TrackIops : Real-Time NFS Performance Metrics Extractor

Théophile Dubuc
ENS de Lyon, Outscale
France
theophile.dubuc@ens-lyon.fr

Pascale Vicat-Blanc
Inria, ENS de Lyon
France
pascale.vicatblanc@inria.fr

Pierre Olivier
The University of Manchester
United Kingdom
pierre.olivier@manchester.ac.uk

Mar Callau-Zori
Outscale
France
mar.callau-zori@outscale.com

Christophe Hubert
Outscale
France
christophe.hubert@outscale.com

Alain Tchana
Grenoble INP
France
alain.tchana@grenoble-inp.fr

Abstract

Network File System (NFS) is commonly used in cloud environments as a cost-effective file storage solution that is easy to set up. However, the multi-tenant nature of cloud infrastructures makes distributed file systems prone to instability and unpredictability. These performance issues can be very harmful to both Cloud Service Providers (CSPs) and tenants. Therefore, CSPs and their customers require more and more real-time granular metrics (per-file, high-frequency) for dynamically optimizing data placement, resource usage and ensuring file access performance as well as for provisioning resources cost-effectively, billing and troubleshooting them rapidly. In this paper, we propose TrackIops, a novel NFS tracer that provides these metrics without effort and at low cost. TrackIops is an eBPF-based client-side request-oriented tracing solution. The main contribution of this paper is a smart kernel-level solution that reconstructs NFS request and response threads and analyses them online without requiring server instrumentation. TrackIops provides real-time per-tenant, per-file, per-second NFS metrics extractor, easy to integrate in any optimization or troubleshooting solution, with an overhead lower than 3.5% on the client in a worst-case scenario.

ACM Reference Format:
Théophile Dubuc, Pascale Vicat-Blanc, Pierre Olivier, Mar Callau-Zori, Christophe Hubert, and Alain Tchana. 2024. TrackIops : Real-Time NFS Performance Metrics Extractor. In *4th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3642963.3652202

1 Introduction

Cloud applications and services rely on complex infrastructures made of thousands of nodes. A typical infrastructure dissociates compute and storage nodes [19]: compute nodes are used for their large amount of CPU cores and memory, and they contain the hypervisor which runs the virtual machines (VMs) or containers. Storage nodes are used for their large amount of storage capacity and are used to store the virtual storage, which can be virtual drives for VMs, or container persistent volumes. They are accessed from the compute nodes through remote file systems like NFS. With NFS, modern cloud applications can transparently share data and collaborate on files accessed by multiple virtual machines or containers.

Cloud providers need fine-grained storage performance metrics like bandwidth, latency and I/O operations per second (iops) to perform load-balancing [23], or to know customers' resource usage, for billing purposes [25, 35] or carbon emission estimation [33]. For these use-cases, the cloud providers need storage metrics at the granularity of the resource reserved by the customer. For storage, this granularity is the file, as virtual drives for VMs are stored as files on the host machine, and a container volume can be a directory mounted in the container. Thus, the use-cases cited above require *per-file* and *real-time* NFS performance metrics.

In production environments, storage nodes are usually industrial scale closed-source solutions which cannot be instrumented. Therefore, collecting NFS metrics is harder for it should be done from the compute nodes (NFS clients) only.

To the best of our knowledge, there is no existing tool which provides NFS performance metrics per-file and in real time from the client side only.

To fill this gap, we introduce TrackIops, a *real-time per-file and client-side NFS performance metrics extractor.* Some of the challenges which explain the gap in existing tooling, and that TrackIops successfully overcomes, are:

- (C1) The tool should have a negligible overhead on the compute node performance, as it should be always-on in production environments.
- (C2) It is often not desirable or even not possible to modify the kernel of production machines, for stability reasons: a modified kernel or a kernel module could lead to kernel crashes, which is not a risk that can be taken in production.
- (C3) The NFS system is asynchronous. The I/O requests from the client are not emitted by the same process that receives the answer, therefore it is difficult to measure requests latency.

We argue that basing the tracer on *eBPF* [12] (called *BPF* for convenience) allows tackling challenge (C2), and also helps for (C1) as BPF enables lightweight instrumentation.

