PAIO: General, Portable I/O Optimizations With Minor Application Modifications

Ricardo Macedo, Yusuke Tanimura[†], Jason Haga[†], Vijay Chidambaram[‡], José Pereira, João Paulo
INESC TEC and University of Minho [†]*AIST* [‡]*UT Austin and VMware Research*

Abstract

We present PAIO, a framework that allows developers to implement portable I/O policies and 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. PAIO adopts ideas from the Software-Defined Storage community, building data plane stages that mediate and optimize I/O requests across layers and a control plane that coordinates and fine-tunes stages according to different storage policies. We demonstrate the performance and applicability of PAIO with two use cases. The first improves 99^{th} percentile latency by 4× in industry-standard LSM-based key-value stores. The second ensures dynamic per-application bandwidth guarantees under shared storage environments.

1 Introduction

Data-centric systems such as databases, key-value stores (KVS), and machine learning engines have become an integral part of modern I/O stacks [12, 19, 32, 43, 53, 55]. Good performance for these systems often requires storage optimizations such as I/O scheduling, differentiation, and caching. However, these optimizations are implemented in a sub-optimal manner, as these are *tightly coupled to the system implementation*, and can *interfere with each other due to lack of global context*. 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 [16]) requires a deep understanding of the system, and are not portable across other KVS. Similarly, optimizations from applications deployed at shared infrastructures may conflict due to not being aware of each other [27, 51, 61, 62].

In this paper, we argue that there is a better way to implement such storage optimizations. We present PAIO, a user-level framework that enables building portable and generally applicable storage optimizations by adopting ideas from the Software-Defined Storage (SDS) community [38]. 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. PAIO does not require any modifications to the kernel (critical for deployment). 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.

Building PAIO is not trivial, as it requires addressing multiple challenges that are not supported by current solutions. To perform complex I/O optimizations outside the application, PAIO needs to *propagate context* down the I/O stack, from high-level APIs down to the lower layers that perform I/O in smaller granularities.[1] It achieves this by combining ideas from *context propagation* [36], enabling application-level information to be propagated to data plane stages with minor code changes and without modifying existing APIs.

PAIO requires the design of new abstractions that allow differentiating and mediating I/O requests between user-space I/O layers. These abstractions must promote the implementation and portability of a variety of storage optimizations. PAIO achieves this with four main abstractions. The *enforcement object* is a programmable component that applies a single user-defined policy, such as rate limiting or scheduling, to incoming I/O requests. PAIO characterizes and differentiates requests using *context objects*, and connects I/O requests, enforcement objects and context objects through *channels*. To ensure coordination (*e.g.,* fairness, prioritization) across independent storage optimizations, the control plane, with global visibility, fine-tunes the enforcement objects by using *rules*.

With these new features and abstractions, system designers can use PAIO to develop custom-made SDS data plane stages. To demonstrate this, we validate PAIO under two use cases. First, we implement a stage in RocksDB [9] and demonstrate how to prevent latency spikes by orchestrating foreground and background tasks. Results show that a PAIO-enabled RocksDB improves 99^{th} percentile latency by 4× under different workloads and testing scenarios (*e.g.,* different storage devices, with and without I/O bandwidth restrictions) when compared to baseline RocksDB, and achieves similar tail latency performance when compared to SILK [16]. Our approach demonstrates that complex I/O optimizations, such as SILK's I/O scheduler, can be decoupled from the original layer to a self-contained, easier to maintain, and portable stage. Second, we apply PAIO to TensorFlow [11] and show how to achieve dynamic per-application bandwidth guarantees under a real shared-storage scenario at the ABCI supercomputer [1]. Results show that all PAIO-enabled TensorFlow instances are

[1] We refer to the term *"layer"* as a component of a given I/O stack that handles I/O requests (*e.g.,* application, KVS, file system, device driver).

provisioned with their bandwidth goals. This shows that PAIO enables enforcing storage policies with system-wide visibility and holistic control.

In summary, the paper makes the following contributions:

- PAIO, a user-level framework for building programmable and dynamically adaptable data plane stages (§3-§7). PAIO is publicly available at https://github.com/dsrhaslab/paio.
- Implementation of two stages to (1) reduce latency spikes in an LSM KVS; and (2) achieve per-application bandwidth guarantees under shared storage settings (§8).
- Experimental results demonstrating PAIO's performance and applicability under synthetic and real scenarios (§9).

2 Motivation and Challenges

We now describe the problems of system-specific I/O optimizations and how these drive the proposal of PAIO.

Problem 1: tightly coupled optimizations. Most I/O optimizations are single-purposed as they are tightly integrated within the core of each system [16, 29, 50]. Implementing these optimizations requires deep understanding of the system's internal operation model and profound code refactoring, limiting their maintainability and portability across systems that would equally benefit from them. For instance, to reduce tail latency spikes at RocksDB, an industry-standard LSM-based KVS, SILK proposes an I/O scheduler to control the interference between foreground and background tasks. However, applying this optimization over RocksDB required changing several core modules made of thousands of LoC, including *background operation handlers*, *internal queuing logic*, and *thread pools* [5, 15]. Further, porting this optimization to other KVS (*e.g.,* LevelDB [21], PebblesDB [47]) is not trivial, as even though they share the same high-level design, the internal I/O logic differs across implementations (*e.g.,* data structures [20, 47], compaction algorithms [34, 47]).

Solution: decouple optimizations. I/O optimizations should be disaggregated from the system's internal logic and moved to a dedicated layer, becoming generally applicable and portable across different scenarios.

Resulting challenge: rigid interfaces. Decoupling optimizations comes with a cost, as we lose the granularity and internal application knowledge present in system-specific optimizations. Specifically, the operation model of conventional I/O stacks requires layers to communicate through rigid interfaces that cannot be easily extended, discarding information that could be used to classify and differentiate requests at different levels of granularity [13]. For instance, let us consider the I/O stack depicted in Fig. 1 made of an *Application*, a *KVS*, and a POSIX-compliant *File System*. POSIX operations submitted from the *KVS* can be originated from different workflows, including foreground (a) and background flows *i.e.,* flushes (b) and compactions (c). The *File System* however, can only observe the request's size and type (*i.e.*, read and write), making it impossible to infer its origin. Implementing SILK's I/O scheduler at a lower layer (*e.g., File System*, layer between the *KVS* and the *File System*), would make the optimization portable to other KVS solutions. However, it would be ineffective since it could not differentiate between foreground and background operations.

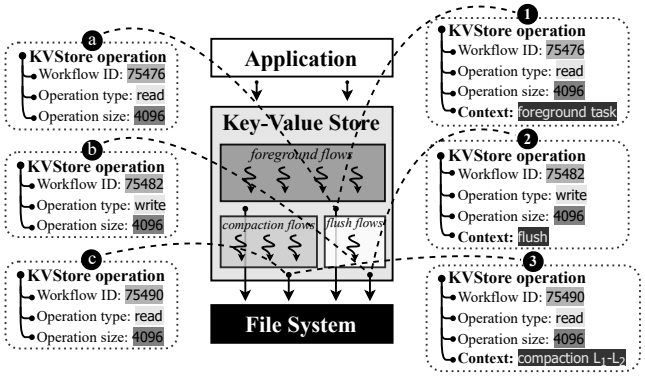

Figure 1: ***Operations submitted from different workflows.*** *Example of the operation flow of a multi-layered I/O stack. Left side depicts the regular information that can be extracted from operations between the KVS and File System, while the right side propagates additional request information throughout layers.*

Solution: information propagation. Application-level information must be propagated throughout layers to ensure that decoupled optimizations can ensure the same level of control and performance as system-specific ones.

Resulting challenge: kernel-level layers. While implementing SILK's I/O scheduler at the kernel (*e.g.,* file system, block layer) would promote its applicability across other KVS solutions, it poses several disadvantages. First, for application-level information to be propagated to these layers, it requires breaking user-to-kernel (*i.e.,* POSIX) and kernel-internal interfaces (*e.g.,* VFS, block layer, page cache), decreasing portability and compatibility [13]. Further, kernel-level development is more restricted and error prone than in user-level [42, 56]. Finally, these optimizations would be ineffective under kernel-bypass storage stacks (*e.g.,* SPDK [10], PMDK [8]), since I/O requests are submitted directly from the application (user-space) to the storage device.

Solution: actuate at user-level. I/O optimizations should be implemented at a dedicated user-level layer, promoting portability across different systems and scenarios, and easing information propagation throughout layers.

Problem 2: 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 [27, 62], I/O contention, and performance variation for both applications and storage backends [51, 61].

Solution: global control. Optimizations should be aware of the surrounding environment and operate in coordination to ensure holistic control of I/O workflows and shared resources.

3 PAIO in a Nutshell

PAIO is a framework that enables system designers to build custom-made SDS data plane stages. A data plane stage built with PAIO targets the workflows of a given user-level layer, enabling the classification and differentiation of requests and the enforcement of different storage mechanisms according to user-defined storage policies. Examples of such policies can be as simple as rate limiting greedy tenants to achieve resource fairness, to more complex ones as coordinating workflows with different priorities to ensure sustained tail latency. 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 I/O control. PAIO classifies, differentiates, and enforces I/O requests with different levels of granularity, enabling a broad set of policies to be applied over the I/O stack.

