System-Call Based Problem Diagnosis for PVFS

Michael P. Kasick, Keith A. Bare, Eugene E. Marinelli III, Jiaqi Tan, Rajeev Gandhi, Priya Narasimhan
Carnegie Mellon University; 5000 Forbes Ave; Pittsburgh, PA 15213; Tel.: 412–268–9474
E-mail: `{kbare, mkasick, emarinel, jiaqit, rgandhi, priyan}@andrew.cmu.edu`

Abstract

We present a syscall-based approach to automatically diagnose performance problems, server-to-client propagated errors, and crash/hang problems in PVFS. Our approach compares the statistical and semantic attributes of syscalls across PVFS servers in order to diagnose the culprit server, under these problems, for different file-system benchmarks–`dd`, `postmark` and `IOzone`–in a PVFS cluster.

1. Introduction

The Parallel Virtual File System (PVFS) [3] is an open-source, parallel file-system that provides high-performance computing (HPC) applications with high-speed data read/write access to files in a cluster of commodity computers. PVFS is designed as a client-server architecture, with many clients communicating with multiple I/O servers and one or more metadata servers. To facilitate parallel access to a file, PVFS distributes (or "stripes") that file across multiple disks located on physically distinct I/O servers.

Problem diagnosis is important in long-running jobs in HPC environments, where the effects of problems can be magnified due to computations exhibiting long durations and being performed at large scales. Server-to-client propagated errors and crash/hang failures are two other relevant categories of PVFS problems. Current diagnosis involves the manual analysis of the logs that record PVFS' execution; this has high runtime overheads and requires code-level (white-box) PVFS instrumentation.

On the other hand, syscall tracing does not require any modification of either the traced client application or PVFS, making it a black-box diagnosis strategy. Syscall instrumentation of the PVFS server and client processes yields *statistical* data about disk/network I/O transfer times that then enable us to detect performance anomalies. Syscall instrumentation of the client application yields *semantic* data about hung or failed requests, which we then trace back to misconfiguration or resource-exhaustion.

Concretely, our contributions are: (1) a new syscall-based approach to diagnose problems automatically and transparently in parallel file-systems, such as PVFS, (2) a statistical diagnosis algorithm that correlates syscall service-times across PVFS servers, to localize the culprit server, and (3) a semantic diagnosis algorithm that correlates errors returned by syscalls at the PVFS client and servers, to diagnose non-performance problems.

2. Problem Statement

Our research is motivated by PVFS developers' experience [2] of problems faced/reported in production PVFS deployments, one of which is Argonne National Laboratory's 557 TFlop Blue Gene/P (BG/P). We emulate and study these real-world problems and attempt to diagnose them.

We aim for our approach to be: (i) transparent, i.e., no modifications to the PVFS applications, and independent of PVFS' operation; (ii) able to differentiate between anomalous and legitimate behavioral changes (e.g., workload shifts); (iii) able to diagnose the culprit server under performance problems, misconfigurations, and resource exhaustions that cause degraded or halted PVFS operation. Fine-grained diagnosis, which would trace the bug to culprit lines of PVFS source-code, is outside our current scope.

We assume that a majority of the PVFS I/O servers exhibit fault-free behavior. We assume that the physical clocks on the cluster's nodes are synchronized (via NTP) so that their time-stamped data can be temporally correlated.

Hypotheses: Based on PVFS' design, we have specific expectations of its behavior. For performance problems, we expect non-faulty I/O servers to exhibit similar wall-clock service time for read/write syscalls, while we expect a faulty I/O server will exhibit longer service times for the same syscalls, in comparison. Thus, the statistical comparison of read/write syscall durations, across PVFS servers, should identify an anomalous server.

For server-to-client error propagation, we expect a server's host file-system's errors to propagate, through the server, and manifest as an error at a client application's syscall. By observing and correlating such errors at both the client application and the servers, we should be able to identify the culprit server. For crash/hang failures, we expect that client application syscalls will never terminate or will take an inordinate amount of time. Thus, by waiting for an explicit time-out, we should be able to identify when a hang has occurred and then trace it back to the culprit server.

3. Instrumentation

We have developed a tool, `syscap`, that uses `ptrace` (a user-space Linux API for syscall interception and modification, *without* changes to the monitored process) to trace syscalls and signals. Each local process' traced events are written, in the order of event completion, as a record to a syscall-event log (`sclog`). Record fields include a record number, a sequence number (identifying events in order of start), timestamp (at the start of the event), light-weight process ID, syscall/signal number, syscall register arguments, syscall result, and syscall wall-clock service time.

