IOscope: A Flexible I/O Tracer for Workloads' I/O Pattern Characterization

Abdulqawi Saif, Lucas Nussbaum, Ye-Qiong Song

IOscope: A Flexible I/O Tracer for Workloads' I/O Pattern Characterization

Abdulqawi Saif[1,2], Lucas Nussbaum[1], and Ye-Qiong Song[1]

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
[2] Qwant Enterprise, F-88000, Épinal, France

Abstract. Storage systems are getting complex to handle *HPC* and Big Data requirements. This complexity triggers performing in-depth evaluations to ensure the absence of issues in all systems' layers. However, the current performance evaluation activity is performed around high-level metrics for simplicity reasons. It is therefore impossible to catch potential I/O issues in lower layers along the Linux I/O stack. In this paper, we introduce *IOscope* tracer for uncovering I/O patterns of storage systems' workloads. It performs filtering-based profiling over fine-grained criteria inside Linux kernel. *IOscope* has near-zero overhead and verified behaviours inside the kernel thanks to relying on the *extended Berkeley Packet Filter* (eBPF) technology. We demonstrate the capabilities of *IOscope* to discover patterns-related issues through a performance study on *MongoDB* and *Cassandra*. Results show that clustered *MongoDB* suffers from a noisy I/O pattern regardless of the used storage support (HDDs or SSDs). Hence, *IOscope* helps to have better troubleshooting process and contributes to have in-depth understanding of I/O performance.

1 Introduction

Storage systems become complex to keep pace with the requirements of both *HPC* and Big Data. The current way of evaluating storage systems, especially the data stores, has not changed adequately. It still focuses on high-level metrics which completely ignores potential issues in lower interfaces. For instance, studies like [12, 1, 11, 8] use *YCSB* [5] benchmark. Their core evaluation metrics are limited to workloads' throughput and execution time. One would know more why a given system achieves modest or strange results, but unfortunately such metrics cannot explain I/O performance. They only give indications that something goes wrong. Hence, in-depth evaluations such as evaluating I/O activities and interactions in lower layers is required. It leads to explain high-level measurements and to examine production workloads. Thus, our main concern here is to analyze how data files are accessed during workloads' execution, investigating if such experience leads to discover potential I/O issues.

Henceforth, we define workloads' I/O pattern as the order of files' offsets targeted by I/O requests during accessing on-disk data. Potential I/O issues may appear due to various reasons. For instance, reordering I/O requests in lower layers of I/O stack or a content distribution issue in the applicative layers

may transform a sequential access into a random access or vice versa. However, analyzing I/O pattern is less practiced during evaluations for two reasons. Firstly, there is a lack of specific tools to directly analyze in-production I/O workloads. Secondly, this is considered as an internal testing procedure which is often faced by a convention that all storage systems are well tested internally.

Tracing is highly used for evaluating storage systems [23]. Tracing tools often collect generic I/O traces from several layers of I/O stack. This leads not only to incur high overheads regarding the large number of diversified interceptions inside and outside the Linux kernel, but also to generate large quantity of tracing files that need huge effort for post-analysis. In contrast, multiple tracing tools are specific enough to collect precise data. However, they partially cover the dominant methods the storage systems use for issuing I/O workloads.

The key contributions of this paper are as follows. Firstly, we introduce the *IOscope*[3] tracer. *IOscope* applies both tracing and filtering techniques by relying on the *eBPF* which incurs a negligible overhead [21, 20]. *IOscope* addresses the above-mentioned limitations by generating specific and ready-to-visualize traces about workloads' I/O patterns; it also covers the dominant methods of issuing I/O workloads including *mmap I/O*. Secondly, we describe original experiments on *MongoDB* and *Cassandra* using *IOscope*. The results show that a pattern-related issue in clustered *MongoDB* is behind the performance variability of experiments. We then propose an ad hoc solution to fix that issue.

The rest of this paper is organized as follows. We describe *IOscope* design and validation is Section 2, before describing the performed experiments on *MongoDB* and *Cassandra* in Section 3. We present the related work of *IOscope* in Section 4. Finally, we conclude with Section 5.

2 IOscope Design & Validation

This sections presents *eBPF* before describing the *IOscope* design and validation.

