User-level Software-Defined Storage Data Planes

Ricardo Macedo

INESC TEC & U. Minho

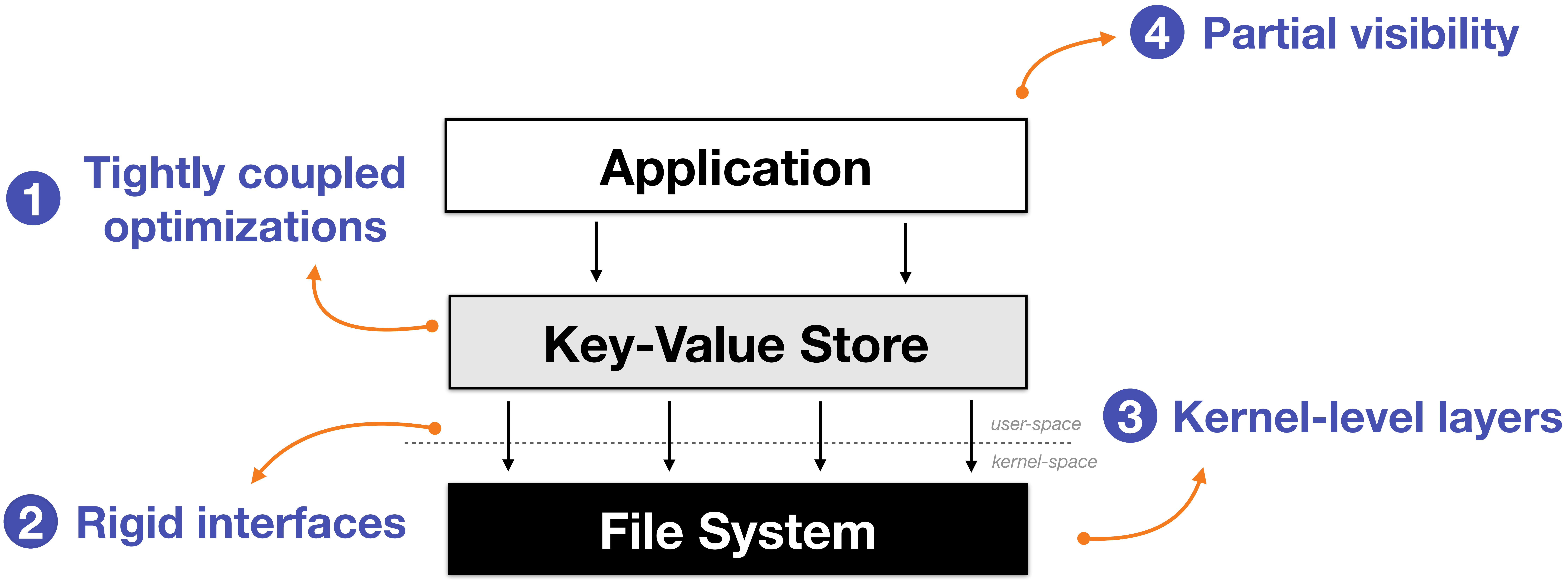
Part 1

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

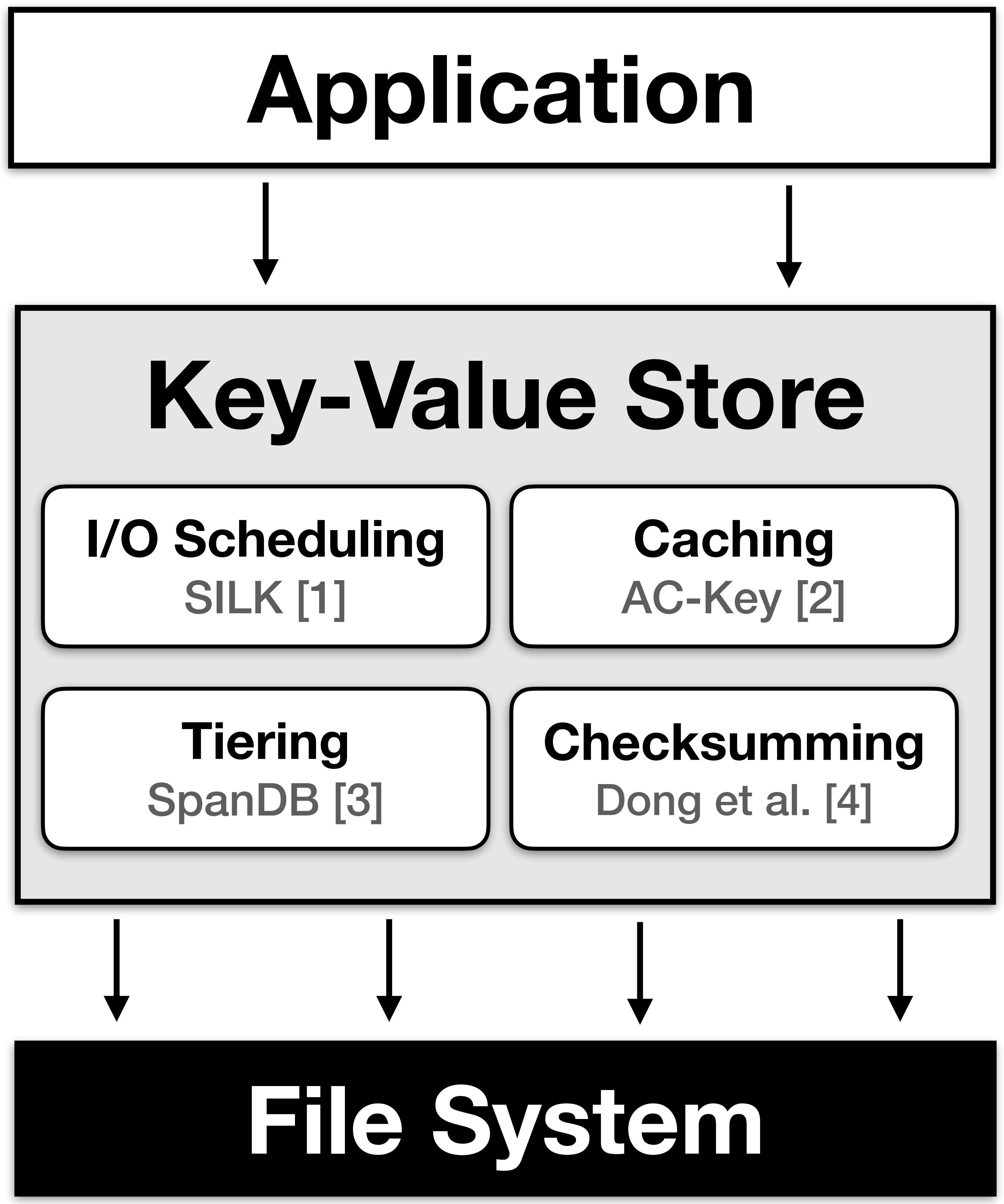
Challenges


4 Partial visibility
1 Tightly coupled optimizations
Application
Key-Value Store
user-space
kernel-space
3 Kernel-level layers
2 Rigid interfaces
File System

