A Case for Dynamically Programmable Storage Background Tasks

Ricardo Macedo
INESC TEC and U. Minho
ricardo.g.macedo@inesctec.pt

Alberto Faria
INESC TEC and U. Minho
alberto.c.faria@inesctec.pt

João Paulo
INESC TEC and U. Minho
joao.t.paulo@inesctec.pt

José Pereira
INESC TEC and U. Minho
jop@di.uminho.pt

***Abstract*—Modern storage infrastructures feature long and complicated I/O paths composed of several layers, each employing their own optimizations to serve varied applications with fluctuating requirements. However, as these layers do not have global infrastructure visibility, they are unable to optimally tune their behavior to achieve maximum performance. Background storage tasks, in particular, can rapidly overload shared resources, but are executed either periodically or whenever a certain threshold is hit regardless of the overall load on the system.**

In this paper, we argue that to achieve optimal holistic performance, these tasks should be dynamically programmable and handled by a controller with global visibility. To support this argument, we evaluate the impact on performance of compaction and checkpointing in the context of HBase and PostgreSQL. We find that these tasks can respectively increase 99th percentile latencies by 955.2% and 61.9%. We also identify future research directions to achieve programmable background tasks.

***Index Terms*—Storage Systems, Background Tasks, I/O Interference, Programmable Storage, Software-Defined Storage**

I. Introduction

A massive, ever-growing amount of digital information is continuously generated, stored, and processed in both public and private premises [1]. The consequent continual need for greater storage and processing capacity has significantly increased the complexity of the underlying infrastructures. These feature long and complicated I/O paths composed of several layers (*e.g.*, hypervisors, operating systems, device drivers, I/O schedulers, file systems, databases), each employing its own optimizations (*e.g.*, caching [2], I/O sequentialization [3], task prioritization [4]) to serve a variety of applications with requirements that fluctuate over time.

However, as these layers do not have global infrastructure visibility, they are unable to tune their behavior in order to achieve optimal system-wide performance. Thus, the configuration of each layer is currently done on an individual basis. Under this design, ensuring optimal end-to-end performance is hard, and if not correctly assessed can lead to high levels of I/O interference and performance degradation [4, 5]. This effect becomes further amplified when the same resources are operated on by concurrent I/O services, coming from other layers of the I/O stack or even from internal background activities of a given layer that are competing for resources with the corresponding foreground tasks.

In particular, background tasks such as compaction [6], checkpointing [7], and replication [8] are predefined I/O-intensive activities that can rapidly overload shared resources, introducing significant I/O interference and workload burstiness, ultimately impacting overall throughput and latency. Currently, to minimize interference with foreground activities, background tasks are processed in a best-effort manner, being executed either periodically or whenever a certain threshold is hit. In many cases, these tasks are also rate-limited to avoid impacting other foreground activities. Again, the decision of when and how to execute such operations is taken by the layer itself regardless of the overall load on the infrastructure at the time.

In this paper, we argue that in order to achieve optimal holistic performance, storage background tasks should be dynamically programmable and their execution handled by a control module with system-wide visibility and end-to-end I/O control. Each layer should thus expose its set of supported background tasks and corresponding configurations to such a controller, which would in turn provide the building blocks for executing each task with optimal I/O performance. Such a programmable environment is aligned with those proposed by current Software-Defined Storage (SDS) solutions, which ensure holistic control of the I/O stack by breaking the vertical alignment of conventional infrastructures and separating storage policies from the control mechanisms that enforce them, thus enabling Quality of Service (QoS) provisioning, performance isolation, and resource fairness [5, 9].

To support this argument, we evaluate the impact of different background tasks, namely compaction and checkpointing, and corresponding configurations on the performance of two data stores: *HBase*, a highly-available distributed key-value store, and *PostgreSQL*, a relational database. Under *HBase* deployments, results show that background tasks introduce a performance degradation of up to 87.3% and 955.2% for mean and 99th percentile latencies, respectively. Under *PostgreSQL* deployments, mean and 99th percentile latencies experienced a performance degradation of at most 20.7% and 61.9% respectively, when executed under varying task settings.

Through the observed results we further substantiate our argument and provide future research directions to achieve the aforementioned design. The full evaluation results are also publicly available at https://rgmacedo.github.io/paper-background-tasks.

This paper is organized as follows. §II surveys related work. §III depicts the general organization of the contemplated I/O stacks. §IV describes the evaluation methodology, while §V presents the obtained results. §VI discusses current challenges and future directions, and concludes the paper.