2.1 Foundation: eBPF

The basic idea of *eBPF* is to inject byte-code programs into the kernel for extending a given kernel functionality. For example, injecting a program to perform statistics on a specific kernel function. *eBPF* consists of a virtual machine and a *syscall* called *bpf syscall*. The virtual machine has three major features. Firstly, it has eleven 64-bit registers. One of them is only readable for holding the frame pointer of the injected *eBPF* program. Secondly, it has an extended verifier which checks that the *eBPF* byte code is free of loops, has no side effect behaviors (e.g., could not lead to crashing the kernel), and terminates without problems. It also has an in-kernel data structures (*eBPF-maps*), which are accessible by user-space processes, suitable to use for data communications between *eBPF* and userspace programs [19]. Through those data structures, the *bpf syscall* bidirectionally transfers the data between the kernel/userspace pair. It carries out both

[3] https://github.com/LeUnAiDeS/IOscope

injecting the *eBPF* byte code into the kernel, and communicating the target kernel data towards a userspace process.

eBPF & Tracing
eBPF can be used to do in-kernel tracing thanks to the aforementioned features as well as its ability to connect to various data sources. Whenever an *eBPF* program is attached to a data source (e.g. a kernel function), the *bpf syscall* introduces a breakpoint on the target function. This allows *eBPF* stack-pointer to capture the functions' context. The *eBPF* program is then supposed to run as configured, i.e., before and/or after every single event on the target function). This program cannot alter the execution context of traced functions because the context is held by a not-writable register as mentioned in 2.1.

2.2 IOscope Design

IOscope contains two tools: *IOscope_classic* and *IOscope_mmap* (see *Fig. 1*). The key idea of *IOscope* is to construct the workloads I/O patterns by tracing and filtering the sequences of workloads' I/O requests. Both tools work with files offsets. An offset is a placement value (in byte) held by an I/O request. It indicates where is the beginning placement in the target file to read from or to write into. The tools apply many filters in earlier steps, e.g., reporting the I/O activities of a specific process or a specific data file. They aim at performing a precise-objective tracing and incurring less overhead. They are injected into the kernel via the *bpf syscall*. They are attached to the targets functions using the *kprobe & kretprobe* modes which allow to execute the tracing code before and after the target functions' body, respectively. Indeed, tracing based on internal kernel functions is dependent on kernel changes. However, the target functions of *IOscope* tools seem to be stable over multiple kernel releases.

IOscope tools are designed using the Python frontend of the *BCC* [18] project (a project that facilitates the development of eBPF-based tools). This implies having two processes for each tool. The first process which runs in userspace is responsible for 1) injecting the *IOscope*-core code into the kernel and 2) performing posterior filtering tasks on the received data from the kernel. The injected program runs as a second process inside the kernel. It intercepts the target

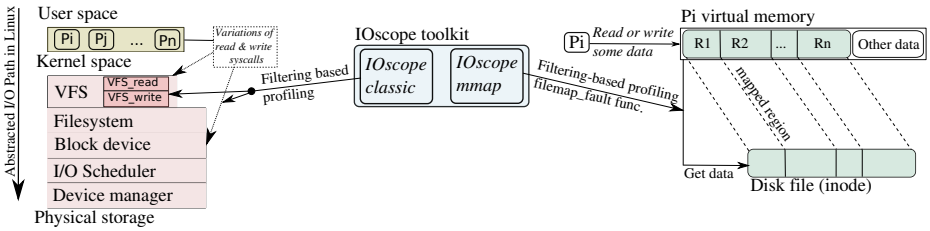


Fig. 1. *IOscope* tools and their instrumentation points inside the Linux kernel

functions to perform the filtering-based profiling task. It regularly exposes the matched traces into a ring buffer for which the userspace process is connected as a consumer.

a) IOscope_classic

IOscope_classic addresses different kinds of I/O methods. For instance, synchronous I/O, asynchronous I/O, and vectored I/O (scatter/gather I/O from multiple buffers). It obtains the tracing data from both 1) the *virtual file system* (*VFS*) as many I/O *syscalls* terminate at this layer in the kernel I/O path, and 2) from different *syscalls* that bypass the *VFS*. Thus, it covers almost all I/O methods that are based on the variations of *read, write syscalls* (e.g., *pread, pwritev, read, preadv, pwritev2*). This tool catches the mixed I/O workloads (read & write workloads) even if targeting the same file. It reports a pure view of how the analyzed system is accessing data over the execution time.

The tool should run for a specific userspace process in order to prevent profiling all the I/O requests found in the kernel. Thus, the *PID* of the target process must be provided to allow *IOscope* to check every issued request if it belongs to that process or not. If so, the tracing code of *IOscope_classic* will be executed, otherwise the I/O request continues its path without being traced.