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

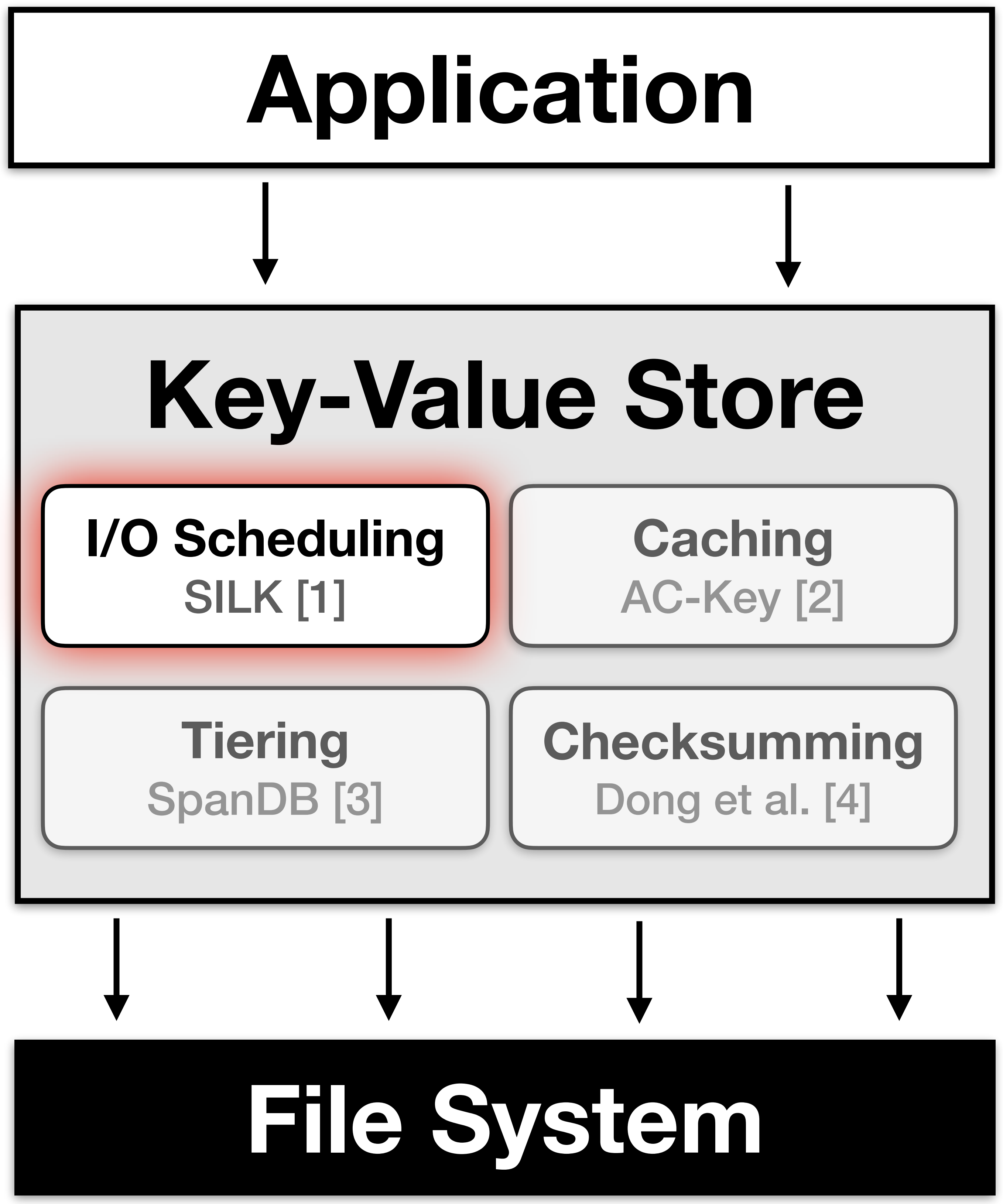


[1] *"SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores"*. Balmau et al. USENIX ATC 2019.
[2] *"AC-Key: Adaptive Caching for LSM-based Key-Value Stores"*. Wu et al. USENIX ATC 2020.
[3] *"SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage"*. Chen et al. USENIX 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