II. Related Work

Recent works on SDS have been proposed to achieve holistic control of the I/O stack by breaking the vertical alignment of conventional infrastructures and separating storage policies from the mechanisms that employ them [5, 9]. However, current solutions do not consider the programmability and holistic control of background tasks, thus limiting their ability to enforce policies at higher levels of performance and QoS.

The impact of mixed background and foreground tasks in storage I/O performance has already been contemplated in the literature [4, 10]. These studies are mainly focused on the I/O stack of modern operating systems, *i.e.*, kernel caching, file system, and block device layers, and propose novel scheduling solutions for orchestrating I/O resources across the different tasks according to their priority. Similarly, several studies have investigated the root causes of tail latency for different I/O layers (*e.g.*, Linux schedulers, file systems) while taking into account the performance impact and I/O interference generated by background processes, such as garbage collection and file system initialization [11, 12, 13].

For the *HBase* software stack, previous studies have drawn conclusions on the overall I/O impact induced by compaction processes when performed in local and distributed settings [14]. Furthermore, recent works have also implemented several optimizations over Log-Structured Merge (LSM) tree [15] data stores. [16, 17, 18] provide internal LSM-level optimizations to decrease the amount of work performed during background operations, while [19] proposes an LSM-based I/O scheduling framework to efficiently manage regular I/O flows and internal maintenance work, and thus reduce latency spikes. However, such solutions are only applicable to LSM-based data stores, comprehending partial visibility of the I/O stack, and are designed to exclusively improve either overall throughput or tail latency.

This paper sets out to show that the impact of background tasks is not limited to the above-mentioned systems and components and that, in many cases, these tasks must be efficiently managed at the user level. The conducted study and the corresponding observations allow us to argue two main points that are not contemplated by previous studies. First, for complex applications, background and foreground tasks need to be considered in a holistic fashion, acknowledging both kernel- and user-space components that may have an impact in shared I/O resources. Secondly, simply applying scheduling and prioritization mechanisms may not be sufficient to ensure complex storage policies for applications and users (*e.g.*, enforcing throughput objectives and latency percentiles under high I/O interference). We argue that this will only be possible by increasing the programmability of critical I/O components. To the best of our knowledge, these points are not addressed by previous work.

III. Case Studies

Modern storage infrastructures comprehend multiple independent layers throughout the I/O path. Typical I/O stack settings of these infrastructures can be composed of applications, databases, caches, file systems, and storage devices. Figure 1 illustrates two examples of such stacks. The left side depicts a typical *HBase* deployment, composed of different I/O applications that submit read and write requests to *HBase*, a distributed, scalable, open-source NoSQL data store, which in turn reads/writes data from/to *HDFS*, a highly available distributed file system, finally backed by a local file system. On the right side, a *PostgreSQL* relational database serves incoming requests of different applications, and is directly backed by a local file system.

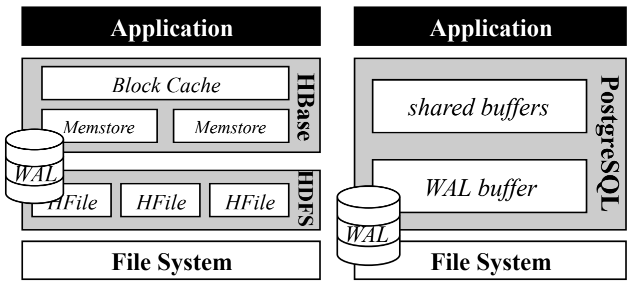

Fig. 1. I/O stack of typical *HBase* and *PostgreSQL* deployments.

A. HBase

A typical *HBase* deployment is composed of several *RegionServers* and an *HMaster*. *RegionServers* are the data units of the database, serving read and write requests from applications, while the *HMaster* is responsible for coordinating the overall database infrastructure and redirecting clients to the *RegionServers* that hold their data. To improve performance, database tables are horizontally partitioned by row key range into *Regions*, which are then assigned to *RegionServers*.