`syscap` also produces a file-descriptor update-log (`fdlog`) with records that describe updates to traced processes' file-descriptor tables due to syscalls (e.g., `open`, `close`) that change the open-file table. The target file name is provided by the `/proc/pid/fd/#` symlink. For anonymous "files" (e.g., network sockets), a special identifier captures other characteristics, e.g., TCP-socket IP address and port number. On observing the creation of a file-descriptor, we automatically track all syscalls that subsequently use the file-descriptor, and extract the service times of all associated read and write syscalls.

Diagnosing performance problems. The metrics of interest are: (i) disk-read service time (`dread`), (ii) disk-write service time (`dwrite`), (iii) server's network-read time (`nsread`), and (iv) client's network-read time (`ncread`). `dread` and `dwrite` are the values of the wall-clock service times for read and write syscalls, respectively, on I/O server file-objects. `nsread` and `ncread` represent the amount of time that it takes for the server (client) to read a single request (response) over the network.

Diagnosing propagated errors. We also seek to study errors that are returned from PVFS server syscalls to an I/O server's host file-system, which are then propagated to clients, and finally appearing as return values from client application read/write syscalls. The metrics of interest for diagnosing propagated errors are: (i) the `errno`s of failed syscalls, and (ii) timestamps of failed syscalls.

Diagnosing crash/hang faults. We also seek to study cases where a PVFS server crashes (or otherwise stops responding to requests), leaving clients in a hung state. Although all execution halts, it is still difficult to manually diagnose the faulty server. The metrics of interest for diagnosing crash/hang faults are: (i) service time of application's PVFS I/O syscalls, and (ii) most recent client syscall to each I/O server socket.

4. Experimental Set-up

We perform our experiments on a cluster of AMD Opteron 1220 machines, each with 4 GB RAM, two Seagate Barracuda 7200.10 320 GB disks (one dedicated for PVFS storage), and a Broadcom NetXtreme BCM5721 Gigabit Ethernet controller. Each node runs Debian GNU/Linux 4.0 (etch) with Linux kernel 2.6.18 and PVFS 2.8.0. The machines run in stock configuration with no background tasks. The results that we report[1] are from experiments in a PVFS cluster with 10 I/O+metadata servers and 10 clients.

Our study of performance problems involves the workload running for 120 seconds in fault-free mode, the fault injected for 300 seconds and then deactivated. The experiment continues to the completion of the benchmark, typically taking 600 seconds in the fault-free case. Our study of propagated errors and crash/hang faults involves the fault persisting for the entire experiment, which runs for shorter durations (approx 60 seconds), long enough for the workload to activate the fault.

Workloads. Our first two workloads, `ddw` and `ddr`, consist of the same benchmark, `dd`, either writing zeros (to `/dev/zero`) to a client-specific temporary file in PVFS, or reading the contents of a previously written client-specific temporary file and writing the output to `/dev/null`. `dd` models HPC scientific workloads with constant data-write rates.

Our next two workloads, `iozonew` and `iozoner`, comprise the same common file-system benchmark, IOzone v3.283. We run `iozonew` in write/rewrite mode and `iozoner` in read/reread mode. IOzone is a large-file I/O-heavy benchmark with few metadata operations, with an `fsync` and a workload change half-way through the benchmark. Our fifth benchmark is PostMark v1.51, a metadata-server heavy workload with small file writes (all writes < 64 kB; thus, writes occur only on a single server per file).

For the `ddw` workload, we use a 17 GB file with a record size of 40 MB. Write-record sizes were chosen so that 4MB of data is sent to each server in a single bulk transfer, which is a requirement for good PVFS performance. This ensures that the server where each request starts is rotated through all I/O servers to eliminate any potential unfairness in always starting transmissions with the same server. For `ddr`, we use a 27 GB file with a record-size of 40 MB. For `iozonew`, we use a 8 GB file with a record-size of 16 MB. For `iozoner`, we use a 8 GB file with a record-size of 16 MB. For `postmark`, we use the default configuration with 9,000 transactions.

Injecting performance problems. We simulate a *disk-hog* problem by running a `dd` process[2], overwriting an un-

[1]Due to space constraints, we do not report results from our experiments on different PVFS cluster configurations.