Also, TrackIops solves (C3) by reconstructing the NFS requests directly in the kernel using the RPC (remote procedure call) layer, on which NFS is based, to match NFS answers received from the NFS server to the requests that were issued by the client. This method is also what allows real-time metrics extraction, contrary to methods which would rely on logging events and reconstructing the requests offline.

After motivating the use-case for the metrics and providing background about existing tools and BPF (Section 2), this paper presents TrackIops's design and implementation (Section 3), then evaluates its overhead on the system (Section 4). We find that even in a scenario where the NFS server has a very low latency, the overhead of TrackIops on storage performance is always below 3.5%, which makes it suitable for real-world production environments.

2 Background and Motivations

2.1 Use-Cases for Per-File NFS Performance Metrics

A fundamental practice for a cloud provider managing many servers is *load-balancing* [23, 24, 27]. It consists in deciding where to place the resources requested by a client, or move existing ones for optimization. The goal of load-balancing is to save energy by enabling *over-commitment*, while maintaining the same Quality of Service. If customers use 10% of the resources (for instance CPU) they reserved, then 90% of the resources on the server are wasted, and so are energy and money, as energy consumption of physical servers is not proportional to server usage [17]. Therefore, the cloud provider can over-commit on the resources and allocate more virtual resources to the customers than the actual physical resources, and thus reduce energy consumption by increasing server usage. In case of a peak of usage of a server, load-balancing is also used to move parts of the reserved resources to other machines, and prevent resources starvation inducing Service-Level Agreement violation and financial penalties [30].

In order to perform load-balancing, the cloud provider needs to know at any time what resources are used, and it needs this information to the granularity of the unit of allocation (per-VM, per-container), so it knows what VMs or containers can be moved away to free the required resources. For CPU and memory, this information is provided directly by Linux, as a container or a VM is a process (or a set of processes) on the host, and Linux built-in tools can provide CPU and memory usage per process identifier (PID) [9, 11]. However, a single VM could have several virtual drives stored in several files, on different drives or even different storage nodes. Thus, per-PID storage consumption is not of any use as it aggregates the information of all the drives or volumes of a single VM or container. Therefore, for storage load-balancing, the required granularity is (at least) *per-file*. Obtaining per-file size is straightforward, so this work focuses on collecting storage *performance* metrics: *I/O operations per second (iops), throughput* and *latency*.

A second use-case of per-file storage metrics is carbon footprint estimation [31, 33]. As emissions reporting is becoming mandatory in a growing number of countries [2], companies need to have the means to compute their carbon footprint. If a company IT infrastructure is cloud-based, the cloud provider should report the carbon emissions to their customers. Providing the data per-volume rather than per-VM or container allows finer decision-taking from the companies relying on cloud services. This per-file precision also enables better billing precision. It should finally provide help in troubleshooting performance issues, especially when a single virtual drive of a VM is split onto several files on the storage host, which is possible with copy-on-write drive formats (like QCOW2 [10], the default for KVM-based virtualization). For instance, if a customer experiences latency on their VM drive, knowing per-file latency helps the provider pinpointing one problematic file in the QCOW2 chain and taking action on the responsible storage node.

2.2 Closed-Source Storage Nodes

The first challenge for collecting these metrics from a production system is that storage nodes are usually industrial-scale storage solutions, like NetApp ONTAP or Dell EMC. These solutions usually provide performance metrics, but they are aggregated at the scale of a whole NFS share at best [15, 16]. Unfortunately, as they are closed-source, they cannot be instrumented to collect more data. Thus, the storage node must be considered as a *closed-box*, and storage performance metrics should be collected from the *client side only*.

2.3 Extended Berkley Packet Filter (eBPF)

BPF [12] is a technology which allows attaching *BPF programs* to some kernel events, like kernel function entry (*kprobes*) or exit (*kretprobes*), or *tracepoints*, which are special hooks placed at strategic points in the kernel by its developers. BPF programs are written in a restricted C-like language, and are verified and compiled to byte code which is executed in a kernel virtual machine. The BPF verifier and the isolation from the kernel ensures that BPF-based applications are perfectly stable and do not risk crashing the kernel, contrary to modifying the kernel code directly or through modules. It is therefore safer and used by many companies in production. Furthermore, as it runs in kernel space where NFS resides, it is much faster than user-space tracing. Additionally, BPF developers argue that tracepoints have a faster interface than kprobes [21], and they are not vulnerable to function signature changes. Therefore, we base TrackIops on available NFS tracepoints, for Challenge C1 - low overhead.

