TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 01/15 pp337-343
Volume 16, Number 4, August 2011

FastDu: Efficient Directory Summaries Harvest by Tracking File System Changes*

LIU Likun (刘立坤), WU Nuo (吴 诺), XU Chuncong (许春聪),
WU Yongwei (武永卫), YANG Guangwen (杨广文)**

Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

Abstract: FastDu is a file system service that tracks file system changes by intercepting file system calls to maintain directory summaries, which play important roles in both storage administration and improvement of user experiences for some applications. In most circumstances, directory summaries are independently harvested by applications via traversing the file system hierarchy and calling stat () on every file in each directory. For large file systems, this brute-force traverse-based approach can take many hours to complete, even if only a small percentage of the files have changed. This paper describes FastDu, which uses a pre-built database to store harvested directory summaries, and tracks the file system changes by intercepting file system calls, so that new harvesting is restricted to the small subset of directories that contain modified files. Tests using FastDu show that this approach reduces the time needed to get a directory summary by one or two orders of magnitude with almost negligible penalty to application-aware file system performance.

Key words: file system metadata; metadata crawl; file system changes; file system intercepts

Introduction

Directory summaries (e.g., total size of a directory tree and the number of files in the tree) are not only important in storage management (e.g., understanding the overall file system make-up, estimating the amount of migration data between different tiers, and storage space reclamation), but are also very helpful for improving user experience in many applications, such as the GUI front ends of cp, tar, brute-force file search, and anti-virus scanning.

However, most file systems provide no active support of this kind of information, so this functionality is generally implemented independently by applications via a just-in-time traverse approach that traverses the target directory hierarchy and calls stat() on each file in the directory. Obviously, the performance of brute-force traverse-based approaches depends strongly on the number of files in the target directory tree.

For large file systems that contain hundreds of millions or even billions of files, performing such a harvest on a relatively large directory (e.g., the users' home directories) can take minutes or even hours due to the large number of disk seeks and (in the case of network file systems) communication overhead[1]. As an example, we have observed a commercial NFS file system taking more than five hours for executing a du[2] command on the /home directory containing 16 million

Received: 2011-03-17; revised: 2011-06-12

* Supported by the National Key Basic Research and Development Program (973) of China (No. 2011CB302505), the National Natural Science Foundation of China (Nos. 60803121 and 61073165), and the National High-Tech Research and Development (863) Program of China (Nos. 2010AA012401 and 2009AA01A130)

** To whom correspondence should be addressed.
E-mail: ygw@tsinghua.edu.cn; Tel: 86-10-62797142

files. As modern file systems are becoming increasingly large[3,4], such awkward solutions are becoming increasingly time- consuming, making them unacceptable sometimes.

Given the inefficiency of the traverse-based harvest, other solutions have been proposed. The most commonly used approach uses a pre-built database generated and updated regularly (e.g., every day) by a periodic scan (e.g., used by locate[5]). This manages to walk around the time-consuming traverse by doing it in the background, especially with improvements[1,6] to accelerate the scan. However, the results lag to some extent due to the delayed database updating, which restricts its usefulness. Another alternative used mostly by desktop search services is to update the database based on the file system event mechanism (e.g., "file-system-change notification" on Windows and "inotify" in Linux[7]). This mechanism is able to provide real-time results, but may introduce too much runtime overhead depending on the operating system implementation (e.g., the memory consumption of inotify[1]). In addition, updating the database with each file system change is inefficient since most file system changes will be overwritten by later changes. However, such approaches are worth exploring, especially when new notification mechanisms[8] become more mature.

This paper describes FastDu, an approach that maintains directory summaries by tracking file system changes. FastDu operates by intercepting file access calls[9] and passing the modification information to a user land daemon. The daemon adjusts the external database. Therefore, FastDu not only provides real-time enough results, but also significantly reduces the work involved with harvesting such information since the percentage of files changed in any given day is relatively small in most file systems[1,3,10,11]. An inherent danger of hooking into the critical I/O path (i.e., intercepting file system calls) is that it may hurt application-aware file system performance. Two optimization methods, lazy re-harvesting and batch change propagation, are used here to minimize the impact.

1 FastDu Design

Two design choices were made in designing FastDu. First, unlike both the periodic scanning based and file-system-event based approaches, FastDu harvests the directory summaries by tracking file system changes. It operates by intercepting file system calls. The same approach was used by Dazuko[9], Connections[12], and TraceFS[13] to address similar requirements for different tasks. Since this can capture all the file system changes, this solution is usually able to provide better real-time support. Although there is a potential danger of hurting application-aware file system performance, performance penalty is shown here to be minimal, if the system is carefully designed and optimized.

