A Reproducible I/O Analyzer

Rodrigo P. do Nascimento[1], **Prof. Dr. Alfredo Goldman Vel Lejbman**[2]

[1]Instituto de Pesquisas Tecnológicas - IPT
São Paulo - SP - Brazil

[2]Instituto de Matemática e Estatística - Universidade de São Paulo
São Paulo - SP - Brazil

rodrigo.nascimento@ensino.ipt.br, gold@ime.usp.br

Abstract. *This paper presents the Stealth Metrics Framework, an I/O characterization and data-driven I/O replay framework that users can utilize to gain detailed insights into how their applications issue I/O to storage systems. The study provides a clear snapshot of how a MongoDB instance manages recovery from a crash from a storage I/O perspective, and explores potential strategies to enhance recovery performance.*

1. Introduction

Any organization that utilizes computer systems aims to use the computational resources of these systems as efficiently as possible. This is closely linked to understanding the performance capacity of a system. The extensive use of computing in science is a concrete example that maximizing the performance of computer systems is a matter of extreme importance [Wright 2019].

Performance is generally quantified through metrics such as throughput, utilization, and response time, yet measuring a system's performance encompasses a broad range of factors. Several HPC applications rely on I/O, therefore it is also important to understand the behavior of the file system to improve performance.

[Reisel et al. 2020] pointed out the projected production of approximately 175 ZettaBytes of data by 2025 across various platforms. Therefore understanding the I/O characteristics of applications to enhance storage system performance is crucial for optimizing overall system performance. Tracing I/O system calls is crucial for analyzing application I/O characteristics, with strace being a widely used tool. However, as [Gebai and Dagenais 2018] have shown, its overhead makes it unsuitable for production environments.

This paper introduces a non-intrusive method for not only characterizing I/O operations for Linux-based applications, but also replaying these I/O operations regardless of the storage subsystem type.

2. Background

The Linux Kernel I/O System Calls. A system call is a controlled entry point into the kernel, allowing a process to request that the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication

[Kerrisk 2010]. Therefore system calls are crucial for executing tasks that require protected access to hardware resources, which cannot be directly accessed from user space.

The Linux Kernel Tracepoint System. The Linux kernel's tracepoint system is a sophisticated mechanism designed for monitoring and debugging kernel operations with minimal impact on system performance. According to [Desnoyers 2020], tracepoints are effectively placed hooks within the kernel code that can be activated by users to gather diagnostic and performance data. This feature is essential for comprehensively understanding the Linux kernel's behavior, particularly in production settings where stability and performance are crucial.

eBPF - extended Berkeley Packet Filter. The Extended Berkeley Packet Filter (eBPF) is a significant feature of the Linux kernel that enables the secure execution of concise programs within kernel space, without the need to modify kernel code or load modules. Originally developed for network packet filtering, eBPF's functionality has expanded to include a general-purpose framework for executing code in response to diverse events such as system calls, network activities, and tracepoints. eBPF programs, written in a restricted C-like syntax and compiled into bytecode, can be linked to multiple kernel hooks, facilitating their activation by specified events.

FIO - Flexible I/O. Flexible I/O (FIO) is a widely-used, open-source tool for benchmarking and stress testing storage system performance, compatible with Linux, Windows, and macOS. It is valued in industry and academia for its versatility, configurability, and detailed performance analytics. It can replay I/O operations from trace files, simulating real-world workloads with high accuracy — an advanced feature crucial for benchmarking, testing, and analyzing storage system performance and reliability under specific recorded conditions — this facilitates detailed testing and analysis in a controlled environment.

3. Stealth Metrics Framework

The Stealth Metrics Framework (SMF) is designed for low-impact tracing of I/O workloads in production systems, featuring an architecture that supports workload characterization and data-driven I/O replay. It consists of four main components: a non-intrusive I/O tracer that minimally disrupts system operations, an I/O trace parser that organizes captured data, an I/O visualizer for graphical workload representation, and an I/O trace generator for replicating workloads through synthetic operations.

The Non-Intrusive eBPF-based I/O Tracer. The bpf-io-trace.bt (biot) script, written in the higher-level eBPF language bpftrace, strategically attaches probes to kernel-level I/O system calls using the kernel's tracepoint system. It captures data and transfers it to an eBPF performance ring buffer, a data structure optimized for high-performance logging. This buffer allows asynchronous data consumption from user space, separating the data capture from processing. This component operates within the server that initiates the I/O requests.

The I/O Trace Parser. The I/O Trace Parser, a Python script, utilizes a publisher/subscriber design pattern. In this setup, a file reader instance sequentially reads and publishes each line from a trace file to its subscribers. There are three subscribers in the current implementation: one counts all system calls during the observed period, another counts system calls by file, and the third generates a FIO trace file.

The I/O Characterization Views. A Jupyter notebook is used to display data visualization charts that assist in I/O characterization.

4. Results and Discussion

This section demonstrates the potential of the Stealth Metrics Framework through its application in analyzing the I/O patterns of a MongoDB database during its recovery following a crash.

Beginning with an analysis of the system call distribution created by SMF, as shown in Figure 1, it is evident that read operations are the predominant component of the I/O workload.

