Accelerated Data Analytics and Computing Institute Seminar

Building user-level storage data planes with PAIO

Ricardo Macedo
INESC TEC & University of Minho

part 1

background and motivation

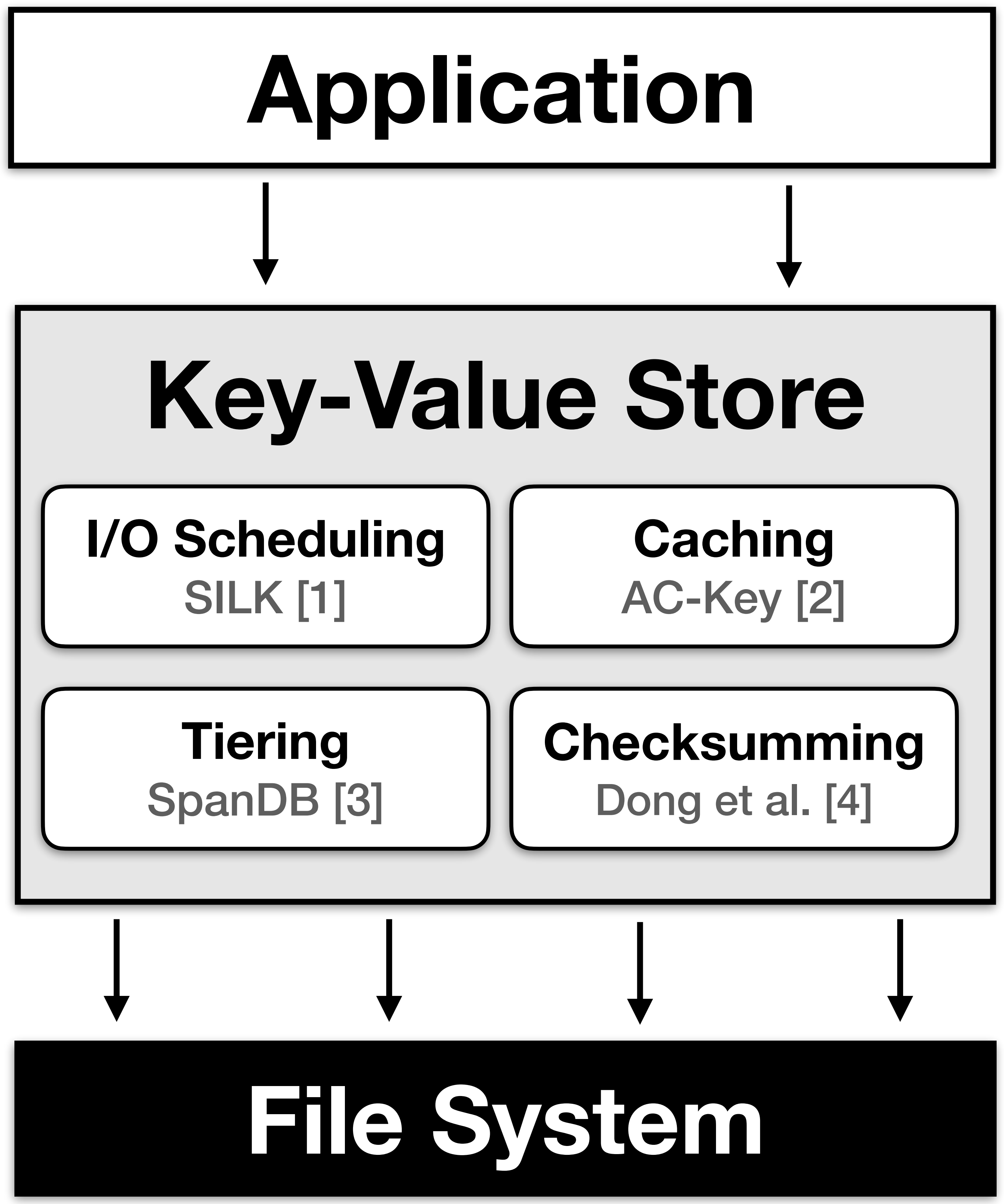
Data-centric systems

- Data-centric systems have become an integral part of modern I/O stacks
- Good performance for these systems often requires storage optimizations
  - **Scheduling**, **caching**, **tiering**, **replication**, …
- Optimizations are implemented in sub-optimal manner

Challenge #1

❌ Tightly coupled optimizations

- I/O optimizations are single purposed
- Require deep understanding of the system's internal operation model
- Require profound system refactoring
- Limited portability across systems

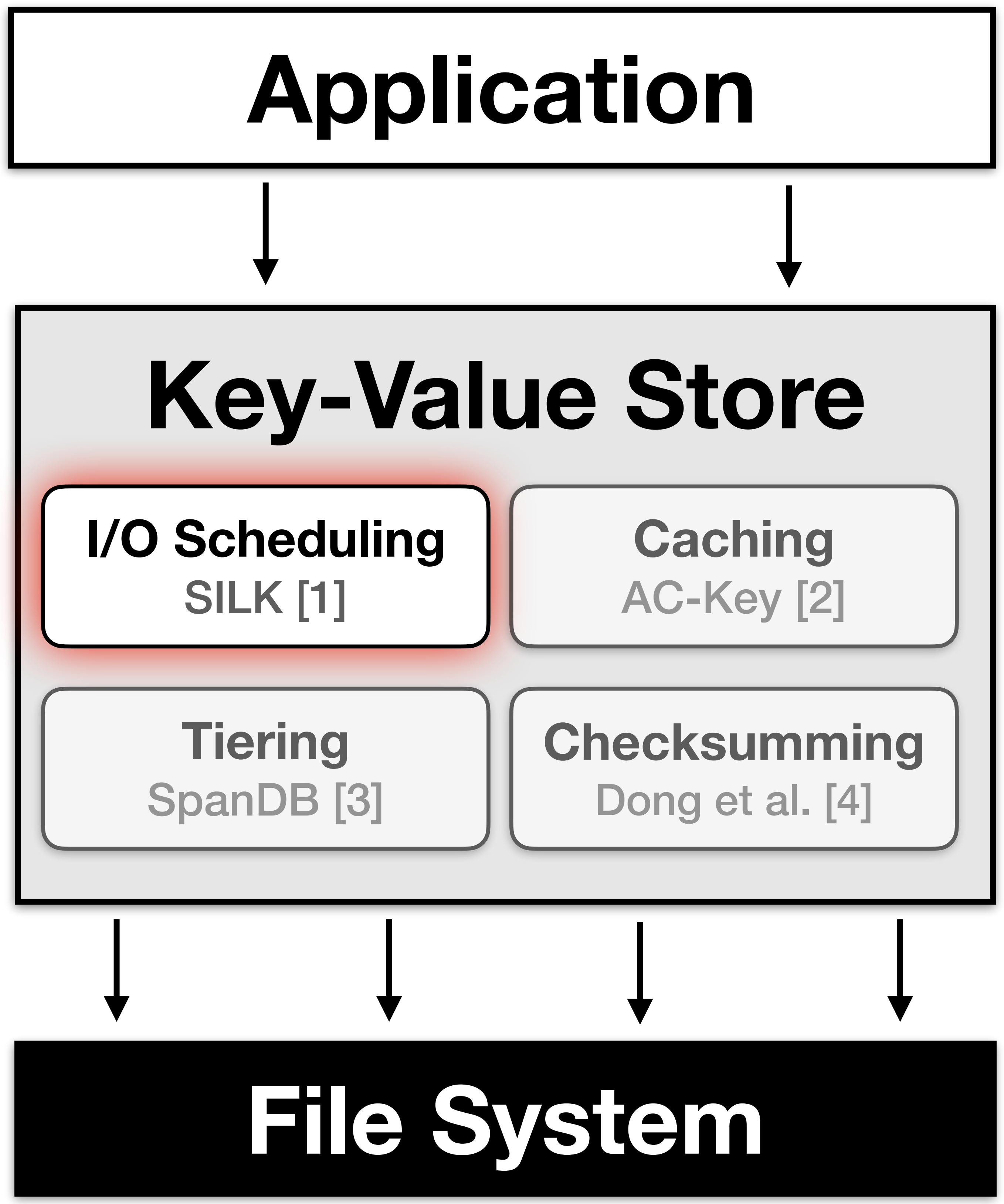


[1] *"SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores"*. Balmau et al. USENIX ATC 2019.
[2] *"AC-Key: Adaptive Caching for LSM-based Key-Value Stores"*. Wu et al. USENIX ATC 2020.
[3] *"SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage"*. Chen et al. USENIX FAST 2021.
[4] *"Evolution of Development Priorities in Key-Value Stores Serving Large-scale Applications: The RocksDB Experience"*. Dong et al. USENIX FAST 2021.