IOscope_classic collects various informations from the target function parameters. The principal ones are the *offsets* of I/O requests, their *data size*, their *target file name* or *file identifier*, and the *request type* (read or write). This is done before executing the target function body. This information is stored in a hash map (a type of *eBPF-maps*). The tool also measures the latency for every I/O request as the elapsed time to execute the body of the target function. This indicates the time taken to write into or to read from a given offset on a target file. This is done by the *kretprobe* mode, which allows running a part of *IOscope_classic* code before the closing bracket of the target function.

b) IOscope_mmap

IOscope_mmap tackles the I/O activities that access the *memory mapped files* (*mmap*). This method allows the system to map the data files into its private address space, manipulating data like it is already located in memory. The CPU and some internal structures of the kernel bring out the data from the physical storage each time requested by a given process. Because of the absence of *syscalls* that carry the data access, the effective way to obtain the I/O access patterns would be to trace inside the kernel. However, the main challenge is to find a stable instrumentation point through which the I/O patterns can be obtained.

We find that the kernel function *filemap_fault* can serve as an instrumentation point. This function is responsible for processing the memory faults of mapped files. When a process attempts to read or to write some data, a generated memory fault occurs. The CPU investigates if the data is already in memory (during previous retrieval) or not yet loaded. Each memory fault has an offset that indicates where the required data is found inside a memory page. The offset is still helpful in case of having several memory regions of the mapped file.

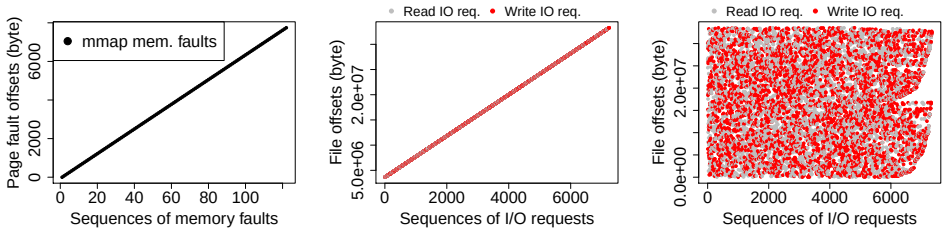


Fig. 2. Selected results from the experimentation campaign of validating *IOscope* via *Fio* benchmark, over a 32 MB file. a) shows I/O patterns of a read workload on *Mmap IOengine*, b) Shows I/O patterns of a readwrite workload on *Posixaio IOengine*, and c) presents I/O patterns of a random readwrite workload on *Psync IOengine*

Table 1. *Validated I/O access modes and workloads*

Fio IOengine	Target syscalls	Tested workloads: *read, write, randread, randwrite, readwrite,* and *randreadwrite*
Sync	read, write	all
Psync	pread, pwrite	all
Pvsync	preadv, pwritev	all
Pvsync2	preadv2, pwritev2	all
posixaio	aio_read, aio_write	all
Mmap	mmap, memcpy	all

The typical workflow of using this tool is to provide either the *inode* number or a data path with an extension of target files. The first is required in case of targeting only one file while the path and extension are required to trace the matched files. *IOscope_mmap* then starts to examine only the memory faults over the given file/files thanks to its preliminary filter. Its mechanism for connecting to the target function is similar to the *IOscope_classic* tool (*kprobe* and *kretprobe*). *IOscope_mmap* also reports the same data as the *IOscope_classic* tool.

2.3 IOscope Validation

IOscope is experimentally validated over synthetic workloads. The flexible I/O benchmark (*Fio*) is used to generate workloads that encompass the applied workloads by real systems and applications. These workloads are tested against the *Fio IOengines* which represents various I/O methods for accessing data (e.g., direct access, vectored I/O, memory-mapped access). *Table 1* lists those engines, the *syscalls* through which the I/O requests pass, and the tested workloads.

The validation experiments are executed on a single machine running *Ubuntu 14.04 LTS*, *Linux kernel 4.9.0* and *Fio-v2.1.3*. *Figure 2* shows some validation results. The flow of sequential workloads is always represented as a diagonal line of file offsets. The random workloads are represented as scratch dots, indicating

that the I/O requests target random offsets during the workload execution. It is noticeable from *Fig. 2-c* that the *Fio* random workloads are totally shapeless.

***IOscope* overhead.** We measure the overhead of using *IOscope* over the realistic workloads described in the next section. This is done by getting the difference between the execution time of an experiment with & without using *IOscope*. The maximum overhead obtained for an experiment was less than 0.8% of the execution time regardless of analyzing millions of I/O requests. In terms of memory overhead, the ring buffer of *IOscope* is limited up to 8 MB. No single event is lost with this configuration even in case of having high frequency I/Os.