2.4 Existing Tools

There exist many tools for monitoring an NFS mount point from the client side. nfsiostat [1] provides statistics per NFS mount, nfsstat [7] only counts the number of operations. There exist BPF-based tools, like nfsdist [3] and nfsslower [4], which provide some stats about NFS mounts, but they both filter per PID and not per file.

Performance metrics can also be measured at the VFS level. Examples of tools providing performance metrics are blktrace [5], atop [13], or pidstat [8]; but they provide only block device or per-process metrics. inotifywatch [6] provides file-level metrics, but only the number of read and write operations, not the latency. There are several bpftools [14] provided with BCC (a Python-based eBPF framework and toolkit) which trace some storage metrics, but none provides all the required performance metrics at the file granularity.

In the state of the art, there are also a lot of works aiming at tracing I/O in user-space [18, 20, 26, 34] or even mixing both user and kernel tracing [28] for better comprehension of the system; but these tools have a purpose of tracing and collect too many data than what we require here, thus they usually have an overhead that discards them for real-time metrics extraction. There also exist distributed tracing frameworks for cloud native applications [22, 29, 32], which can thus be used for NFS-based storage; however they usually require a lot of post-processing for trace reconstruction and are thus not intended for real-time metrics extraction. There is, to the best of our knowledge, no recent work focusing on efficiently extracting per-file NFS performance metrics.

3 Design and Implementation

TrackIops is composed of two parts: a kernel-space agent and a user space consumer, as shown in Figure 1. The kernel-space agent (A) is based on BPF, for light and stable instrumentation (challenges C1 and C2). Its role is to collect data at different stages of the NFS requests, and reconstruct them using a unique identifier available at the RPC layer. It aggregates statistics on the reconstructed requests in a BPF map (B) readable from user space. The user space consumer (C) collects the data and writes it to a pipe for use by external services. This section gives the key points of the design and implementation of these components.

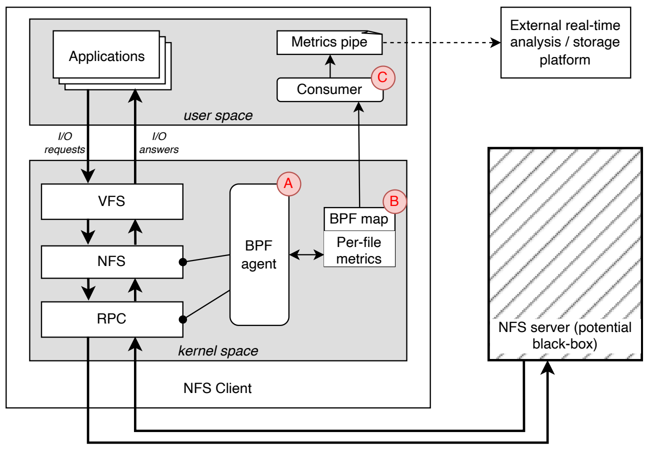

Figure 1. The global design of TrackIops. It is composed of an eBPF agent which attaches to NFS and RPC tracepoints, and a user space consumer which collects the storage metrics and writes them to a database.

Table 1. Data exposed by `nfs_readpage_done`, a tracepoint triggered when an NFS read request is completed.