For write requests, applications submit a key-value pair to the *RegionServer*, which is first written to a Write Ahead Log (WAL) file that is used to persist all data on permanent storage and assist data recovery in case of server failure. Once persisted in the WAL, data is then written to a write-oriented caching instance known as *Memstore*, finally sending an acknowledgment message back to the application. The *Memstore* holds sorted key-value pairs of write requests. Moreover, each *Region* comprises several *Memstore* instances, each for a different part of the table. When the *Memstore* reaches a predefined size (*e.g.*, 128 MiB), its data is flushed to *HDFS* as a new *HFile*, which is then persistently written at the underlying file system. Internally, *HBase* stores its data in an LSM tree, similarly to other data stores [6, 20].

For reads, requests are first sent to a read-oriented caching instance known as *Block Cache*. If the requested records are not found, the *Memstore* is then checked. If the records are held in neither of these caching instances, *HBase* then reads a number of *HFiles* until they are found. After successfully reading the requested records, they are inserted into the *Block Cache* and sent back to the application.

The generation of different *HFiles* per *Memstore* instance leads to multiple files being examined during read requests, thus resulting in read amplification. To address this problem, *HBase* executes a background compaction process that merges a number of small-sized *HFiles* into fewer larger ones, a process termed *minor compaction*. Moreover, *HBase* provides an additional compaction process, namely *major compaction*, that merges and rewrites all *HFiles* comprehended in a *Region* into a single large file, removing all previously deleted entries in the process. While they significantly improve read performance, compaction processes can severely overload disk and network resources, introducing increased I/O interference and burstiness in the I/O stack.

B. PostgreSQL

Regarding *PostgreSQL*, typical deployments consist of a database server that handles requests from multiple applications. To access the database, a client connects to a running *postmaster* that establishes a communication channel between the application and the database.

For write operations (*e.g.*, INSERT, UPDATE, and DELETE statements), the database server maps the corresponding blocks in the *shared buffers* memory area and directly modifies them. The changes are written to a WAL buffer, usually as logical deltas, but in some cases by fully copying the complete physical 8 KiB block (*i.e.*, immediately after a checkpoint).

On commit, changes are persistently written to a WAL file on disk. Directly writing data blocks to the respective data files would lead to significant performance overhead due to the resulting random write pattern, and endanger recovery as writing 8 KiB blocks cannot be done atomically with common file system semantics. Instead, writes at the WAL file are sequential and can be truncated to include only complete records. Dirty blocks in shared memory are eventually written to the respective data files by a separate background process, becoming available again for other uses.

For read operations (*e.g.*, SELECT statements), required file blocks are also mapped to *shared buffers*, if not already present, being fetched from the kernel caching layer or read from disk. When the block is no longer needed, and if it is not dirty as a consequence of concurrent write operations, it can be immediately removed from *shared buffers* and remains only in the operating system cache.

In order to truncate the log and allow for fast recovery, *PostgreSQL* periodically performs *checkpoints*, a synchronization event that flushes all dirty data pages in the *shared buffers* to disk. Such an event guarantees that, in case of a failure, a crash recovery procedure seeks for the latest checkpoint record to determine from which point it should start the REDO operation. Checkpointing tasks are conducted in the background and are triggered either when the WAL file is about to exceed a certain size (1 GiB by default) or upon a checkpoint timeout (5 minutes by default). Setting these values to a lower bound will lead *PostgreSQL* to conduct checkpoints more often, allowing the database to recover faster after a failure.

To reduce I/O interference and workload burstiness, *PostgreSQL* throttles the write performance of checkpointing, leading to dirty buffers being written over a predefined period of time. To balance these trade-offs, *PostgreSQL* provides a *checkpoint completion target* parameter that allows database administrators to adjust the throughput at which checkpoints are created. When this parameter is set to a low value, checkpoints are performed faster, resulting in I/O burstiness. Setting it to a higher value ensures sustained performance and reduced I/O interference for foreground activities, leading however to higher recovery times after failure.

IV. Methodology

To illustrate the impact of background tasks on storage system performance and underline the importance of making them programmable, we evaluated the overhead imposed by each of the two types of background task identified above. Here, we describe the adopted evaluation methodology, which in particular aims to answer the following questions:

- *How much overhead do these background tasks impose?*
- *How does their overhead vary across operation types?*
- *How does their overhead vary across time?*
- *How do these tasks impact tail latencies?*
- *How do these tasks' configuration parameters influence their impact on performance?*

a) ***Evaluated deployments****:* To quantify the overhead imposed by the compaction process, we considered a local deployment consisting of an *HBase* v2.0.5 instance (1 *HMaster*, 1 *RegionServer*, and 1 *Zookeeper* instance, all in the same machine; dedicated heap size = 4 GiB, *Memstore* size = 0.4, block cache size = 0.25), backed by an *HDFS* v2.9.2 instance (1 *NameNode* and 1 *DataNode*, both in the same machine as *HBase*; replication factor = 1; block size = 128 MiB; dedicated heap size = 1 GiB), in turn backed by an ext4 file system.