Challenge #1

Tightly coupled optimizations

- I/O optimizations are single purposed
- Require deep understanding of the system's internal operation model
- Require profound system refactoring
- Limited portability across systems



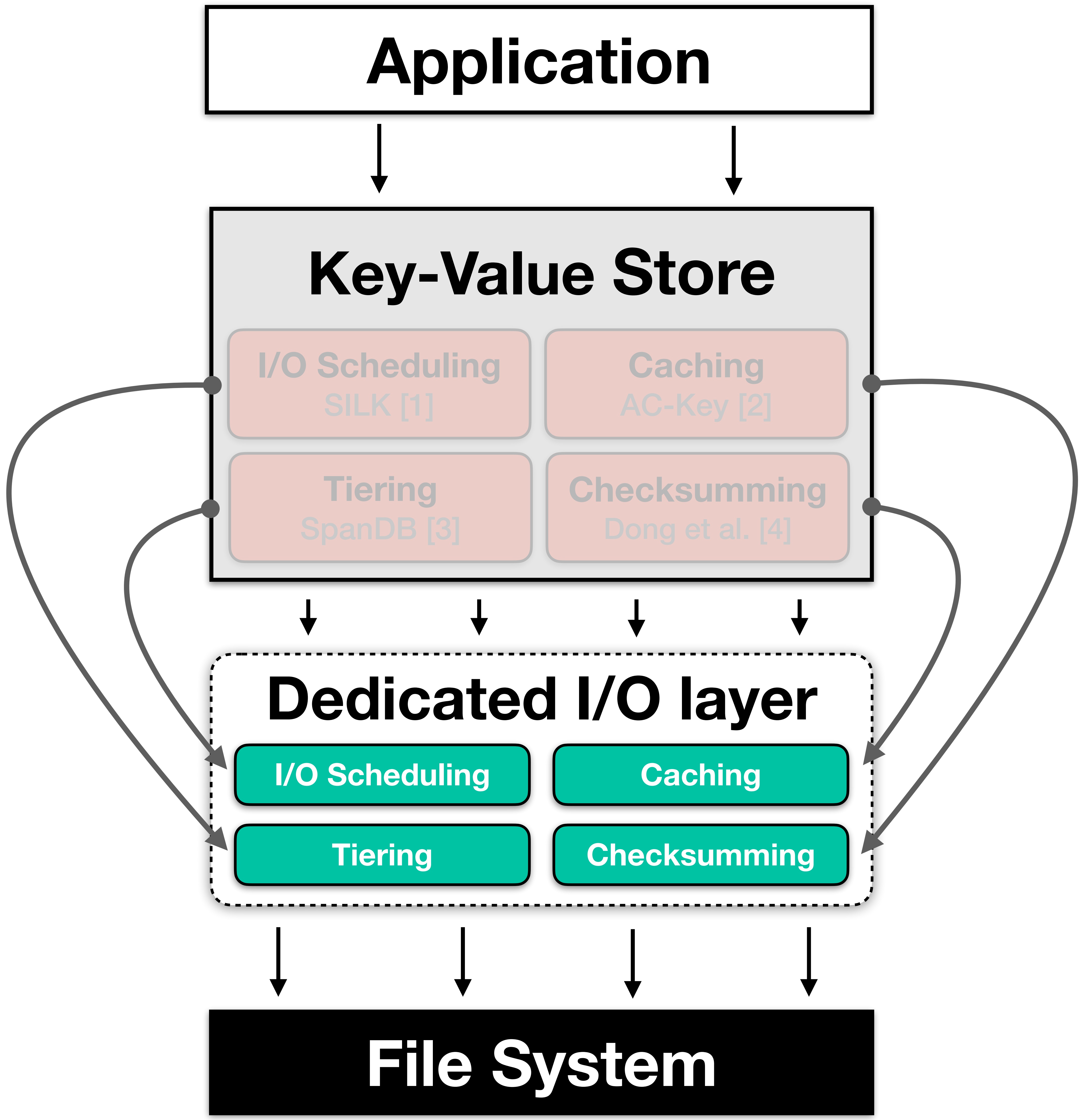
SILK's I/O Scheduler

- Reduce tail latency spikes in RocksDB
- Controls the interference between foreground and background tasks
- Required changing several modules, such as *background operation handlers*, *internal queuing logic*, and *thread pools*

Challenge #1

✅ Decoupled optimizations

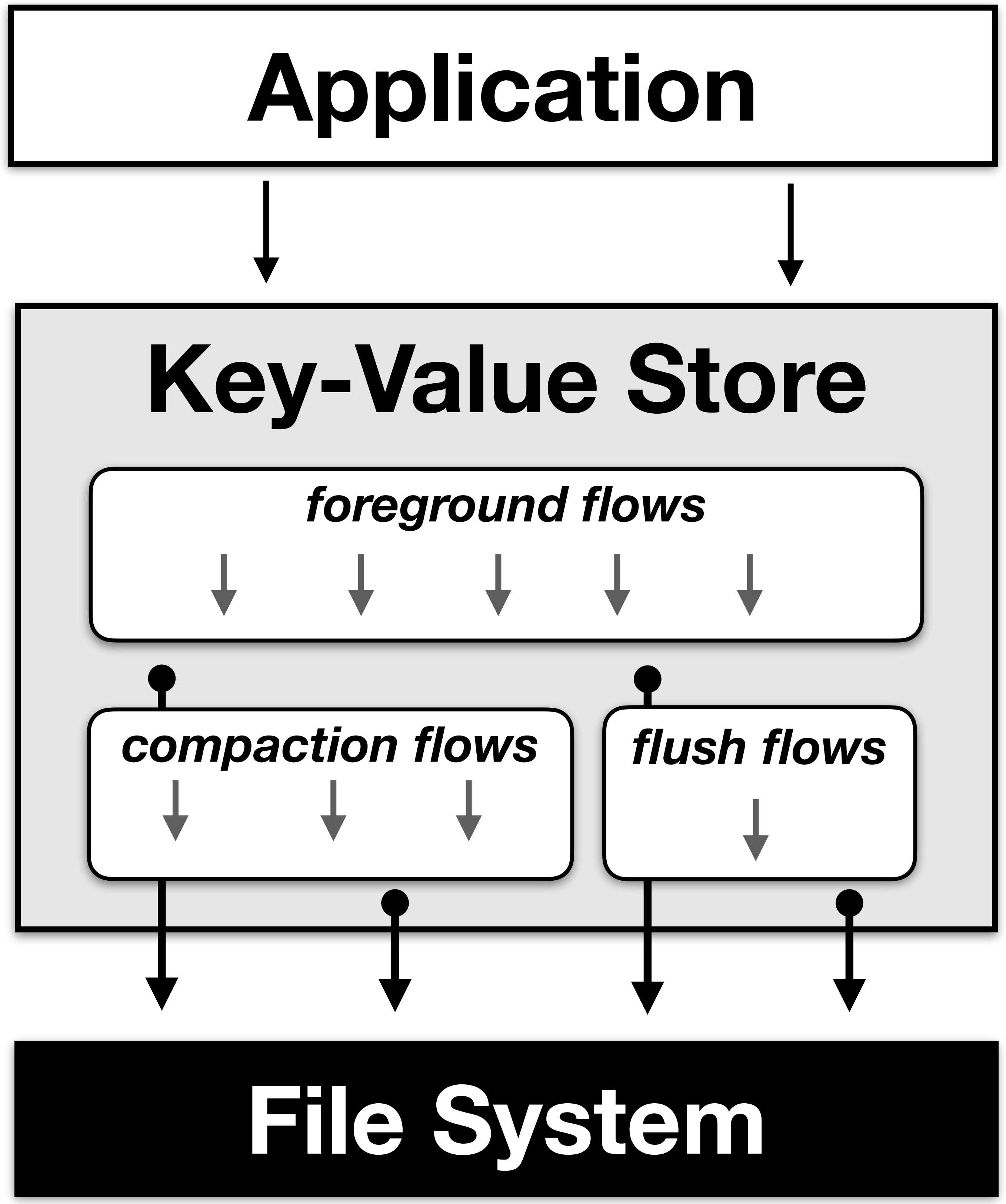
- I/O optimizations should be disaggregated from the internal logic
- Moved to a dedicated I/O layer
- Generally applicable
- Portable across different scenarios



Challenge #2

Rigid interfaces

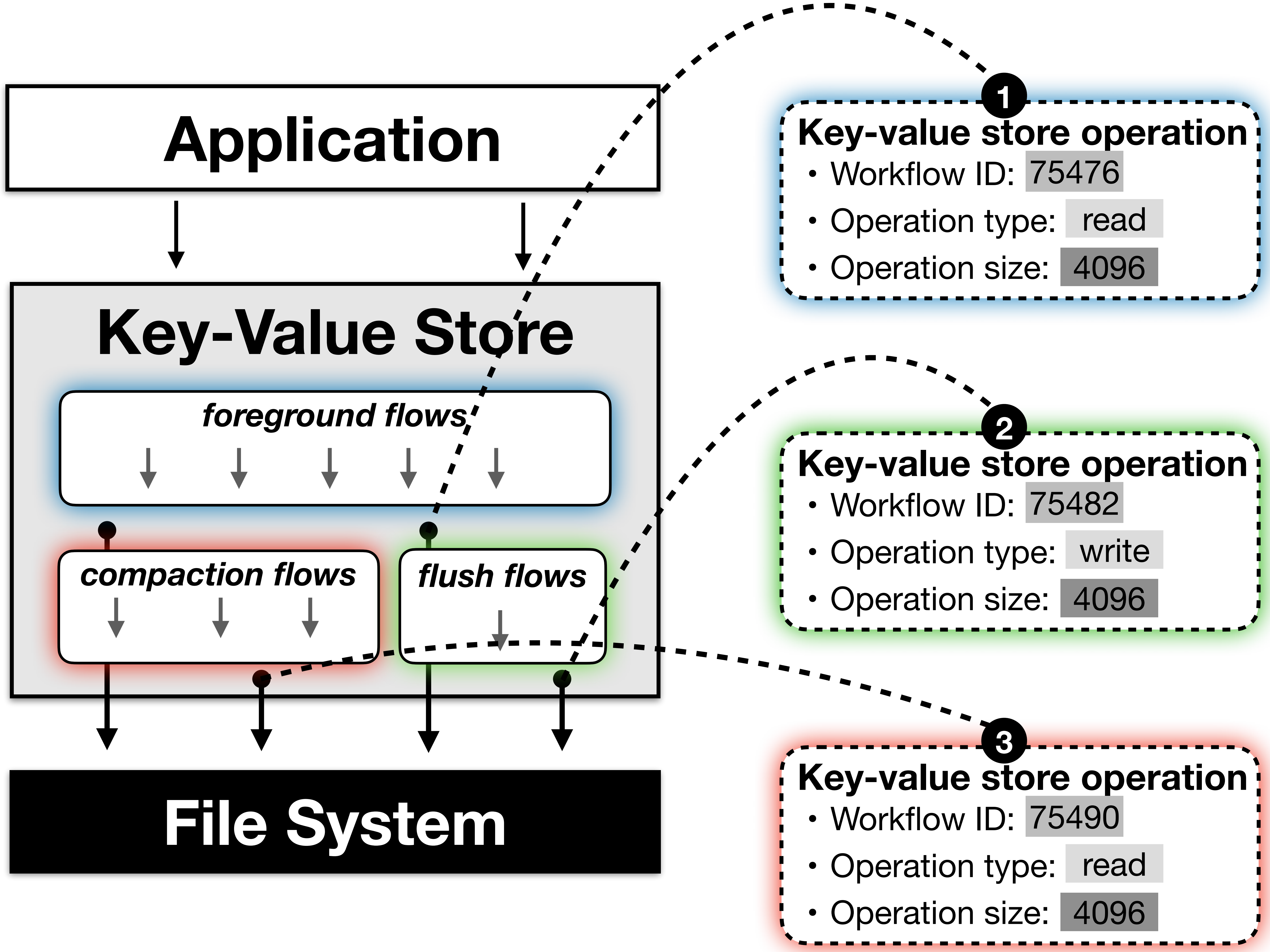
- Decoupled optimizations lose granularity and internal application knowledge
- I/O layers communicate through rigid interfaces
- Discard information that could be used to classify and differentiate requests