Field name	Explanation
`dev`	Device to which the read is issued
`fhandle`	File handle
`fileid`	File ID (inode number)
`offset`	Offset in the file of the requested read
`arg_count`	Requested number of bytes to read
`res_count`	Actual number of bytes read

3.1 BPF for In-Kernel Data Collection

TrackIops leverages eBPF for in-kernel lightweight data collection. It allows writing *BPF programs* that are compiled and verified before being attached to *kernel events*. When an event occurs, the hooked BPF program is executed in a kernel VM. This approach offers efficiency for the executed BPF code, as it is directly in kernel and has access to kernel data structures. It also offers security, as the BPF verifier and some constraints on the BPF code ensure it is safe to execute in the kernel, unlike kernel modules or kernel patching. Kernel events to which the BPF programs are attached can be kernel function entry (*kprobe*), function return (*kretprobe*), *tracepoints*, which are placed at key points inside the kernel code by the kernel developers, or even user-defined probes. Tracepoints are designed for tracing and directly expose context-related information like shown in Table 1, through a faster interface compared to kprobes. Thus, they are preferred when available, which is the case for NFS.

Table 2. The conversion formulas for the different target metrics. g is the granularity, *i.e.* the time interval during which the raw metrics were collected.

Metric	Formula
IOPS	I/O count $/g$
Throughput	Cumulated size $/g$
Average latency	Cumulated latency / I/O count

3.2 Metrics Sources

The role of the BPF agent is to collect data that the user space consumer can use to compute NFS storage performance metrics: IOPS (number of I/O operations per second), throughput (read / written size per second), and latency (average time before an I/O request gets a response). These metrics can be derived respectively from the *count* of I/O operations on a given time, the *cumulated size* of I/O operations on a given time, and the *cumulated latencies* of a known number of I/O operations. Table 2 summarizes these conversions.

Therefore, the BPF agent must collect the following data: *read count, read cumulated size, read cumulated latency*, and the same ones for write. For read requests, the tracepoint `nfs_readpage_done` allows collecting the first two: counting the number of times it is triggered gives the read count, and by summing the `res_count` values, one gets the cumulated read size. Finally, the read latency is the difference between the timestamp when `nfs_initiate_read` is triggered and the one when `nfs_readpage_done` is triggered for a same request. This is all analogous for write requests (with corresponding write tracepoints). However, to compute this difference, when an NFS request (read or write) occurs, one must link the two events corresponding to its beginning and end in order to compute this latency. This *request reconstruction* is performed directly in the kernel by the BPF programs, and is explained next in Section 3.3.

3.3 NFS Requests Reconstruction

In order to measure the NFS read and write latencies, a tracer should be able to link the tracepoints related to requests beginning and end. However, this information is not directly provided by the NFS tracepoints. In order to get a common request identifier, TrackIops hooks two extra `sunrpc` tracepoints. The RPC (remote procedure call) is the layer below NFS, which is responsible for packing the NFS request and sending it through the network to the server. The following observations allow to reconstruct an NFS request:

1. The process initiating an NFS request is the same that will create the RPC task responsible for transporting it.
2. The RPC identifies its tasks with a task ID which is unique on the client even if mounted from different NFS servers are present.

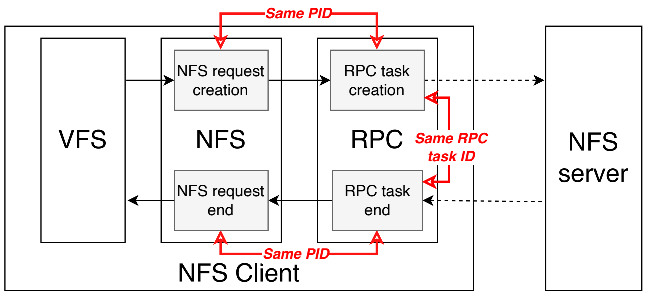

Figure 2. One can link the beginning and end of an NFS request using the information available at kernel tracepoints. The reconstruction uses the RPC subsystem and its tasks IDs as an intermediary between the two NFS tracepoints.

3. The process ending an RPC request when the answer has been received is also the one that completes the NFS request.

Therefore, by collecting the task ID when an RPC request is created and when it is done, along with the process ID (PID), and collecting the ID of the process which initiates and the one that ends an NFS request, TrackIops can reconstruct the NFS request by linking its beginning and end. The RPC task IDs and PIDs are collected by hooking the `rpc_task_begin` and `rpc_task_done` tracepoints. Figure 2 summarizes the different collect points along an NFS request and how they relate to each other.