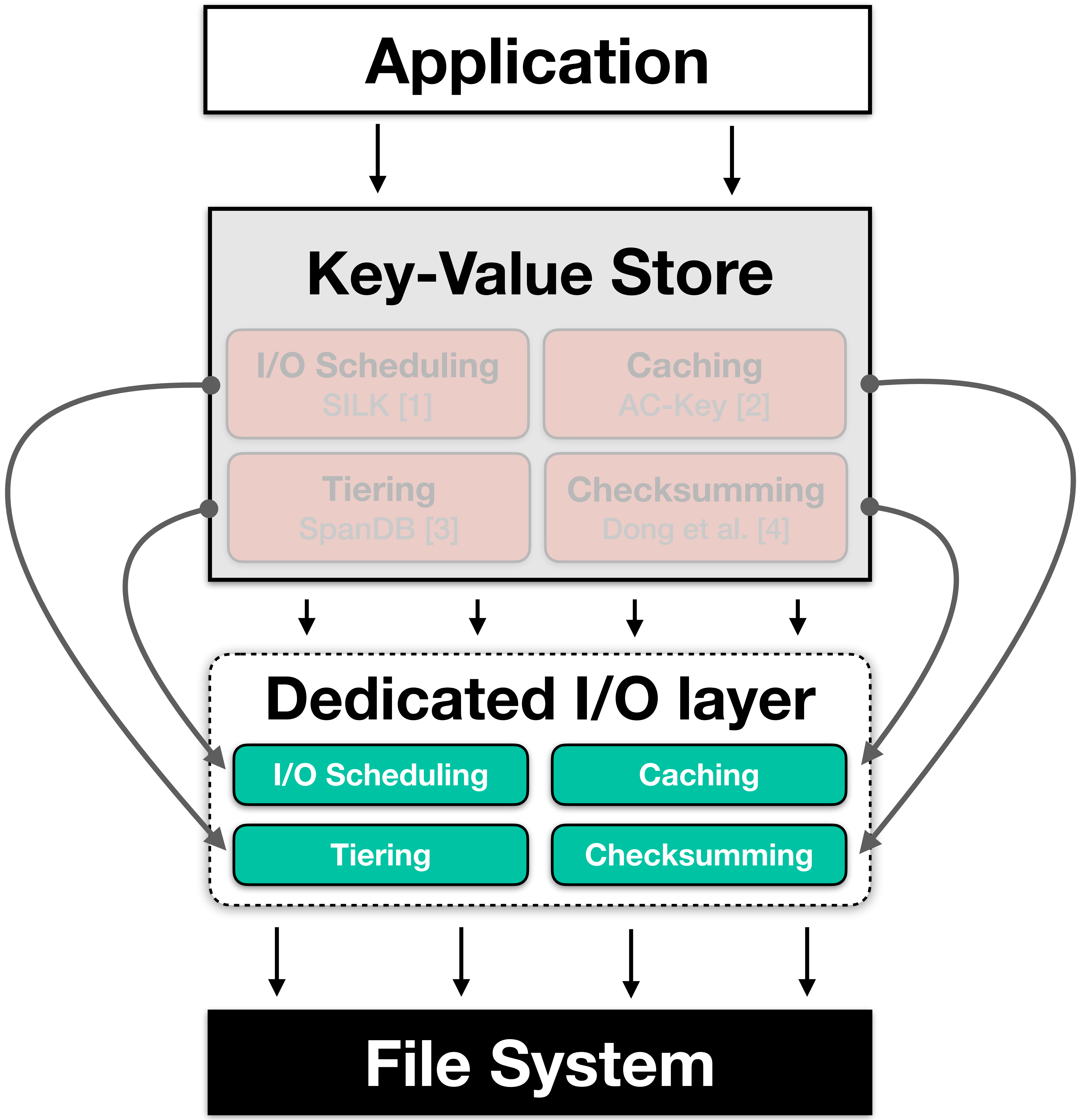
SILK's I/O Scheduler

- 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

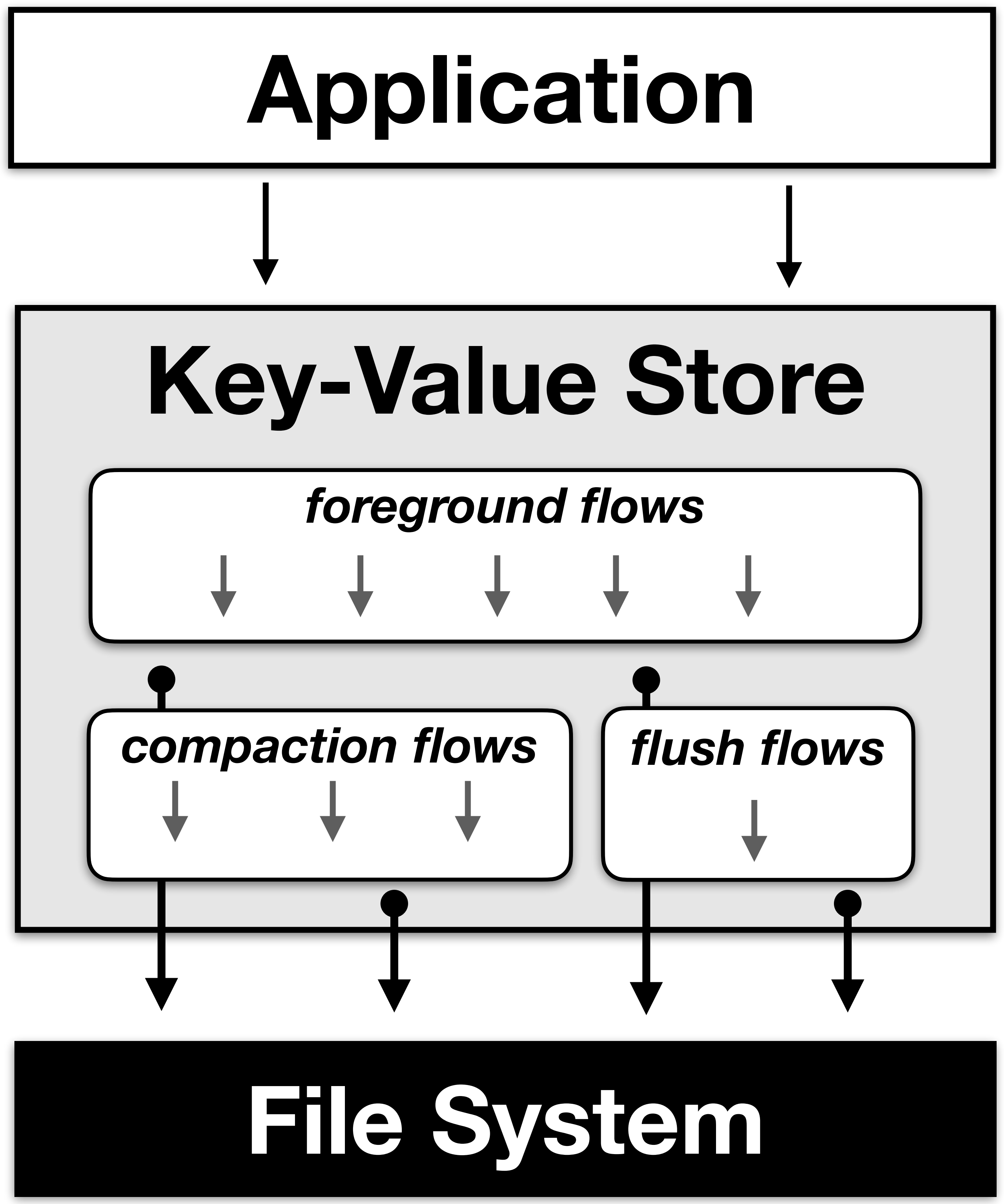
- I/O optimizations should be disaggregated from the internal logic
- Moved to a dedicated I/O layer
- Generally applicable
- Portable across different scenarios