Stage coordination. To ensure stages have coordinated access to resources, PAIO exposes a control interface that enables the control plane to dynamically adapt each stage to new policies and workload variations.

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

3.1 Abstractions in PAIO

PAIO uses four main abstractions, namely *enforcement objects*, *channels*, *context*, and *rules*.

Enforcement object. An enforcement object is a self-contained, single-purposed mechanism that applies custom I/O logic over incoming I/O requests. Examples of such mechanisms can range from *performance control* and *resource management* such as token-buckets and caches, *data transformations* as compression and encryption, to *data management* (*e.g.,* data prefetching, tiering). This abstraction provides to system designers the flexibility and extensibility for developing new mechanisms tailored for enforcing specific storage policies.

Channel. A channel is a stream-like abstraction through which requests flow. Each channel contains one or more enforcement objects (*e.g.,* to apply different mechanisms over the same set of requests) and a *differentiation rule* that maps requests to the respective enforcement object to be enforced.

Context object. A context object contains metadata that characterizes a request. It includes a set of elements (or *classifiers*), such as the *workflow id* (*e.g.,* thread-ID), *request type* (*e.g.,* `read`, `open`, `put`, `get`), *request size*, and the *request context*, which is used to express additional information of a given request, such as determining its origin, context, and more. For each request, PAIO generates the corresponding *Context* object that is used for classifying, differentiating, and enforcing the request over the respective I/O mechanisms.

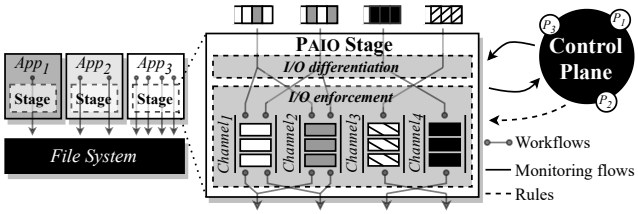

Figure 2: PAIO *overview.* PAIO *is a user-level framework that allows implementing programmable and adaptable data plane stages.*

Rule. In PAIO, a rule represents an action that controls the state of a data plane stage. Rules are submitted by the control plane and are organized in three types: *housekeeping rules* manage the internal stage organization, *differentiation rules* classify and differentiate I/O requests, *enforcement rules* adjust enforcement objects upon workload variations.

3.2 High-level Architecture

Fig. 2 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. To achieve this, PAIO is organized in four main components.

Stage interface. Applications access stages through a stage interface (§6.1) 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.

Differentiation module. The differentiation module (§4) classifies and differentiates requests based on their *Context* object. To ensure requests are differentiated with fine-granularity, we combine ideas from *context propagation* [36] to enable application-level information, only accessible to the layer itself, to be propagated to PAIO, broadening the set of policies that can be enforced.

Enforcement module. The enforcement module (§5) is responsible for applying the actual I/O mechanisms over requests. It is organized with channels and enforcement objects. 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 (§6.1) that enables the control plane to (1) orchestrate the stage lifecycle by creating channels, enforcement objects, and differentiation rules, 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 holisti-

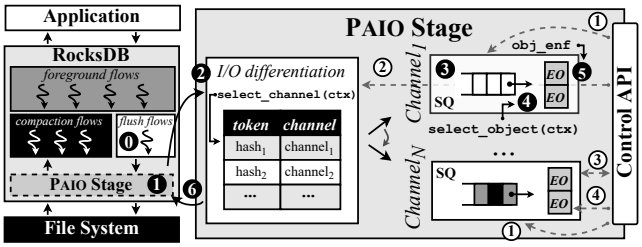

Figure 3: PAIO *operation flow.* *Black circles depict the execution flow of a request in the* PAIO *stage. White circles depict the control flow between the SDS control plane and the stage.*

cally. Exposing this interface allows stages to be managed by existing control planes [22, 35, 54].

3.3 A Day in the Life of a Request

Before delving into PAIO's internal modules, we first illustrate how it orchestrates the workflows of a given layer. We consider the I/O stack depicted on Fig. 3, which is made of an *Application*, *RocksDB*, a PAIO stage, and a POSIX-compliant *File System*; and the enforcement of the following policy: *"limit the rate of RocksDB's flush operations to X MiB/s"*. *RocksDB*'s background workflows generate flush and compaction jobs, which are translated in multiple POSIX operations that are submitted to the *File System*. Flushes are translated in `writes`, while compactions in `reads` and `writes`.

At startup time, *RocksDB* initializes the PAIO stage, which connects to an already deployed control plane. The control plane submits housekeeping rules to create a channel and an enforcement object that rate limits requests at *X* MiB/s (①). It also submits differentiation rules (②) to determine which requests should be handled by the stage, namely flush-based `writes`. Details on how the *differentiation* and *enforcement* processes work are given in §4 and §5, respectively.

At execution time, *RocksDB* propagates the context at which a given operation is created (❶) and redirects all `write` operations to PAIO (❶). Through ❶, we ensure that only `write` operations are enforced at PAIO, while with ❶, we differentiate flush-marked `writes` from others that can be triggered by compactions jobs. Upon a flush-based `write`, a *Context* object is created with its *request type* (`write`), *context* (`flush`), and *size*, and submitted, along the request, to the stage (❶). Then, the stage selects the channel (❷) to be used, enqueues the request (❸), and selects the enforcement object to service the request (❹), which in turn rate limits the request at *X* MiB/s (❺). After enforcing the request (❻), the original `write` operation is submitted to the *File System* .

The control plane continuously monitors and fine-tunes the data plane stage. Periodically, it collects from the stage the throughput at which requests are being serviced (③). Based on this metric, the control plane may adjust the enforcement object to ensure flush operations flow at *X* MiB/s, generating enforcement rules with new configurations (④).

Table 1: *Examples of the type of requests a channel receives.*

Channel	Workflow ID	Request context	Request type
$channel_1$	$flow_1$	—	—
$channel_2$	—	*background tasks*	`read`
$channel_3$	$flow_5$	*compaction*	`write`

4 I/O Differentiation

PAIO's differentiation module provides the means to classify and differentiate requests at different levels of granularity, namely *per-workflow*, *request type*, and *request context*. The process for differentiating requests is achieved in three phases.

Startup time. At startup time, the user defines *how* requests are differentiated and *who* should handle each request. First, it defines the granularity of the differentiation, by specifying which I/O classifiers should be used to differentiate requests. For example, to provide per-workflow differentiation PAIO only considers the *Context's workflow id* classifier, while to differentiate requests based on their context and type, it uses both *request context* and *request type* classifiers. Second, the user attributes specific I/O classifiers to each channel to determine the set of requests that a given channel receives. Table 1 provides examples of this specification: $channel_1$ only receives requests from $flow_1$, while $channel_2$ only handles `read` requests originated from *background tasks*; $channel_3$ receives compaction-based `writes` from $flow_5$. To generate a unique identifier that maps requests to channels, the classifiers can be concatenated into a string or hashed into a fixed-size token (§7). Further, this process can be set by the control plane (*i.e.,* differentiation rules) or configured at stage creation.

Execution time. The second phase differentiates the I/O requests submitted to the stage and routes them to the respective channel to be enforced. This is achieved in two steps.

Channel selection. For each incoming request, which is accompanied by its *Context* object, PAIO selects the channel that must service it (Fig. 3, ❷). PAIO verifies the *Context*'s I/O classifiers and maps the request to the respective channel to be enforced. This mapping is done as described in the first phase of the differentiation process.

Enforcement object selection. As each channel can contain multiple enforcement objects, analogously to channel selection, PAIO selects the correct object to service the request (Fig. 3, ❹). For each request, the channel verifies the *Context*'s classifiers and maps the request to the respective enforcement object, which will then employ its I/O mechanism (§5).

Context propagation. Several I/O classifiers, such as *workflow id*, *request type*, and *size*, are accessible from observing raw I/O requests. However, application-level information, that is only accessible to the layer that submits the I/O requests, could be used to expand the policies to be enforced over the I/O stack. An example of such information, as depicted in Fig.1, is the *operation context*, which allows to determine the origin or context of a given request, *i.e.,* if it comes from a foreground or background task, flush or compaction, or other.

As such, PAIO enables the propagation of additional information from the targeted layer to the stage. It combines ideas from *context propagation*, a commonly used technique that enables a system to forward context along its execution path [36, 37, 41, 62], and applies them to ensure fine-grained control over requests. To achieve this, system designers instrument the data path of the targeted layer where the information can be accessed, and make it available to the stage through the process's address space, shared memory, or thread-local variables. The information is included at the creation of the *Context* object as the *request context* classifier. Propagating the context without this method would require changing all core modules and function signatures between where the information can be found and its submission to the stage.

As an example, consider the I/O stack of Fig. 3. To determine the origin of POSIX operations submitted by *RocksDB*'s background workflows, system designers instrument the *RocksDB*'s critical path responsible for managing flush or compaction jobs (❶) to capture their context. This information is then propagated to the *stage interface*, where the *Context* object is created with all I/O classifiers, including the *request context*, and submitted to the stage (❶).

Note that *this step is optional*, as it can be skipped for policies that do not require additional information to be enforced.