An important aspect of the design is that this reconstruction is done *inside the kernel*, directly by the BPF programs. This method allows minimizing data movement between kernel and user spaces, when compared to the other common option which consists in sending an event to the user space every time a tracepoint is triggered and reconstructing the request offline there. To achieve this, the BPF side of TrackIops uses different hash maps that it uses to store temporary information while waiting for an NFS request to end. We detail next the instrumented tracepoints and what data is used to reconstruct the requests.

3.4 In-kernel Reconstruction Implementation

This section presents the different tracepoints instrumented by the BPF agent. To each of these tracepoints in attached one BPF program, and we describe what information it collects and stores in the different hash maps in order to reconstruct the requests. The whole process is presented in Figure 3, which is referenced to all along the following paragraphs.

***`nfs_initiate_read`* (1R).** This tracepoint is triggered at the beginning of an NFS read request. Here, the tracer collects only a timestamp and the PID of the process which triggered the tracepoint. It stores this information in the hash table `link_begin`, whose key is the PID and value the timestamp.

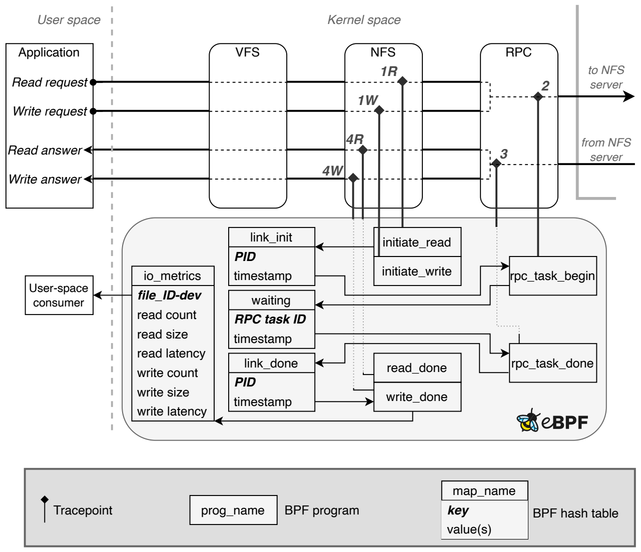

Figure 3. The BPF agent detail, the different steps are detailed in Section 3.4. The names in the "BPF program" boxes are the names of the BPF programs attached to the various tracepoints, whose names are given as the paragraph titles of Section 3.4.

***`nfs_initiate_write`* (1W).** This is analogous to the above `nfs_initiate_read` but for write requests.

***`rpc_task_begin`* (2).** This tracepoint is triggered at the beginning of an RPC task, which can be triggered after a NFS read or write request alike. The BPF program attached to this tracepoint collects the current PID and the RPC task ID, exposed by the tracepoint. It reads the hash map `link_begin` using the PID, and collects the timestamp associated to it, filled in step (1R/W). It then writes back the timestamp to the `waiting_RPC` hash table, with the RPC task ID as the key. Note that the RPC might be used by other services than NFS; therefore this tracepoint can be triggered outside the scope of TrackIops. However, the first step of reading the hash map `link_begin` with the key as PID acts like a filter: if the read fails, it means that the map was not filled by this process, therefore the current RPC task is not for an NFS read or write.

***`rpc_task_done`* (3).** This tracepoint is triggered when an RPC request received an answer, right before returning to the requesting application (here, NFS). On this tracepoint, TrackIops gets the PID and RPC task ID, reads the timestamp stored in the `waiting_RPC` table at step (2), and writes it in `link_end` with the PID as the key. Now the link between the request and its answer has been made, the next step is to collect the last pieces of information about this request.

***`nfs_readpage_done`* (4R).** This is the last tracepoint of NFS read requests instrumented by the BPF agent. The program attached to it gets the current PID, reads the `link_end` table to fetch the timestamp of when the NFS request was started. It also fetches the size of the read operation from the `res_count` of the tracepoint context, like shown in Table 1, as well as the ID (inode number) of the read file (`fileid`) and the device hosting it (`dev`). The keys of the `io_metrics` hash table are the concatenation of the file ID and the device number. This is for in NFS, the inode number are borrowed to the server. Thus, when several NFS shares are mounted from different servers to a single client, nothing prevents inode number conflicts, and the device number is required to disambiguate them. Using that key, the BPF program can enrich the `io_metrics` map by:

- Incrementing the read count by one
- Incrementing the total read size by this operation size
- Incrementing the total latency by the difference between the current time and the timestamp of the request beginning.