[2]Indeed, we leverage `dd` both as a benchmark and an external disk-hog.

used partition on one of our PVFS storage disks with zeros. We simulate a *disk-busy* problem by running a `sgmdd` process (from Linux sg3_utils), which issues low-level SCSI I/O commands to read 1 MB blocks from the same unused partition on one of our storage disks.

We simulate a *write-network-hog* problem by having a third-party open a TCP connection to a listening port on one of the PVFS I/O servers and sending zeros to it, and a *read-network-hog* problem by having the I/O server send zeros to the third-party. We simulate a server receive-packet-loss (*receive-pktloss* problem) by a netfilter firewall rule that probabilistically matches packets received at one of the I/O servers, and then drops them with probability 5%. We simulate a server send-packet-loss (*send-pktloss* problem) by a firewall rule on all clients that matches packets incoming from a single server.

Injecting propagated errors. The three faults we inject that manifest as propagated errors are:

- *err-inodes*—Insufficient inodes for storage space. An ENOSPC `errno` is returned by the I/O server's host file-system and propagated to the client application through a `write` syscall, potentially misleading the client into believing that PVFS ran out of metadata structures. To inject this, we set one of the I/O server's storage partition size to be unusually low (8 MB).
- *err-files*—Insufficient maximum open files (`rlimit`). An EMFILE `errno` is returned by the I/O server to the application, unusually as the result of a `read` or `write` call, misleading the client into believing its resource limit (and not the server's) is misconfigured. To inject this, we execute one of the PVFS I/O servers as an unprivileged user with `RLIMIT_NOFILE` set to 50.
- *err-remount*—Emergency remount read-only. This occurs when a PVFS server experiences an error due to the underlying storage device or file-system corruption. To inject this, at 15 seconds into the experiment, we write a "w" to `/proc/sysrq-trigger`, forcing the I/O server to remount as read-only. We run a custom workload for this experiment that repeatedly opens a new file and writes a single null byte, then waits half a second. This ensures that the next PVFS operation after the fault-injection is an `open` (not a `write`) call.

Injecting crash/hang faults. The four faults we inject that manifest as crash/hang failures are:

- *err-space*—Insufficient storage partition space. The server's host file-system returns an ENOSPC `errno` for a `write` syscall. The server reacts by killing the message-handling thread, hanging all subsequent client requests. To inject this, we select an I/O server and set its storage-partition size lower (256 MB).
- *err-fsize*—Insufficient maximum file size (`rlimit`) and *err-vmsize*—Insufficient process maximum virtual-memory size (`rlimit`). *err-fsize* causes an I/O server `write` call to return an EBIG `errno`. *err-vmsize* causes an I/O server memory-allocation syscall (e.g., `mmap`) to return an ENOMEM `errno`. To inject *err-fsize*, we execute one of the PVFS I/O server processes as an unprivileged user with the maximum file-size set to 128 MB. To inject *err-vmsize*, we follow the same procedure, but instead set the maximum virtual-memory size to 40,000 kB.
- *err-remount*–Emergency remount read-only.

5. Statistical Syscall-Based Diagnosis

This diagnosis algorithm relies on our hypothesis that fault-free I/O servers have similar average behavior. However, even under fault-free conditions, servers can behave differently (even small hardware differences can cause some of the servers to be saturated). We account for such heterogeneity through a *fault-free training phase*. The underlying hypothesis is that while a saturated server's service time would likely differ from those of non-saturated servers under fault-free conditions, the deviation of a faulty server would be more pronounced.

For each syscall of interest, we generate a time-series of syscall service times at each server by dividing the average syscall duration by the number of syscalls at 1-second intervals. Our algorithm compares the time-series of syscall service times at all of the servers to detect any anomalous time-series (and thence, the associated anomalous server). Every second, we compute a representative value (currently, we use the median of the syscall service times across all the non-faulty/non-saturated servers) of the syscall service times for the non-faulty nodes. A server is flagged as anomalous if its syscall service time differs by more than a predetermined threshold from this representatie value.

In the training phase, we determine the maximum deviation of a server's syscall service times from the representative (median) value under fault-free conditions. This phase also determines which servers are likely to be saturated under fault-free conditions. We currently use the `ddw` and `ddr` workloads only under fault-free conditions to determine the threshold values. For each server, we detect the maximum deviation of its service times from the median value and use the maximum deviation to choose our threshold value for that server. For actual deployment, a PVFS administrator would first run the `ddr` and `ddw` workload under fault-free conditions to determine the threshold values for each server.