Challenge #2

Rigid interfaces

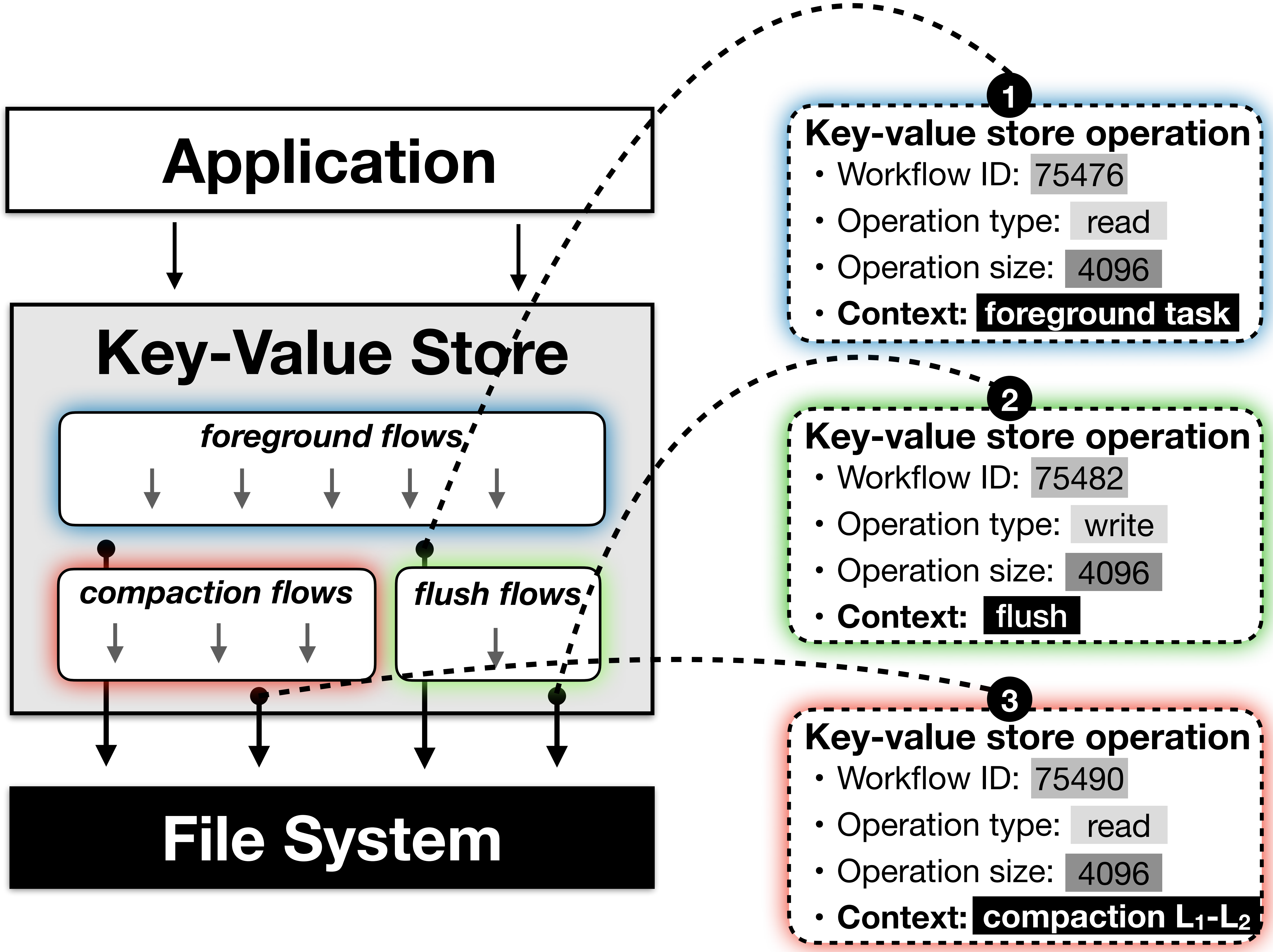
- Decoupled optimizations lose granularity and internal application knowledge
- I/O layers communicate through rigid interfaces
- Discard information that could be used to classify and differentiate requests



Challenge #2

✅ Information propagation

- Application-level information must be propagated throughout layers
- Decoupled optimizations can provide the same level of control and performance

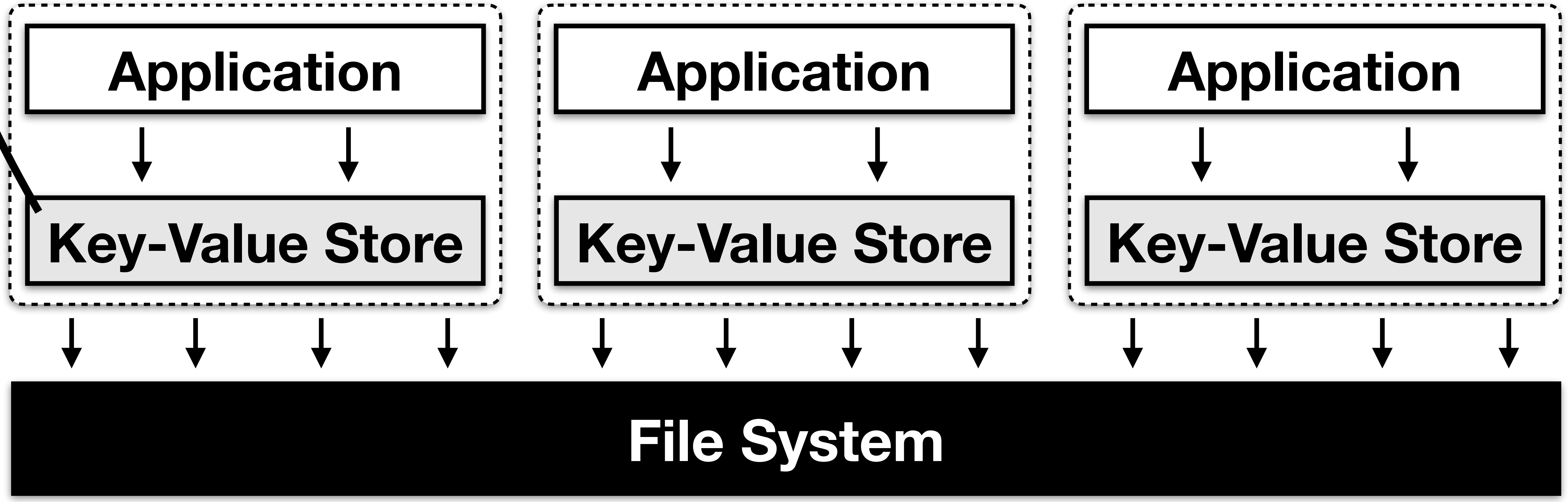


Challenge #3

Partial visibility

- Optimizations are oblivious of other systems
- Lack of coordination
- Conflicting optimizations, I/O contention, and performance variation

Note: in this case it's a key-value store, but could be any other data-centric layer (e.g., DL framework, scientific application, database)

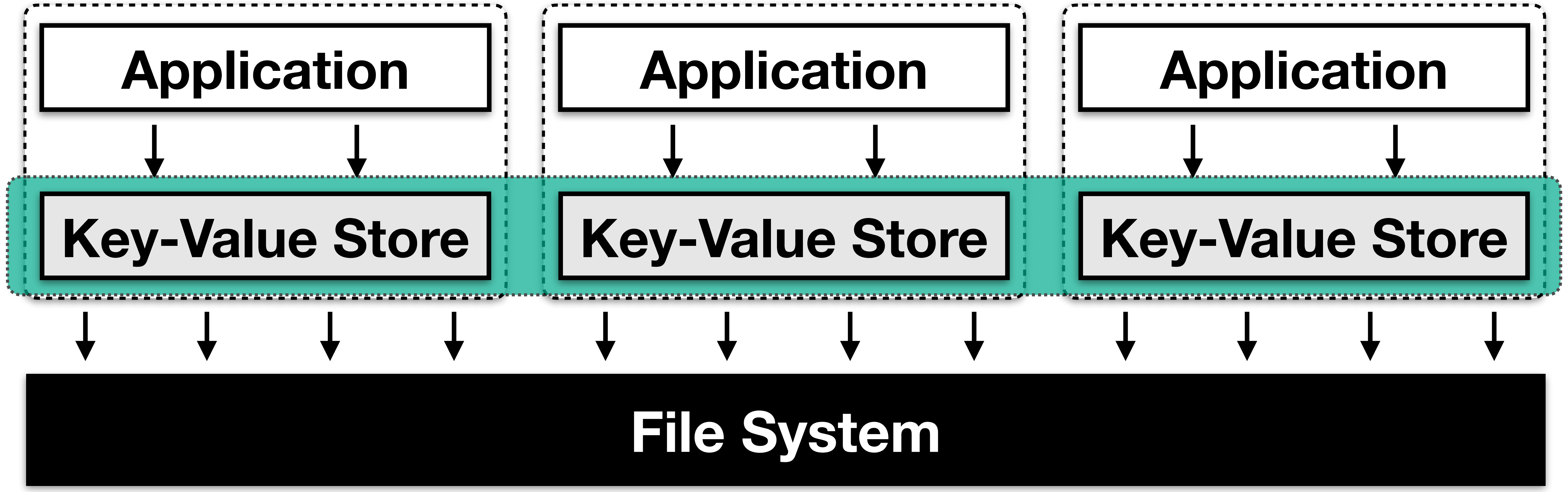


Note: the storage backend can either be local (*e.g.,* ext4, xfs) or distributed (*e.g.,* Lustre, GPFS)