***`nfs_writeback_done`* (4W).** This is the equivalent to `nfs_readpage_done` for write requests.

Table 3. Fields of **`struct`** `raw_metrics`. Values are counted or summed since last reset.

Field	Description
`read_count`	Number of read operations
`read_size`	Total read size
`read_lat`	Cumulated read latencies
`write_count`	Number of write operations
`write_size`	Total written size
`write_lat`	Cumulated write latencies

3.5 Metrics Polling by the User Space Consumer

As the BPF hash map containing the metrics is being filled by the BPF programs, the role of the user space consumer is to collect those metrics at regular time intervals (the granularity g, in seconds, which can be as low as 1). This is possible as the BPF maps are accessible from the user space for read and write. Every g seconds, the consumer reads the `io_metrics` hash table, whose keys are the file IDs, and the values are the **`struct`** `raw_metrics` described in Table 3. The hash table entries are reset right after they are read, so the BPF programs which are constantly running can keep filling them with fresh data for the next collect. The collected information is converted to the actual storage metrics like described in Table 2. Therefore, the granularity does not impact the collect of the data by the BPF programs in the kernel, which is always exhaustive. Instead, it defines the rate at which the user space consumer will fetch the data in the shared map, allowing to refine the granularity, or reduce the overhead and amount of data generated by the consumer, depending on the needs of the user.

4 Evaluation

4.1 Performance Overhead

This section evaluates the overhead of the tracer and the factors that impact it. The main sources of overhead for a given workload are:

- The code injected in the kernel at BPF hooks. It slows down the application that makes NFS calls, as it adds instructions to the critical path of its execution. Each NFS request will be slowed down by the same amount of time. So (1) the lower the NFS latency, the higher the impact of the tracer is visible *per request.* And (2) the higher the frequency of NFS requests made by an application, the higher the impact of slowing NFS, so the higher the impact of the tracer.
- The consumer, which adds some CPU and memory load when reading the hash table every g seconds. The longer the consumer runs, the higher this overhead; therefore the parameters that influence it are the granularity g (increases the number of times that the consumer runs) and the size of the hash table (increases the execution time of every full read), which directly depends on the number of active NFS files.

Here is a summary of the identified parameters that impact the overhead, and how they are treated in this experiment:

1. NFS server performance. The lower the NFS latency, the more the tracer's impact is visible. Therefore, the experiment emulates a very fast server by running locally on the client (no network latency) and in memory (very low storage latency).
2. Frequency of NFS requests. As the tracer supposedly impacts NFS performance, the workload chosen for this experiment is an fio test on files accessed via NFS. This storage-intensive workload will maximize the impact of the tracer in the measurements.
3. Tracer's granularity. The experiment is run several times with variable granularity, going from infinitely coarse (the user space consumer never fetches the data) to the finest available with our implementation (fetch the data every 1 second).
4. Simultaneously active files. The experiment also varies the number of workers run in the fio workload, which is equivalent to the number of files read and written to. It goes from a single file to 4000 simultaneous files, which is the maximum we could get fio to run.

This experiment ran on a single Dell PowerEdge C6420 with two Intel Xeon Gold 5118 CPUs (12 cores/CPU, 2.30GHz), and 512 GiB of memory, with Linux 5.15 in Ubuntu 22.04.

The workload consists in running a fio experiment with k workers. Each worker opens a 1 MB file, and performs a total of 1 GB of random read/write operations on it. The same experiment is run several times with different values for the parameters cited above: granularity and number of fio workers. For every (granularity, workers) couple of parameters, the experiments was run 15 times. The results presented here are the arithmetical mean of 13 of these runs after removing the min and max value. We present the degradation in percent compared to a control experiment, which is the same workload running without any tracer at all.