6. Semantic Syscall-Based Diagnosis

Propagated errors. Our strategy consists of two phases: (i) a training phase to identify (and ignore) `errnos` that result from "failed" syscalls, as a part of normal PVFS operation, and (ii) a diagnosis phase to identify unexpected failed syscalls with propagated errors.

For training, we invoke the `pvfs2-ping` utility on a single client immediately after launching all of the PVFS servers. This exercises the code paths involved in a complete client session. As the utility executes, we examine the `errno`s returned by "failed" syscalls on the I/O servers and flag these `errno`s as normal so that they can be safely considered as a part of normal PVFS operation. For diagnosis, if a server syscall returns an `errno` that we did not previously flag as normal, we examine whether the same `errno` is returned to the client application during a 3-second window around the server call's timestamp. If the `errno` is indeed propagated to the client, we declare the I/O server where the `errno` originated as the culprit.

Crash/hang faults. Our examination of PVFS' behavior leads to two observations: (i) application syscalls that take "unusually" long times typically indicate that one or more PVFS servers have crashed, and (ii) the crashed server can be identified based on whether or not the client daemon has received a response from the server. Thus, we first examine the client application's syscalls to find I/O syscalls that either exceed a timeout threshold of 10 seconds[3], or that never complete due to forced process termination. Second, we examine the PVFS client daemon's I/O syscalls to I/O server sockets, keeping track of the syscall that was most recently performed in communication with each I/O server, until the time of the application-syscall timeout. Based on these observations, we expect that the last syscall completed with the faulty I/O server will differ from that completed with the non-faulty servers.

7. Results

Table 1 shows the performance (true- and false-positive rates) of our statistical algorithm for diagnosing performance faults using different syscalls for the different workloads. For each fault, we only show those syscalls causing a non-zero true/false positive.

The `dread` and `ncread` syscalls, which are not used by the write-intensive (`ddw` and `iozonew`) workloads, do not help with diagnosis for those workloads. Thus, the left half of the table omits the `ddw` and `iozonew` workloads. Similarly, the `dwrite` and `nsread` syscalls, which are not used by the read-intensive (`ddr` and `iozoner`) workloads, do not help with diagnosis for those workloads. Thus, the right half of the table omits the `ddr` and `iozoner` workloads. We include the `postmark` workload in both halves of the table since it include read as well as write syscalls.

[3]We do not seek to find an optimal timeout threshold. For our purposes, it suffices that we use a threshold that is both many orders of magnitude greater than expected I/O-syscall completion times.

Instrumentation	Overhead for Workload				
	ddr	ddw	iozoner	iozonew	postmark
`syscap`	0.2%	0.6%	-0.4%	0.7%	64.4%
`strace`	-0.1%	-0.8%	-3.3%	0.0%	138.8%
`sysstat`	0.7%	0.4%	-0.9%	0.4%	5.4%

Table 2. Instrumentation overhead: Increase in run-time w.r.t. non-instrumented workload.

Our low false-positive rates are an artifact of our threshold selection (which minimizes only the false-positive rate). An ROC-based approach to threshold selection would likely increase the false-positive (as well as the true-positive) rate. Different syscalls are useful for diagnosing different problems. For instance, `dread` and `dwrite` are useful for diagnosing disk-related problems while `nsread` and `ncread` are useful for diagnosing network-related problems. While each syscall diagnoses only a subset of the problems, a combination of syscalls can effectively diagnose many performance problems.

For propagated errors and crash/hang faults, our diagnosis algorithms successfully diagnosed the faulty server with no false positives.

Overheads. Table 7 reports overheads for three different kinds of black-box instrumentation for our five workloads. Overheads are calculated as the increase in mean workload runtime, w.r.t. the uninstrumented counterparts.

Since four of our five workloads are I/O-heavy, with large bulk transfers, relatively few syscalls are made for the amount of data transferred. Since network and disk data-transfer consume the majority of time in these I/O operations, the added overhead (< 1%) is negligible.

In contrast, the metadata-heavy `postmark` workload has many small data transfers or metadata operations (create, remove, etc.) on many small files. Since the time to issue the syscall takes as long as (if not longer than) the time to carry out the requested operation, syscall instrumentation has a significant overhead (64% and 135% for `syscap` and `strace`, respectively). Because `sysstat`-based instrumentation does not alter the operation of syscalls, its overheads are more modest, making it more appropriate for metadata-heavy workloads.