Challenge #3

Global I/O control

- Optimizations should be aware of the surrounding system stack
- Operate in coordination
- Holistic control of I/O workflows and shared resources



part 2

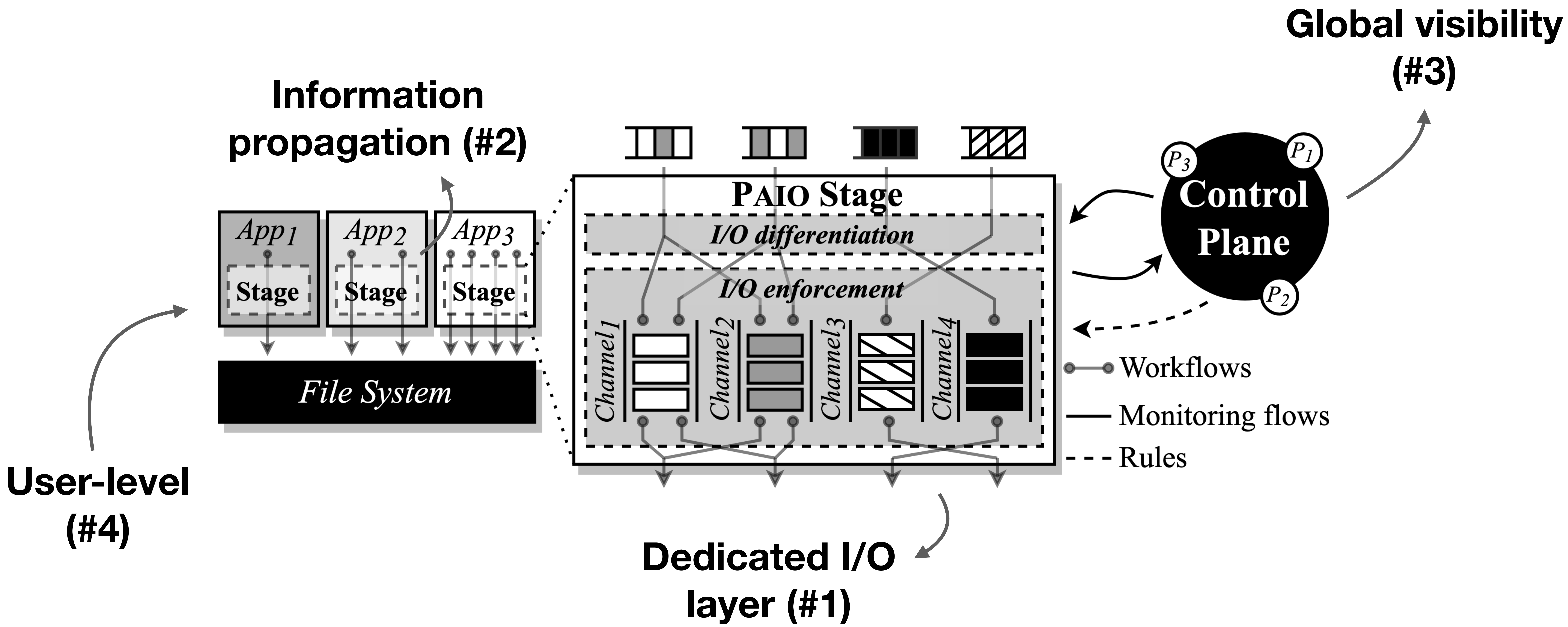
designing a storage data plane framework

PAIO

- **User-level** framework for building **portable** and **generally applicable** optimizations
- Adopts ideas from **Software-Defined Storage** [6]
  - I/O optimizations are implemented ***outside*** applications as **data plane stages**
  - **Stages** are controlled through a **control plane** for coordinated access to resources
- Enables the propagation of application-level information through **context propagation**
- Porting I/O layers to use PAIO requires **none to minor** code changes

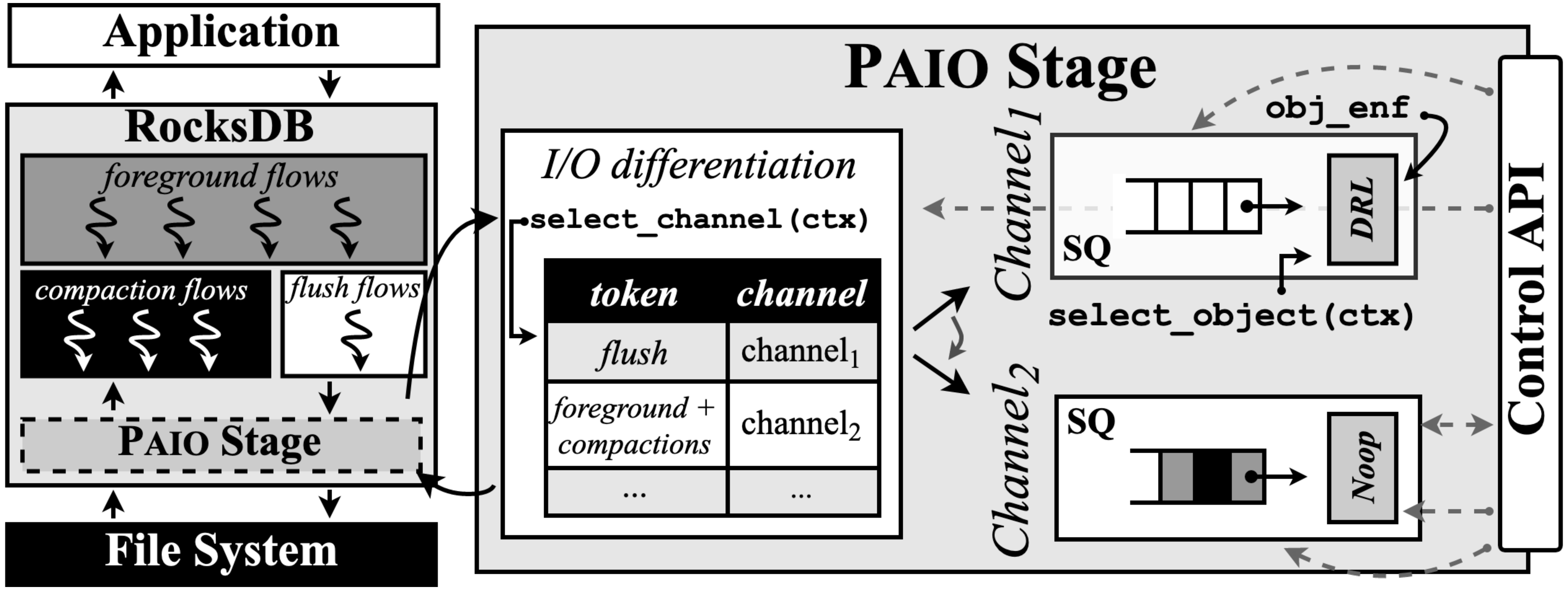
[5] *"PAIO: General, Portable I/O Optimizations with Minor Application Modifications"*. Macedo et al. USENIX FAST 2022.
[6] *"A Survey and Classification of Software-Defined Storage Systems"*. Macedo et al. ACM CSUR 2020.

PAIO design


Global visibility (#3)
Information propagation (#2)
App1
App2
App3
Stage
Stage
Stage
File System
PAIO Stage
I/O differentiation
I/O enforcement
Channel1
Channel2
Channel3
Channel4
P3
P1
P2
Control Plane
Workflows
Monitoring flows
Rules
User-level (#4)
Dedicated I/O layer (#1)

PAIO design

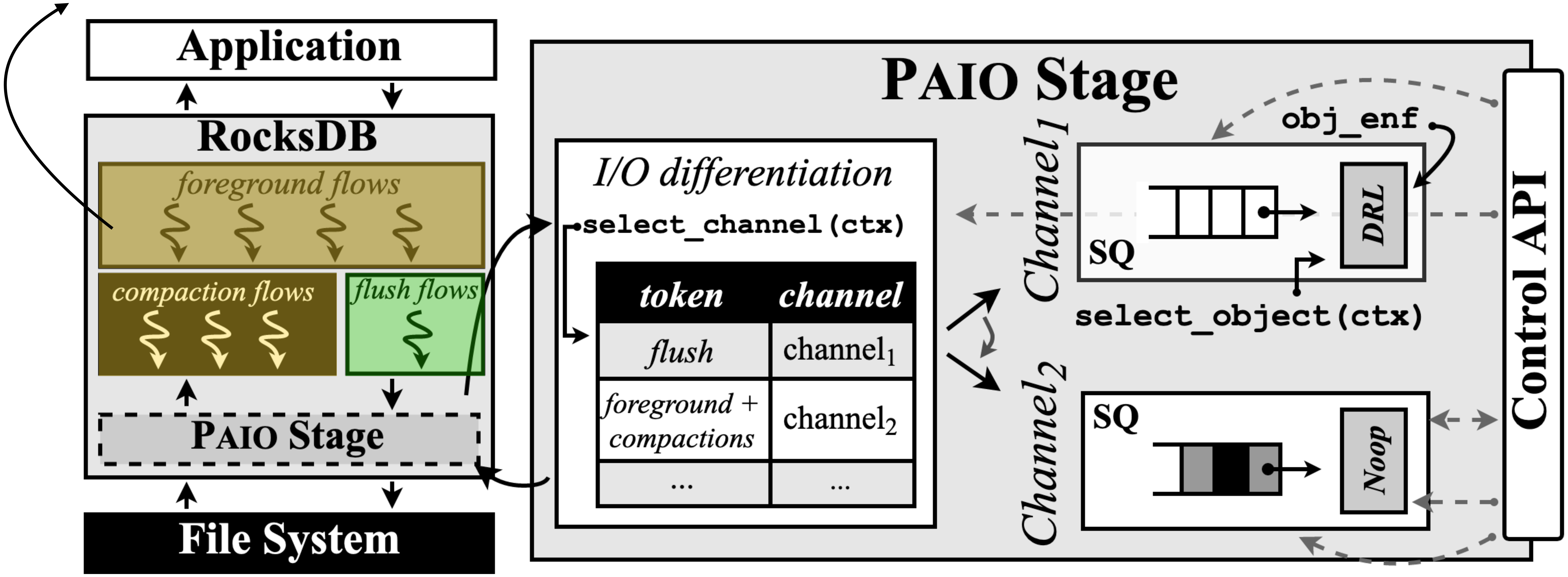
- I/O differentiation
- I/O enforcement
- Control plane interaction