5 I/O Enforcement

The enforcement module provides the building block for developing the actual I/O mechanisms that will be employed over requests. It is composed of several channels, each containing one or more enforcement objects.

As depicted in Fig. 3, requests are moved to the selected channel and placed in a *submission queue* (❸). For each dequeued request, PAIO selects the correct enforcement object (❹) and applies its I/O mechanism (❺). Examples of these mechanisms include token-buckets, caches, encryption schemes, and more; we discuss how to build enforcement objects in §6.3 Since several mechanisms can change the original request's state, such as data transformations (*e.g.,* encryption, compression), during this phase, the enforcement object generates a *Result* that encapsulates the updated version of the request, including its content and size. The *Result* object is then returned to the stage interface, that unmarshalls it, inspects it, and routes it to the original data path (❻). After this process, PAIO ensured that the request has met the objectives of the specified policy.

Optimizations. Depending on the policies and mechanisms to be employed, PAIO can enforce requests using only their I/O classifiers. While data transformations are directly applicable over the request's content, performance-driven mechanisms such as token-buckets and schedulers, only require specific request metadata to be enforced (*e.g.,* type, size, priority, storage path). As such, to avoid adding overhead to the system execution, PAIO allows for the request's content to be copied to the stage's execution path only when necessary.

Table 2: Interface definitions of PAIO.

1†	`paio_init()`	Initialization of PAIO stage
	`enforce(ctx,r)`	Enforce context `ctx` and request `r`
2*	`obj_init(s)`	Initialize enforcement object with state `s`
	`obj_enf(ctx,r)`	Enforce I/O mechanism over `ctx` and `r`
	`obj_config(s)`	Configure enforcement object with state `s`
3*	`stage_info()`	Get data plane stage information
	`hsk_rule(t)`	Housekeeping rule with tuple `t`
	`dif_rule(t)`	Differentiation rule with tuple `t`
	`enf_rule(id,s)`	Enf. rule over enf. object `id` with state `s`
	`collect()`	Collect statistics from data plane stage

†**Stage API**; *__Enforcement object API__; *__Control API__.

6 PAIO Interfaces and Usage

We now detail how PAIO interacts with I/O layers and control planes, how to integrate PAIO in user-level layers, and how to build enforcement objects.

6.1 Interfaces

Stage interface. PAIO provides an application programming interface to establish the connection between an I/O layer and PAIO's internal mechanisms. As depicted in Table 2, it presents two functions: `paio_init` initializes a stage, which connects to the control plane for internal stage management and defining how workflows should be handled; `enforce` intercepts requests from the layer and routes them, along the associated *Context* object, to the stage (§6.2 details how requests should be intercepted and submitted to PAIO). After enforcing the request, the stage outputs the enforcement result and the layer resumes the original execution path.

Control interface. Communication between stages and the control plane is achieved through five calls, as depicted in Table 2. A `stage_info` call lists information about the stage, including the *stage identifier* and *process identifier* (PID). Rule-based calls are used for managing and tuning the data plane stage. *Housekeeping rules* (`hsk_rule`) manage the stage lifecycle (*e.g.,* create channels and enforcement objects), *differentiation rules* (`dif_rule`) map requests to channels and enforcement objects, and *enforcement rules* (`enf_rule`) dynamically adjust the internal state (`s`) of a given enforcement object (`id`) upon workload and policy variations. The control plane also monitors stages though a `collect` call, that gathers key performance metrics of all workflows (*e.g.,* IOPS, bandwidth) and can be used to tune the data plane stage.

This interface enables the control plane to define how PAIO stages handle I/O requests. Nonetheless, concerns related to the dependability of data plane stages, as well as the resolution of conflicting policies are responsibility of the control plane [38], and are thus orthogonal to this paper.

6.2 Integrating PAIO in User-level Layers

Porting I/O layers to use PAIO stages can require a few steps.

Using PAIO with context propagation. To integrate a stage within a layer, the system designer typically needs to:

1. Create the stage in the targeted layer, using `paio_init`.
2. Instrument the critical data path, where the layer-level information is accessible, and propagate it to the stage upon the *Context* object creation. This might entail creating additional data structures.
3. Create the *Context* object that will be submitted, alongside the request, to the stage. It can include the *workflow id*, *request type* and *size*, and the propagated information.
4. Add an `enforce` call to the I/O operations that need to be enforced at the stage before being submitted to the next layer. For example, to enforce the POSIX `read` operations of a given layer, all `read` calls need to be first routed to PAIO before being submitted to the file system.
5. Verify if the request was successfully enforced by inspecting the *Result* object, returned from `enforce`, and resume the execution path.

Using PAIO transparently. When context propagation is not required, PAIO stages can be used transparently between I/O layers, such as applications and file systems. PAIO exposes layer-oriented interfaces (*e.g.,* POSIX) and uses `LD_PRELOAD` to replace the original interface calls at the top layer (*e.g.,* `read` and `write` calls invoked by applications) for ones that are first submitted to PAIO before being submitted to the bottom layer (*e.g.,* file system) [7]. Each supported call defines the logic to create the *Context* object, submits the request to the stage, verifies the *Result*, and invokes the original I/O call. This enables layers to use PAIO without changing any line of code.

6.3 Building Enforcement Objects

PAIO exposes to system designers a simple API to build enforcement objects, as depicted in Table 2.

- **`obj_init`.** Create an enforcement object with initial state `s`, which includes its type and initial configuration.
- **`obj_config`.** Provides the tuning knobs to update the enforcement object's internal settings with a new state `s`. This enables the control plane to dynamically adapt it to workload variations and new policies.
- **`obj_enf`.** Implements the actual I/O logic to be applied over requests. It returns a *Result* that contains the updated version of the request (`r`), after applying its logic. It also receives a *Context* object (`ctx`) that is used to employ different actions over the I/O request.

By default, PAIO preserves the operation logic of the targeted system (*e.g.,* ordering, error handling), as both enforcement objects and operations submitted to PAIO follow a synchronous model. While developing asynchronous enforcement objects is feasible, one needs to ensure that both correctness and fault tolerance guarantees are preserved.

7 Implementation

We have implemented PAIO prototype with 9K lines of C++ code. It targets layers at the user-level, enabling the construction of new stage implementations and simple integration, requiring none or minor code changes.

Enforcement objects. We implemented two enforcement objects. `Noop` implements a pass-through mechanism that copies the request's content to the *Result* object, without additional data processing. *Dynamic rate limiter* (`DRL`) implements a token-bucket to control the rate and burstiness of I/O workflows [17]. The bucket is configured with a maximum token capacity (*size*) and period to replenish the bucket (*refill period*). The rate at which the bucket serves requests is given in *tokens/s*. On `obj_init` the bucket is created with an initial *size* and *refill period*. On `obj_config`, a `rate(r)` routine changes the *size* according to a function between `r` and *refill period*. For each request, `obj_enf` verifies the *context's size* classifier and computes the number of tokens to be consumed. If not enough tokens are available, the request waits for the bucket to be refilled. To demonstrate the portability and maintainability of PAIO's I/O mechanisms, we apply the `DRL` object over two use cases composed of different layers and objectives.

I/O cost. We consider a constant cost for requests *e.g.,* each byte of a `read` or `write` request represents a token. Although the cost depends on several factors (*e.g.,* workload, type, cache hits), we continuously calibrate the token-bucket so its rate converges to the policies' goal. Our experiments show that this approach works well in our scenarios, as the bucket's rate converges within few interactions with the control plane. Nevertheless, determining the I/O cost is complementary to our work [24,50]. Combining PAIO with these could be useful under scenarios where policies are sensitive to the I/O cost.

Statistics, communication, and differentiation. PAIO implements per-workflow statistic counters at channels to record the bandwidth of intercepted requests, number of operations, and mean throughput between collection periods. Communication between the control plane and stages is established through UNIX Domain Sockets. To create unique identifiers that map requests to channels and enforcement objects, we used a computationally cheap hashing scheme [14] (*i.e.,* MurmurHash3) that hashes classifiers into a fixed-size token.

Context propagation. To propagate information from layers, we implemented a shared map, indexed by the *workflow identifier* (*e.g.,* thread-id), that stores the *context* of the requests being submitted, which is similar to those used in [36, 37].

Transparently intercepting I/O calls. PAIO uses `LD_PRELOAD` to intercept POSIX calls and route them either to the stage or to the kernel. It supports `read` and `write` calls, including different variations (*e.g.,* `pread`, `pwrite64`). We found that supporting this set of calls is sufficient to enforce data-oriented policies, as presented in §8.2. We defer the support of other calls and interfaces (*e.g.,* KVS, object store) to future work.

Control plane. We built a simple but fully-functional control plane with 3.6K lines of C++ code that enforces policies for the two use cases of this paper (§8). Policies were implemented as control algorithms. To calibrate enforcement objects, besides stage statistics, it collects I/O metrics generated by the targeted layer from the `/proc` file system [44]. Specifically, it inspects the `read_bytes` and `write_bytes` I/O counters, which represent the number of bytes read/written from/to the block layer, and compares them with the stage statistics to converge to the targeted performance goal.

8 Use Cases and Control Algorithms

We now present two use cases that showcase the applicability of PAIO for different applications and performance goals.

