Behavior-Based Problem Localization for Parallel File Systems

Michael P. Kasick, Rajeev Gandhi, Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University; Pittsburgh, PA 15213–3890
{mkasick, rgandhi, priyan}@andrew.cmu.edu

Abstract

We present a behavior-based problem-diagnosis approach for PVFS that analyzes a novel source of instrumentation—CPU instruction-pointer samples and function-call traces—to localize the faulty server and to enable root-cause analysis of the resource at fault. We validate our approach by injecting realistic storage and network problems into three different workloads (dd, IOzone, and PostMark) on a PVFS cluster.

1 Introduction

Large scientific applications exhibit compute-intense behavior intermixed with periods of intense parallel I/O, and therefore, depend on file systems that support high-bandwidth concurrent writes. The Parallel Virtual File System (PVFS) [2] is an open-source, parallel file systems that provides such applications with high-speed data access to files. PVFS has a client-server architecture, with many clients communicating with multiple I/O servers and one or more metadata servers.

Diagnosing performance problems is important in high-performance computing (HPC) where the effects of these problems are magnified due to long-running, large-scale computations. Our previous work sought to diagnose PVFS performance problems through system-call service times [3] and OS-level performance metrics [4]. While this performance-metric driven diagnosis was effective, we discovered that problems with a lower-but-still-significant performance impact (e.g., *network-hog*s) are difficult to diagnose from performance metrics alone.

Our contribution in this paper is a *problem-diagnosis approach that analyzes system behavior*, as captured by the servers' executing functions' (i) computational demand (derived from CPU instruction-pointer samples), (ii) call frequency, and (iii) execution time. To the best of our knowledge, this is the first approach to exploit *both* sample profiles and function-call traces (traditionally used for single-node optimization) as instrumentation sources for distributed fault localization. In doing so, we automatically localize the faulty server and to enable manual root-cause analysis by highlighting the functions that are most impacted by the problem. We validate our approach by studying realistic storage and network problems injected into three file-system benchmarks (dd, IOzone, and PostMark).

2 Problem Statement

Our research is motivated by the following questions: (i) *do the behaviors captured by sample profiles and function-call traces enable the diagnosis of performance problems in parallel file systems*, and (ii) *if so, how do the two methods differ in their capability and overhead?*

Hypotheses. Under a performance fault in a PVFS cluster, metrics derived from sample profiles and function-call traces should exhibit observable anomalous behavior on the culprit servers. Additionally, the statistical trends of these metrics: (i) should be similar (albeit with inevitable minor differences) across fault-free I/O servers, even under workload changes, and (ii) should differ on the culprit server, as compared to the fault-free I/O servers. This hypothesis drives our *peer-comparison* diagnosis approach, where we statistically compare the same metrics across servers to discover the faulty server.

Assumptions. We assume that a majority of the I/O servers exhibit fault-free behavior, that all server nodes have identical software configurations, and that the clocks on the various nodes are synchronized so that instrumented data can be temporally correlated across the system. We also assume that clients and servers run on homogeneous hardware and execute similar workloads.

3 Instrumentation

3.1 Sample Profiling

Sample profiling is the act of periodically sampling the CPU instruction pointer to determine what (program and function) the CPU is executing. Sample counts serve as a statistical approximation of the amount of time that CPU spends executing a particular program and function.

We use OProfile 0.9.6 to collect sample profiles on each PVFS server. By default, OProfile generates a non-maskable interrupt for every 100,000 cycles that a CPU executes outside of the halt state (`CPU_CLK_UNHALTED` events), during which it records the instruction pointer along with execution contexts. We run OProfile with the `-separate=kernel` option so that OProfile can attribute samples to an executing program name (user-space processes and kernel threads), a binary image name (executables, shared libraries, and kernel modules), and a function name (if symbols are present in the binary image). We also modified OPro-

file to periodically save profiles into session directories at a fixed time interval, allowing us to reconstruct sample profiles for (and thus, localize faults to) specific windows of time. Currently, we generate sample profiles at an interval of 10 seconds, which is a low enough resolution to avoid significant jitter. While this method of instrumentation does not require application source code, it does require function symbols to be present in the binary image in order to attribute samples to individual functions.

Although sample profiling is not perfectly accurate—in particular, it has difficulty observing functions that execute quickly but infrequently—it has the advantage of being able to collect data globally at low overhead, including all user-space processes and kernel threads.

