

VFS Interceptor: Dynamically Tracing File System Operations in real environments^{*}

Yang Wang, Jiwu Shu, Wei Xue, Mao Xue

Department of Computer Science and Technology, Tsinghua University

iodine01@mails.tsinghua.edu.cn, {shujw, xuwei}@tsinghua.edu.cn, lion_xuemao@163.com

Abstract: File system traces have been used for file system evaluation, user behavior analysis, file system debugging and intrusion detection. Current file system tracing tools are hardly applicable in real environments, because they require restarting the applications or are designed for specific kinds of file systems. We developed VFS Interceptor, a low-overhead tool for capturing file system traces dynamically. First, it detects VFS function pointers by recursive detecting, replaces them with stackable ones, in which traces are recorded. The detecting and replacing process is completely transparent to the applications. Second, VFS Interceptor uses a low-overhead compressing method to reduce the size of trace files, thus to reduce I/O overhead and save disk space. Finally, implemented between the VFS layer and the low-level file system, it can capture all important information and can be used for any low-level file system. The evaluations showed that the overhead of VFS Interceptor is no more than 4% on various kinds of file systems. The compressing method can achieve a compressing ratio of up to 9 times and slightly reduce the overall overhead simultaneously.

1 Introduction

1.1 Motivation

File system traces have long been used for file system evaluation and optimization [2][8]. Researchers can replay a file system trace [4] to evaluate a newly designed file system or to find the bottleneck or special access patterns. Currently, file system traces are also used for debugging and intrusion detection [1].

Traces from real environments are most representative and helpful. However, in real environments, there are many restrictions to the tracing tools. First, many environments require a high availability. For example, in supercomputing environments, tasks may last for months and if stopped, they must restart from the beginning and this is not affordable. Second, in real environments, tools cannot add too much overhead to the target machines, since they are used for production activities and overhead means economic loss. Moreover, a large overhead will change the behavior of the target applications and the traces may not be accurate.

To summarize, a tracing tool must satisfy three properties to be applicable in real environments: 1) Transparent to applications: Loading the tool cannot interrupt the target

^{*} This work was supported by the National Natural Science Foundation of China under Grant No. 10576018, and the National Grand Fundamental Research 973 Program of China under Grant No. 2004CB318205, and Supported by Program for New Century Excellent Talents in University (NCET-05-0067).

Correspondence author: Jiwu Shu, email: shujw@tsinghua.edu.cn

applications. 2) Low overhead: The tracing process cannot add too much CPU and I/O overhead or occupy too much disk space. 3) Able to collect all important information: The tool should be able to collect information about all the necessary operations which are critical to trace analysis. This is the basic requirement of a tracing tool.

To the best of our knowledge, no existing tools can satisfy all these three properties. System-call level tracing tools like `strace` [5] can be loaded transparently. However, they cannot trace applications which bypass the system call layer, such as the NFS server. Moreover, they cannot trace memory map operations [1]. File system level tracing tools can collect all the necessary information. However, some of them require modifying the kernels [8] and rebooting the system. Some of them like `tracefs` [1] require a new mount point and thus require reconfiguring and restarting the applications. Some of them like NFS tracing tools [3] can be loaded without interrupting the applications, but they cannot be used for other file systems.

In summary, to collect traces in real environments, it is highly desirable to develop a tool that can meet all three properties.

1.2 Contributions

To achieve the above goal, we proposed VFS Interceptor, a tool that can capture file system traces dynamically. First, it dynamically detects VFS function pointers by recursive detecting. Then it replaces these functions with stackable ones, which records the traces. With this technique, VFS Interceptor is completely transparent to the applications and loading the tool does not require restarting the applications. Second, we proposed a low-overhead compressing method to reduce the size of trace files. By comparing adjacent trace records, VFS Interceptor can eliminate common information redundancies in trace files. Finally, implemented between the VFS layer and the low-level file system, VFS Interceptor can trace all necessary file system operations and can be used upon any low-level file system.

The evaluations showed that the overhead of this tool was no more than 4% on different kinds of low-level file systems. The compressing method can achieve a compressing ratio of up to 9 times and it could slightly reduce the overall overhead simultaneously.