FastDu only re-harvests the directory summaries before a query, rather than updating the database with each file system change, even if the latter would result in much better query performance. This is done because, on one hand, most file system changes are overwritten by subsequent changes so updating the database with each change is inefficient, on the other hand, directory summary queries are much less infrequent than normal file system operations.

1.1 Architecture

Figure 1 illustrates the FastDu system architecture. From the application's perspective, FastDu is a file system service separating from the file system that takes in a directory identification and returns a just-in-time summary related to that directory. Internally, the FastDu system consists mainly of: a stackable in-kernel tracer that monitors file system changes as they happen, a FastDu database that holds the summary and modification information for each directory, and a FastDu daemon service responsible for updating the database with the changes captured by the tracer and for answering directory summary queries from applications. When the service receives a query request from an application, it looks in the database to determine whether the information from the database is up-to-date. The information is returned to the application if up-to-date, otherwise a re-harvest is launched on the related directories and the new result from the re-harvest is returned. When any change to a file is captured by the in-kernel tracer, the tracer marks all the ancestor directories as changed and notifies the daemon service. The ancestor directories are marked as changed so that any given directory always reflects the latest changes in that directory.

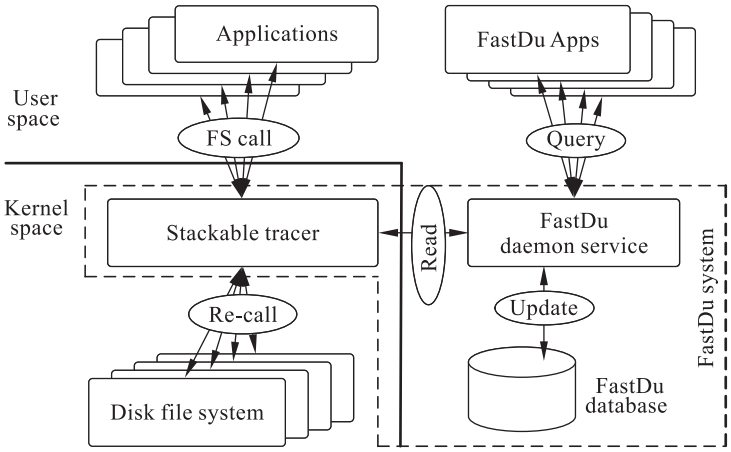

Fig. 1 Architecture of FastDu

The file system remains unchanged, since the only information required by FastDu can be gathered either by a transparent tracing module or directly from existing file system interfaces.

1.2 Interfaces

The core FastDu interface can be expressed as

AggregatedProperty[] QueryDirStatistics(path, <dev, ino>)

(1) The first input parameter path denotes the full path of the directory being queried. This is required for the lazy database update when outdated summaries are detected during query processing, because only inode information is kept in the database and standard file systems do not provide interfaces to traverse a sub-tree with only this information (Section 1.3).

(2) The second input parameter is the <dev, ino> pair, in which dev denotes the device number holding the queried directory and ino denotes the inode number of the queried directory. dev is required since directories in different file systems may have the same ino.

(3) The return value is the AggregatedProperty array, each element of which is a <key, value> pair in which key denotes which aggregated property (e.g., total size of the sub-tree or file count in the sub-tree) and value denotes the associated value.

Internally, FastDu maintains a set of directory summary records. Each record is a tuple:

{<dev, ino>, mtime, dutime, AggregatedProperty[]}

in which <dev, ino> and AggregatedProperty[] are as just described, mtime denotes the timestamp of the newest change in this directory tree, and dutime denotes when the last harvest was performed. The record is up-to-date if and only if mtime < dutime.

1.3 Lazy re-harvesting and delayed database updating

Updating the database on every change is inefficient as most changes are quickly overwritten. FastDu is unable to update the database on each individual due to the lack of the full path in change information from the kernel, which is designed purposely to minimize anonymity introduced by the database and to minimize the tracer overhead and system resource consumption. Thus, the summary of a changed directory is re-harvested only during query processing.

To determine whether a summary record is outdated, FastDu keeps timestamps for both the latest change happen, mtime, and last re-harvest performed on the target directory, dutime. Whenever a queried record from the database satisfies mtime > dutime, a re-harvesting is launched. The path information required to launch the re-harvesting is provided by the application, as shown by the interface in Section 1.2. During the re-harvesting, dutime for each touched directory is also updated. mtime is updated under the following circumstances:

(1) before a query is performed;

(2) during a re-harvesting after all the subdirectories have been updated; and

(3) an excessive period, timeout, (e.g., more than 30 min) after the last harvest.