8.1 Tail Latency Control in Key-Value Stores

LSM KVSs [34] (*e.g.,* RocksDB) use *foreground flows* to attend client requests, which are enqueued and served in FIFO order. *Background flows* serve internal operations, namely flushes and compactions. Flushes are sequentially written to the first level of the tree (L_0) and only proceed when there is enough space. Compactions are held in a FIFO queue, waiting to be executed by a dedicated thread pool. Except for low level compactions ($L_0 \rightarrow L_1$), these can be made in parallel. A common problem of these however, is the interference between I/O workflows, generating latency spikes for client requests. Latency spikes occur when flushes cannot proceed because $L_0 \rightarrow L_1$ compactions and flushes are slow or on hold [16].

SILK. SILK [16], a RocksDB-based KVS, prevents this through an I/O scheduler that: allocates bandwidth for internal operations when client load is low; prioritizes flushes and low level compactions, as they impact client latency; and preempts high level compactions with low level ones. It employs these techniques through the following control algorithm. As these KVSs are embedded, the KVS I/O bandwidth is bounded to a given rate (KVS_B). It monitors clients' bandwidth (Fg), and allocates leftover bandwidth ($left_B$) to internal operations (I_B), given by $I_B = KVS_B - Fg$. To enforce rate I_B, SILK uses RocksDB's rate limiters [4]. Flushes and $L_0 \rightarrow L_1$ compactions have high priority and are provisioned with minimum I/O bandwidth (min_B). High level compactions have low priority and can be paused at any time. Because all compactions share the same thread pool, it is possible that, at some point, all threads are handling high level compactions. As such, SILK preempts one of them to execute low level compactions.

Applying these optimizations however, required reorganizing RocksDB's internal operation flow, changing core modules made of thousands of LoC including *background operation handlers*, *internal queuing logic*, and *thread pools allocated for internal work* [15]. Further, porting these optimizations to other KVS that would equally benefit from them, such as LevelDB [21] and PebblesDB [47], requires deep system knowledge and substantial re-implementation efforts.

Algorithm 1 Tail Latency Control Algorithm

Initialize: $KVS_B = 200$; $min_B = 10$
1: $\{Fg, Fl, L_0, L_N\} \leftarrow collect\ ()$
2: $left_B \leftarrow KVS_B - Fg$
3: $left_B \leftarrow max\ \{left_B \mid min_B\}$
4: **if** $Fl > 0 \wedge L_0 > 0$ **then**
5: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B/2,\ left_B/2,\ min_B\}$
6: **else if** $Fl > 0 \wedge L_0 = 0$ **then**
7: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B,\ min_B,\ min_B\}$
8: **else if** $Fl = 0 \wedge L_0 > 0$ **then**
9: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ left_B,\ min_B\}$
10: **else**
11: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ min_B,\ left_B\}$
12: *enf_rule* $(\{B_{Fl},\ B_{L_0},\ B_{L_N}\})$
13: *sleep (loop_interval)*

PAIO. Rather than modifying the RocksDB engine, we found that several of these optimizations could be achieved by orchestrating the I/O workflows. Thus, we applied SILK's design principles as follows: a PAIO *data plane stage provides the I/O mechanisms for prioritizing and rate limiting background flows*, while the *control plane re-implements the I/O scheduling algorithm* to orchestrate the stage.

The stage intercepts all RocksDB workflows. We consider each RocksDB thread that interacts with the file system as a workflow. Channel differentiation is made using the *workflow id*. We instrumented RocksDB to propagate the *context* at which a given operation is created, namely flush (`flush`) or compaction (*e.g.,* `compaction_L0_L1`). Foreground flows are monitored for collecting clients' bandwidth (Fg). Background flows are routed to channels made of `DRL` objects. Flushes flow through a dedicated channel. As compactions with different priorities can flow through the same channel, each channel contains two `DRL` objects configured at different rates. The enforcement object differentiation is made through the *request context* classifier, and requests are enforced with the optimization described in §5. PAIO also collects the bandwidth of flushes (Fl), and low (L_0) and high level compactions (L_N).

The control plane implements the control portion of SILK's scheduling algorithm (Alg. 1). It uses a feedback control loop that performs the following steps. First, it collects statistics from the stage (`1`) and computes leftover disk bandwidth ($left_B$) to assign to internal operations (`2`). To ensure that background operations keep flowing, it defines a minimum bandwidth threshold (`3`), and distributes $left_B$ according to workflow priorities (`4-11`). If high priority tasks are executing it assigns them an equal share of $left_B$, while ensuring that high level compactions keep flowing (min_B), preventing low level ones from being blocked in the queue (`5`). If a single high priority task is being executed, $left_B$ is allocated to it and min_B to others (`6-9`). If no high priority task is executing, it reserves $left_B$ to low priority ones (`11`). It then generates and submits `enf_rule`s to adjust the rate of each enforcement object (`12`). For low priority compactions, it splits B_{L_N} between all `DRL`

Table 3: *Lines of code added to RocksDB and TensorFlow.*

	Lines added	
	RocksDB	TensorFlow (`LD_PRELOAD`)
Targeted code base size	≈335K [5]	≈2.3M [6]
Initialize PAIO stage	10	—
Context propagation	47	—
Create *Context* object	7	—
Instrument I/O calls	17	—
Verify *Result* object	4	—
Total	**85**	**0**

objects that handle these. Since high priority compactions are executed sequentially [9, 16], it assigns B_{L_0} to the respective objects. Rate B_{Fl} is assigned to those responsible for flushes.

Integration with RocksDB. Integrating PAIO in RocksDB only required adding 85 LoC (Table 3). Specifically:

1. Initialize PAIO stage and create additional structures to identify the task that each workflow is executing (10 LoC).
2. Instrument RocksDB's internal thread pools for identifying the workflows that run flush and compaction jobs (17 LoC). To differentiate high priority compactions from low priority ones, we instrumented the code where compaction jobs are created. For each job, we verify its level and update the structure with the task that the workflow will be executing (*e.g.,* `compaction_L0_L1`) (30 LoC).
3. Create a *Context* object with *workflow id*, *request type*, *context*, and *size* I/O classifiers (7 LoC).
4. Submit all `read` and `write` calls to the stage (17 LoC).
5. Verify the *Result* of the enforcement (4 LoC).

8.2 Per-Application Bandwidth Control

The ABCI supercomputer is designed upon the convergence between AI and HPC workloads. One of the most used AI frameworks on it is TensorFlow [11]. To execute TensorFlow jobs users can reserve a full node or a fraction of it (*i.e.,* jobs execute concurrently). Nodes are partitioned into resource-isolated *instances* through Linux's cgroups [39]. Each instance has exclusive access to CPU cores, memory space, a GPU, and local storage quota. However, the local disk bandwidth is still shared, and because each instance is agnostic of others, jobs compete for bandwidth leading to I/O interference and performance variation. Even if the block I/O scheduler is fair, all instances are provisioned with the same service level, preventing the assignment of different priorities.

Using cgroups's block I/O controller (*blkio*) allows static rate limiting `read` and `write` operations of each instance [2]. However, under ABCI, once the rate is set it cannot be dynamically changed at execution time, as it requires stopping the jobs, adjust the rate of all groups, and restart the jobs, being prohibitively expensive in terms of overall execution time. This creates a second problem where if no other job is executing in the node, the instance cannot use leftover bandwidth.

PAIO. To address this, we use a PAIO stage that implements the mechanisms to dynamically rate limit workflows at each instance, while the control plane implements a proportional sharing algorithm to ensure all instances meet their policies.

Algorithm 2 Max-min Fair Share Control Algorithm

Initialize: $Max_B = 1GiB$; $Active > 0$; $demand_i > 0$
1: $\{I_1, I_2, I_3, I_4\} \leftarrow collect\ ()$
2: $left_B \leftarrow Max_B$
3: **for** $i = 0$ in $[0, Active-1]$ **do**
4: **if** $demand_i \leq \frac{left_B}{Active-i}$ **then**
5: $rate_i \leftarrow demand_i$
6: **else**
7: $rate_i \leftarrow \frac{left_B}{Active-i}$
8: $left_B \leftarrow left_B - rate_i$
9: **for** $i = 0$ in $[0, Active-1]$ **do**
10: $rate_i \leftarrow \frac{left_B}{Active}$
11: $enf\_rule\ (\{rate_1, I_1\}, \{rate_2, I_2\}, \{rate_3, I_3\}, \{rate_4, I_4\})$
12: $sleep\ (loop\_interval)$

Our use case focuses on the model training phase, where each instance runs a TensorFlow job that uses a single workflow to read dataset files from the file system. TensorFlow's `read` requests are intercepted and routed to the stage, which contains a channel with a `DRL` enforcement object. Requests are enforced with the optimization described in §5.

The control plane implements a max-min fair share algorithm to ensure per-application bandwidth guarantees (Alg. 2), which is typically used for resource fairness policies [35, 54]. The overall disk bandwidth available (Max_B) and bandwidth demand of each application (*demand*) are defined *a priori* by the system administrator or the mechanism responsible for managing resources of different job instances [63]. The algorithm uses a feedback control loop that performs the following steps. First, the control plane collects statistics from each active instance's stage, given by I_i (`1`), as well as the bandwidth generated by each TensorFlow job (collected at `/proc`). Then, it computes the rate of each active instance (`3-10`). If an instance's *demand* is less than its fair share, the control plane assigns its *demand* (`4-5`), assigning the fair share otherwise (`7`). It then distributes leftover bandwidth ($left_B$) across instances (`9-10`). Then, it calibrates the rate of each instance in a function of I_i and $rate_i$, generating the `enf_rules` to be submitted to each stage (`11`). Finally, the control plane sleeps for *loop_interval* before beginning a new control cycle (`12`).

