User-level Software-Defined Storage Data Planes

Ricardo Macedo
ricardo.g.macedo@inesctec.pt
INESC TEC & University of Minho
Portugal

ABSTRACT

We present Paio, a framework that allows developers to implement portable I/O optimizations for different applications with minor modifications to their original code base. The chief insight behind Paio is that if we are able to intercept and differentiate requests as they flow through different layers of the I/O stack, we can enforce complex storage policies without significantly changing the layers themselves. We demonstrate the performance and applicability of Paio by: (1) improving 99th percentile latency by 4× in LSM-based key-value stores; and (2) ensuring dynamic per-application bandwidth guarantees under shared storage environments.

1 MOTIVATION

Data-centric systems such as databases, key-value stores (KVS), and machine learning engines have become an integral part of modern I/O stacks. Good performance for these systems often requires storage optimizations such as I/O scheduling, differentiation, and caching. However, these optimizations are implemented in sub-optimal manner.

Tightly coupled optimizations. Most I/O optimizations are single-purposed as they are tightly integrated within the core of each system. Implementing these requires deep understanding of the system's internal operation model and profound code refactoring, limiting their maintainability and portability. For example, optimizations such as differentiating foreground and background I/O to reduce tail latency are broadly applicable; however, the way they are implemented in KVS today (*e.g.,* SILK [1]) requires a deep understanding of the system, and are not portable across other KVS.

Rigid interfaces. To address the previous challenge, one could decouple optimizations from the internal system's logic. However, this comes with a cost, since one loses the granularity and internal application knowledge present in system-specific optimizations. Specifically, conventional I/O stacks require layers to communicate through rigid interfaces that cannot be easily extended (*e.g.,* POSIX), discarding information that could be used to classify and differentiate requests at different levels of granularity.

Kernel-level layers. While promoting general applicability, implementing I/O optimizations at the kernel (*e.g.,* file system, block layer) poses several disadvantages. For application-level information to be propagated to kernel, it requires breaking user-to-kernel and kernel-internal interfaces, decreasing portability and compatibility. Further, these optimizations are ineffective under kernel-bypass storage stacks (*e.g.,* SPDK, PMDK), since I/O requests are submitted directly from user-space to the storage device.

Partial visibility. Optimizations implemented in isolation are oblivious of other systems that compete for the same storage resources. Under shared infrastructures (*e.g.,* cloud, HPC), this lack of coordination can lead to conflicting optimizations, I/O contention, and performance variation for both applications and storage backends.

2 PAIO DATA PLANE FRAMEWORK

To address these challenges, we present Paio, a user-level framework that enables system designers to build portable and generally applicable storage optimizations by adopting ideas from the Software-Defined Storage community [3].

The key idea is to implement the optimizations *outside* the applications, as *data plane stages*, by intercepting and handling the I/O performed by these. These optimizations are then controlled by a logically centralized manager, the *control plane*, that has the global context necessary to prevent interference among them. Using Paio, one can decouple complex storage optimizations from current systems, such as I/O differentiation and scheduling, while achieving results similar to or better than tightly coupled optimizations. Paio's design is built over five core principles.

General applicability. To ensure applicability across different I/O layers, Paio stages are disaggregated from the internal system logic, contrary to tightly coupled solutions.

Programmable building blocks. Paio follows a decoupled design that separates the I/O mechanisms from the policies that govern them, and provides the necessary abstractions for building new storage optimizations to employ over requests.

Fine-grained control. Paio classifies, differentiates, and enforces requests with different levels of granularity, enabling a broad set of policies to be applied over the I/O stack.

Stage coordination. Paio exposes a control interface that enables an external control plane to coordinate access to resources of each stage.

Low intrusiveness. Porting I/O layers to use Paio requires none to minor code changes.

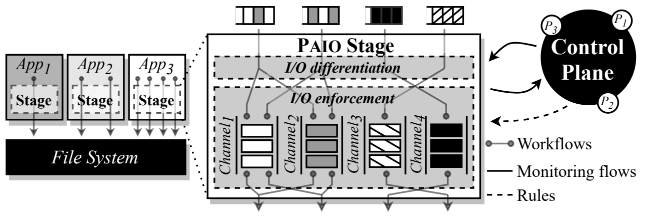

Figure 1: PAIO allows implementing programmable and adaptable user-level storage data plane stages.