The first case ensures that each query will get up-to-date results. The second case makes sure that mtime correctly indicates the record status with the

delayed change propagation described here; and the third case helps reclaim the kernel resource (in particular memory) used to track file system changes. This lazy re-harvest allows many changes, such as those written to a large file, to be absorbed by a change buffer in the kernel, significantly reducing the changes that must be processed by the daemon service as well as related context switches.

To minimize the normal file system workload interference caused by storing the harvested changes, such as mtime updating, a change is only propagated to the database if and only if the new mtime is greater than and the old mtime is less than the dutime, which is referred to as delayed change propagation. The correctness of such optimization is ensured by the second case.

The core loop of the daemon service is outlined as Algorithm 1 as follows.

Algorithm 1: The core loop algorithm of the daemon service

```
while Not completed do
  event←wait for next event;
  if event is query then
    harvest changes from tracer;
    update mtimes if needed;
    R←get directory statistic from the DB;
    if R is outdated then
      traverse to update R and store to the DB;
      adjust dutime and mtime for R in the DB
    end
    send R as response;
  end
  if event is timeout then
    harvest changes from tracer;
    for change C do
      if C.mtime > dutime and old_mtime <
      dutime then
        update mtime for C in the DB;
      end
    end
  end
end
```

1.4 Batch change propagation

Propagating each file system change to the daemon service is inefficient since most changes will be quickly overwritten, usually within several seconds. Furthermore, batch propagation will improve the efficiency by significantly reducing the communication overhead and kernel-user context switches.

The batch propagation enables FastDu to perform early change combination, which greatly helps reduce kernel resources (e.g., memory). Specifically, the tracer maintains a set of changes that have occurred but have not been propagated to the daemon service. Here, a change is a triad {dev, ino, timestamp} in which timestamp denotes the change time. When a file is changed, the tracer generates triads for all the directories on the path and adds them to the set. If a triad with the same dev and ino already exists, the timestamp is updated.

2 Implementation

The prototype implementation of FastDu had the three components shown in Fig. 1, a stackable tracer, a daemon service, and a FastDu database.

The tracer sits at the system call (VFS) layer in the kernel and watches application activities to trace all file system calls. This component was implemented by modifying the DazukoFS[9]. Unlike DazukoFS which communicates with user-land applications during most file system operations, this tracer used a change buffer and batch change propagation analyzed and described in Section 1.4. The tracer is operating system specific, and FastDu currently runs exclusively under Linux 2.6 kernels. However, similar system call tracing infrastructures exist in other systems, such as Windows, so FastDu can be easily ported to other systems.

The FastDu database stores directory summaries and change information using BerkeleyDB[14] which are updated according to the algorithms described in Section 1.3. The key of each record is the <dev, ino> pair; and the value of each record is the summary information for the directory.

The daemon service runs as a privilege background process so that it can traverse the file system without encountering permission problems. It communicates with the FastDu applications via a local network. Applications specify each query as a <dev, ino> pair. The daemon service returns the directory summary by looking in the FastDu database, during which it also updates the database if needed according to the algorithm described in Section 1.3. The daemon service harvests file system changes from the tracer via Linux's proc file system by reading a specific file

under the /proc directory. Harvests are scheduled with a fixed delay or before a query is performed.

3 Evaluation

The FastDu evaluation consists of two parts. The first part compares the performance of getting directory summaries via FastDu with the traditional brute-force namespace-traverse-based solution. As expected, FastDu was much more efficient. The second part evaluated the impact of FastDu on an application-aware file system performance via several typical tasks and iozone, a widely adopted file system benchmark. It turns out that the performance penalty of intercepting file system calls is minimal.

3.1 Experimental setup

All experiments were run on machines with Intel (R) Core (TM) i3 @2.93GHz, 4 GB of main memory, and 1 TB Samsung 7200 RPM hard drives running Ubuntu Linux 8.04. Unless otherwise specified, each experiment was run ten times with the average reported. Snapshots from two production file systems were used to evaluate the benefits of FastDu, as summarized in Table 1. Server A is a storage appliance used by more than 100 university researchers for code development, paper writing, and data analysis. Server B is a file server used by more than 3000 students at another university for personal data backup and online sharing[15]. The snapshots were collected at 3:00 a.m. each day during the period from Sept. 1, 2010 to Sept. 14, 2010 using a modified version of fsstats[1,16].

Table 1 File system snapshots used to evaluate FastDu

File system	Number of files	Number of directory	Used capacity (TB)
Server A	16.3 million	1.4 million	1.70
Server B	3.4 million	0.3 million	15.10

3.2 Performance

The FastDu evaluations used the merit speedup ratio which is the time for a task without FastDu divided by the time with FastDu enabled. These tests used the standard du task which calculates the total size of a directory tree.