Policy: *limit the rate of RocksDB's flush operations to X MiB/s*

I/O differentiation

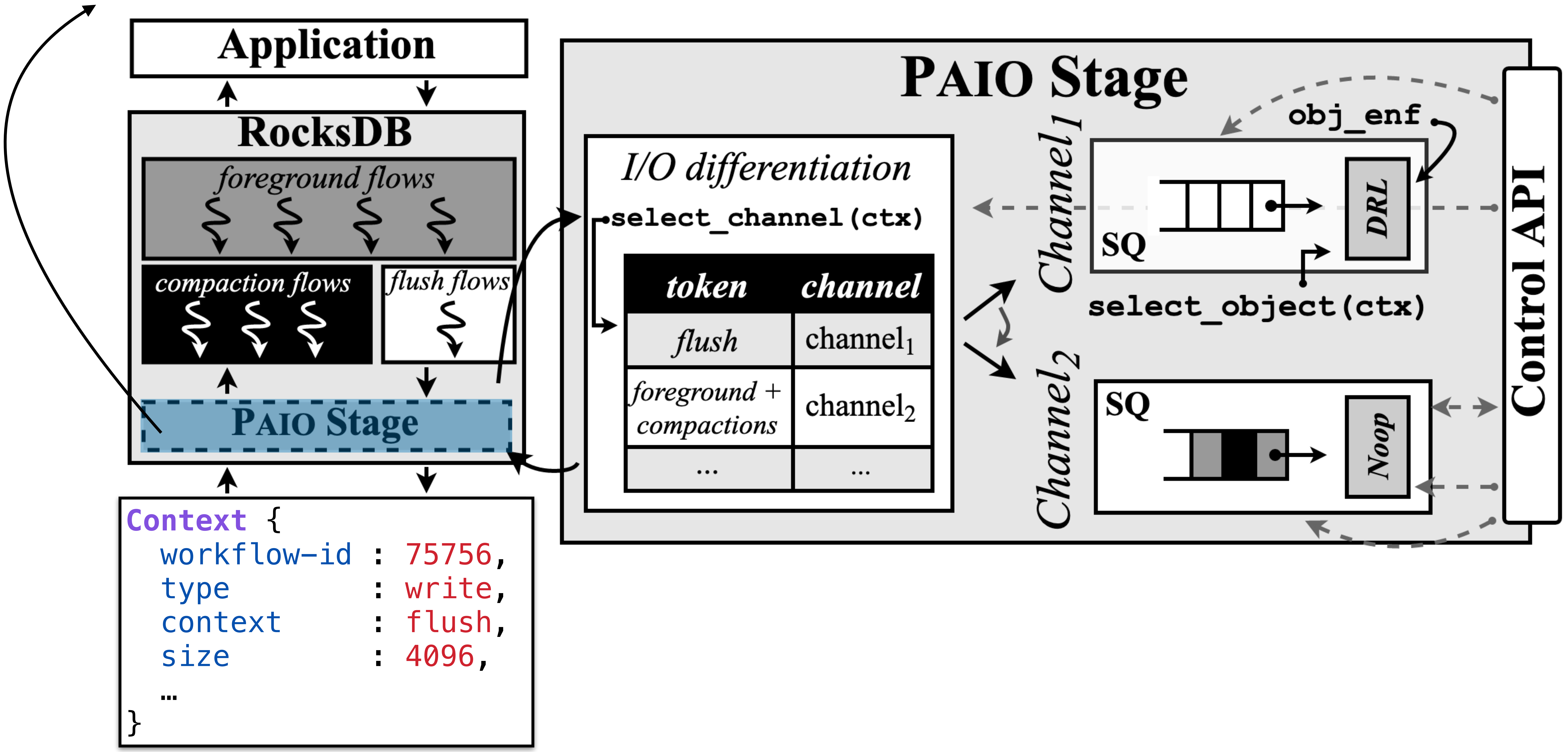
Context propagation:
instrumentation + propagation phases



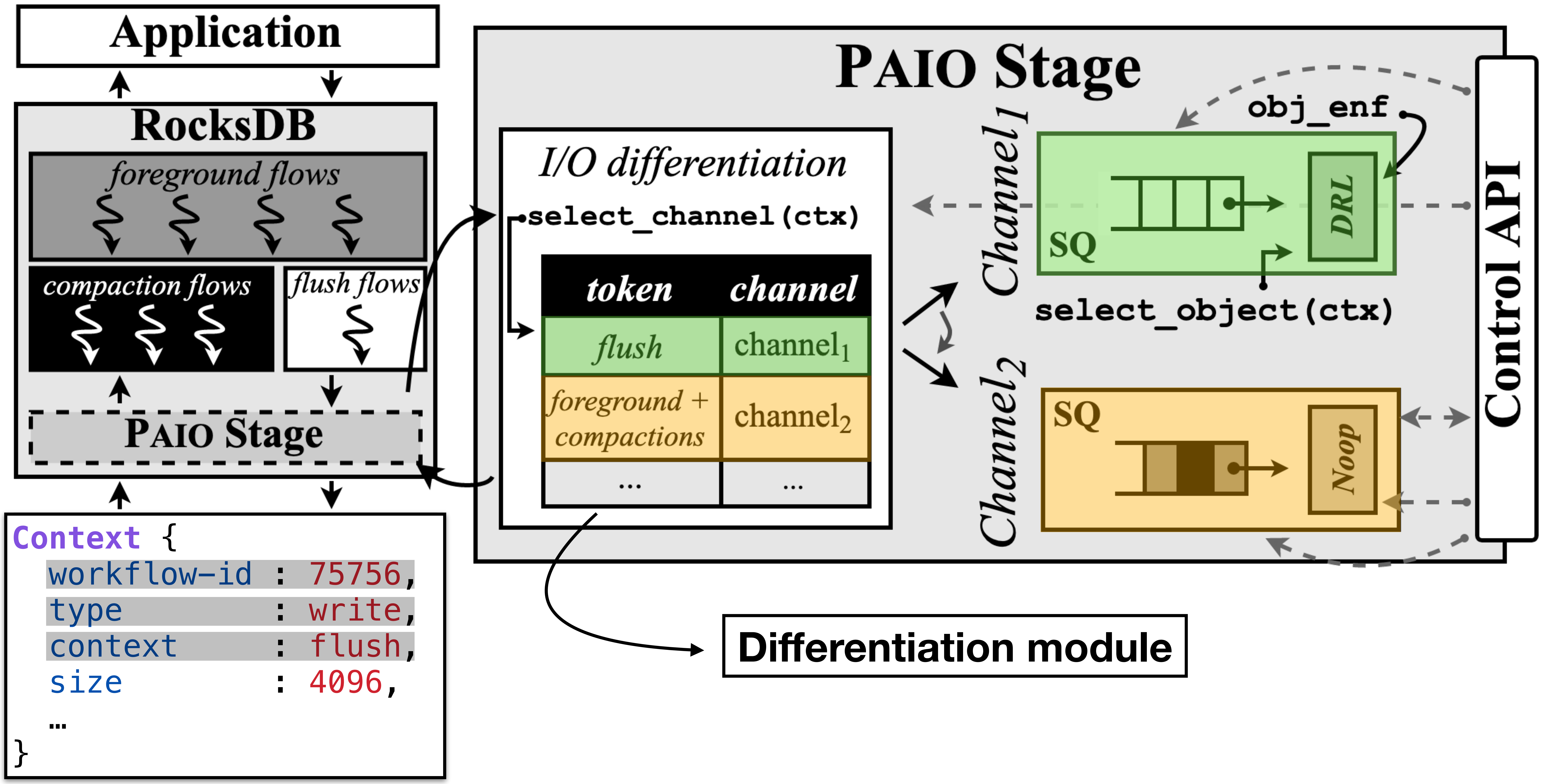
Identify the origin of POSIX operations (i.e., **foreground**, **compaction**, or **flush** operations)