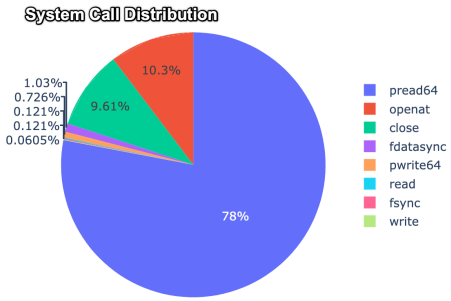

Figure 1. System call distribution.

Subsequently, an examination of the top 10 files ranked by the total number of system calls reveals significant activity on the journal files since they feature the top 3 in the list, as illustrated in Figure 2.

	openat	write	fsync	total	read	close	pread64	fdatasync	pwrite64
/data/db/journal/WiredTigerLog.0000000004	8	0.0	0.0	641	0.0	8.0	624.0	1.0	0.0
/data/db/journal/WiredTigerLog.0000000005	1	0.0	0.0	627	0.0	0.0	618.0	4.0	4.0
/data/db/journal	132	0.0	0.0	264	0.0	131.0	0.0	1.0	0.0
/data/db/WiredTiger.turtle	11	0.0	0.0	33	0.0	11.0	11.0	0.0	0.0
/data/db/WiredTiger.wt	1	0.0	0.0	13	0.0	0.0	6.0	2.0	4.0
/data/db/index-3--4169581049979291868.wt	1	0.0	0.0	6	0.0	0.0	5.0	0.0	0.0
/data/db/_mdb_catalog.wt	1	0.0	0.0	6	0.0	0.0	5.0	0.0	0.0
/data/db/collection-2--4169581049979291868.wt	1	0.0	0.0	6	0.0	0.0	5.0	0.0	0.0
/data/db/collection-0--4169581049979291868.wt	1	0.0	0.0	6	0.0	0.0	5.0	0.0	0.0
/data/db/index-1--4169581049979291868.wt	1	0.0	0.0	6	0.0	0.0	5.0	0.0	0.0

Figure 2. Top 10 files ranked by total number of system calls.

Delving deeper into the analysis, the data presented in Figure 3, captures in detail the mongod process as it opens and processes the WiredTiger journal data located at /data/db/journal/WiredTigerLog.0000000004. The process begins with the database reading 128-byte segments starting from offset 0, continuing in sequence to offset 128. It then jumps to offset 4992 and continues reading up to offset 5632 in similar 128-byte steps. After this point, the reading process remains sequential but with a substantial increase in the request size to 1,048,576 bytes.

This detailed observation allows us to conclude that the file access pattern is sequential, as the mongod process methodically progresses through the file, moving from one offset to the next in increments determined by the request size.

How can the findings from this analysis enhance performance? According to [MongoDB 2024], it is advised to set storage device read-ahead to a low value like 8KB due to mainly random reads in typical MongoDB operations. Yet, for a MongoDB instance recovering from a crash, the data suggests increasing read-ahead to a value larger than the request size could be advantageous. Such an adjustment might speed up recovery by prefetching more data into the cache, thereby boosting read throughput.

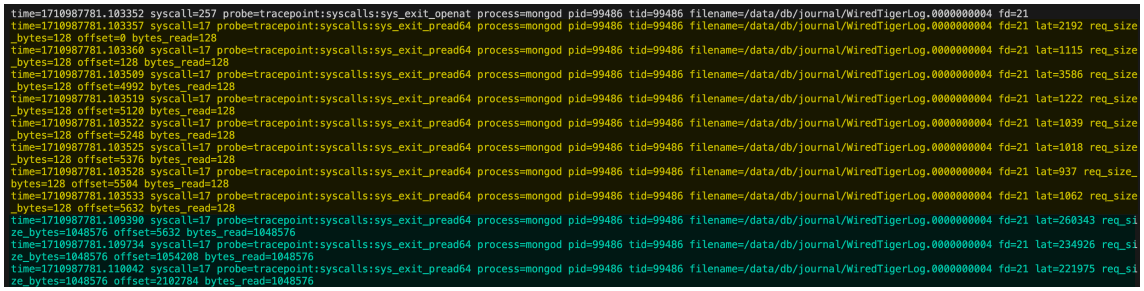

Figure 3. mongod process reading a journal file for recovery.

Limitations. Currently, the Stealth Metrics Framework is unable to provide information on applications conducting I/O operations via memory-mapped files. Although the data collected by the eBPF-based tracer is comprehensive and rich in details, enhancements in data visualization within the framework are ongoing to yield deeper insights into the I/O characteristics of workloads. Additionally, the development of the SMF subscriber responsible for producing the FIO trace file is still in progress.

5. Conclusion

The Stealth Metrics Framework shows significant promise in offering insights into I/O-bound workloads, aiming to equip developers and systems administrators with a thorough understanding of the I/O characteristics issued by applications. It is designed to capture sufficient information about these I/O requests to allow them to be comprehensively understood and replicated, regardless of the underlying storage system.

References

Desnoyers, M. (2020). Using the linux kernel tracepoints. Accessed on: 10 April 2024.

Gebai, M. and Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys*, 51(2):1–33.

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

MongoDB (2024). Production notes - mongodb manual. Accessed on: 11 April 2024.

Reisel, D., Gantz, J., and Rydning, J. (2020). The digitalization of the world from edge to core. White paper, International Data Corporation.

Wright, S. A. (2019). Performance modeling, benchmarking and simulation of high performance computing systems. *Future Generation Computer Systems*, 92:900–902.

This research is funded by FAPESP grant number 19/26702-8.