Point-in-time file system statuses were constructed from snapshots of the two production file systems. First, an empty file system was created. Then, a small file (8 KB, in particular) was created for each path in the first snapshot with the size and timestamp then adjusted using truncate() and utime() to reflect the metadata status in the snapshot. Files were not filled to actual size due to the disk capacity limitation, and empty files were not used to avoid having all metadata clustered together. The tests will underestimate the benefits of FastDu because the seek time between metadata readings for large files is significantly under represented. For all subsequent snapshots, the differences between adjacent snapshots were used to change the file attributes as just described.

The tests were performed as follows. A FastDu database was pre-built based on the first file system status. And then the following steps were repeated: (1) enabled the FastDu tracer, (2) patched the current status to reflect the change for the next status, (3) performed a modified du using FastDu service on the file system root and then disabled the FastDu tracer, and (4) performed the standard du. Both the file system and operating system caches were cleared before each du was performed to ensure the results were comparable.

Figure 2 shows the speedup ratio results. Except for Server A on Sept. 11, the speedup ratio is greater than one hundred, meaning that for most of the time, FastDu can make getting summary information for large directories 2-3 orders of magnitude faster, depending on the number of changes in the target directory. The much lower speedup ratio of Server A on Sept. 11 is due to the importation of several projects containing many files. The existence of such exception may imply automatic FastDu database background refreshing could be useful when large file system modifications are detected.

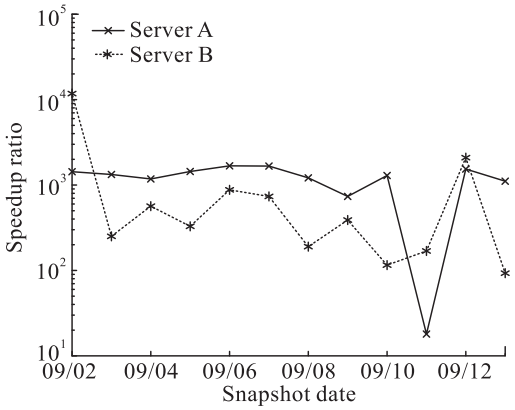

Fig. 2 FastDu speedup ratio

3.3 File system impact

An inherent danger of hooking into critical I/O paths is that it may hurt application-aware file system performance. Obviously, such impact depends on the work involved with each file system call in the tracer. This effect was evaluated using five typical file system tasks and the iozone file system benchmark. The five tasks were (1) untar, unpacking the kernel source code in the experimental file system, (2) tar, to reverse the untar process with /dev/null as the output file, (3) cp, to duplicate the kernel source directory from the experimental file system to a target directory on the same file system, (4) rm, to remove the kernel source directory using the rm -rf command, and (5) make, to compile the kernel in the experimental file system. The kernel source code used in the tests is 2.6.20.20. To reduce the compile time, the kernel configuration was minimized by turning off all optional modules. The experiments were performed on a newly created ext3 file system. All tests were run ten times with FastDu tracer enabled and ten times with FastDu tracer disabled. Both the file system and operating system caches were cleared before each run.

The average experiment results shown in Table 2 show that enabling the tracer has little impact to the application-aware file system performance. The two shortest tasks even have better performance with the tracer enabled, implying that the impact is even less than the impact of uncertain factors in the environment, such as operating system scheduling. However, for the longer tasks, the impact was small but distinguishable. Given the great performance improvement in getting directory summaries, this impact seems worthwhile most of the time.

Table 2 Impact of FastDu of typical storage tasks

Task	No FastDu		FastDu	
	AVG* (s)	STDEV+	AVG* (s)	STDEV+
Untar	14.9	0.7	15.1	0.7
tar	5.5	0.3	5.3	0.1
cp	68.1	1.3	68.3	0.8
rm	4.2	1.0	3.6	1.9
make	219.4	0.9	219.4	1.4

*AVG is the average time in terms of seconds consumed by each task and +STDEV is the corresponding standard deviation.

The impact of the tracer on the data path was also measured using iozone on the same file system using

iozone -s 512m -i 1 -i 0 -y 1k -q 8m –C

Linux was forced to use 256 MB memory by setting the boot time parameter mem=256 m, to ensure that iozone would get accurate results. Both the file system and operating system caches were cleared before each run to ensure that the results were comparable.