I/O differentiation

Context propagation:
propagation + classification phases

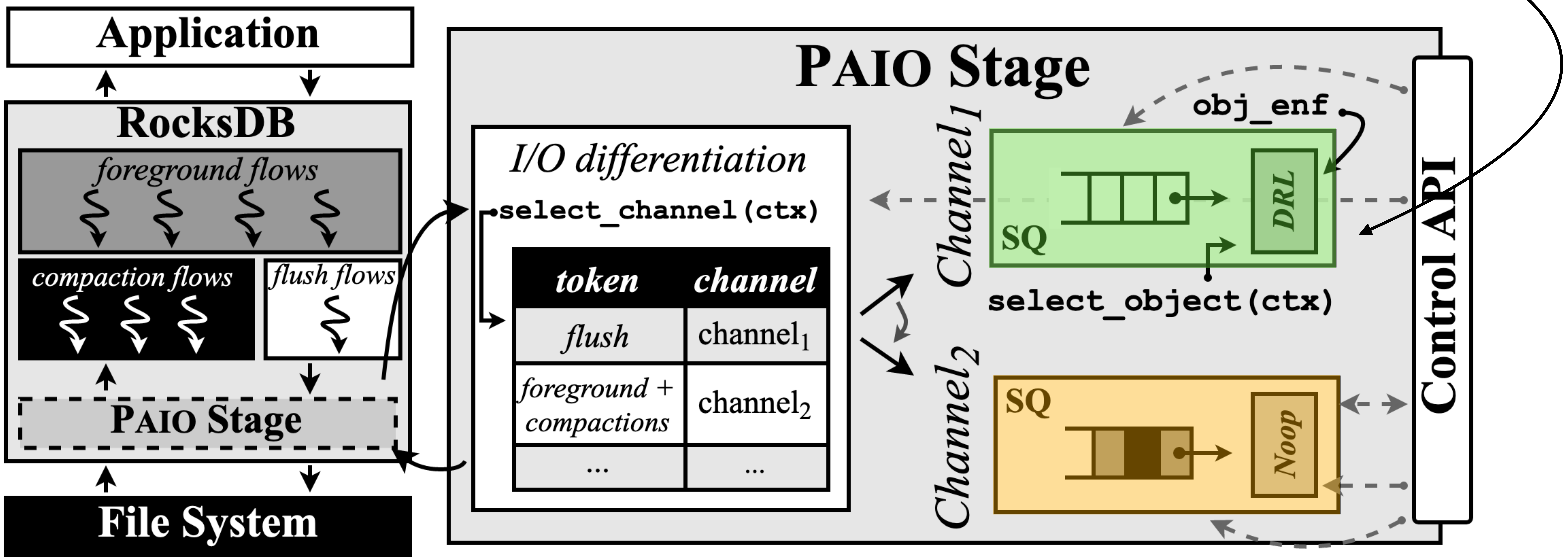


I/O differentiation



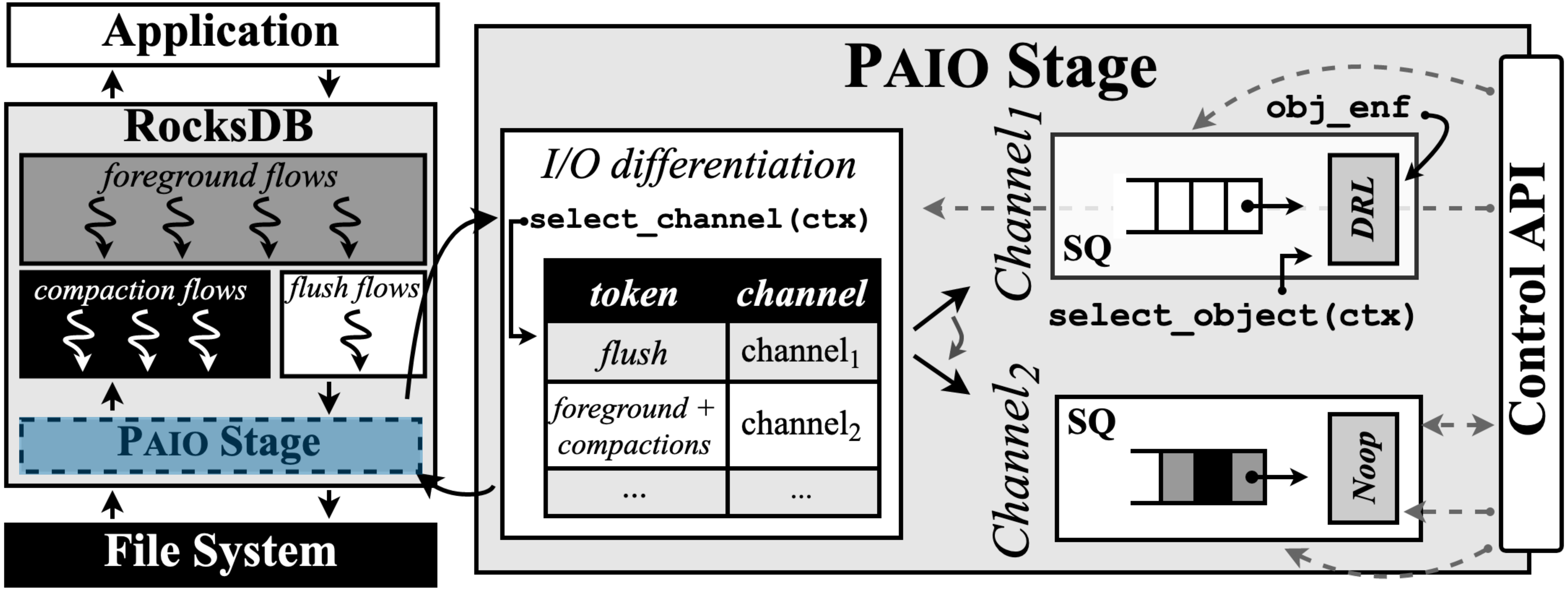
I/O enforcement

Enforcement module



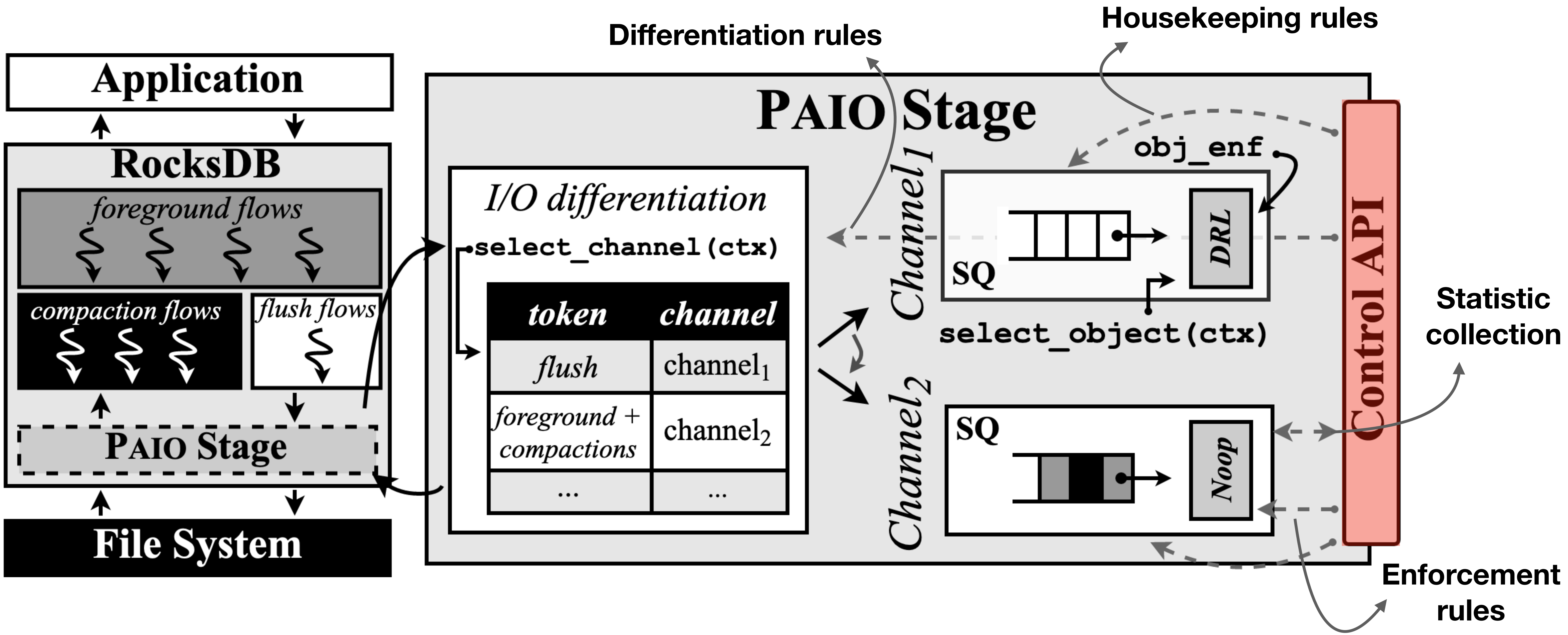
PAIO currently supports **Noop** (passthrough) and **DRL** (token-bucket) enforcement objects

I/O enforcement



Requests return to their original I/O path

Control plane interaction



Implements the **control algorithms** for orchestrating stages (e.g., **tail latency control**, **per-application bandwidth guarantees**)

More about PAIO

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 99th 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 99th 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).

USENIX Association 20th USENIX Conference on File and Storage Technologies 413

PAIO paper

- Context propagation
- PAIO interfaces
- Control algorithms
- Micro and macro experiments

part 3

building storage data planes

Per-application bandwidth control

ABCI supercomputer

- Jobs can be co-located in the same compute node
- Each job runs with dedicated CPU cores, memory, GPU, and storage quota
- Local **disk bandwidth** is **shared**, leading to **I/O interference** and **performance variation**

BLKIO

- cgroup's block I/O controller allows static rate limiting read and write operations
- Adjusting the rate requires stopping and restarting jobs
- Cannot leverage from leftover bandwidth

PAIO

- Stage provides the I/O mechanisms to dynamically rate limit workflows at each instance
  - Integrating PAIO in TensorFlow did not required any code changes (LD_PRELOAD)
- Control plane provides a proportional sharing algorithm to ensure per-application bandwidth QoS guarantees

Per-application bandwidth control

ABCI supercomputer

- Jobs can be co-located in the same compute node
- Each job runs with dedicated CPU cores, memory, GPU, and storage quota
- Local disk bandwidth is shared, leading to I/O interference and performance variation

BLKIO

- cgroup's block I/O controller allows **static rate limiting** read and write operations
- Adjusting the rate requires **stopping** and **restarting jobs**
- **Cannot leverage** from **leftover bandwidth**

PAIO

- Stage provides the I/O mechanisms to dynamically rate limit workflows at each instance
  - Integrating PAIO in TensorFlow did not required any code changes (LD_PRELOAD)
- Control plane provides a proportional sharing algorithm to ensure per-application bandwidth QoS guarantees

Per-application bandwidth control

ABCI supercomputer

- Jobs can be co-located in the same compute node
- Each job runs with dedicated CPU cores, memory, GPU, and storage quota
- Local disk bandwidth is shared, leading to I/O interference and performance variation