3.2 Function-Call Tracing

Function-call tracing is the act of recording function-call entries and exits in a given program. From these traces, we derive a profile of function-call *count* (the number of times a particular function is called) and execution *time* (the amount of time spent executing within a function).

Our custom, C-based instrumentation module, along with GCC's `-finstrument-functions` option, enables function-call tracing in PVFS at compile-time. The GCC option automatically instruments all PVFS function-call sites, calling our module on entry and exit events. On entry, we record the function address and entry-time to a thread-local stack. On exit, we increment the call count and compute the execution time by subtracting any child (called) function execution times from the wall-clock-elapsed time since entry. This execution time includes both the time that the CPU spends executing the function as well as any time spent, potentially blocked, in system calls. This time is then added to the parent stack frame as part of its child execution-time. Finally, the current stack frame is popped and the incremented call count & time are added to the current profile—a global table of function addresses with accumulated call counts & times. Finally, we write this profile out to a trace file once every second. Subsequently, the in-memory profile is zeroed to await new data.

Our current function-call tracing method requires access to application source-code but does not need to modify source-code as the module is injected at build-time. While our module currently targets threaded, single-process programs, support for multi-process programs is possible by recording profiles in a shared memory segment, and using `ptrace` to trace process fork and exit events. Alternatively, if source-code is not available then binary-instrumentation can be used [7].

Function-call traces provide exact metrics, and not approximations as sample profiles do. This allows us to observe call trends that otherwise escape in sampled data.

4 Experimental Set-up

We perform our experiments on 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 5.0 (lenny) with Linux kernel 2.6.26 and PVFS 2.8.1. The machines run in stock configuration with no background tasks. We conduct experiments with 10 combined I/O and metadata servers and 10 clients.

PVFS is used in the default server configuration with the same three modifications used in our previous work [4]. First, we use the Direct I/O method to bypass the Linux buffer cache for PVFS I/O server storage, a requirement for diagnosis in our previous work that is preserved here both for comparability of results and since it improves large write throughput by 10%. Second, we increase the Flow buffer size to 4 MB, which is is standard practice in PVFS performance tuning and is required to maximize our testbed performance. Finally, we patch the PVFS kernel client to eliminate the 128 MB total size restriction on the device request buffers, and we invoke the client with 64 MB request buffers in order to make the 4 MB data transfers to each of the I/O servers.

The nodes are rebooted prior to the start of each experiment. Time synchronization is performed at boot-time using `ntpdate`. In the sample-profiling experiments, the OProfile monitoring daemon starts capturing profiles to a local (non-storage dedicated) disk once the servers are initialized and the client is mounted. In the function-call tracing experiments, the PVFS server daemon writes trace profiles to the same local disk used for sample profiling. `sync` is then performed, followed by a 15-second sleep, then the experiment benchmark starts. The benchmark first runs fault-free for 120 seconds. Then, a fault is injected for 300 seconds and deactivated after. The experiment continues to the completion of the benchmark, which is at least 180 seconds after fault deactivation to determine if there are any delayed effects. We run ten experiments for each workload and fault combination, using a different faulty server each time.

4.1 Workloads

One of five experiment workloads (derived from dd, IOzone, and PostMark) is invoked concurrently on all clients. The first two workloads, `ddw` and `ddr`, either write zeros (from `/dev/zero`) to a client-specific temporary file or read the contents of a previously written temporary file and write the output to `/dev/null`. dd performs a constant-rate large-file read/write from/to disk that models the behavior of scientific-computing workloads with constant data-write rates.

Our next two workloads, `iozonew` and `iozoner`, consist of the same 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, an `fsync`, and a workload change half-way through. 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 I/O server per file).

Configurations of Workloads. For `ddw` and `ddr`, we use 19 GB and 28 GB files respectively with a record-size of 40 MB for both. File sizes are chosen to result in a fault-free runtime of approximately 600 seconds. The record-size was chosen to result in 4 MB data transfers to each I/O server, which we empirically determined to be the knee of the performance vs. record-size curve. For `iozonew` and `iozoner`, we use 9 GB and 12 GB files respectively with a record-size of 16 MB (the largest that IOzone supports). For `postmark`, we use 14,000 transactions for sample profiling and 6,500 for function-call tracing to give sufficiently long-running benchmarks.