3 Experiments

This section describes a set of experiments on two NoSQL storage databases (*MongoDB* and *Cassandra*) for which *IOscope* is used to analyze I/O patterns. It starts by presenting the environmental setup, and the applied experimental scenarios. It then describes the experiments done on *MongoDB* and its revealed performance issue before describing how *IOscope* is used to explain that issue. It ends by describing the experiments done on *Cassandra*.

3.1 Setup, datasets, and scenarios

Environmental setup. We perform those experiments on the *Grid'5000* [2] testbed. Each machine has two Intel Xeon E5-2630 v3 CPUs (8 cores/CPU), 128 GB of RAM, and a 10 Gbps Ethernet. Every machine is equipped with an *HDD* of 558 GB, and an *SSD* of 186 GB. The disks are connected as *JBOD*, using Symbios Logic MegaRAID SAS-3 3008 (rev 02). We deploy *Ubuntu 14.04 LTS* with *Linux 4.9.0* in which a resident *eBPF* virtual machine is enabled. The *Ext4* filesystem is used. The *deadline* I/O scheduler is used by default. Linux I/O schedulers (*Noop, deadline,* and *CFQ*) are interchangeable in our experiments due to the absence of concurrent I/O processes. The *Native Command Queuing* (*NCQ*) of disks is 2^6 by default; it minimizes the mechanical seeks of disks via rescheduling groups of concurrent I/O requests. We clean out the cache data including the memory-resident data of *MongoDB* and *Cassandra* between experiments. No more than one experiment is executed at the same time.

MongoDB v3.4.12[4] is used. It is tested with default configuration in which *WiredTiger* is the main storage engine. We use *Cassandra v 3.0.14* with its default configuration too. In both databases, nodes hold equal portions of data in case of cluster experiments, thanks to the built-in partitioner/balancer.

Datasets. Two equally sized datasets are created (each has 71 GB) to perform our experiments on *MongoDB* and *Cassandra*. Their contents are randomly generated to eliminate data-biased results. Each dataset has 20,000,000 data elements (called documents in *MongoDB* and rows in *Cassandra*) with 3.47 kB as

[4] A major version of *MongoDB* (v3.6) has been released during writing this paper. It suffers from the same performance issue discussed in Section 3.2, regardless of the optimized throughput

Table 2. Average throughputs of single-server and clustered experiments. Results of clustered *MongoDB* show a performance variability issue

		HDD (MB/s)	SSD (MB/s)
Single-server experiment		48.4	306.7
Two-Shards cluster	1st Shard (51% of data)	25.5	161.2
	2nd shard (49% of data)	11.8	182.6
Another execution of Two-Shards cluster	1st shard (50.27% of data)	47.8	539.6
	2nd shard (49.72% of data)	30.6	401.4

an average size. Each element consists of 1) one integer with random values, 2) a timestamp, 3) two string fields, 4) and one array which has a random number of fields up to four. *MongoDB* stores the dataset as a single file. In contrast, the number of *Cassandra*s' *SSTables* depends on how many times the data is flushed into disks (one or more SSTables).

Workload & Scenarios. The experiments are executed on either a single server or a distributed cluster of two shard nodes. In both scenarios, one client running on another node performs an indexation workload. The workload aims at indexing an integer field, pushing the target databases to read the entire datasets in order to construct a corresponding index tree. The objective is to look at how each database is accessing data and to see if some hidden issues could be revealed. This does not necessarily mean that the results of both databases are comparable due to an absence of a tuning phase for making an *apple-to-apple* comparison. In each scenario, All experiments cover both technologies of storage: *HDDs* and *SSDs*. They are executed one time using *HDDs* as principal storage of the involved machines, and another time on *SSDs*. We test the data contiguity on disks by profiling their physical blocks using filesystem `FIBMAP` command. Each data file resides on about 99.9% of contiguous blocks.

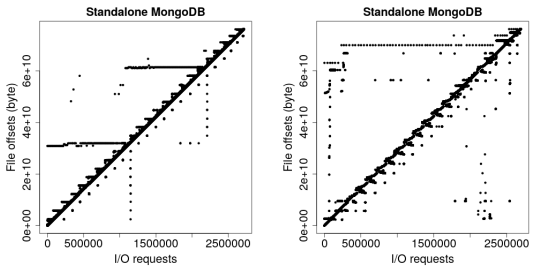
For the distributed experiments, the data is distributed using a hash-based mechanism. This is achieved over the *_id* field in *MongoDB* and over the primary key in *Cassandra* (*uuid* field), both in order to obtain evenly distributed data. In the rest of the paper, *executions* means that the experiment is re-performed from the step of pushing the dataset into the distributed cluster.