BLKIO

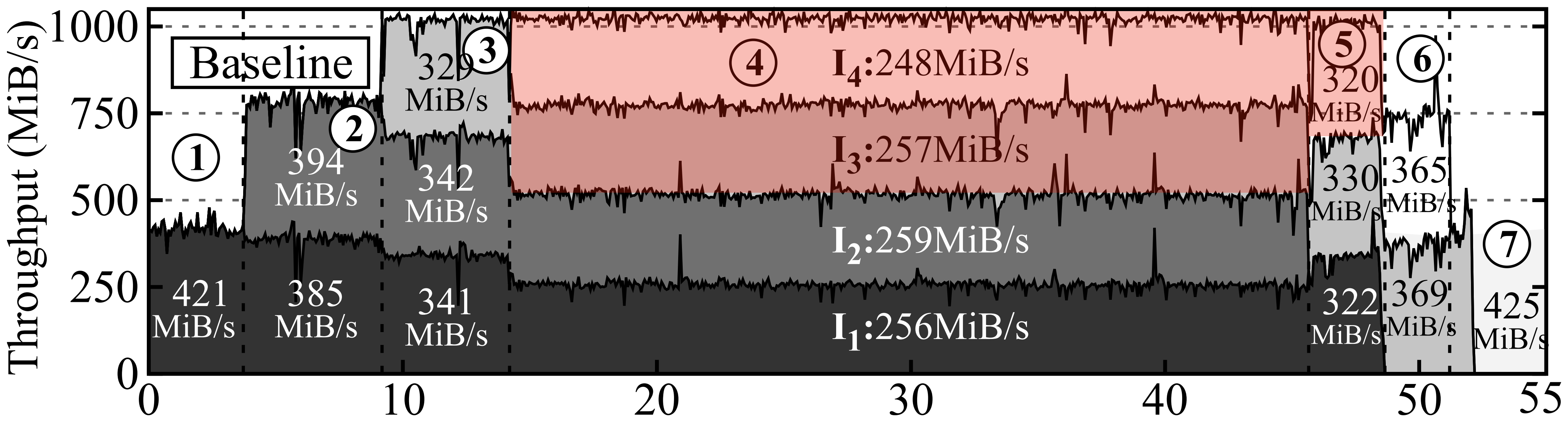
- cgroup's block I/O controller allows static rate limiting read and write operations
- Adjusting the rate requires stopping and restarting jobs
- Cannot leverage from leftover bandwidth

PAIO

- Stage provides the I/O mechanisms to **dynamically rate limit** workflows at each instance
  - Integrating PAIO in TensorFlow did **not required any code changes** (LD_PRELOAD)
- Control plane provides a **proportional sharing algorithm** to ensure per-application bandwidth QoS guarantees

Per-application bandwidth control

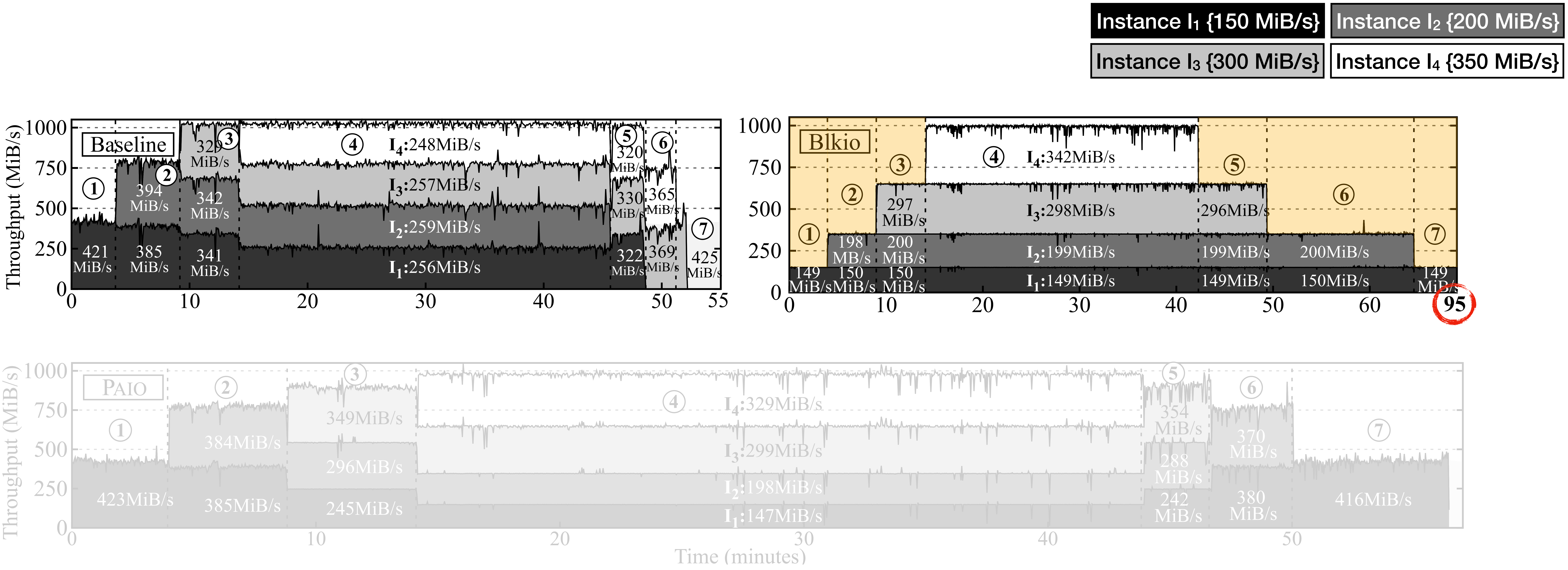
Instance I_1 {150 MiB/s}	Instance I_2 {200 MiB/s}
Instance I_3 {300 MiB/s}	Instance I_4 {350 MiB/s}



System configuration and workload
- 4 working instances, each running a TensorFlow job
- Dedicated compute and memory resources
- Disk bandwidth limited to 1GiB/s
- Jobs start at different times

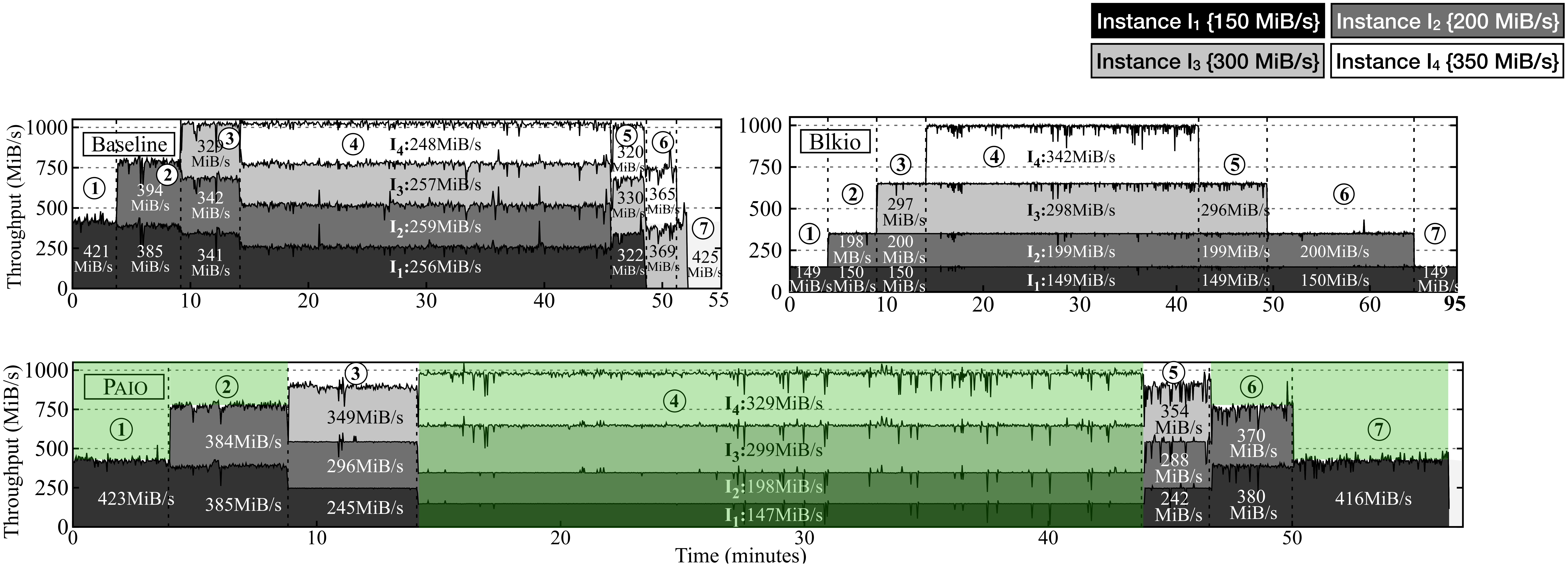
I_3 and I_4 cannot meet their bandwidth targets during 31 and 34 minutes

Per-application bandwidth control



Instances cannot be dynamically provisioned with available disk bandwidth

Per-application bandwidth control



PAIO ensures that policies are met at all times, and whenever leftover bandwidth is available, PAIO shares it across active instances

Tail latency control in LSM-based KVS

RocksDB

- Interference between foreground and background tasks generates high latency spikes
- Latency spikes occur due to L_0-L_1 compactions and flushes being slow or on hold

SILK

- I/O scheduler
  - Allocates bandwidth for internal operations when client load is low
  - Prioritizes flushes and low level compactions
  - Preempts high level compactions with low level ones
- Required changing several core modules made of thousands of LoC

PAIO

- Stage provides the I/O mechanisms for prioritizing and rate limiting background flows
  - Integrating PAIO in RocksDB only required adding 85 LoC
- Control plane provides a SILK-based I/O scheduling algorithm

Tail latency control in LSM-based KVS

RocksDB

- Interference between foreground and background t
- Latency spikes occur due to L_0-L_1 compactions and

SILK

- I/O scheduler
  - **<u>Allocates bandwidth for internal operations when client load is low</u>**
  - **<u>Prioritizes flushes and low level compactions</u>**
  - ~~Preempts high level compactions with low level ones~~
- Required changing several core modules made of thousands of LoC