4.2 Fault Injection

In fault-induced experiments, we inject a single fault into an I/O server to induce degraded storage or network performance. Our faults are modeled after real-world problems experienced by operators of PVFS clusters [4]:

- *disk-hog*: a `dd` process that reads 256 MB blocks (using direct I/O) from an unused storage disk partition.
- *disk-busy*: an `sgm_dd` process that issues low-level SCSI I/Os via the Linux SCSI Generic (`sg`) driver to read 1 MB blocks from the same unused partition.
- *network-hog*: a third-party node opens a TCP socket to a listening port on a server and sends zeros to it (*write-network-hog*), or a server opens a socket and sends zeros to a third party node (*read-network-hog*).
- *pktloss*: a netfilter firewall rule that drops packets received at a single server with probability 5% (*receive-pktloss*), or a rule on all clients that drops packets sent from one server with probability 5% (*send-pktloss*).

5 Diagnosis Algorithm

Our diagnosis algorithm has two objectives: (i) to automatically identify the faulty server by analyzing *samples*, *count*, and *time* metrics across servers; and (ii) to enable manual root-cause analysis by identifying the functions most affected, indirectly hinting at the resource at fault.

5.1 Finding the Faulty Server

Overview. To find the faulty server we peer-compare *samples*, *count*, and *time* metrics across servers to determine those behaving anomalously. We analyze one metric at a time across all servers. On each server, we generate F-dimensional feature vectors, where each component of the feature vector contains the sum of the metric quantity (*samples*, *count*, or *time*) attributed to a particular function over a time window of *WinSize* seconds. We then compute the distance between feature vectors for each pair of servers, which represents the degree to which servers behave differently. We then flag a server as anomalous over a window if its median feature-vector distance (relative to the other servers) exceeds a predefined threshold. We then shift the window by $WinShift$ seconds, leaving an overlap of $WinSize - WinShift$ seconds between consecutive windows, and repeat the analysis. We classify a server to be faulty if it shows anomalous behavior for at least k of the past $2k-1$ windows.

Feature-Vector Dimensionality. For a particular metric (*samples*, *count*, or *time*), the maximum dimensionality (F) of the feature vector is the number of functions to which the metric is attributed at least once, across all nodes, in a single experiment. We observe, however, that this creates feature vectors with very large dimensionality (an average of 7446 for *samples* and 765 for *count*/*time*). However, many components of these feature vectors have small values (and thus, little diagnosis influence) as they represent functions that execute infrequently in PVFS. Thus, we reduce the dimensionality of the feature vectors by including only those functions that contain a minimum of 100 *samples* or 1 sec *time* for at least one window on a single node, reducing the dimensionality to 201 for *samples* and 10 for *count*/*time*.

Window Aggregation & Anomaly Filtering. We use a window of 60 seconds (*WinSize*) to generate the feature vectors for our analysis. Thus, the components of the feature vectors in each window contain the six most recent per-function *samples* sums, or the 60 most recent per-function *count* or *time* sums. We use a $WinShift$ of 30 seconds between each window, leaving a consecutive window overlap of 30 seconds. This aggregation ensures that each window reflects average behavior of PVFS request processing. However, the aggregation process increases the diagnosis latency since samples have to be collected for 60 seconds (*WinSize*) before they can be analyzed. In general, we find that diagnosis is insensitive to *WinSize*s and $WinShift$s of 60+ and 30+ seconds, respectively. We classify a server as faulty if it shows anomalous behavior for 3 out of the past 5 windows ($k = 3$). This filtering process reduces false-positives in the event of sporadic anomalies when no underlying fault is actually present, but adds to the diagnosis latency. The combined effect of aggregation and anomaly filtering results in a fault-injection-to-diagnosis latency of 90 seconds.

Distance Measure. We use the Manhattan distance, $d(\vec{p},\vec{q}) = \sum_{i=1}^{F} |p_i - q_i|$, to compute a distance measure between two feature vectors, $\vec{p}$ & $\vec{q}$. The Manhattan distance, which is also used in [7], is a measure of absolute distance that performs well to discriminate anoma-

lous features. We also tried relative distance measures, $d(\vec{p},\vec{q}) = \sum_{i=1}^{F} \frac{|p_i - q_i|}{\max(p_i, q_i)}$; relative scaling of absolute distance, $d(\vec{p},\vec{q}) = \sum_{i=1}^{F} \frac{(p_i - q_i)^2}{\max(p_i, q_i)}$; and symmetric KL divergence, $d(\vec{p},\vec{q}) = \frac{1}{2}\sum_{i=1}^{F}\left(p_i \log \frac{p_i}{q_i} + q_i \log \frac{q_i}{p_i}\right)$. While some of these produced better results in certain experiments, we found Manhattan results to be the best overall.