8. Related Work

Past work on fault diagnosis in distributed systems has focused mainly on Internet services [4, 8], with some work examining debugging performance problems in high-performance computing environments [10]. Our work is different in targeting HPC file-systems instead of the deployed workload application.

Tracing and instrumentation generate system views that are useful for failure diagnosis. File system-specific trac-

Fault	Syscall	**read-heavy workloads**						Syscall	**write-heavy workloads**					
		`ddr`		`iozoner`		`postmark`			`ddw`		`iozonew`		`postmark`	
		TP	FP	TP	FP	TP	FP		TP	FP	TP	FP	TP	FP
disk-hog	`dread`	1	0	1	0	1	0.1	`dwrite`	0.4	0	0.4	0	1	0.1
disk-busy	`dread`	1	0	1	0	0.9	0.1	`dwrite`	0.4	0	0.4	0	1	0.1
write-network-hog	`ncread`	0	0	0	0	0	0	`nsread`	0	0	0	0	0.9	0
read-network-hog	`ncread`	0	0	0	0	1	0	`nsread`	0	0	0	0	1	0
								`dwrite`	0.22	0	0	0	1	0
send-pktloss	`ncread`	0	0	0	0	1	0							
receive-pktloss	`ncread`	0	0	0	0	0.8	0	`nsread`	0	0	0	0	0.9	0

Table 1. Results of our statistical syscall-based diagnosis. TP (FP) = true (false) positive ratio.

ing mechanisms include Stardust [14], which traces causal request flows through a distributed storage system, and TraceFS [1], which uses a thin file-system interpositioned between the Linux VFS layer and the underlying file-system to provide operation traces at multiple granularities. Performance tools for HPC environments include TAU [13] and Paradyn Parallel Performance Tools [9].

Forensix [5] captures syscall times and parameters to reconstruct security incidents. Our approach is similar to Forensix but applies to diagnosing failures rather than logging security intrustions. [6] also uses sequences of syscalls to discriminate abnormal user program behavior.

Other related work includes studies of storage system failures. Jiang et al. [7] conclude from storage logs of 1,800,000 disks deployed at Network Appliance customer sites that disk failures contributed to 20-55% of storage subsystem failures, while other causes included physical interconnects and protocol stacks that led to disk replacement. Prabhakaran et al. [11] analyze failures in journaling file-systems by building models of file system behavior. Work to increase the fault-tolerance of file-systems takes failures into consideration. IRON File Systems [12] re-examines the approach of traditional file-systems towards error handling, analyzes file-system failure policies, and boosts the end-to-end robustness of the *ext3* file-system.

9. Conclusion

Exploiting syscall instrumentation, we were able to detect and diagnose, with a low false-positive rate, a number of performance problems, propagated errors, and crash/hang faults in PVFS. We have effectively shown the value of syscall-based statistical and semantic analyses for diagnosing common PVFS problems. To minimize instrumentation overheads, we plan to develop a low-overhead, in-kernel utility that would allow us to efficiently use syscall tracing for continuous monitoring and online diagnosis.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *FAST*, pages 129–143, San Francisco, CA, Mar/Apr 2004.

[2] P. H. Carns, S. J. Lang, K. N. Harms, and R. Ross. Private communication, Dec. 2008. `http://www.ece.cmu.edu/~fingerpointing/pvfs-issues.html`.

[3] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct 2000.

[4] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, pages 105–118, Brighton, UK, Oct 2005.

[5] A. Goel, W.-C. Feng, D. Maier, W.-C. Feng, and J. Walpole. Forensix: A robust, high-performance reconstruction system. In *International Workshop on Security in Dist. Comp. Sys.*, pages 155–162, Washington, DC, 2005.

[6] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. of Comp. Security*, 6(3):151–180, Aug 1998.

[7] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. In *FAST*, pages 111–125, 2008.

[8] E. Kiciman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Trans. on Neural Networks*, 16(5):1027– 1041, Sep 2005.

[9] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[10] A. Mirgorodskiy. Automated problem diagnosis in distributed systems. Technical Report Unpublished, University of Wisconsin-Madison, 2006.

[11] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *DSN*, pages 802–811, 2005.

[12] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP*, pages 206–220, Brighton, UK, Oct 2005.

[13] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[14] E. Thereska, O. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Sigmetrics*, pages 3–14, Saint-Malo, France, 2006.