3.2 MongoDB experiments

This section describes the high-level results of executed workloads before showing *IOscope* results. High-level results are presented to convince that they only give insights into understanding I/O performance but cannot explain issues. The section ends by describing an *ad hoc* solution to the discovered issue by *IOscope*.

a) High-level results

The indexation workload described in Section 3.1 is executed. This workload is an intensive read workload in *MongoDB*. The dataset must be entirely retrieved



(a) *HDD* (b) *SSD*

Fig. 3. I/O patterns of the single-server experiments described in *Tab. 2*

from the storage support to be parsed document by document. Table 2 shows average throughputs of single-server and distributed experiments.

Single-server experiments. The execution time is reduced by a factor of 6.3 when using SSD as a principal storage instead of HDD. This indicates how the storage technology can improve accessing data. Repeating these experiments makes no changes over the execution time and throughput values. Hence, we use them as a reference for the distributed experiments to investigae *MongoDB* scalability.

Distributed experiments. For the two-shards experiments, we expect to decrease the execution time by half as an ideal case of linear scalability. However, the performance results of several runs are not as expected (see *Tab.* 2). *MongoDB* reports variable performance over both *HDDs* and *SSDs*. Performing other executions over the same dataset brings out variable results too.

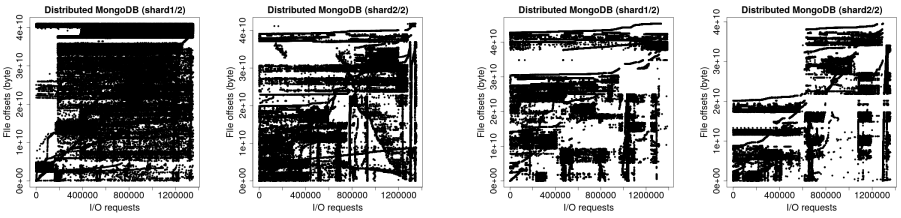
There is a hidden issue behind obtaining variable performance results. This issue can not be explained using high-level metrics as shown. This implies to go beyond those results by doing further investigations with *IOscope*.

b) IO pattern analysis using IOscope

We expect that *IOscope* reports a clear diagonal line of file offsets if the data is sequentially accessed. Otherwise, a noisy access pattern or even a shapeless one is obtained. The size of collected trace files of experiments is 1 GB.

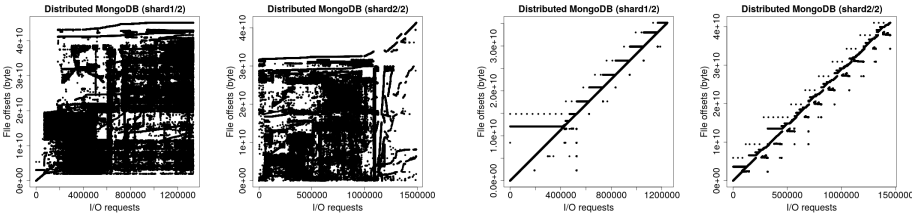
For the single-server experiments, both I/O patterns over the *HDD* and the *SSD* are acceptable. The files are sequentially accessed as shown in *Fig. 3*. The diagonal lines are present in both sub-figures regardless of the tiny noises that might refer to file alignments operations.

IOscope uncovers the reasons behind the performance variability of the distributed experiments. *Figure 4* and *Fig. 5* show the I/O patterns of these experiments. The same analysis can be done for *HDDs* and *SSDs* experiments as the obtained I/O patterns of each shard correspond to its execution time. In regard to the *SSDs* experiments, *Fig. 5-a* shows that both shards have totally-random I/O patterns. Thy take about 97%, 82% of the execution time of the single-server experiment. In contrast, the I/O patterns of both shards shown in *Fig. 5-b* are



(a) First experiment results (b) Second experiment results

Fig. 4. I/O patterns of the distributed experiments on *HDDs* described in *Tab. 2*



(a) First experiment results (b) Second experiment results

Fig. 5. I/O patterns of the distributed experiments on *SSDs* described in *Tab. 2*