In turn, to characterize the impact of checkpointing on performance, we considered a local deployment consisting of a *PostgreSQL* v11.3 instance backed by an ext4 file system.

Unless stated otherwise, remaining configuration parameters were kept at their default values for all components of both deployments.

b) ***Workloads****:* The aforementioned deployments were evaluated under several workloads generated using YCSB v0.15.0 [21] (running locally with the deployments), with different operation type proportions and access distributions:

- *Workload A:* 50% read, 50% update, Zipfian;
- *Workload B:* 100% update, Zipfian;
- *Workload C:* 100% read, uniform;
- *Workload D:* 5% read, 95% insert, Zipfian;
- *Workload E:* 95% scan, 5% insert, Zipfian;
- *Workload F:* 50% read, 50% read-modify-write, Zipfian.

These workloads were previously used in [22], with the exception of workload C which we modified to follow a uniform distribution instead of the Zipfian distribution employed by the remaining workloads, as it would also be interesting to analyze a different access pattern. Workloads were executed with both 1 and 10 threads.

We performed between 3 and 11 runs per combination of deployment, configuration, workload, and number of threads. A loading phase was conducted before each run, populating

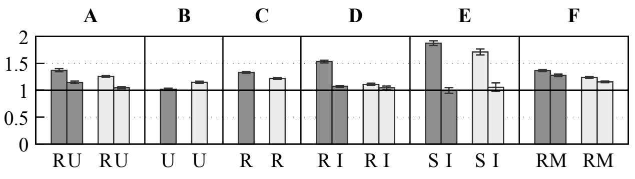

Fig. 2. *HBase* mean latency under the configuration *with* compaction effects normalized against the configuration *without* compaction effects, under workloads with 1 thread (▆) and 10 threads (▭). Error bars represent the 95% confidence interval, calculated as per Fieller's theorem. (`R` – read, `U` – update, `I` – insert, `S` – scan, `M` – read-modify-write)

the database with 12.5 million records (using approximately 16 GiB of disk space). After each run, the file system was recreated, caches were purged, and a 2 minute cool down period was given. Runs were configured to end after 10 million operations were performed or 17 minutes had elapsed.

*c) **Deployment configurations**:* The *HBase* deployment was evaluated under two scenarios (which we refer to as configurations): (1) *with compaction effects*, performing each workload immediately after the corresponding loading phase and thus evaluating the deployment under the effect of ongoing compactions; and (2) *without compaction effects*, waiting 30 minutes after the loading phase — enough time for resulting compactions to finish. This allowed us to measure the overhead imposed by the compaction process.

To understand the performance impact of checkpointing, we evaluated 6 configurations of the *PostgreSQL* deployment, each with a different combination of values for the *maximum WAL size* parameter — either 128 or 1024 MiB — and the *checkpoint completion target* parameter — 0.1, 0.5, or 0.9. We will denote these configurations by (x, y) tuples, where x represents the maximum WAL size (in MiB) and y is the completion target value. The default *PostgreSQL* settings correspond to the $(1024, 0.5)$ configuration.

*d) **Collected metrics**:* YCSB was configured to report the achieved mean and percentile latencies both for each run as a whole and for each 10 second period. Since YCSB reports throughput as simply the inverse of latency multiplied by the number of threads, we only present latency values.

For each combination of deployment, configuration, workload, and number of threads, the sample mean of each collected metric was calculated, and Student's *t*-distribution was used to compute the 95% confidence intervals for the corresponding population means. The half-width of those confidence intervals for values presented later or otherwise contemplated in our analysis is under 10% of the respective sample mean.

Finally, Dstat v0.7.3 was used to observe CPU, memory, and disk utilization.

*e) **Experimental environment:*** Experiments were conducted using machines with the following specifications: one Intel Core i3-4170 CPU, clocked at 3.70 GHz, with 2 physical and 4 logical cores; 8 GiB of DDR3 RAM, clocked at 1600 MHz; and one 119 GiB, SATA-III Samsung MZ7LN128 solid-state drive. Software-wise, the machines used Ubuntu Server 18.04 LTS for AMD64 with Linux kernel v4.15.0.