Integration with TensorFlow. Integrating TensorFlow with PAIO did not required any code changes (Table 3). We used `LD_PRELOAD` to intercept, and route to PAIO, TensorFlow's `read` and `write` calls. All supported calls implement the logic necessary for the request to be enforced, including the creation of the *Context* object using the *request type* and *size* classifiers; stage enforcement; verification of the enforcement *Result*; and its submission to the original execution path (file system).

9 Evaluation

Our evaluation seeks to demonstrate the performance of PAIO, and its ability and feasibility of enforcing policies over different scenarios. The results show that:

- Its performance scales with the number of channels, achieving high throughput and low latency (§9.1).
- It can be used to enforce policies over different I/O layers with distinct requirements (§9.2 and §9.3).
- By propagating application-level information to the data plane stage, PAIO outperforms RocksDB by at most 4× in tail latency, while enabling similar control and performance as system-specific optimizations (SILK) (§9.2).
- When internal system knowledge is not required, PAIO can enforce policies without application changes. By having global visibility, it provisions per-application bandwidth guarantees at all times, and improves overall execution time when compared to a static rate limiting approach (§9.3).

Experimental setting. Experiments were conducted under two hardware configurations. ***A***: a compute node of the ABCI supercomputer with two 20-core Intel Xeon processors (80 cores), 4 NVidia Tesla V100 GPUs, 384GiB of RAM, and a 1.6TiB Intel SSD DC P4600, running CentOS 7.5 with Linux kernel 3.10 and the `xfs` file system. ***B***: a server with two 18-core Intel Xeon processors (72 cores), 192GiB of RAM, a 1.6TiB Dell Express Flash PM1725b SSD (NVMe) and a 480GiB Intel D3-s4610 SATA SSD, running Ubuntu Server 20.04 LTS with kernel 5.8.9 and the `ext4` file system.

9.1 PAIO Performance and Scalability

We developed a benchmark that simulates an application that submits requests to a PAIO stage. This benchmark aims to demonstrate the maximum performance achievable with PAIO by stress-testing it in a loop-back manner. It generates and submits multi-threaded requests in a closed loop through the *Instance*'s `enforce` call, under a varying number of clients (*e.g.,* workflows) and request sizes. Request size and number of client threads range between 0 – 128KiB and 1 – 128, respectively. Each client thread submits 100M requests. A PAIO stage is configured with varying number of channels (matching the number of client threads), each containing a `Noop` enforcement object that copies the request's buffer to the *result* object. All reported results are the mean of at least ten runs and standard deviation is kept below 5%.

IOPS and bandwidth. Fig. 4 depicts the cumulative IOPS ratio with respect to a single channel. 0B represents a *context*-only request, as described in §5. Results marked with ∗ and + were conducted under configurations *A* and *B*, respectively.

For configuration *A*, under a 0B* request size, a single PAIO channel achieves a mean throughput of 3.05 MOps/s and a 327 ns latency. Since the workload is CPU-bound, the performance does not scale linearly, as client threads compete for processing time. Under 128 channels, it achieves a cumulative throughput of 97.4 MOps/s, a 31× performance increase. As the request size increases so does the total bytes processed by PAIO. When configured with 128 channels, it processes 128KiB* requests at 384 GiB/s. For a single channel, PAIO processes requests at 2.1 GiB/s and 11.7 GiB/s

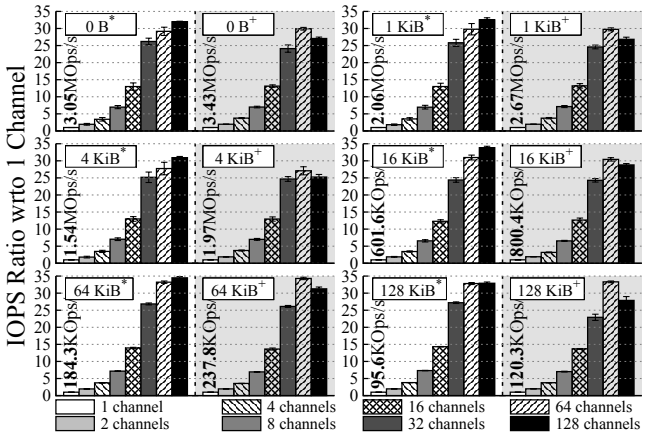

Figure 4: ***Cumulative IOPS*** *of* PAIO *under varying number of channels (1 – 128) and request sizes (0 – 128 KiB). Absolute IOPS value is shown above the 1 channel bar.*

for 1KiB* and 128KiB* request sizes. For configuration *B*, PAIO achieves higher throughput results as it operates under a later kernel version. Since the machine is configured with 72 cores, PAIO's performance peaks at 64 client threads. Under a 0B+ request size, PAIO achieves 3.43 MOps/s (1 channel) and 102.7 MOps/s (64 channels), representing a 30× performance increase. When configured with 64 channels, it is able to process 128KiB+-sized requests at 489 GiB/s. For a single channel, PAIO processes requests at 2.5 GiB/s and 14.7 GiB/s for 1KiB* and 128KiB* request sizes, respectively.

Profiling. We measured the execution time of each PAIO operation that appears in the main execution path. Depending on the hardware configuration, *Context* object creation takes between 17 – 19 ns, while the channel and enforcement object selection take 85 – 89 ns to complete (each). The duration of `obj_enf` ranges between 20 ns and 8.45 μs when configured with 0B and 128KiB request sizes.

Summary. Results show that PAIO has low overhead, as it is provided as a user-space library, which does not require costly context-switching operations. We expect that the main source of overhead will always be dependent on the type of enforcement object applied over requests. For the enforcement object used in the use cases of this paper (§9.2 – §9.3), we have not observed significant performance degradation.

9.2 Tail Latency Control in Key-Value Stores

We now demonstrate how PAIO achieves tail latency control under several workloads. We compare the performance of RocksDB [5]; Auto-tuned, a version of RocksDB with auto-tuned rate limiting of background operations enabled [28]; SILK [16]; and PAIO, *i.e.,* a PAIO-enabled RocksDB.

System configuration. Experiments were conducted under hardware configuration *B* using the available NVMe device (unless stated otherwise). All systems are tuned as follows. The `memtable-size` is set to 128MiB. We use 8 threads for client operations and 8 background threads for flush (1) and

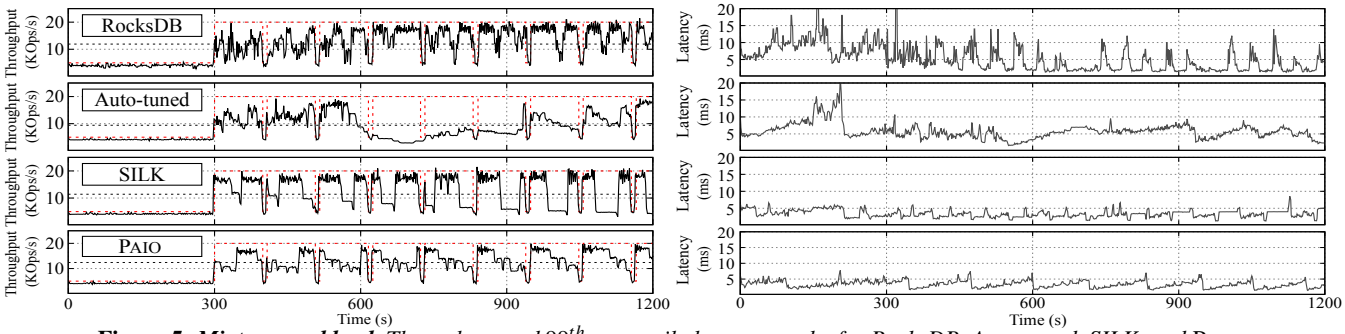

Figure 5: ***Mixture workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

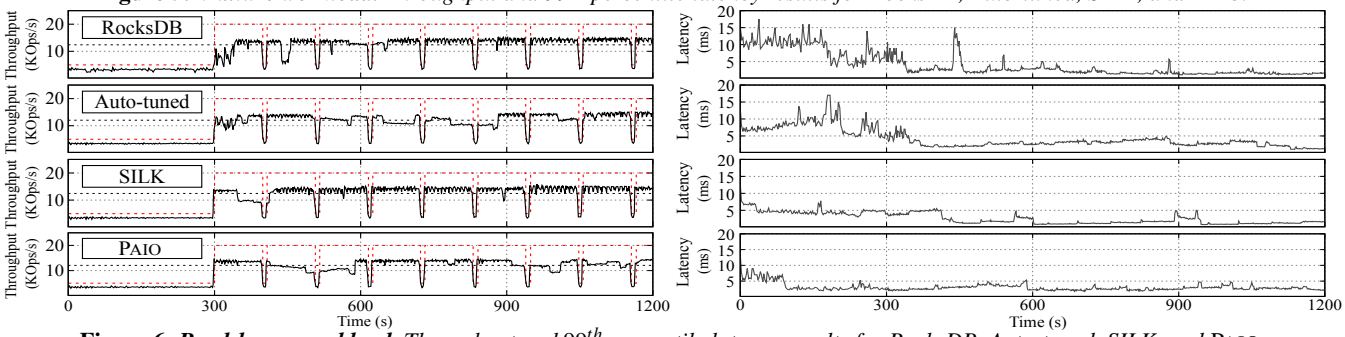

Figure 6: ***Read-heavy workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