Table 4. Read latency degradation (in %) compared to the control experiment for the all the tested values of granularity and number of workers.

workers	−s	60s	30s	15s	5s	1s
1	3.10	3.30	3.14	2.99	3.12	0.48
5	0.83	0.38	0.86	1.12	0.54	0.45
10	0.62	0.75	0.71	1.31	0.74	0.47
100	0.23	0.21	0.28	0.18	0.22	0.00
1000	1.57	1.70	1.83	1.98	1.79	1.84
4000	0.00	0.05	0.51	0.00	0.00	0.73

Table 5. The worst degradation observed for every metric, and the couple (granularity, workers) of parameters for which each worst performance was met.

Metric	max loss	workers	granularity
Read bandwidth	2.47%	1	60s
Write bandwidth	2.47%	1	60s
Read latency	3.30%	1	60s
Write latency	2.65%	5	5s
Read iops	2.48%	1	60s
Write iops	2.47%	1	60s

Table 4 shows the results for read bandwidth experiments. The biggest degradation values are reached for the single worker configurations. The reason for that is the high performance of fio in that configuration: the average read latency with a single worker in the control experiment is 169.4μs. Thus, the 2.47% observed degradation corresponds to an extra 5.6μs latency on average. The baseline latency of 169.4μs is extremely low thanks to our experimental setup which almost completely removes storage and network latencies. As the latency of an SSD drive is usually higher than 200μs, this 5.6μs degradation is negligible when the NFS server is distant rather than local and using SSDs rather than memory.

The worst degradation per metric is shown in Table 5. The overall maximum degradation observed during this evaluation is the read latency one presented above. All other metrics degradation for every couple (granularity, workers) of parameters is less than 3%, even in this worst-case setup. This makes TrackIops suitable for environment production, which was Challenge C1.

4.2 Volume of Generated Data

Here we briefly provide an insight on the quantity of data generated by TrackIops. The data generated per day on a single node is given by the formula:

$$(86400/g) * w * sizeof(log\_entry)$$

86400 is the number of seconds in a day, g is the granularity, w the number of workers. In its current implementation, the size of an entry of the `io_metrics` map is 28 bytes, and the key (fileid and device) is 12 bytes long, for a total of 40 bytes. So with the worst-case 1s granularity and 4000 simultaneously open files, TrackIops generates less than 13 GiB of data per day, which is reasonable in terms of storage and network overhead in a cloud provider production environment, and can be easily reduced by increasing the granularity.

5 Conclusion

This paper proposes TrackIops, an eBPF-based client-side request-oriented tracer which extracts NFS performance metrics (iops, throughput and latency) and exposes them for services that requires them, like a load balancer. The main contribution of this paper is the in-kernel NFS session reconstruction which allows lightweight and real-time metrics collection while reducing the amount of generated data compared to traditional eBPF-based tools. Our evaluation shows that TrackIops is suitable for production always-on use as it has a very low overhead on the NFS client running it (less than 3.5% in a worst-case scenario).