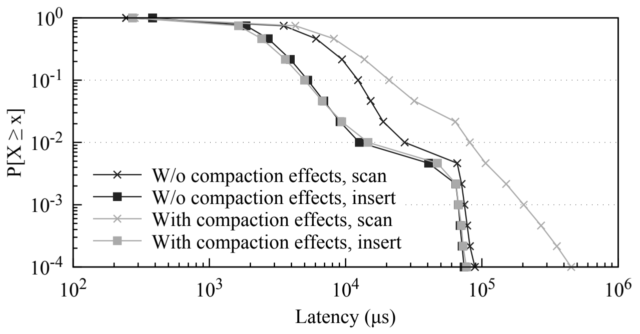

Fig. 3. Complementary cumulative distribution function for the latency of scan and insert operations attained by both *HBase* configurations under a single run of workload E with 10 threads.

V. Evaluation

Here, we present the results of the evaluation conducted on the *HBase* (§V-A) and *PostgreSQL* (§V-B) deployments described above. The full results are also available at https://rgmacedo.github.io/paper-background-tasks.

A. *HBase – Compaction*

Figure 2 depicts the mean latency achieved by the *HBase* deployment under the configuration *with* compaction effects, normalized against the configuration *without* compaction effects (taken as the baseline in the discussion that follows), under all workloads defined previously with both 1 and 10 threads.

Write-oriented operations, namely updates and inserts, present a performance degradation of at most 14.7% for mean latency, while read-oriented operations, namely reads and scans, exhibit a performance degradation of at most 87.3%. At the 99th percentile latency, depending on the workload, write operations experience performance degradations of at most 40.0%, while for reads, overhead ranges from being imperceptible to being as high as 955.2%. For read-modify-write operations, mean latency results experience overheads of at least 27.6%, while at the 99th percentile overheads of up to 281.0% are observed.

With respect to resource utilization, effects of compaction processes do not entail significant differences. CPU utilization remains mostly unaltered, while read and write disk throughput experiences low variation — a maximum absolute difference of 33 and 15 MiB/s, respectively. Interestingly, during compaction effects, this observed increase of disk throughput remains constant throughout the overall execution time (for all observations). This is due to the default throttling policy that *HBase* employs over compactions, limiting their I/O performance so as not to introduce high interference and performance overhead in incoming I/O requests [23].

As observed, write operations have a much lower impact in mean latency compared to read operations. This is due to two main factors: (1) the write operation flow at *HBase* introduces less overhead than read-oriented ones, as writes are first sequentially written to the WAL and then stored at the in-memory *Memstore*; and (2) the experienced compactions are *minor compactions*, thus not imposing major disk overload and I/O interference to write operations.

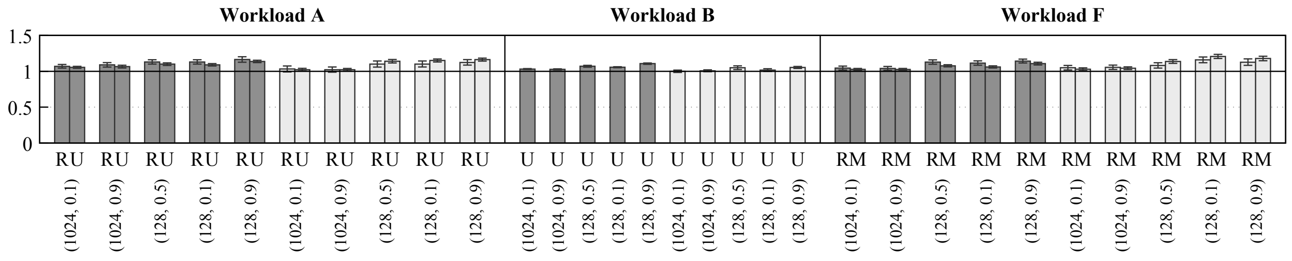

Fig. 4. *PostgreSQL* mean latency normalized against the $(1024, 0.5)$ configuration, under workloads with 1 thread (▇) and 10 threads (□). Error bars represent the 95% confidence interval, calculated as per Fieller's theorem. (R – read, U – update, M – read-modify-write)