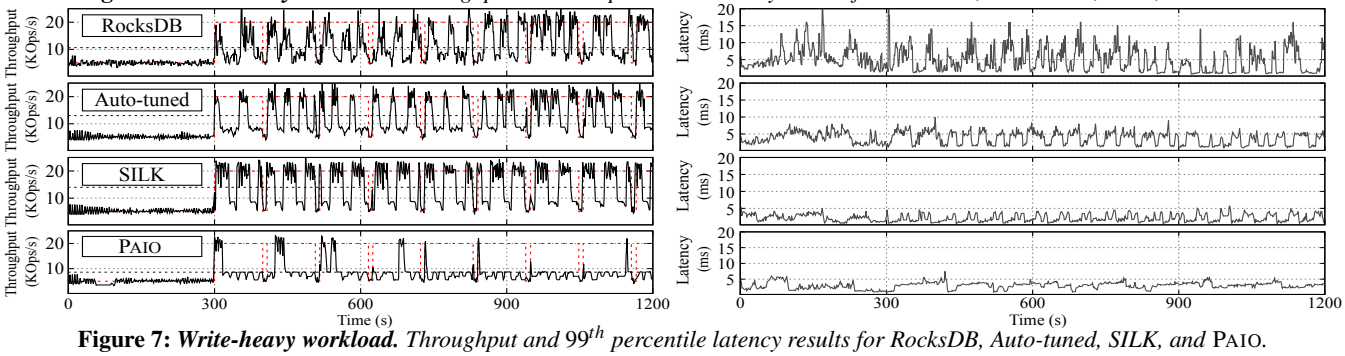

Figure 7: ***Write-heavy workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

compactions (7). The minimum bandwidth threshold for internal operations is set to 10MiB/s. To simplify results compression and commit logging are turned off. All experiments are conducted using the `db_bench` benchmark [3]. As used in the SILK testbed [16], we limit memory usage to 1GiB and I/O bandwidth to 200MiB/s (unless stated otherwise).

Workloads. We focus on workloads made of bursty clients, to better simulate existing services in production [16, 18]. Client requests are issued in a closed loop through a combination of peaks and valleys. An initial valley of 300 seconds submits operations at 5kops/s, and is used for executing the KVS internal backlog. Peaks are issued at a rate of 20kops/s for 100 seconds, followed by 10 seconds valleys at 5kops/s. All datastores were preloaded with 100M key-value pairs, using a uniform key-distribution, 8B keys and 1024B values.

We use three workloads with different read:write ratios: *mixture* (50:50), *read-heavy* (90:10), and *write-heavy* (10:90). *Mixture* represents a commonly used YCSB workload (workload A) and provides a similar ratio as Nutanix production workloads [16]. *Read-heavy* provides an operation ratio similar to those reported at Facebook [18]. To present a comprehensive testbed, we include a *write-heavy* workload. For each system, workloads were executed three times over 1-hour with uniform key distribution. For figure clarity, we present the first 20 minutes of a single run. Similar performance curves were observed for the rest of the execution. Fig. 5–9 depict throughput and 99th percentile latency of all systems and workloads. Theoretical client load is presented as a red dashed line. Mean throughput is shown as an horizontal dashed line.

Mixture workload (Fig. 5). Due to accumulated backlog of the loading phase, the throughput achieved in all systems does not match the theoretical client load. RocksDB presents high tail latency spikes due to constant flushes and low level compactions. Auto-tuned presents less latency spikes but degrades overall throughput. This is due to the rate limiter being agnostic of background tasks' priority, and because it increases its rate when there is more backlog, contending for disk bandwidth. SILK achieves low tail latency but suffers periodic drops in throughput due to accumulated backlog. Compared to RocksDB (11.9 kops/s), PAIO provides simi-

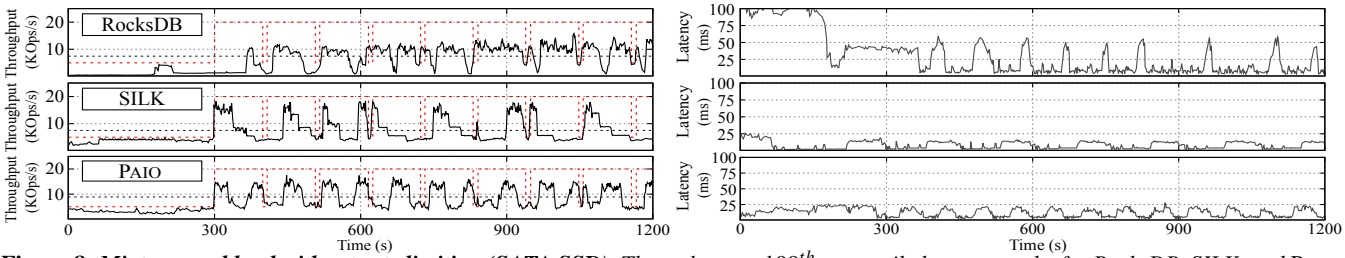

Figure 8: ***Mixture workload without rate limiting (SATA SSD).*** *Throughput and 99th percentile latency results for RocksDB, SILK, and* PAIO.

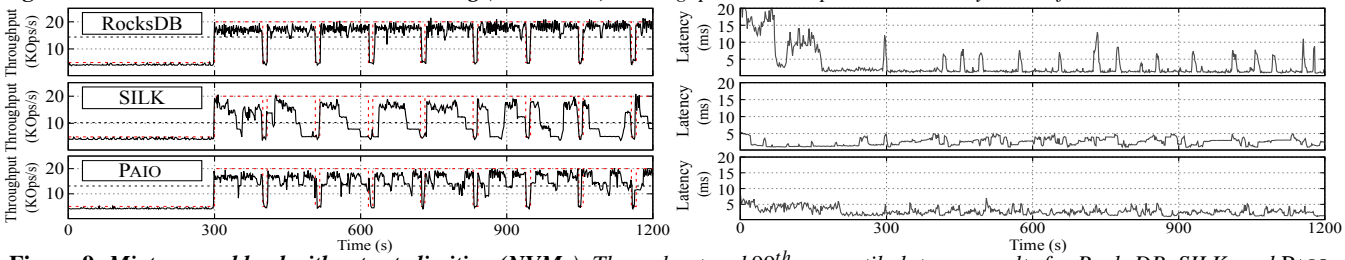

Figure 9: ***Mixture workload without rate limiting (NVMe).*** *Throughput and 99th percentile latency results for RocksDB, SILK, and* PAIO.

lar mean throughput (12.4 kops/s). As for tail latency, while RocksDB experiences peaks that range between 3–20 ms, PAIO and SILK observe a 4× decrease in absolute tail latency, with values ranging between 2–6 ms.

Read-heavy workload (Fig. 6). Throughput-wise all systems perform identically. At different periods, all systems demonstrate a temporary throughput degradation due to accumulated backlog. As for tail latency, the analysis is twofold. RocksDB and Auto-tuned present high tail latency up to the 400 s mark. After that mark, RocksDB does not have more pending backlog and achieves sustained tail latency (1–3 ms), while on Auto-tuned, some compactions are still being performed due to rate limiting, increasing latency by 1–2 ms. SILK and PAIO have similar latency curves. During the initial valley both systems significantly improve tail latency when compared to RocksDB. After the 400 s mark, SILK pauses high level compactions and achieves a tail latency between 1–2 ms. By preempting high level compactions and serving low level ones through the same thread pool as flushes, it ensures that high priority tasks are rarely stalled. SILK achieves this by modifying the RocksDB's queuing mechanism. In PAIO, while sustained, its tail latency is 1 ms higher than SILK's in the same observation period. Since PAIO does not modify the RocksDB engine, it cannot preempt compactions (§8.1).

Write-heavy workload (Fig. 7). Write-intensive workloads generate a large backlog of background tasks, leading RocksDB to experience high latency spikes. Auto-tuned limits all background writes, reducing latency spikes but still exceeding the 5 ms mark over several periods. SILK pauses high level compactions and only serves high priority tasks, improving mean throughput and keeping latency spikes below 5 ms. In PAIO, since flushes occur more frequently, the control plane slows down high level compactions more aggressively, which leads to low level ones to be temporary halted at the compaction queue, waiting to be executed. Even though mean throughput is decreased, PAIO significantly reduces tail latency, never exceeding 6 ms. The throughput difference between PAIO and SILK is justified by the latter preempting high level compactions, as described in the read-heavy workload.

Mixture workload without rate limiting (Fig. 8–9). We conducted an additional set of experiments to assess the impact of the tail latency control algorithm under a scenario where the KVS has access to the full storage device bandwidth. We compared the performance of RocksDB, SILK, and PAIO under both SSD and NVMe devices, without rate limiting, using the *mixture* workload. The KVS_B parameter was set with a value closer to the device's limit. For Auto-tuned, we report similar conclusions to those presented in Fig. 5.

