As a result, the new crawler can identify just the data blocks that have changed and crawl only their data. This approach greatly enhances scalability because crawl performance is a function of the number of files that have changed rather than the total number of files.

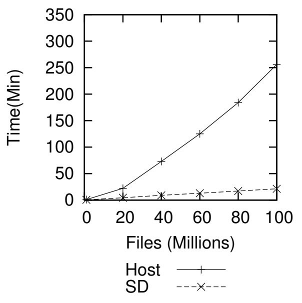
Snapshot based crawling has many benefits compared to alternative approaches. Periodically walking the file system can be extremely slow because each file must be traversed. Moreover, traversal can utilize significant system resources and alter file access times on which file caches depend. Another approach, file system event notifications (e. g., inotify [7]), requires hooks into crit- ical code paths, potentially impacting performance. A changelog, such as the one used in NTFS, is another alternative; however, since we are not interested in every system event, a snapshot-based scheme is more efficient.

3.1 Performance

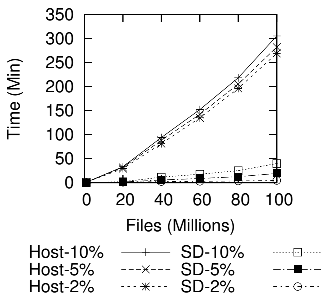
We first evaluated our snapshot-based metadata crawler and compared it to a the logical file system crawler. Fast collection performance impacts how often updates occur and system resource utilization. As described earlier the logical file system crawler performs a parallelized walk of the file system using `stat()` to extract metadata. Figure 7(a) shows the performance of a baseline crawl of all file metadata. The snapshot-based metadata crawler is up to 10× faster than the logical crawler for 100 million files because a snapshot based crawler simply scans the inode file. As a result, a 100 million file system is crawled in less than 20 minutes.

Figure 7(b) shows the time required to collect incremental metadata changes. We examine systems with 2%, 5%, and 10% of their files changed. For example, a baseline of 40 million files and 5% change has 2 million changed files. For the 100 million file tests, each of our crawls finishes in under 45 minutes, while the logical file system crawler takes up to 5 hours. The snapshot-based crawler is able to crawl the inode file at about 70,000 files per second. The snapshot-based crawerl effectively scales because it incur only a fractional overhead as more files are crawled; this is due to crawling only changed blocks of the inode file.

4. RELATED WORK



(a) Baseline



(b) Incremental: 2%, 5%, and 10% changes from baseline

Figure 7: Metadata collection performance. We compare the Snapshot-based crawler (SD) to a host-based file system crawler. The snapshot-based crawler has good scalability; performance is a function of the number of changed files rather than system size.

There are alternatives to periodically recrawling a file system to get the list of files and directories that have changed. One technique is to notify an application whenever a file changes, which is what the Apple® search utility, Spotlight® [13] relies on. Although file notification works on the desktop, where relatively few files change per hour or day, on a large enterprise file system, many millions of files change per day, making the overhead of file notification difficult to manage.

A second technique is to write a log each time the file system is modified, as is done by the Windows® NTFS file system [8]. This log is leveraged by the Windows XP indexing service to decide which files to re-index.

Finally, there are existing tools such as `find`. However, using `find` repeatedly to crawl a file system is slow and inefficient. Additionally, python has a function called `os.walk`, which recursively crawls a file system tree. However, it is single threaded and inefficient.

4.1 Future Work

The current local baseline and incremental crawlers are not resistant to network or host failures. If for any reason the crawler or one of its components dies, the entire crawl process needs to be restarted. We plan to add checkpoints in future so that it isn't necessary completely restart after failure.

Although we limit the number of threads in use to control CPU usage on the host, the number of threads we've picked is based on empirical measurements from a single host and storage system pair. We plan to add functionality to dynamically monitor CPU usage from within the crawler, which will allow the number of active threads to be adjusted at runtime.

Finally, we believe there are optimizations we can make around our use of locks (or lack thereof) to improve parallelism, especially in the case of thin and deep file system trees.

4.2 Summary

We have described the design of two fast file system crawlers: one that is agnostic to the underlying file system and the other that leverages the underlying file system layout. Our benchmarks show that our crawlers are significantly faster than commercial file system crawlers we are aware of. Additionally, we have shown how it is possible to manage the host CPU and memory resources without affecting crawl performance.

5. ACKNOWLEDGMENTS

We acknowledge Priyamvad Deshmukh's contributions to the baseline crawler during his internship at NetApp Bangalore in 2007.

6. REFERENCES

[1] Kazeon. http://www.kazeon.com, 2007.

[2] CHO, J., AND GARCIA-MOLINA, H. Parallel Crawlers. In *11th International World Wide Web Conference"* (2002).

[3] CHO, J., GARCIA-MOLINA, H., AND PAGE, L. Efficient Crawling through URL Ordering. In *Computer Networks and ISDN systems* (1998).

[4] DILIGENTI, M., MAGGINI, M., PUCCI, F. M., AND SCARSELLI, F. Design of a crawler with bounded bandwidth. In *Procedings of the World Wide Web 2004* (2004).

[5] ELLIS, C. S. The case for higher level power management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS)* (Rio Rico, AZ, March 1999), pp. 162–167.

[6] HEYDON, A., AND NAJORK, M. Mercator: A scalable, extensible web crawler. In *World Wide Web, 2(4):219-229* (1999).

[7] KERNEL.ORG. inotify official readmine. http://www.kernel.org/pub/linux/kernel/people/rml/inotify/README, 2008.

[8] MICROSOFT. Windows Change Journal. http://msdn2.microsoft.com/en-us/library/aa363798(VS.85).aspx, 2007.

[9] NAJORK, M., AND WIENER, J. L. Breadth-first search crawling yields high-quality pages. In *Proc. of the 10th International World Wide Web Conference* (2001).

[10] NETAPP. Introduction to Data ONTAP 7G. Tech. rep., 2005.

[11] SHIVAKUMAR, N. Google in a box - building the google search appliance. In *SIGMOD Conference* (2003), p. 635.

[12] SHKAPENYUK, V., AND SUEL, T. Design and Implementation of a High-Performance Distributed Web Crawler. In *ICDE* (2002).

[13] SINGH, A. *Mac OS X Internals*. Addison-Wesley Professional, 2006, ch. File Systems.

[14] SVARCAS, R. DataManip. http://web.netapp.com/engineering/ontap_tests/agnostic/docs/lib-DataManip-local-pm.html, 2004.