# Understanding Storage I/O Patterns Through System Call Observability

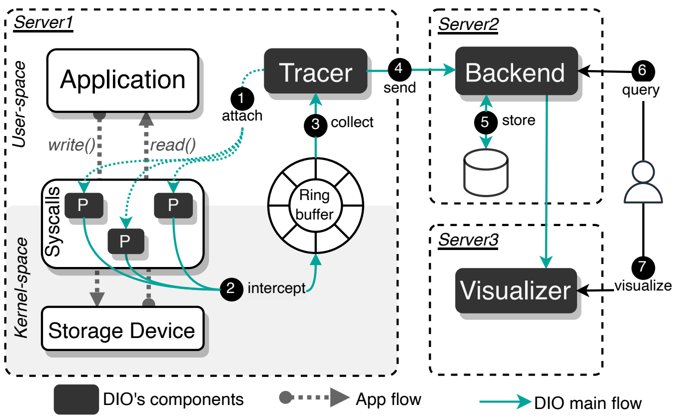
*Tânia Esteves, Ricardo Macedo, Rui Oliveira and João Paulo*
*INESC TEC & University of Minho*

## *Motivation*

- **Diagnosing inefficient I/O patterns** done by applications is **complex** and **time-consuming**.
- Existing tools suffer from **intrusiveness**, **high performance overhead**, **lack of analysis pipelines** or **narrowed scopes**.

<u>We introduce DIO, a practical solution that transparently traces applications' syscalls, parses collected data, and sends it to a pipeline for customized data analysis and visualization in near real-time.</u>

## *DIO in a Nutshell*




- The *tracer* uses eBPF to **automatically and non-intrusively** capture applications' syscalls information with enriched context from the kernel and forward it to the *backend*.
- The *backend* indexes the data and provides a querying API for accessing it and **building correlation algorithms**.
- The *visualizer* automatically queries the *backend* and summarizes the data through **customizable visualizations**.

## ***Use Case:*** *Finding the root cause of performance anomalies*

***Problem***: RocksDB clients observe high latency spikes.

***Diagnosis***: Using DIO to observe the syscalls submitted over time by different RocksDB threads *(Fig. 1)*, we see that when:

- **multiple compaction threads** perform I/O simultaneously, db_bench **performance decreases** (1&3).
- **few compaction threads** perform I/O simultaneously, db_bench **performance improves** (2&4).

***Root cause***: Latency spikes occur when threads compete for shared disk bandwidth, leading to performance contention.

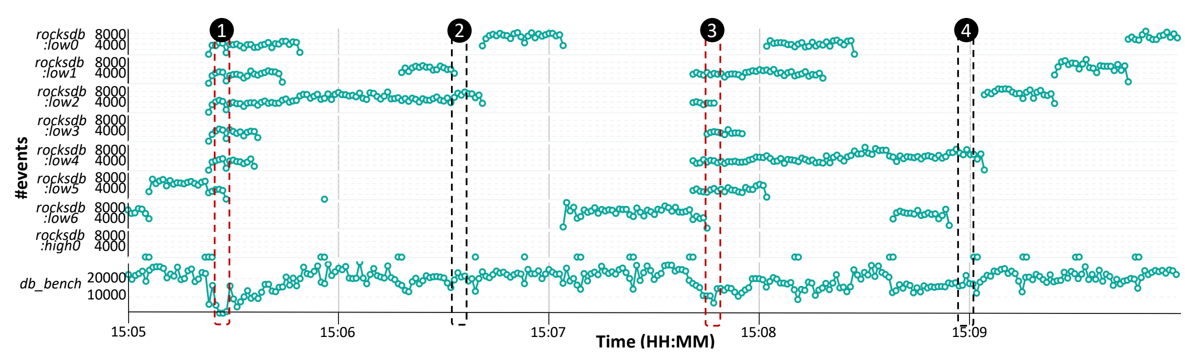



*Fig. 1 - Syscalls issued by RocksDB over time, aggregated by thread name. db_bench includes the 8 client threads, rocksdb:low[0-6] refers to each compaction thread, and rocksdb:high0 refers to the flush thread.*

## ***Use Case:*** *Identifying erroneous actions that lead to data loss*

***Problem***: Data loss when using Fluent Bit's tail input plugin.

***Diagnosis***: With DIO, one can observe that:

- *app* writes 26 bytes to offset 0 of "app.log" file.
- *fluent-bit* reads the whole content (26 bytes).
- *app* deletes the "app.log" file, creates a new one with the same name, and writes 16 bytes to offset 0.
- *fluent-bit* tries to **read from offset 26 instead of offset 0**, losing the 16 bytes written by *app*.

***Root cause***: Fluent Bit tracks the last processed offset for each inode, which is not reset when the file is removed.

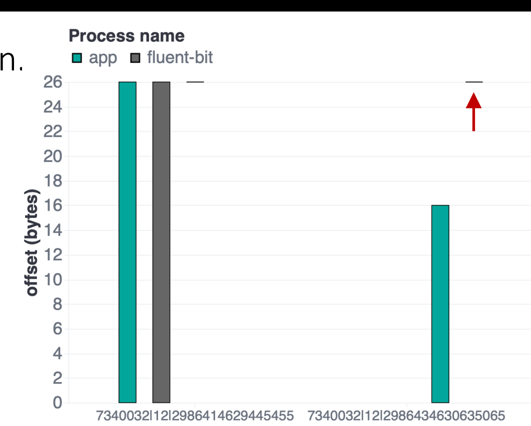



*Fig. 2 – Fluent Bit (v1.4.0) erroneous access pattern*

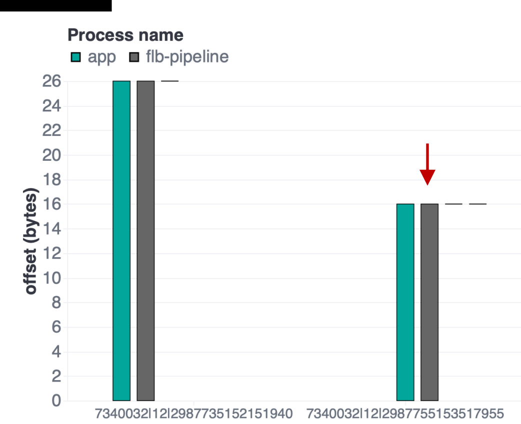



*Fig. 3 – Fluent Bit (v2.0.5) correct access pattern*

## *Future Directions*

### Automate the detection of key I/O patterns

Build correlation algorithms that:

- find sequences of syscalls repeated multiple times for a given file.
- find redundant operations, such as opening and closing a file for every write.

Scan me!

### Assist research in other areas like security

Analyze I/O patterns performed by malware to:

- observe and compare how different malware families interact with the storage.
- find distinctive I/O behavior to assist in building and improving malware detection tools.

***Acknowledgments:*** *This work was financed by the FCT - Portuguese Foundation for Science and Technology, through Ph.D. grant DFA/BD/5881/2020, and realized within the scope of the project POCI-01-0247-FEDER-045924, funded by the ERDF - European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalization and by National Funds through FCT, I.P. within the scope of the UT Austin Portugal Program.*

arXiv.2304.08569

tania.c.araujo@inesctec.pt