Threshold selection. The distance thresholds used to differentiate faulty from fault-free servers are determined through a fault-free training phase that captures the maximum expected deviation in server behavior. Instead of training against all potential workloads, we train on workloads that are expected to stress the system to its limits of performance. Since server performance (and thus, behavior) deviates the most when resources are saturated, these thresholds represent the maximum expected behavioral deviation under normal operation.

In our experiments, we train with 10 fault-free iterations of `ddr`, `ddw`, and (optionally) `postmark`. For each metric, we perform a binary search of threshold values until the minimum integer threshold is determined that eliminates all anomalies on a particular server. This server-specific threshold is doubled to provide a cushion that masks minor manifestations occurring during faults.

5.2 Root-Cause Analysis

To enable manual root-cause analysis we identify the functions most affected by a performance problem. For each faulty server, we compute the component-wise sum, across each anomalous window, of a metric's component distances to the median node. We then rank the component (function) sums, and present the top ten anomalous functions of that server for further inspection.

6 Results

Table 1 shows the accuracy rates of our diagnosis algorithm using *samples*, *count*, and *time* metrics. We present two sets of accuracy rates, for both when `postmark` is included and excluded from training and testing data. The *combined* columns shows the better true-positive rate and worse false-positive rate when inspecting (i) both *count* and *time* metrics simultaneously, or (ii) *samples* alone. This provides a lower-bound approximation of using both sample profiling and function-call tracing in the same experiment. In general, we note that while certain metrics are excellent discriminators of specific faults, no single metric is alone sufficient to diagnose a variety. However, when considering the combination of all three metrics, nearly all fault types are diagnosable.

We present data both without and with `postmark` to illustrate the diagnosis capability for (i) the specific class of large-I/O comprising workloads that parallel file systems target, and (ii) the more general class of large-I/O, small-I/O, and metadata-heavy workloads. Our diagnosis capability is greater for the large-I/O workload subset since `postmark`'s random requests and uneven metadata distribution results in behavioral asymmetry across fault-free servers, attenuating the *count* metric capability.

6.1 Fault Manifestations

The *disk-hog* and *disk-busy* faults exhibit similar behavioral manifestations; both introduce disk contention that significantly increases the service time of storage I/O requests. Since PVFS storage I/O is performed in separate threads that issue blocking I/O calls, *time* metric asymmetries best discriminate storage-related faults due to the increased time spent blocked on disk I/O. Since PVFS storage I/O uses relatively few CPU cycles and is synchronized by client requests, the *disk-hog* and *disk-busy* influence on *samples* & *count* metrics is less prominent.

The *network-hog* faults significantly increase TCP traffic volume and primarily manifest as increases in kernel-level computation (CPU cycles) to validate and process data & ACK packets. Thus the *samples* metric, which best discriminates computational asymmetries and is collected globally at kernel & user levels, is the metric most influenced by this fault. As *network-hog*s have relatively little behavioral influence on the PVFS server process, the *count* & *time* metrics are mostly unaffected.

The *pktloss* faults manifest as disruptions in PVFS network I/O which reduces the network throughput rate. In PVFS, network I/O is implemented with a poll loop that performs non-blocking socket reads as soon as any network data is received and socket writes when the write buffer is half-depleted. During *receive-pktloss* there is an asymmetric increase in non-blocking socket read calls, which affects the *samples* and especially *count* metrics, as network data is received over a longer period of time and each call returns less data compared to fault-free servers. However, this capability is limited to write-heavy loads as otherwise there is insufficient incoming data to discriminate read counts. During *send-pktloss* the number of write calls made is unaffected as the amount of data written per-call is independent of the network transfer rate. Although there is an increase in timed-out poll operations, their effect on the *count* metric is insufficient to reliably diagnose *send-pktloss* problems. Since PVFS network I/O does not make use of long-running or blocking function calls, the *time* metric is unaffected.