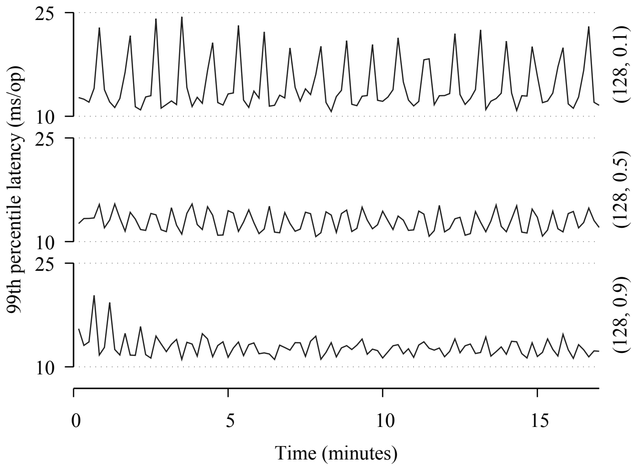

Fig. 5. 99th percentile latency variation for the update operation over the duration of a single run for each of the $(128, 0.1)$, $(128, 0.5)$, and $(128, 0.9)$ *PostgreSQL* configurations, under workload A with 1 thread.

On the other hand, read operations experience significant mean latency overhead due to the inherent dependency over *HFiles* currently being compacted. Specifically, compactions are mainly motivated to improve read performance, rewriting several small *HFiles* into fewer larger ones. As such, under compactions effects, read operations will be severely influenced by the concurrent merging operations being conducted. Moreover, this becomes further amplified at the 99th and higher percentile latencies, as depicted in Figure 3, in which a point (x, y) signifies that y is the fraction of requests that experience a latency of x µs or higher. As observed, insert requests present similar latency for all percentiles. Contrarily, scan requests experience a significant latency increase between the 50th and 99.99th percentiles, achieving a 452 ms latency at the 99.99th percentile, a 509% increase compared to the baseline.

B. *PostgreSQL – Checkpointing*

Figure 4 shows the mean latency achieved by the various *PostgreSQL* configurations defined previously, normalized against the default $(1024, 0.5)$ configuration, under workloads A, B, and F with both 1 and 10 threads.

Results show that the completion target value does not significantly impact performance when configured with a 1024 MiB WAL size ($(1024, y)$ configurations). Specifically, the overhead on mean latency ranges from being imperceptible to up to 8.9%. Analogously, at the 99th percentile latency, performance is degraded by at most 8.7%. Contrarily, under a 128 MiB WAL size, all workloads experienced a much more noticeable performance degradation in both mean and 99th percentile latencies. As opposed to the *HBase* deployments (§V-A), both read- and write-oriented operations were equally exposed to latency overheads. Specifically, experiments conducted for the $(128, 0.5)$, $(128, 0.1)$, and $(128, 0.9)$ configurations showed a performance degradation for the mean latency of at most 13.9%, 20.7%, and 17.8%, respectively. Likewise, this effect is further amplified at 99th percentile latencies, exposing an overhead of 33.2%, 61.9%, and 33.3% under the same configurations. This is due to the high checkpoint frequency entailed by small-sized WAL files. Regarding resource utilization, no significant differences are observed when varying the WAL size and completion target value.

As observed, *PostgreSQL* WAL size plays a major role in the overall performance, as flushing larger WAL files to the file system provides better end-to-end performance when compared to smaller sizes, thus ensuring sustained latency performance. On the other hand, despite the low differences in mean latency, *PostgreSQL* checkpoint completion target noticeably impacts latencies at the 99th and higher percentiles, introducing severe I/O interference and performance variability. Figure 5 depicts a representative case, showing the performance variation under different checkpoint completion target values. While the achieved mean latency was 4.598 ms, 4.559 ms, and 4.755 ms for the $(128, 0.5)$, $(128, 0.1)$, and $(128, 0.9)$ configurations, respectively, latency at the 99th percentile obtained 13.060 ms, 14.729 ms (+12.7%), and 12.769 ms (-2.2%) for the same configurations. When the completion target is set to a higher value, namely 0.9, *PostgreSQL* throttles the write performance for WAL files, providing a more conservative and sustained performance for incoming I/O requests. When set to a lower value, namely 0.1, the system experiences increased I/O burstiness, achieving 10.687 ms and 24.383 ms as minimum and maximum 99th percentile latencies (observed at 10 second intervals), respectively.

VI. Discussion and Conclusion

Results show that both compaction and checkpointing background tasks heavily impact the performance of foreground activities, introducing significant I/O interference and performance degradation of both mean and 99th percentile latencies.