Challenge #2

Rigid interfaces

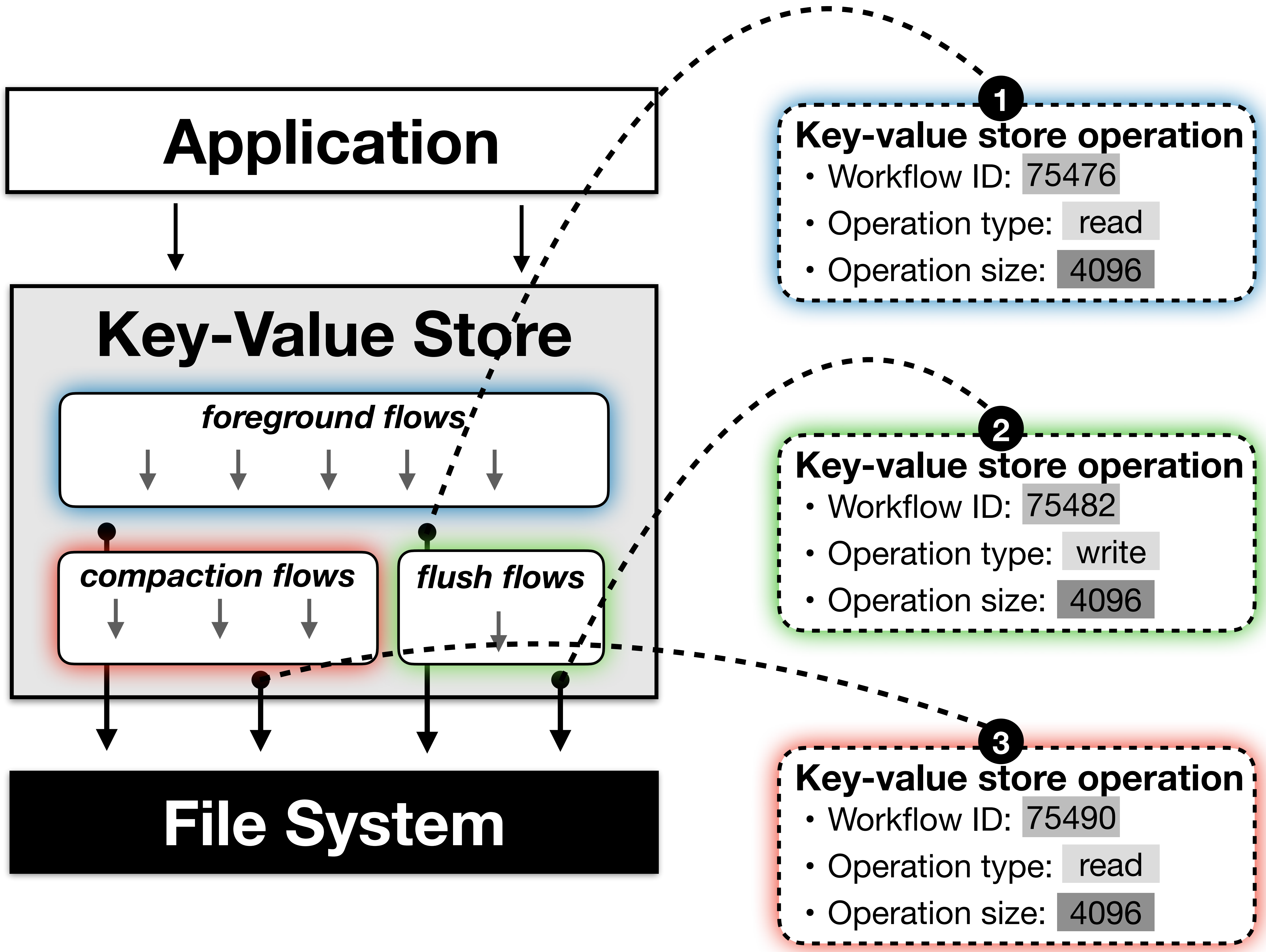
- Decoupled optimizations lose granularity and internal application knowledge
- I/O layers communicate through rigid interfaces
- Discard information that could be used to classify and differentiate requests