2.1 High-level Architecture

Fig. 1 outlines PAIO's high-level architecture. It follows a decoupled design that separates policies, implemented at an external control plane, from the mechanisms that enforce them, implemented at the data plane stage. PAIO targets I/O layers at the user-level. Stages are embedded within layers, intercepting all I/O requests and enforcing user-defined policies. PAIO is organized in four main components.

Stage interface. Applications access stages through a stage interface that routes all requests to PAIO before being submitted to the next I/O layer (*i.e., App*$_3$ $\rightarrow$PAIO $\rightarrow$*File System*). For each request, it generates a *Context* object with the corresponding I/O classifiers.[1]

Differentiation module. The differentiation module classifies and differentiates requests based on their *Context* object. To ensure requests are differentiated with fine-granularity, we combine ideas from *context propagation* [2] to enable application-level information, only accessible to the layer itself, to be propagated to PAIO.

Enforcement module. The enforcement module is responsible for applying the actual I/O mechanisms over requests. It is organized with channels and enforcement objects.[2] For each request, the module selects the channel and enforcement object that should handle it. After being enforced, requests are returned to the original data path and submitted to the next I/O layer (*File System*).

Control interface. PAIO exposes a control interface that enables the control plane to (1) orchestrate the stage lifecycle by creating channels and enforcement objects, and (2) ensure all policies are met by continuously monitoring and fine-tuning the stage. The control plane provides global visibility, ensuring that stages are controlled holistically.

3 RESULTS

We validate the feasibility of using PAIO under two use cases.

[1]A **context object** contains metadata that characterizes a request, including *workflow id, request type, request size,* and *request context.*

[2]**Enforcement objects** are self-contained, single-purposed mechanisms (*e.g.,* token-buckets, caches, compression) that apply custom I/O logic over I/O requests, while **channels** are a stream-like abstraction through which requests flow. Each channel contains one or more enforcement objects.

Tail latency control in KVS. We implement a stage in RocksDB, an industry-standard KVS, and demonstrate how to prevent latency spikes by orchestrating foreground and background tasks. Results show that by propagating application-level information to the data plane stage, PAIO outperforms RocksDB by at most 4× in 99^{th} percentile latency, and achieves similar control and performance when compared to system-specific, latency-oriented optimizations (SILK). Integrating PAIO in RocksDB only required adding 85 LoC.

Per-application bandwidth control. We apply PAIO to TensorFlow and show how to achieve per-application bandwidth guarantees under a real shared-storage scenario at the ABCI supercomputer. Results show that, by having global visibility, PAIO provisions per-application bandwidth guarantees at all times, and improves overall execution time when compared to a static rate limiting approach. Integrating TensorFlow with PAIO did not required any code changes.

4 CONCLUSION

We have presented PAIO, a framework that enables system designers to build data plane stages applicable over different I/O layers. It provides differentiated treatment of requests and allows implementing storage mechanisms adaptable to different policies. By combining ideas from SDS and context propagation, we demonstrated that PAIO decouples system-specific optimizations to a more programmable environment (*i.e.,* a self-contained, easier to maintain, and portable stage), while enabling similar I/O control and performance. PAIO was published and presented at USENIX FAST 2022 [4].

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020, and through the PhD Fellowship SFRH/BD/146059/2019.

REFERENCES

[1] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference.* USENIX Association.

[2] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *13th European Conference on Computer Systems.* ACM. https://doi.org/10.1145/3190508.3190526

[3] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2020. A Survey and Classification of Software-Defined Storage Systems. *ACM Computing Surveys* 53, 3, Article 48 (2020). https://doi.org/10.1145/3385896

[4] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. 2022. PAIO: General, Portable I/O Optimizations With Minor Application Modifications. In *20th USENIX Conference on File and Storage Technologies.* USENIX Association.