6.2 Root-Cause Analysis

A cursory investigation of the top ten reported anomalous functions indicates that root-cause analysis is possible for some combinations of faults and metrics. *Disk-* and *network-hog*s exhibit *samples* attributed to specific rogue processes (e.g., `dd` and `socat`). *Disk-hog* & *disk-busy* faults on read workloads exhibit *time* blocked in

	Without `postmark` training or testing								With `postmark` training & testing							
Fault	*samples* (%)		*count* (%)		*time* (%)		*combined* (%)		*samples* (%)		*count* (%)		*time* (%)		*combined* (%)	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
None (control)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	2.0	0.0	0.0	0.0	2.0
disk-hog	12.5	2.5	77.5	0.0	75.0	0.0	92.5	2.5	0.0	0.0	0.0	4.0	70.0	0.0	70.0	4.0
disk-busy	20.0	0.0	42.5	0.0	65.0	2.5	77.5	2.5	0.0	0.0	0.0	0.0	62.0	2.0	62.0	2.0
write-network-hog	100.0	0.0	7.5	0.0	0.0	0.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0
read-network-hog	100.0	0.0	27.5	2.5	0.0	10.0	100.0	10.0	100.0	4.0	0.0	2.0	0.0	8.0	100.0	10.0
receive-pktloss	30.0	2.5	50.0	0.0	0.0	2.5	50.0	2.5	0.0	4.0	8.0	2.0	0.0	2.0	8.0	4.0
send-pktloss	2.5	0.0	25.0	0.0	0.0	0.0	25.0	0.0	0.0	0.0	0.0	6.0	0.0	0.0	0.0	6.0
Aggregate	44.2	0.7	38.3	0.4	23.3	2.1	74.2	2.9	33.3	1.4	1.3	2.3	22.0	1.7	56.7	4.0

Table 1: Diagnosis accuracy (true- and false-positive) rates. TP is the percentage of experiments where all faulty servers are correctly indicted as faulty, FP is the percentage where at least one non-faulty server is misindicted.

Overhead for Workload	Instrumentation	
	Sample Profiling	Call Tracing
`ddr`	–1.43% ± 0.55%	2.39% ± 0.77%
`ddw`	2.67% ± 1.32%	1.28% ± 1.75%
`iozoner`	–0.28% ± 0.25%	0.74% ± 0.28%
`iozonew`	3.23% ± 1.47%	0.82% ± 1.14%
`postmark`	–0.36% ± 1.06%	122.18% ± 3.72%

Table 2: Instrumentation overhead: Increase in runtime w.r.t. non-instrumented workload ± standard error.

a high-ranking `dbpf_pread` function, indicating that time is spent blocked on storage read syscalls. *Count*-manifesting faults exhibit no specific high-ranking attributable functions, although their rank/ordering may serve as useful signatures for root-cause categorization.

6.3 Instrumentation Overhead

Table 2 reports runtime overheads for sample profiling and function-call tracing for our five workloads. Overheads are calculated as the increase in mean workload runtime (for 10 iterations) with respect to their uninstrumented counterparts. Negative overheads are result of sampling error, which is high due runtime variance across experiments. We conclude that sample profiling and function-call tracing have comparable overheads for the large-I/O workloads, both less-than 7% with 98% confidence. While `postmark`'s sample profiling overhead is also comparable, its function-call tracing overhead is extremely high, representing a deployment challenge for function-call tracing for general workloads.

6.4 Comparison to Performance Metrics

The *combined* rate of the behavior-based approach is comparable to that of our performance metric diagnosis (avg. 69.8% TP & 2.6% FP) [4]. Instrumentation overhead is also comparable except for `postmark`, which in [4] is superior and consistent with the other workloads.

In *disk-hog* and *disk-busy* problems, the *time* metric is equivalent to [4]'s `await` (average I/O wait time) metric, scaled to the size of the PVFS I/O requests. [4] better diagnoses these problems since their behaviors manifest in delayed performance (blocked storage-I/O call time), and `await` is a more direct representation of that metric. In contrast, the behavior-based approach more reliably diagnoses *network-hog* faults, which [4] occasionally masks, as these faults manifest greater in behavior (extra TCP computation) than in performance metrics.

Both approaches diagnose *receive-pktloss* similarly well, but [4] requires client-side instrumentation to observe `cwnd` (TCP sending congestion-window) metrics, whereas the behavioral approach uses server-side instrumentation only. Unfortunately, the instrumentation used here is insufficient to reliably diagnose *send-pktloss*; thus the performance metric approach performs much better.