Challenge #2

Rigid interfaces

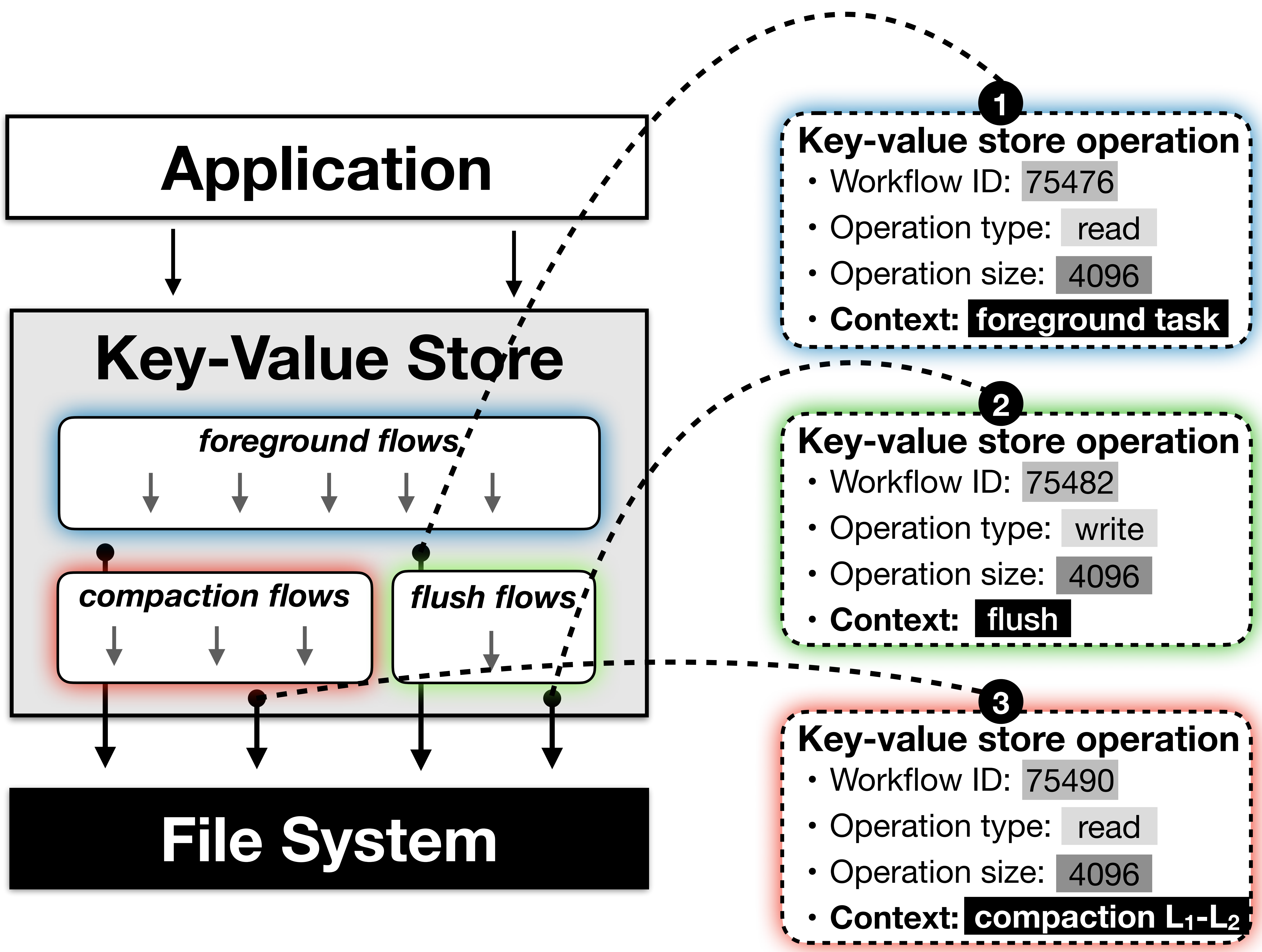
- Decoupled optimizations lose granularity and internal application knowledge
- I/O layers communicate through rigid interfaces
- Discard information that could be used to classify and differentiate requests