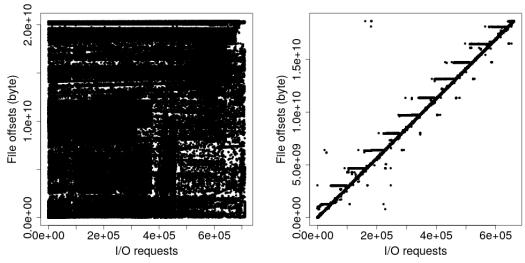
sequential. The shards reach the required performance (near 50% of execution time obtained in the single-server experiment). Hence, it is obvious to see the shards patterns as diagonal lines indicating that the data is accessed as it should be. This example shows that *IOscope* is able to explain issues even over recent storage support like *SSDs* and over a fine-grain execution time. This leaves no doubt that the I/O patterns are behind the reported performance variability.

We performed further experiments with three and four sharding nodes, but the random access patterns are still present. As a result, the inefficient way used by *MongoDB* for accessing data is the main reason of obtaining that issue.

c) An ad hoc solution to fix MongoDB issue

A mismatch between the order applied by *MongoDB* to retrieve data and the order of stored data is behind the above described issue. *MongoDB* tries to sequentially traverse the documents based on its view of pre-stored $\_id_s$. This occurs even if its retrieval plan does not follow the exact order of documents in the collection file. As described, the symptoms of this issue are 1) incurring mechanical seeks and 2) having noisy I/O patterns.

The key idea of our *ad-hoc* is to re-write the shards data locally. This implicitly updates the $\_id_s$ order regarding the documents order in the stored file, i.e., the inaccurate traversal plan of *MongoDB* will be replaced. The detailed steps



(a) (b)

Fig. 6. I/O pattern of a *MongoDB* shard with 20 GB of data a) before, and b) after applying the solution

of this solution are as follows. Firstly, we make a local dump of shards' data; this dump will sequentially retrieve the data from stored file, so it will have an accurate view of documents' order. Secondly, we re-extract the local dump on the corresponding shard, so *MongoDB* takes into account the novel documents' order. Of course, it is unrealistic to perform this solution every time encountered by similar issue due to the enormous overhead of rewriting data. But it gives insights to *MongoDB* community to fix that issue in upcoming versions.

Figure 6 shows a worst I/O pattern obtained on a shard node over an *HDD*. After applying our *ad hoc* solution, the execution time becomes 12.4 times faster (it is reduced from 1341 s to 108 s). On *SSD*, the performance is enhanced with a speedup factor of 2.5 (time is reduced from 89 s to 32 s). This might be related to the nature of the used *SSD* (Toshiba PX04SMQ040) which is optimized for sequential reading workloads.

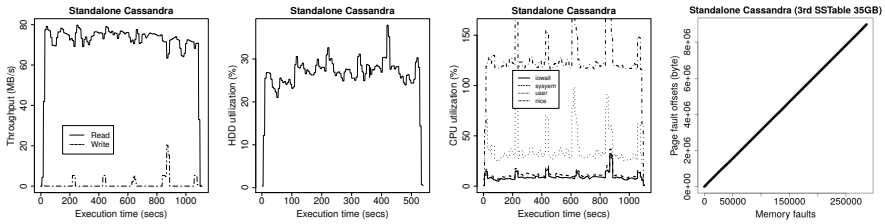
3.3 Cassandra experiments

This section describes the results of experiments performed on *Cassandra*.

a) Results

Single-server experiments. *Cassandra* maintains a stable throughput during the workload execution as shown in *Fig. 7*. However, the workload execution depends more on CPU as the stacked CPU sub-figure shows; the peak CPU reaches more than 150% of a core capacity. We only show the I/O patterns of the biggest *SSTable* in the same *figure*. In fact, the other *SSTables* have the same clear sequential access (the dataset is represented by five different-size *SSTable*s).

The peak value of the disk utilization is near 30%, indicating that the indexing operations are not I/O bounded. Hence, the factor that stresses the performance is the amount of used memory. If we limit the available memory for *Cassandra*, the performance in terms of execution time will increase to some extent. This occurs due to an increase of memory operations being performed such as freeing memory pages. However, the access patterns will not be changed thanks to the metadata that are used to regulate accessing data.