The average results are shown in Fig. 3. Although the file system throughput varies with different record sizes, both the read and write throughput show very little difference for each record size when the tracer is enabled and disabled, meaning that the tracer has little impact to the data path. Both the read and write throughput can be less (e.g., 256 KB record size for write) and greater (e.g., 8 KB record size for write) when the tracer is enabled than when the tracer is disabled, implying that the tracer impact is even less than the impact of uncertain system factors. The figure shows surprisingly high read throughputs for 1 KB and 2 KB record sizes, but the reason is not known. All ten tests showed this trend, but this does not influence the tracer evaluation.

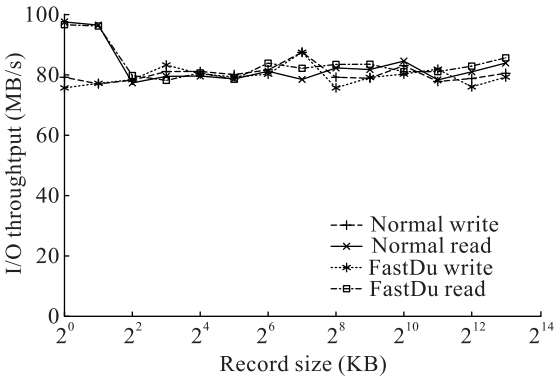

Fig. 3 Impact of FastDu measured with iozone

4 Conclusions and Future Work

Directory summaries can be efficiently maintained by tracking file system changes and intercepting file system calls. This file system interception solution has minimal impact on the application-aware file system performance if carefully designed and optimized. Lazy re-harvest and batch change propagation can be exploited to reduce the impact. These conclusions are confirmed by tests with FastDu, a file system service prototype that intercepts file system calls to maintaining directory summaries.

FastDu will be improved by adding the hard link support, one desired but missed feature of our

prototype, by storing additional information for hard-linked files, and by treating them specially when the database is updated. FastDu will also be enhanced to harvest customized summaries.

References

[1] Liu Likun, Xu Lianghong, Wu Yongwei, et al. Smartscan: Efficient metadata crawl for storage management metadata querying in large file systems. Technical Report CMU-PDL-10-112. Carnegie Mellon University, USA, 2010, 10-112.

[2] The Open Group. Estimate file space usage commands & utilities reference. The Single UNIX Specification: The Authorized Guide to Version 3, UK: the open group publications department, 2002: G906.

[3] Agrawal N, Bolosky W J, Douceur J R, et al. A five-year study of file-system metadata. *ACM Transactions on Storage*, 2007, **3**(3): 9.1-9.32.

[4] Walter C. Kryder's law. *Scientific American Magazine*, 2005, **8**(1).

[5] The Open Group. The find utils. The Single UNIX Specification: The Authorized Guide to Version 3, UK: the Open Group Publications Department, 2002: G906.

[6] Soules C, Keeton K, Morrey C. Scan-lite: Enterprise-wide analysis on the cheap. In: Proceedings of the 4th ACM European Conference on Computer Systems. Nuremberg, Bavaria, 2009: 117-130.

[7] Streicher M. Monitor Linux file system events with inotify. http://www.ibm.com/developerworks/linux/library/l-ubuntu-inotify/index.html?ca= drs, September 2010.

[8] Paris E. Fanotify: The fscking all notification system. http://http://lwn.net/Articles/339253/, February 2011.

[9] Ogness J. Dazuko: An open solution to facilitate on-access scanning. In: Proceedings of Virus Bulletin Conference. Toronto, Canada, 2003: 1-5.

[10] Satyanarayanan M. A study of file sizes and functional lifetimes. In: Proceedings of the Eighth ACM Symposium on Operating Systems Principles. New York, USA, 1981: 96-108.

[11] Roselli D, Lorch J R, Anderson T E. A comparison of file system workloads. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. Berkeley, USA, 2000: 4-4.

[12] Soules C A, Ganger G R. Connections: Using context to enhance file search. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. Brighton, UK, 2005: 119-132.

[13] Aranya A, Wright C P, Zadok E. Tracefs: A file system to trace them all. In: Proceedings of the 3rd Conference on File and Storage Technologies. San Francisco, California, USA, 2004: 129-145.

[14] Olson M A, Bostic K, Seltzer M. Berkeley DB. In: Proceedings of the FREENIX Track. Berkeley, CA, USA, 1999: 6-11.

[15] Xu Pengzhi, Huang Xiaomeng, Wu Yongwei, et al. Campus cloud for data storage and sharing. In: Proceedings of Eighth International Conference on Grid and Cooperative Computing. Lanzhou, China, 2009: 244-249.

[16] Dayal S, Unangst M, Gibson G. Static survey of file system statistics. http://www.pdsi-scidac.org/fsstats/index.html, September 2010.