Challenge #2

Information propagation

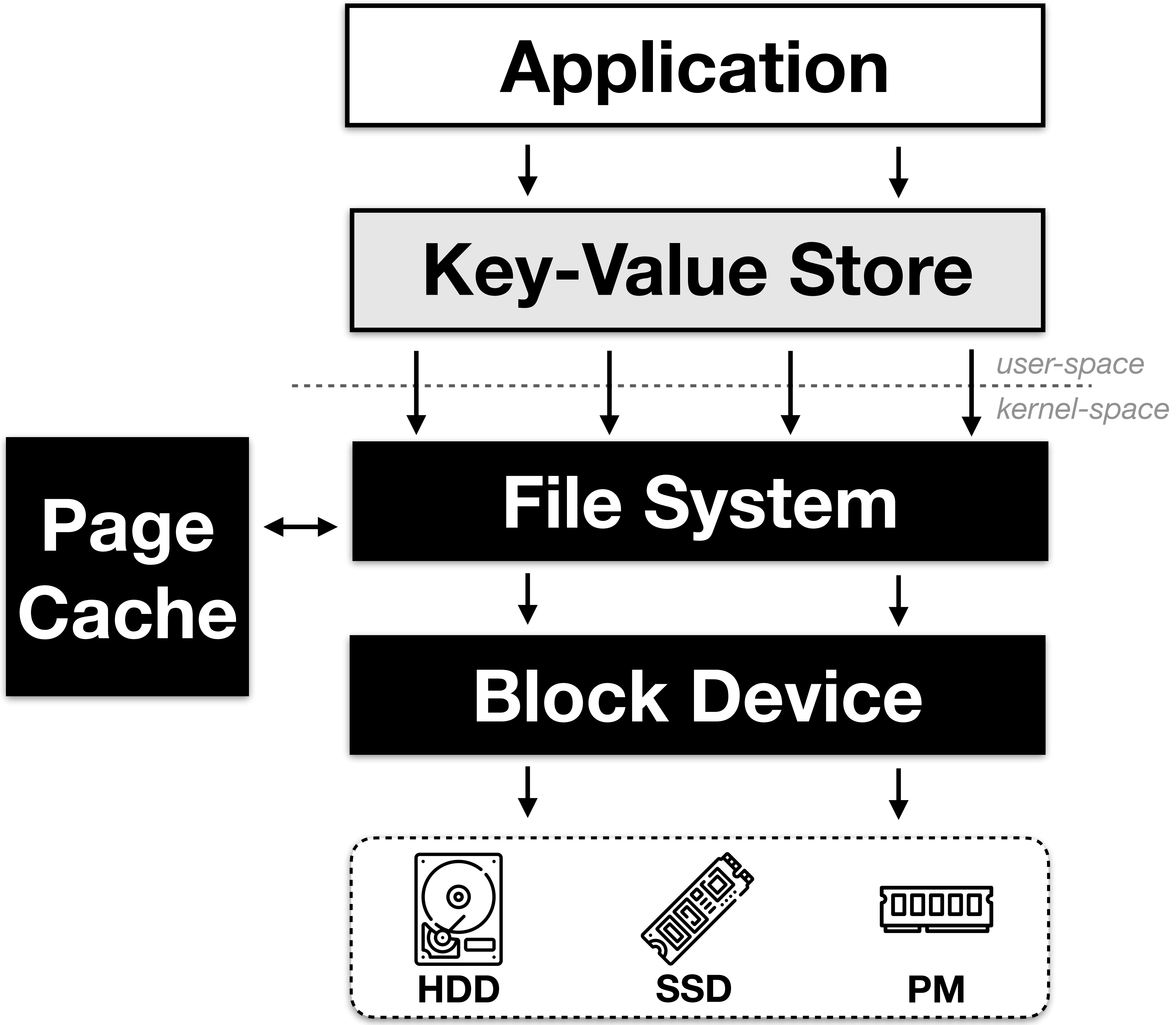
- Application-level information must be propagated throughout layers
- Decoupled optimizations can provide the same level of control and performance