PAIO

- Stage provides the I/O mechanisms for prioritizing and rate limiting background flows
  - Integrating PAIO in RocksDB only required adding 85 LoC
- Control plane provides a SILK-based I/O scheduling algorithm

! **Note:** By propagating application-level information to the stage, PAIO can enable similar control and performance as system-specific optimizations

Mixture workload

50% read 50% write

System configuration and workload

- 8 client threads and 8 background threads
- Memory limited to 1GB and I/O BW to 200MB/s
- Bursty workload with peaks and valleys

Throughput: high variability due to constant flushes and compactions

99th latency: high tail latency with peaks with an average range between 3 and 15 ms

Mixture workload

50% read 50% write

Throughput: suffers periodic throughput drops due to accumulated backlog

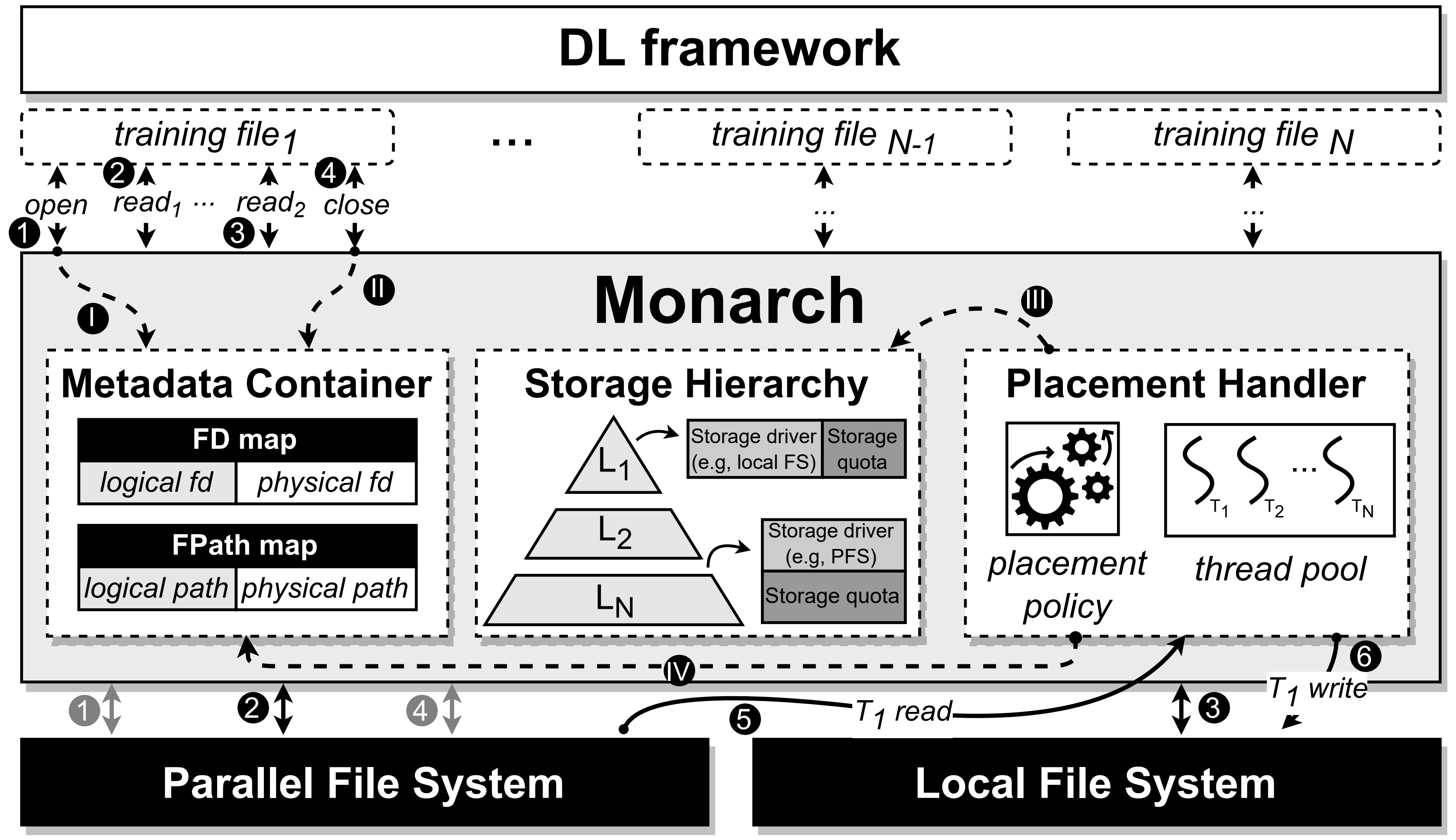
99th latency: low sustained tail latency

PAIO and SILK observe a 4x decrease in absolute tail latency

Data planes for Deep Learning

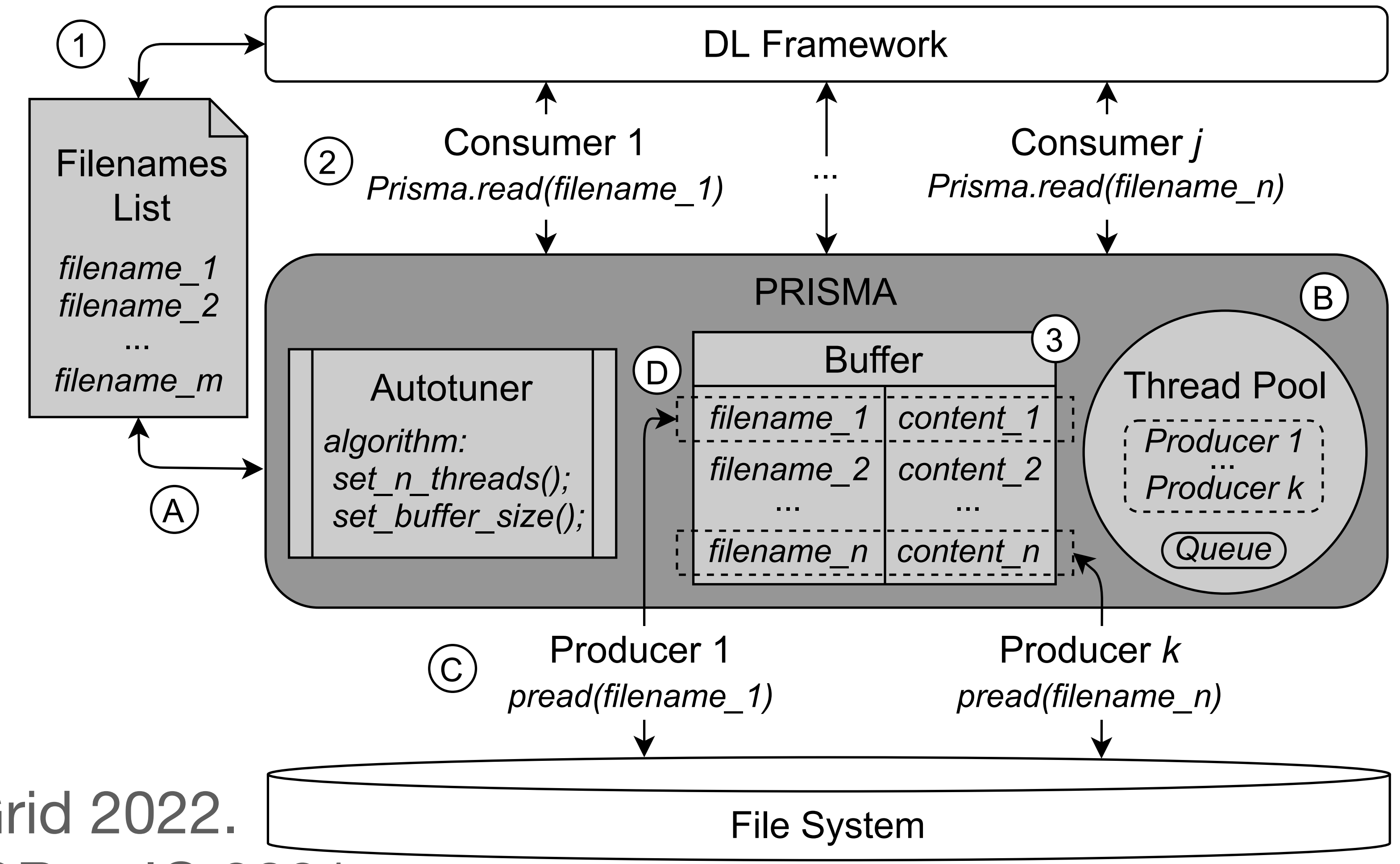
Storage tiering (Monarch)

- Framework-agnostic storage middleware
- Leverages existing storage tiers of supercomputers
- Accelerates DL training time by up to 28% and 37% in TensorFlow and PyTorch
- Decreases the operations submitted to the PFS



Parallel data prefetching (Prisma)

- Data plane for prefetching training data samples
- Significantly outperforms baseline PyTorch and TensorFlow configurations
- Achieves similar performance as carefully engineered I/O optimizations in TensorFlow



[7] *"Accelerating Deep Learning Training Through Transparent Storage Tiering"*. Dantas et al. ACM/IEEE CCGrid 2022.
[8] *"Monarch: Hierarchical Storage Management for Deep Learning Frameworks"*. Dantas et al. IEEE Cluster@Rex-IO 2021.
[9] *"The Case for Storage Optimization Decoupling in Deep Learning Frameworks"*. Macedo et al. IEEE Cluster@Rex-IO 2021.

Summary and takeaways

- **PAIO**, a **user-level** framework to build **custom-made** storage **data plane stages**
- Combines ideas from **Software-Defined Storage** and **context propagation**
- **Decouples** system-specific optimizations to **dedicated** I/O layers
- **User-level data planes** enable similar **control** and **I/O performance** as system-specific optimizations
  - Can be applied over (a lot of) different storage scenarios …

Accelerated Data Analytics and Computing Institute Seminar

Building user-level storage data planes with PAIO

Ricardo Macedo

INESC TEC & University of Minho

ricardo.g.macedo@inesctec.pt

github.com/dsrhaslab

dsr-haslab.github.io