(a) (b) (c) (d)

Fig. 7. Cassandra single-server experiment results on *HDD*. a) shows the I/O throughput, b) shows the disk utilization, c) presents the CPU mode, and d) shows the I/O pattern of the largest SSTable

Distributed experiments. *Cassandra*'s nodes still reach the same throughput of the single-server configuration both over *HDDs* and *SSDs*. As a result, the execution time is optimized as expected on both nodes of *Cassandra*. Each node takes near 50% out of the single-server execution time. *IOscope* shows sequential I/O patterns for both experiments (similar results of *Fig. 7-d*). Because of space limitation, those results are not shown here.

4 Related Work

Betke and Kunkel [3] proposed a framework for real-time I/O monitoring. It does not implement a filtering mechanism like *IOscope* during the interception of I/O traces, leading to collecting a huge number of generic traces. Daoud and Dagenais [6] proposed a *LTTng*-based framework for collecting disks metrics. Their framework only analyses generated traces of *HDDs*, and no information is provided about its applicability on *SSDs*. In addition, it does not collect file offsets, which is our main metric for analyzing workloads' I/O patterns. Jeong et. al [10] proposed a tool to generate I/O workloads and to analyze I/O performance for Android systems. Their I/O performance analyzer requires a modified kernel and runs only for custom filesystems (ext4, fat32). In contrast, *IOscope* needs no kernel modification and mainly works on the *VFS* layer to support wide number of filesystems. Other tools [14, 15, 4, 24, 22] aim to predict and extrapolate the I/O performance for large scale deployments by analyzing and replaying small set of traces. In contrast, our work focuses on collecting fine-grained traces of I/O workloads under study for discovering and explaining I/O issues.

Several tracing tools such as SystemTap [9], Dtrace [17], LTTng tools [7] load tracing scrip as dynamic modules into the kernel (e.g., using *dkms* package). This makes them unsuitable for usage in some situations, e.g., in case of signed kernels. Using them also implies doing posterior efforts for analyzing massive quantity of collected traces. In contrast, *eBPF* is formally adopted by the Linux kernel [13]. It is mainly known for its filtering capabilities that we leveraged to build *IOscope*.

IOscope performs four activities: profiling, filtering, tracing, and direct analysis of I/O patterns. Its filtering and tracing activities are comparable to several tools of the BCC project [18]. In general, they give an instantaneous view of matched events on target instrumentation points, presenting outputs like the `top` command of Linux. BCC *Slower* tools are built to filter the I/O operations with large latencies. They work on higher layers of Linux I/O stack. The *fileslower* tool traces operations on the VFS layer of Linux I/O stack while the *ext4slower* tool works on *Ext4* filesystem. Both bring out fine-grained filtering of I/Os (e.g., reporting I/Os per process), but deal with partial I/O contexts. The *fileslower* does not work with *pvsync*, *pvsync2*, and *mmap* I/Os, while the *ext4Slower* lacks supporting *mmap* I/Os. Extracting I/O patterns is still possible for the supported I/O contexts. However, this requires much post-analysis effort compared with *IOscope* which needs nothing to prepare final results.

Several tools collect traces from the block I/O layer on Linux I/O stack. For instance, BCCs' *biotop*, BCCs' *bIOsnoop*, DTraces' *IOsnoop*. These tools generate traces in terms of *accessed sectors* of disks. They do not link those sectors to workloads' accessed files, being more close to studying hardware issues rather than analyzing I/O patterns. To explain, collecting disks sectors do not specify how applications access data files. The reason is that I/O requests are expected to be re-ordered in intermediate layers (e.g., in the I/O scheduler layer). A modified tracer [16] of *blktrace* addresses that issue by combining traces from block I/O and VFS layers. However, it lacks supporting *mmap* I/O, and it needs an additional effort to analyze I/O patterns. Hence, replacing *IOscope* by any of these tools cannot explain I/O issues. Analyzing I/O flow in terms of disk sectors has no sense as there is no constraints to allocate data on successive or random sectors. *IOscope* addresses that by working with files offsets. Over a given file, the offsets specify the order of all read/written data throughout workloads' I/Os.

5 Conclusions

Performing in-depth analysis of storage system workloads is necessary to reveal potential I/O issues in lower levels. Robust and flexible tools are needed to perform such detailed evaluations. In this paper, we first described *IOscope* which uncovers I/O patterns of storage workloads. It has less than 1% overhead and inherits other features from *eBPF* technology, being suitable for analyzing production workloads. We then showed use case experiments over *MongoDB* and *Cassandra* using *IOscope*. The results from *MongoDB* experiments reinforce our hypothesis for going beyond high-level evaluations. *IOscope* was able to report the main reasons behind the performance variability of *MongoDB*, over executed workloads. The issue is raised due to unexpected mismatch between the order of the allocated data on disks, and the traversal plan used by *MongoDB* in case of distributed experiments. Moreover, *IOscope* was able to confirm the occurrence of that issue over *HDDs* and *SSDs*. Based on the insights provided by *IOscope*, we proposed an *ad hoc* solution to fix that issue by re-writing the shards data. This allows achieving anew linear and scalable results of the concerned experiments.