The reason for this impact on performance is twofold. First, by default, *HBase* throttles both read and write performance of compactions and simultaneously does not provide sufficient building blocks to dynamically tune such settings, forcing applications to experience I/O interference and performance degradation for longer periods of time. These background tasks should be inherently programmable in order to adjust compaction primitives to the overall infrastructure load, and to comply with varying storage policies submitted by applications. Second, although *PostgreSQL* already provides a wide set of primitives to adjust checkpointing processes, these cannot be dynamically tuned to the overall load of the infrastructure. As such, they are not sufficient to ensure complex storage policies for applications and users with time varying requirements.

We thus restate our argument that to attain optimal holistic performance, storage background tasks should be dynamically programmable and their execution handled by a control module with global infrastructure visibility and holistic I/O control.

Following SDS principles, we identify two steps that must be taken in order to achieve this goal. First, at layer level, storage systems should decouple background mechanisms from the policies that govern them. Specifically, these mechanisms should be agnostic of the I/O layer and exported to another layer with extended visibility and more granular control over I/O flows. Such a layer would expose the building blocks to dynamically adapt and fine-tune background activities to the overall infrastructure load. This would allow, for instance, for *HBase* storage applications to decide when to perform compactions and at which rate. *PostgreSQL* applications would similarly benefit from this design, being able to dynamically configure checkpointing tasks as intended.

Second, at infrastructure level, a policy-enabled SDS controller with system-wide visibility would provide adaptable end-to-end control over storage resources to adjust I/O flows in a holistic fashion. Such a controller would employ control and monitoring endpoints throughout the I/O path to collect different metrics at runtime (*e.g.*, throughput, latency, resource utilization), thus allowing for continuous adaptability of background primitives and achieving sustained performance under heterogeneous infrastructures with evolving requirements.

As opposed to current work, where most improvements have focused rather narrowly on fine-grained, system-specific optimizations, we argue that following an SDS-enabled design with holistic I/O control and programmability would improve the management of regular I/O flows and internal maintenance work, as well as minimize performance variability and I/O interference and ensure higher levels of QoS provisioning and resource fairness.

Acknowledgments

This work was supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within the project UID/EEA/50014/2019.

References

[1] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao *et al.*, "Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018," Tech. Rep., 2018.

[2] E. Lee and H. Bahn, "Caching Strategies for High-Performance Storage Media," *ACM Transactions on Storage*, vol. 10, no. 3, Aug. 2014.

[3] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device," in *15th USENIX Conference on File and Storage Technologies*, 2017.

[4] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O Path: A Holistic Approach for Application Performance," in *15th USENIX Conference on File and Storage Technologies*, 2017.

[5] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-Defined Storage Architecture," in *24th ACM Symposium on Operating Systems Principles*, 2013.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, Jun. 2008.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1992.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.

[9] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted Resource Management in Multi-tenant Distributed Systems," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[10] H. Jo, S. hun Kim, S. Kim, J. Jeong, and J. Lee, "Request-aware Cooperative I/O Scheduling for Scale-out Database Applications," in *9th USENIX Workshop on Hot Topics in Storage and File Systems*, 2017.

[11] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *5th ACM Symposium on Cloud Computing*, 2014.

[12] Z. Cao, V. Tarasov, H. Raman, D. Hildebrand, and E. Zadok, "On the Performance Variation in Modern Storage Stacks," in *15th USENIX Conference on File and Storage Technologies*, 2017.

[13] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of ACM*, vol. 56, no. 2, 2013.

[14] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS Under HBase: A Facebook Messages Case Study," in *12th USENIX Conference on File and Storage Technologies*, 2014.

[15] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996.

[16] N. Dayan and S. Idreos, "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging," in *2018 International Conference on Management of Data*, 2018.

[17] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs," in *14th USENIX Conference on File and Storage Technologies*, 2016.

[18] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *2017 USENIX Annual Technical Conference*, 2017.

[19] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *2019 USENIX Annual Technical Conference*, 2019.

[20] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, Apr. 2010.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *1st ACM Symposium on Cloud Computing*, 2010.

[22] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça, "MeT: Workload aware elasticity for NoSQL," in *8th ACM European Conference on Computer Systems*, 2013.

[23] "Limit compaction speed," Website, 2017, retrieved July 1, 2019. [Online]. Available: https://issues.apache.org/jira/browse/HBASE-8329