2 Architecture

In this section, we have a brief overview of all the modules of VFS Interceptor. Detailed design and implementation is presented in section 3. As shown in Figure 1, VFS Interceptor is composed of three modules:

- 1) The Function Interceptor dynamically detects and replaces VFS operations with stackable ones, where file system operation information is collected. The detecting and replacing process is completely transparent to the applications.
- 2) The Trace Recorder receives information from the Function Interceptor, converts it into a compressed format, output it to disks by buffered write. With compressed trace format, it greatly reduces the size of trace files and I/O overhead. For ease of management, the Trace Recorder cut the trace file into segments, so that old ones can be moved to other places [9].
- 3) The Trace File Manager automatically adds additional information to the trace files, and moves them to dedicated machines to save disk space. This module is used to help the tool

run a long period of time without human assistance.

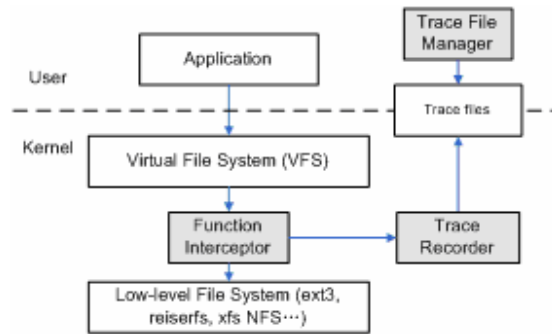


Fig.1. Architecture of VFS Interceptor

3 Design and Implementation

3.1 Detecting and Replacing VFS operations

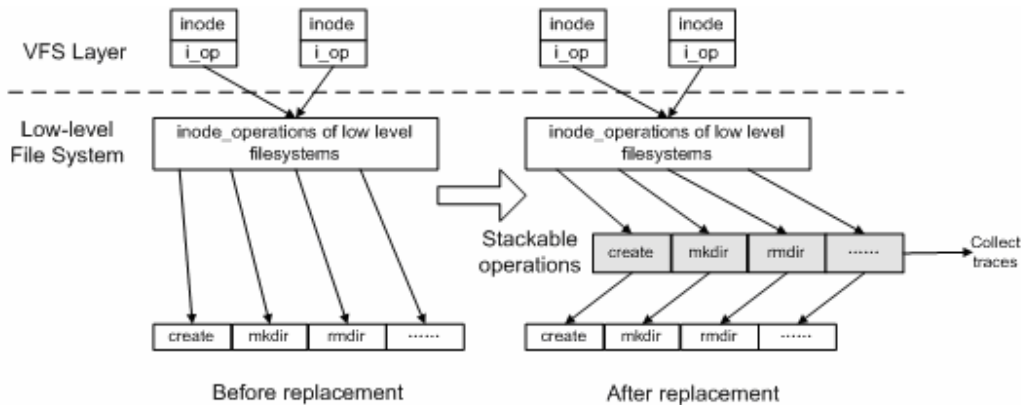


Fig.2. Replacing the *inode_operations*

The Function Interceptor detects VFS operations online and replaces them with stackable ones, so that tracing functionality can be added to the original file system. Figure 2 shows the replacing process using *inode_operations* as an example. As shown in Figure 2, in the VFS model, each *inode* has a pointer *i_op*, which points to its *inode_operations* from the low-level file system. For example, in ext3, it may point to *ext3_dir_inode_operations*. The *inode_operations* is a group of function pointers, such as *create*, *mkdir*, *rmdir*, and so on. The Function Interceptor replaces these functions pointers with its own stackable operations, which will call the original functions and collect required information.

There is a problem that VFS Interceptor has to detect all the VFS operations used by the current low-level file system. However, most low-level file systems implement more than one kind of *inode_operations* or *file_operations* and it is hard to know how many there are. For example, ext3 has *ext3_dir_inode_operations* for directory inodes and *ext3_file_inode_operations* for file inodes. We proposed recursive detecting to solve this problem. First, VFS Interceptor opens the mount point directory, from the *inode* of which VFS Interceptor can get some operations. This step can detect part of the file system operations. Second, VFS Interceptor tries

to detect new operations from the parameters of the detected operations. Still using ext3 as an example, in the first step, VFS Interceptor can detect ext3_dir_inode_operations from the mount point. In the second step, VFS Interceptor can detect ext3_file_operations from the *create* operation in the detected ext3_dir_inode_operations when the *create* operation is executed. By recursive detecting, VFS Interceptor can find all the newly generated VFS operations after the tool is loaded.