References

1. Abramova, V., Bernardino, J.: Nosql databases: Mongodb vs cassandra. In: Proceedings of the international C* conference on computer science and software engineering. pp. 14–22. ACM (2013)
2. Balouek, D., et al.: Adding virtualization capabilities to the Grid'5000 testbed. In: Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013)
3. Betke, E., Kunkel, J.: Real-time i/o-monitoring of hpc applications with siox, elasticsearch, grafana and fuse. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) High Performance Computing. pp. 174–186. Springer International Publishing, Cham (2017)
4. Chahal, D., Virk, R., Nambiar, M.: Performance extrapolation of io intensive workloads: Work in progress. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering. pp. 105–108. ACM (2016)
5. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. ACM (2010)
6. Daoud, H., Dagenais, M.R.: Recovering disk storage metrics from low-level trace events. Software: Practice and Experience **48**(5), 1019–1041 (2018)
7. Desnoyers, M., Dagenais, M.R.: The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In: OLS (Ottawa Linux Symposium). vol. 2006, pp. 209–224. Citeseer, Linux Symposium (2006)
8. Gandini, A., Gribaudo, M., Knottenbelt, W.J., Osman, R., Piazzolla, P.: Performance evaluation of nosql databases. In: European Workshop on Performance Engineering. pp. 16–29. Springer (2014)
9. Jacob, B., Larson, P., Leitao, B., Da Silva, S.: Systemtap: instrumenting the linux kernel for analyzing performance and functional problems. IBM Redbook (2008)
10. Jeong, S., Lee, K., Hwang, J., Lee, S., Won, Y.: Androstep: Android storage performance analysis tool. In: Software Engineering (Workshops). vol. 13, pp. 327–340 (2013)
11. Jung, M.G., Youn, S.A., Bae, J., Choi, Y.L.: A study on data input and output performance comparison of mongodb and postgresql in the big data environment. In: Database Theory and Application (DTA), 2015 8th International Conference on. pp. 14–17. IEEE (2015)
12. Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K., Matser, C.: Performance evaluation of nosql databases: A case study. In: Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems (2015)
13. manual page on Linux, B.: http://man7.org/linux/man-pages/man2/bpf.2.html (2017)
14. Luo, X., Mueller, F., Carns, P., Jenkins, J., Latham, R., Ross, R., Snyder, S.: Hpc i/o trace extrapolation. In: Proceedings of the 4th Workshop on Extreme Scale Programming Tools. p. 2. ACM (2015)
15. Luo, X., Mueller, F., Carns, P., Jenkins, J., Latham, R., Ross, R., Snyder, S.: Scalaioextrap: Elastic i/o tracing and extrapolation. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. pp. 585–594. IEEE (2017)
16. Mantri, S.G.: Efficient In-Depth IO Tracing and its application for optimizing systems. Ph.D. thesis, Virginia Tech (2014)

17. McDougall, R., Mauro, J., Gregg, B.: Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and OpenSolaris. Prentice Hall, (2006)
18. collection project, B.C.: https://github.com/iovisor/bcc
19. Schulist, J., Borkmann, D., Starovoitov, A.: Linux socket filtering aka berkeley packet filter (bpf) (2016)
20. Sharma, S.D., Dagenais, M.: Enhanced userspace and in-kernel trace filtering for production systems. Journal of Computer Science and Technology (6), 1161–1178 (2016)
21. Starovoitov, A.: https://lwn.net/Articles/598545/ (2014)
22. Tak, B.C., Tang, C., Huang, H., Wang, L.: Pseudoapp: Performance prediction for application migration to cloud. In: Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on. pp. 303–310. IEEE (2013)
23. Vef, M.A., Tarasov, V., Hildebrand, D., Brinkmann, A.: Challenges and solutions for tracing storage systems: A case study with spectrum scale. ACM Trans. Storage **14**(2), 18:1–18:24 (Apr 2018). https://doi.org/10.1145/3149376, http://doi.acm.org/10.1145/3149376
24. Virk, R., Chahal, D.: Trace replay based i/o performance studies for enterprise workload migration. In: 2nd Annual Conference of CMG India, page Online (2015)