7 Future Directions

7.1 Analysis

Currently our diagnosis algorithm does not consider the relevance of specific functions in discerning the presence of a fault. We observe across fault-free experiments that some functions exhibit considerable variance within the same node in their *samples* & *count* metrics due to sampling error and random behavior. These random effects, especially in `postmark`, also serve as contributing factors to cross-node variance within the same experiment, and thus, inflate the "maximum expected deviation" captured in our training. This may result in anomalies going undetected which exhibit less overall deviation, but whose deviation is concentrated in a few highly relevant functions that illustrate minimal fault-free variance.

To improve diagnosis sensitivity in the presence of random behavior we may compute the component-wise variances of feature vectors across fault-free experiments and weight components by their observed variance (e.g., divide a component by its standard deviation). Thus, functions that exhibit greater random-behavioral variance (those which are less relevant) would be deemphasized in the overall feature-vector distance, and functions that exhibit little random-behavioral variance (those which are more relevant) would have greater influence.

More generally, we may apply weighting factors to functions known to be semantically or empirically (ir)relevant to the presence of previously-observed faults.

In particular, we may add weight to functions discovered through root-cause analysis to be indicative of past problems, and we may decrease weight (or eliminate) functions discovered to signal false-positives.

7.2 Instrumentation

Missing in our instrumentation is the ability to trace kernel function calls. Kernel-level tracing may be achieved using the same general approach as our user-level instrumentation module. We expect that kernel-level function-call tracing would significantly improve diagnosis of *send-pktloss* faults during read-heavy workloads. Such faults manifest in TCP retransmits visible at the kernel-level, but are observable in the PVFS server daemon only from an increased number of timed-out poll operations.

7.3 Overhead Reduction

The high `postmark` function-call tracing overhead is due the high number of I/O and metadata operations relative to average I/O request size. I/O-bound workloads that utilize small requests make many more PVFS function calls in the same amount of time as workloads utilizing large requests, and thus, significantly increase instrumentation overhead. We propose that this overhead may be reduced by *selectively* instrumenting function call sites. For example, by instrumenting only the call sites that are used in faulty feature vectors, we may reduce the total number of instrumented call sites from 863 to 14. Unfortunately this call site selection requires observation of prior faults to determine which functions need to be included in feature vectors. Previously-unseen faults may go undetected if they manifest in functions not selected for inclusion. Alternatively, we may selectively exclude call sites that are determined via profiling to be frequently called, but which are empirically (via root-cause analysis) or semantically (through code inspection) determined to be irrelevant to diagnosis.

8 Related Work

Mirgorodskiy et al. [7] localizes code-level problems by tracing function calls and peer comparing execution times across nodes to identify anomalous nodes in an HPC cluster. Their debugging tool is designed to locate the specific functions where problems manifest and is demonstrated in a qualitative cluster manager case study. Our approach utilizes execution time along with metrics of function call frequency and computational demand, and is quantitatively assessed in its capability to diagnose a variety of performance problems, many of which escape diagnosis when using execution time alone.

Previous problem diagnosis in Internet Services trace request flows using intercomponent messages (e.g., RPCs) to identify request paths with abnormally long latencies [1] or to identify processing components responsible for failed requests [5]. Our work uses function calls instead of RPCs to model behavior due to relative ease of instrumentation, but shares the goal of determining the components (servers) exhibiting abnormal behavior.

Sample profiling and function-call tracing have long been used to locate performance bottlenecks in program code. Paradyn [6] and Tau [8] are profiling and tracing tools targeted at HPC to measure performance in parallel programs and isolate sources of performance problems.

9 Conclusion

Our new behavior-based problem-diagnosis approach for performance faults in PVFS demonstrates the viability of sample profiling and function-call tracing for problem diagnosis. While neither instrumentation is alone sufficient to diagnose each fault, the combination of both enables diagnosis of nearly all types. Our diagnosis approach automatically identifies the faulty server, and enables manual root-cause analysis by identifying the functions most affected by a performance problem.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, Bolton Landing, NY, Oct. 2003.

[2] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.

[3] M. P. Kasick, K. A. Bare, E. E. Marinelli III, J. Tan, R. Gandhi, and P. Narasimhan. System-call based problem diagnosis for PVFS. In *Hot Topics in System Dependability*, Lisbon, Portugal, June 2009.

[4] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *8th USENIX Conference on File and Storage Technologies*, San Jose, CA, Feb. 2010.

[5] E. Kıcıman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16(5), Sept. 2005.

[6] 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, Nov. 2005.

[7] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *Supercomputing*, Tampa, FL, Nov. 2006.

[8] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Intl. Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.