3.2 Compressed Trace Format

The Trace Recorder receives the primitive trace from the Function Interceptor, compresses them and stores them in a buffer. When the buffer is full, it writes the traces to the disk.

The trace format is shown in Figure 3.

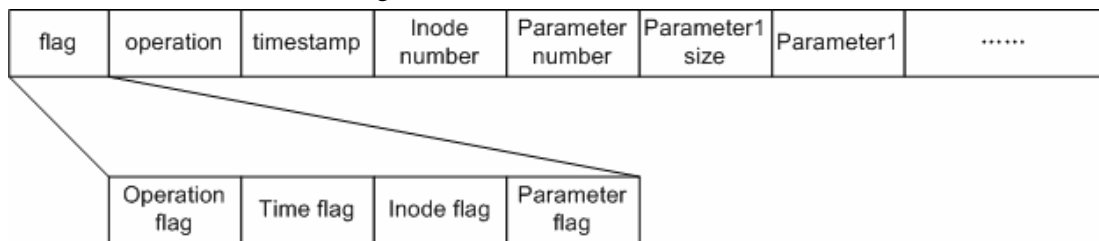


Fig. 3. Trace Format

Every trace record is divided into four segments: operation type, timestamp, inode number and parameters. A flag is placed at the top of the trace to identify the status of the four segments. Every flag occupies 2 bits and has 4 statuses. 00 means that this segment is empty; 01 means that this segment is normal; 10 means that this segment is the same as the last trace; 11 means this is an I/O request and is sequential to the last one. If a flag of a segment is 00, 10 or 11, the corresponding segment is not recorded, so as to save space.

The compressing method can eliminate five types of information redundancy: 1) Operation type redundancy: some operations may repeat several times, such as *permission* and *dirty_inode*. 2) Time redundancy: operations may occur concurrently at the same time, so that timestamps of several consecutive operations are the same. 3) Inode number redundancy: the inode number of the current operation has a high probability to be the same as that of the last operation. 4) Parameter redundancy: some operations can appear in pairs, while their parameters are the same. For example, in ext3, *read* operation will call the *aio_read* operation and the parameters of these two operations are the same. 5) Sequential access redundancy: in sequential access patterns, the parameter of the current I/O operation can be calculated from the last I/O operation.

3.3 Trace File Manager

The Trace File Manager is designed to automatically manage trace files in a cluster environment. First, if all traces are stored in one file, this file will be difficult to manage, since it cannot be moved. Therefore, the Trace Recorder stores traces in separate files, so that earlier trace files can be managed by the Trace File Manager [9].

The Trace File Manager performs three tasks. The first is to collect hardware information (CPU, memory and disks). The second is to add identifiers (timestamp, machine name) to trace files. The third is to move trace files to other locations when the disk space is not enough.

4 Evaluations

We evaluated VFS Interceptor with two benchmarks. The first is Postmark [6], an I/O intensive workload to test disk file systems (ext3, xfs, reiserfs and jfs) on a single machine. The second is NAS Parallel Benchmark (NPB) [7], a workload with interleaving CPU and I/O tasks. NPB is used to test Network File System (NFS) in a cluster.

We performed the Postmark test on a machine with 4 Pentium III 700MHz CPUs, 1GB memory and one IBM DDYS-T36950M 37GB SCSI disk. The machine ran Redhat Linux AS 4.0 with 2.6.11 kernel. Postmark is configured to create 500 files between 512 bytes and 1MB and performed 100,000 transactions in 100 subdirectories. In the NPB test, the NFS server ran on a machine mentioned above. We used 4 NFS clients with 2 Itanium2 1GHz CPUs, 1GB memory and one Seagate ST336753LC disk. VFS Interceptor is installed on both the NFS server and the NFS clients. NPB is configured to perform the bt-io task C with 16 processes.

4.1 Overhead Evaluation

In this test, we first ran Postmark (on ext3, xfs, reiserfs and jfs) and NPB (on NFS) on a plain directory without VFS Interceptor. Then we added VFS Interceptor and ran the same test. We also tested VFS Interceptor without the compressing method. The results are shown in Figure 4.