Fig. 8 depicts the results under the SSD device. Due to accumulated backlog all systems experience poor throughput performance, averaging at 7.46 kops/s (RocksDB), 7.52 kops/s (SILK), and 8.88 kops/s (PAIO). During the loading phase, and until finishing the accumulated backlog (0–400 s), RocksDB experiences long periods of high tail latency, peaking at 111 ms. After that, it observes latency spikes due to constant flushes and low level compactions, with values ranging between 15–60 ms. SILK and PAIO present a more sustained latency performance, never exceeding the 25 ms mark throughout the overall observation period. Specifically, while RocksDB experienced a variability of 21 ms, SILK and PAIO achieved 4.7 ms and 5.8 ms, respectively.[2] Throughput-wise, both systems observe periodic drops due to accumulated backlog. However, PAIO is able to recover faster than SILK. Because it cannot preempt compactions, PAIO reserves more bandwidth (than SILK) to low priority compactions, ensuring that high priority tasks do not wait to be executed. As such, PAIO follows a proactive approach for assigning bandwidth to compactions, while SILK follows a reactive approach.

[2]The variability results correspond to the average of the absolute deviations of data points (*i.e.,* each tail latency measurement) from their mean.

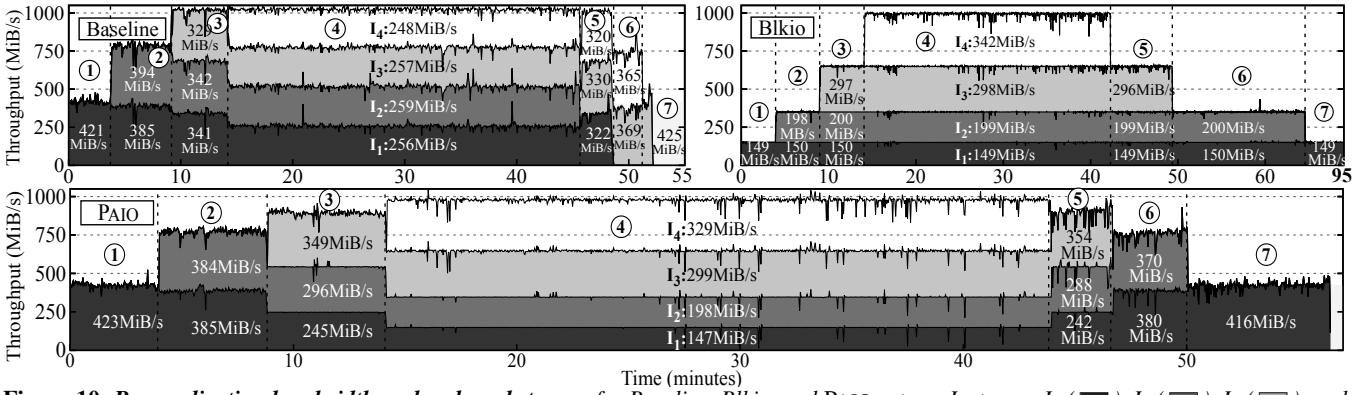

Figure 10: ***Per-application bandwidth under shared storage*** *for Baseline, Blkio, and* PAIO *setups. Instances I_1 (■), I_2 (■), I_3 (■), and I_4 (□) are assigned with minimum bandwidth of 150, 200, 300, and 350 MiB/s, and execute 6, 5, 5, and 4 training epochs, respectively.*

Fig. 9 depicts the results under the NVMe device. All systems experienced higher throughput performance, averaging at 14.39 kops/s (RocksDB), 10.27 kops/s (SILK), and 13.11 kops/s (PAIO). RocksDB follows a similar performance curve as the theoretical client load. The reason behind this is twofold. First, it completes all accumulated backlog during the initial valley (at the cost of high tail latency), which positively reflects in the remainder execution (*i.e,* no significant performance loss is observed). Second, since NVMe devices have higher throughput performance and parallelism than SSD devices (Fig. 8), RocksDB achieves a more sustained performance. After the initial valley, RocksDB observes latency spikes that range between 7–15 ms due to frequent flushes and low level compactions. SILK and PAIO follow similar tail latency curves, never exceeding the 6 ms mark. In detail, throughout the overall observation period, RocksDB observed a variability of 2.5 ms, while SILK and PAIO only observed a variability of 0.8 ms. Similarly to previous results, both system experience periodic throughput drops.

Summary. We demonstrate that through minor code changes, PAIO outperforms RocksDB by at most 4× in tail latency and enables similar control and performance as SILK, which required profound refactoring to the original code base.

9.3 Per-Application Bandwidth Control

We now demonstrate how PAIO ensures per-application bandwidth guarantees under a shared storage scenario. Our setup was driven by the requirements of the ABCI supercomputer.

System configuration. Experiments ran under hardware configuration *A* using TensorFlow 2.1.0 with the LeNet [30] training model, configured with a batch size of 64 TFRecords. We used the ImageNet dataset (≈150GiB) [48]. Each instance runs with a dedicated GPU and dataset, and its memory is limited to 32GiB. Overall disk bandwidth is limited to 1GiB/s. At all times, a node executes at most four instances with equal resource shares in terms of CPU, GPU, and RAM. Each instance executes a TensorFlow job, is assigned with a bandwidth policy, and executes a given number of training epochs. Namely, instances **1** to **4** are assigned with minimum bandwidth guarantees of **150**, **200**, **300**, and **350** MiB/s, and execute **6**, **5**, **5**, and **4** training epochs, respectively.

Setups. Experiments were conducted under three setups. *Baseline* represents the current setup supported at the ABCI supercomputer; all instances execute without bandwidth guarantees. *Blkio* enforces bandwidth limits using blkio [2]. In PAIO, each instance executes with a PAIO stage that enforces the specified bandwidth goals dynamically. Fig. 10 depicts, for each setup, the I/O bandwidth of all instances at 1-second intervals. Experiments include seven phases, each marking when an instance starts or completes its execution.

Baseline. Experiments were executed over 52 minutes. At ①, I_1 reads at 421 MiB/s. Whenever a new instance is added, the I/O bandwidth is shared evenly (②). At ③, the aggregated instance throughput matches the disk limit. At ④, instance performance converges to ≈256 MiB/s, leading to all instances experiencing the same service level. However, I_3 and I_4 cannot meet their goal, since I_1 and I_2 have more than their fair share. After 46 minutes of execution (⑤), I_3 terminates, and leftover bandwidth is shared with the remainder. Again, I_4 cannot achieve its targeted goal. At ⑥ and ⑦, active instances have access to leftover bandwidth and finish their execution.

Summary: I_3 and I_4 were unable to achieve their bandwidth guarantees, missing their objectives during 31 and 34 minutes.

Blkio. Experiments were executed over 95 minutes. From ① to ⑦, whenever a new instance is added, it is provisioned with its exact bandwidth limit. Because the rate of each instance is set using blkio, instances cannot use leftover bandwidth to speed up their execution. For example, while on *Baseline* I_1 executes under the 50-minutes mark, it takes 95 minutes to complete its execution in *Blkio*. To overcome this, a possible solution would require to stop and checkpoint the instance's execution, reconfigure blkio with a new rate, and resume from the latest checkpoint. However, doing this process every time a new instance joins or leaves the compute node would significantly delay the execution time of all running instances.

Summary: All instances achieve their bandwidth guarantees

but cannot be dynamically provisioned with available disk bandwidth, leading to longer periods of execution.

PAIO. Experiments were executed over 56 minutes. At ① and ②, instances are assigned with their proportional share, as the control plane first meets each instance demands and then distributes leftover bandwidth proportionally. At ③, contrary to *Baseline*, the control plane bounds the bandwidth of I_1 and I_2 to a mean throughput of 245 MiB/s and 296 MiB/s, respectively. At ④, instances are set with their bandwidth limit. During this phase, PAIO provides the same properties as blkio. From ⑤ to ⑦, as instances end their execution, active ones are provisioned as in ① to ③.

Summary: We show that PAIO can enforce per-application bandwidth guarantees without any code changes to applications. Contrary to *Baseline*, PAIO ensures that policies are met at all times, and whenever leftover bandwidth is available, PAIO shares it across active instances. Compared to *Blkio*, PAIO finishes 39, 15, and 3 minutes faster for I_1, I_2, and I_3.

10 Related Work

SDS systems. PAIO builds on a large body of work on SDS systems. IOFlow [54], sRoute [52], and PSLO [31] target the virtualization layer (*i.e.,* hypervisor, storage and network drivers) to enforce QoS policies. PriorityMeister [65] enforces rate limiting services at the Network File System. Mesnier *et al.* [41] employ caching optimizations at the block layer. Pisces [51] and Libra [50] enforce bandwidth guarantees in multi-tenant KVS. Malacology [49] improves the programmability of Ceph to build custom applications on top of it. Retro [35] and Cake [58] implement resource management services at the Hadoop stack. SafeFS [45] stacks FUSE-based file systems on top of each other, each providing a different service. Crystal [22] extends OpenStack Swift to implement custom services to be enforced over object requests.