In the future, we intend to perform further testing of TrackIops with our industrial partner to eventually characterize performance metrics at a cloud provider scale. We also plan to associate it with an existing networking analysis tools like ePPing in order to break down storage latency between client, network and server for better troubleshooting.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] 2010. nfsiostat man page. https://man7.org/linux/man-pages/man8/nfsiostat.8.html
[2] 2020. Mandatory Emissions Reporting Around the Globe. https://www.ul.com/news/mandatory-emissions-reporting-around-globe
[3] 2021. nfsdist. https://github.com/iovisor/bcc/blob/master/tools/nfsdist.py
[4] 2021. nfsslower. https://github.com/iovisor/bcc/blob/master/tools/nfsslower.py
[5] 2023. blktrace man page. https://linux.die.net/man/8/blktrace
[6] 2023. inotifywatch man page. https://linux.die.net/man/1/inotifywatch
[7] 2023. nfsstat man page. https://linux.die.net/man/8/nfsstat
[8] 2023. pidstat man page. https://man7.org/linux/man-pages/man1/pidstat.1.html
[9] 2023. ps man page. https://man7.org/linux/man-pages/man1/ps.1.html
[10] 2023. QCOW2 format reference. https://github.com/qemu/qemu/blob/master/docs/interop/qcow2.txt
[11] 2023. top man page. https://man7.org/linux/man-pages/man1/top.1.html
[12] 2023. What Is eBPF? https://ebpf.io/what-is-ebpf/
[13] 2024. atop man page. https://linux.die.net/man/1/atop
[14] 2024. BCC storage tools. https://github.com/iovisor/bcc?tab=readme-ov-file#storage-and-filesystems-tools
[15] 2024. Dell EMC storage metrics. https://www.ibm.com/docs/en/storage-insights?topic=metrics-performance-dell-emc-storage-systems
[16] 2024. NetApp storage metrics. https://docs.netapp.com/us-en/ontap-automation/rest/performance_metrics.html
[17] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (2007), 33–37. Publisher: IEEE.
[18] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Trans. Storage* 7, 3 (Oct. 2011). https://doi.org/10.1145/2027066.2027068 Place: New York, NY, USA Publisher: Association for Computing Machinery.
[19] Tao Chen, Xiaofeng Gao, and Guihai Chen. 2016. The features, hardware, and architectures of data center networks: A survey. *J. Parallel and Distrib. Comput.* 96 (Oct. 2016), 45–74. https://doi.org/10.1016/j.jpdc.2016.05.009
[20] Steven WD Chien, Artur Podobas, Ivy B Peng, and Stefano Markidis. 2020. tf-Darshan: Understanding fine-grained I/O performance in machine learning workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 359–370.
[21] Jonathan Corbet. 2016. Tracepoints with eBPF. https://lwn.net/Articles/683504/
[22] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. 2021. CAT: Content-aware tracing and analysis for distributed systems. In *Proceedings of the 22nd International Middleware Conference*. 223–235.
[23] Pawan Kumar and Rakesh Kumar. 2019. Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey. *ACM Comput. Surv.* 51, 6 (Feb. 2019). https://doi.org/10.1145/3281010 Place: New York, NY, USA Publisher: Association for Computing Machinery.
[24] Daniel Kunkle and Jiri Schindler. 2008. A load balancing framework for clustered storage systems. In *International Conference on High-Performance Computing*. Springer, 57–72.
[25] Haitao Li, Yuliang Yang, and Bin Zheng. 2012. Research on Billing Strategy of Cloud Storage. In *2012 Fourth International Conference on Multimedia Information Networking and Security*. 624–627. https://doi.org/10.1109/MINES.2012.172
[26] Bjørn Lindi. [n. d.]. I/O-profiling with Darshan. *PRACE report* ([n. d.]).
[27] Guoxin Liu, Haiying Shen, and Haoyu Wang. 2015. Computing load aware and long-view load balancing for cluster storage systems. In *2015 IEEE International Conference on Big Data (Big Data)*. 174–183. https://doi.org/10.1109/BigData.2015.7363754
[28] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. 2021. EZIO-Tracer: unifying kernel and user space I/O tracing for data-intensive applications. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. ACM, Online Event United Kingdom, 1–11. https://doi.org/10.1145/3439839.3458731
[29] Francisco Neves, Nuno Machado, and others. 2018. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual*

IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 534–541.

[30] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. 2009. Service Level Agreement in Cloud Computing. *Kno.e.sis Publications* (Jan. 2009). https://corescholar.libraries.wright.edu/knoesis/78%7D

[31] Lorenzo Posani, Alessio Paccoia, and Marco Moschettini. 2018. The carbon footprint of distributed cloud storage. (2018). https://doi.org/10.48550/ARXIV.1803.06973 Publisher: arXiv Version Number: 3.

[32] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, and others. 2023. Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 420–437.

[33] Arie Taal, Dexter Drupsteen, Marc X. Makkes, and Paola Grosso. 2014. Storage to energy: Modeling the carbon emission of storage task offloading between data centers. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. 50–55. https://doi.org/10.1109/CCNC.2014.6866547

[34] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. 2011. EZTrace: a generic framework for performance analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 618–619.

[35] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, and Gregory R Ganger. 2011. Exertion-based billing for cloud storage access. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*.