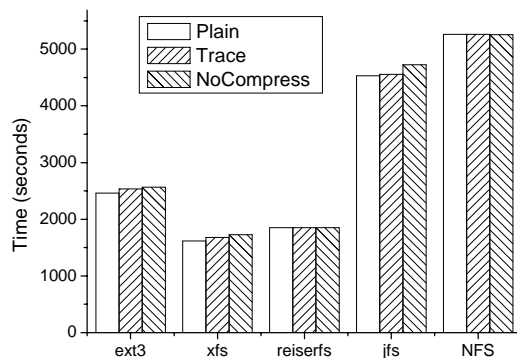


Fig. 4. Overhead of VFS Interceptor

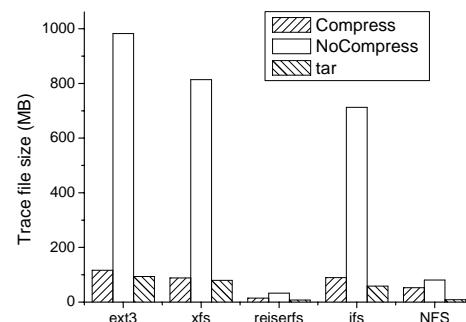


Fig. 5. Size of trace files with different methods

From Figure 4, it can be observed that VFS Interceptor has introduced a low overhead. It is about 4.0% on xfs, 3.0% on ext3 and almost negligible on other file systems. In comparison, Tracefs reported about 12.4% overhead in FULL tracing mode on ext3 with Postmark. VFS Interceptor traces all super_operations, inode_operations and file_operations, but it does not collect as much information as Tracefs, because low overhead is more important to make a tool applicable. Therefore, we give up some less important information to reduce the overhead.

It can also be observed that our compressing method could slightly reduce the overhead. It reduced the overhead from 4.1% to 3.0% on ext3 and from 4.3% to 0.5% on jfs. In comparison, Tracefs reported that its compression with zlib[10] increased the overhead by 6.1% in FULL tracing mode. Therefore, our compressing method has a low overhead.

4.2 The effects of trace compressing

In this test, we observe the size of trace files with and without the compressing method. We

also compare our compressing method with the tar tool. The results are shown in Figure 5. Our compressing method could achieve a compressing ratio of 8.4 times on ext3, 9.1 times on xfs and 7.9 times on jfs. In comparison, the compressing ratio of tar is 10.5, 10.2 and 12.1 times. On reiserfs and NFS, the compressing ratio of our method is significantly lower than tar. However, since the original trace files are not large on these two, the difference is not important.

Since overhead is a critical factor to make a tool applicable in real environments, our method makes a tradeoff between overhead and compressing ratio. In summary, our compressing method has a lower compressing ratio but also a low overhead.

5 Conclusions and Future Work

We proposed VFS Interceptor, a low-overhead file system tracing tool that can be loaded without interrupting the target applications. First, it dynamically detects all the file system function pointers by recursive detecting and replaces them to add the tracing functions. This process does not require restarting the applications. Second, we proposed a low-overhead tracing method to reduce the size of trace files. Third, it is implemented between the VFS layer and the low-level file system, thus it can trace all necessary operations.

The evaluation showed that the overhead of this tool was no more than 4% on five kinds of widely used file systems. The compressing method can achieve a compressing ratio of up to 9 times and it can reduce the overall overhead compared with tracing without compressing.

We are now using this tool in real environments to collect traces. We hope to get the representative traces of real applications over a long period of time. Then we can use them to optimize I/O performance of storage servers, cache systems, etc.

References:

- [1] Aranya A, Wright C. P, Zadok E. Tracefs: A File System to Trace Them All. In Proceedings of FAST 2004, 2004.
- [2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In Proceedings of 13th ACM SOSP, pages 198–212. ACM SIGOPS, 1991.
- [3] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In Proceedings of FAST 2003, March 2003.
- [4] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and Efficient Replaying of File System Traces. In Proceedings of FAST 2005, 2005.
- [5] Strace. <http://sourceforge.net/projects/strace/>
- [6] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [7] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Resources/Software/npb.html>
- [8] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In Proc. of the Annual USENIX Technical Conference, pages 41–54, June 2000.
- [9] S. Zhou, H. Da Costa, and A. J. Smith. A File System Tracing Package for Berkeley UNIX. In Proceedings of the USENIX Summer Conference, pages 407–419, June 1984.
- [10] P. Deutsch and J. L. Gailly. RFC 1050: Zlib 3.3 Specification. Network Working Group, May 1996.