All systems are targeted for specific I/O layers, as their design is *tightly coupled to* and *driven by* the architecture and specificities of the software stacks they are applied to. In contrast, PAIO is disaggregated from a specific software stack, enabling developers to build custom-made data plane stages applicable over different user-level layers, while requiring none to minor code changes — we demonstrate this by integrating PAIO over two different I/O layers (§8). Previous works are also unable to enforce the policies demonstrated in §8.1, as they do not provide context propagation [50, 51], inhibiting differentiating requests at a finer granularity (*i.e.,* foreground *vs* high-priority *vs* low-priority background tasks); or actuate at the kernel-level [41, 54], where the *context* is unreachable without significantly changing legacy APIs. Further, these are also unfit to achieve the policies demonstrated in §8.2, as solutions like [31, 52, 54] cannot be used under scenarios that require bare-metal access to resources, such as HPC infrastructures and bare-metal cloud servers.

Context propagation. Some works use context propagation techniques to tag data across kernel layers. Mesnier *et al.* [41] classifies and tags requests with classes to be differentiated at the block layer. IOFlow [54] tags requests to differentiate tenants that share the same hypervisor. Split-level scheduling [62] identifies the processes that caused a given I/O operation throughout the VFS, page cache, and block layer.

PAIO acts at the user-level and enables the propagation of additional information from the targeted I/O layer to the stage (*e.g.,* propagate the *context* at which a given request was created, as in §8.1), allowing more fine-grained differentiation and control over requests. Enabling the intended granularity by PAIO at kernel-level approaches would require breaking standard user-to-kernel and kernel-internal interfaces, reducing portability and compatibility [13]. Our contributions are also applicable under kernel-bypass storage stacks (*e.g.,* SPDK, PMDK), which is not the case for previous work.

Storage QoS. Many works ensure QoS SLOs at specific storage layers, including the block layer [2, 25, 33, 40, 57, 64], hypervisor [23, 24, 26, 31, 54], and distributed storage [46, 58–60]. These works are targeted for a specific I/O layer and storage objective. In contrast, PAIO is more general, providing a framework for building custom data plane stages applicable over different layers. Also, most of these solutions only differentiate requests based on their type. PAIO provides differentiation at workflow, request type, and request context. Approaches like [26, 33, 40, 64] follow a decoupled design that separates the QoS algorithm from the mechanism that applies it. While complementary to our work, these could be incorporated into our framework as new enforcement objects.

11 Conclusion

We have presented PAIO, a framework that enables system designers to build custom-made SDS data plane stages applicable over different I/O layers. PAIO 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, while enabling similar I/O control and performance, and requiring minor to none code changes.

Acknowledgments

We thank our shepherd, Sudarsun Kannan, and the anonymous reviewers for their insightful comments and feedback. We thank AIST for providing access to computational resources of ABCI. We thank Oana Balmau for discussions about SILK, and Vitor Enes and Cláudia Brito for their valuable input. This work was supported by the Portuguese Foundation for Science and Technology and the European Regional Development Fund, through the PhD Fellowship SFRH/BD/146059/2019 and projects POCI-01-0247-FEDER-045924 and UTA-EXPL/CA/0075/2019.

References

[1] AI Bridging Cloud Infrastructure. https://abci.ai/.

[2] BLKIO: Cgroup's Block I/O Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[3] facebook/rocksdb: Benchmarking tools (db_bench). https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[4] facebook/rocksdb: Rate Limiter. https://github.com/facebook/rocksdb/wiki/Rate-Limiter.

[5] facebook/rocksdb: RocksDB v5.17.2. https://github.com/facebook/rocksdb/tree/v5.17.2.

[6] google/tensorflow: TensorFlow. https://github.com/tensorflow/tensorflow/tree/v2.1.0.

[7] ld.so: LD_PRELOAD. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[8] Persistent Memory Development Kit. https://pmem.io/pmdk/.

[9] RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

[10] Storage Performance Development Kit. https://spdk.io/.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX, 2016.

[12] Ian Ackerman and Saurabh Kataria. Homepage feed multi-task learning using TensorFlow. https://engineering.linkedin.com/blog/2021/homepage-feed-multi-task-learning-using-tensorflow.

[13] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *15th Workshop on Hot Topics in Operating Systems*. USENIX, 2015.

[14] Austin Appleby. appleby/smhasher: SMHasher test suite for MurmurHash family of hash functions. https://github.com/aappleby/smhasher, 2010.

[15] Oana Balmau. theoanab/SILK-USENIXATC2019: Prototype of the SILK key-value store. https://github.com/theoanab/SILK-USENIXATC2019, 2019.

[16] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference*, pages 753–766. USENIX, 2019.

[17] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer Science & Business Media, 2001.

[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*, pages 209–223. USENIX, 2020.

[19] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime Data Processing at Facebook. In *2016 International Conference on Management of Data*, page 1087–1098. ACM, 2016.

[20] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference*, pages 49–63. USENIX, 2020.

[21] Sanjay Ghemawat and Jeff Dean. google/leveldb: LevelDB, A Fast Key-Value Storage Library. https://github.com/google/leveldb.

[22] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *15th USENIX Conference on File and Storage Technologies*, pages 243–256. USENIX, 2017.

[23] Ajay Gulati, Irfan Ahmad, and Carl A Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *7th USENIX Conference on File and Storage Technologies*, pages 85–98. USENIX, 2009.

[24] Ajay Gulati, Arif Merchant, and Peter Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 437–450. USENIX, 2010.

[25] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, page 13–24. ACM, 2007.

[26] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. Demand Based Hierarchical QoS Using Storage Resource Pools. In *2012 USENIX Annual Technical Conference*. USENIX, 2012.

[27] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies*, pages 345–358. USENIX, 2017.

[28] Andrew Kryczka. RocksDB Blog: Auto-tuned Rate Limiter. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[29] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies*. USENIX, 2020.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[31] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *11th European Conference on Computer Systems*. ACM, 2016.

[32] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. Elastic Machine Learning Algorithms in Amazon SageMaker. In *2020 ACM SIGMOD International Conference on Management of Data*, page 731–737. ACM, 2020.

[33] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *2nd USENIX Conference on File and Storage Technologies*. USENIX, 2003.

[34] Chen Luo and Michael J Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[35] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 589–603. USENIX, 2015.

[36] Jonathan Mace and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *13th European Conference on Computer Systems*. ACM, 2018.

[37] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Transactions on Computer Systems*, 35(4), December 2018.

[38] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. A Survey and Classification of Software-Defined Storage Systems. *ACM Computing Surveys*, 53(3), 2020.

[39] Paul Menage. Linux Control Groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[40] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *8th ACM International Conference on Autonomic Computing*, page 245–254. ACM, 2011.

[41] Michael Mesnier, Feng Chen, Tian Luo, and Jason Akers. Differentiated Storage Services. In *23rd ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[42] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies*, pages 65–79. USENIX, 2021.

[43] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[44] Linux Man Page. proc - Process Information Pseudo-File Sytem, 1994.

[45] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All. In *10th ACM International Systems and Storage Conference*. ACM, 2017.

[46] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, et al. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 6:1–6:12. ACM, 2017.

[47] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *26th ACM Symposium on Operating Systems Principles*, page 497–514. ACM, 2017.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.

[49] Michael Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *12th European Conference on Computer Systems*, page 175–190. ACM, 2017.

[50] David Shue and Michael Freedman. From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra. In *9th European Conference on Computer Systems*. ACM, 2014.

[51] David Shue, Michael Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 349–362. USENIX, 2012.

[52] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sRoute: Treating the Storage Stack Like a Network. In *14th USENIX Conference on File and Storage Technologies*, pages 197–212. USENIX, 2016.

[53] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *2020 ACM SIGMOD International Conference on Management of Data*, page 1493–1509. ACM, 2020.

[54] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[55] Raghav Tulshibagwale. RocksDB at Nutanix. https://www.nutanix.dev/2021/03/10/rocksdb-at-nutanix/.

[56] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies*, pages 59–72. USENIX, 2017.

[57] Matthew Wachs and Michael Abd-El-Malek. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies*. USENIX, 2007.

[58] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *3rd ACM Symposium on Cloud Computing*. ACM, 2012.

[59] Yin Wang and Arif Merchant. Proportional-Share Scheduling for Distributed Storage Systems. In *5th USENIX Conference on File and Storage Technologies*, pages 47–60. USENIX, 2007.

[60] Joel C. Wu and Scott A. Brandt. Providing Quality of Service Support in Object-Based File System. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 157–170. IEEE, 2007.

[61] Miguel Xavier, Israel De Oliveira, Fabio Rossi, Robson Dos Passos, Kassiano Matteussi, and Cesar De Rose. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260. IEEE, 2015.

[62] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini Kaushik, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Split-Level I/O Scheduling. In *25th ACM Symposium on Operating Systems Principles*, pages 474–489. ACM, 2015.

[63] Andy Yoo, Morris Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[64] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage Performance Virtualization via Throughput and Latency Control. *ACM Transactions on Storage*, 2(3):283–308, 2006.

[65] Timothy Zhu, Alexey Tumanov, Michael Kozuch, Mor Harchol-Balter, and Gregory Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *5th ACM Symposium on Cloud Computing*. ACM, 2014.