Challenge #3

Kernel-level layers

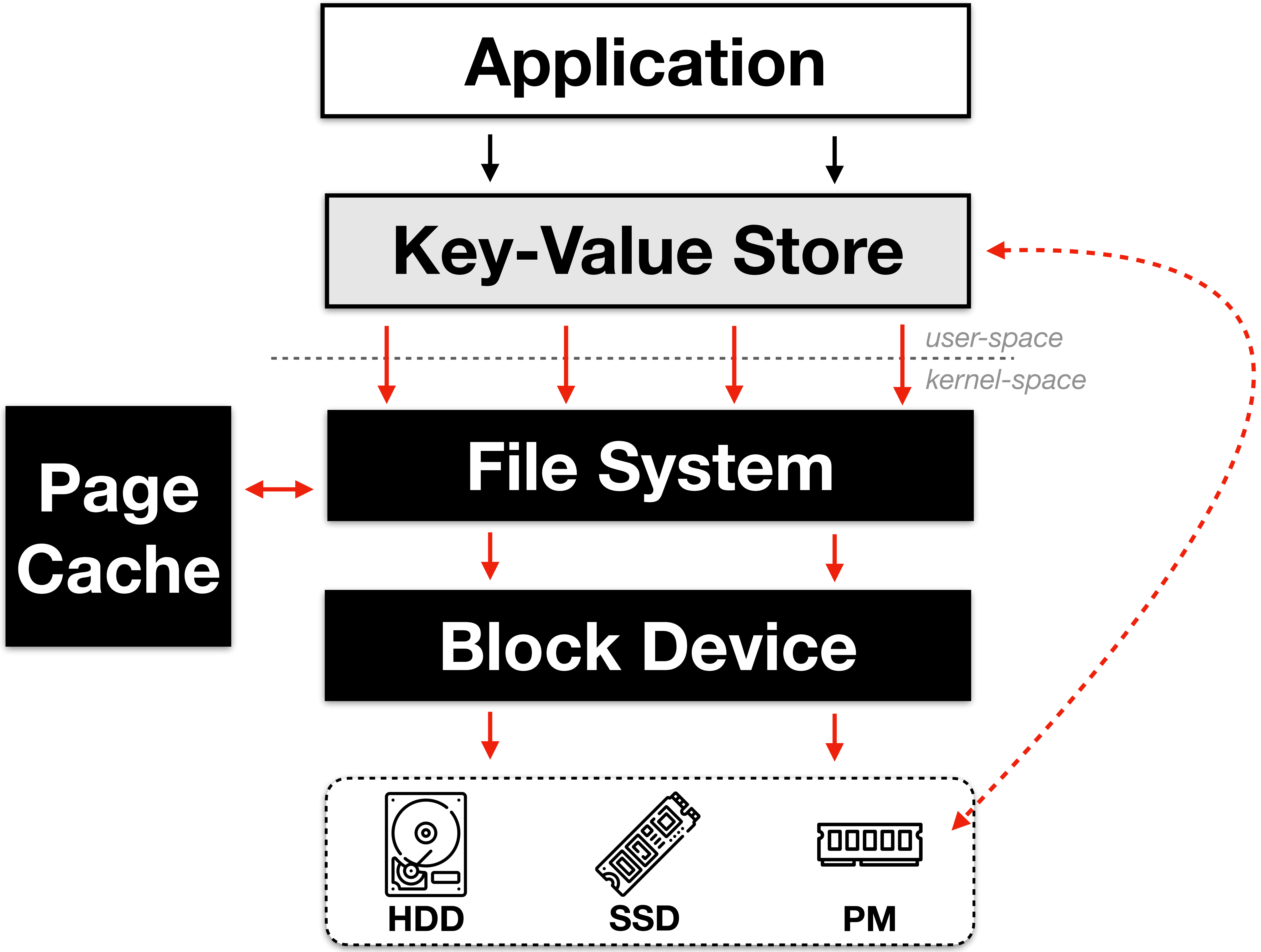
- Propagating context to kernel requires breaking user-to-kernel and kernel-internal APIs
- Kernel-level development is more restricted and error-prone
- Optimizations would be ineffective under kernel-bypass storage stacks (*e.g.,* SPDK, PMDK)



Challenge #3

Kernel-level layers

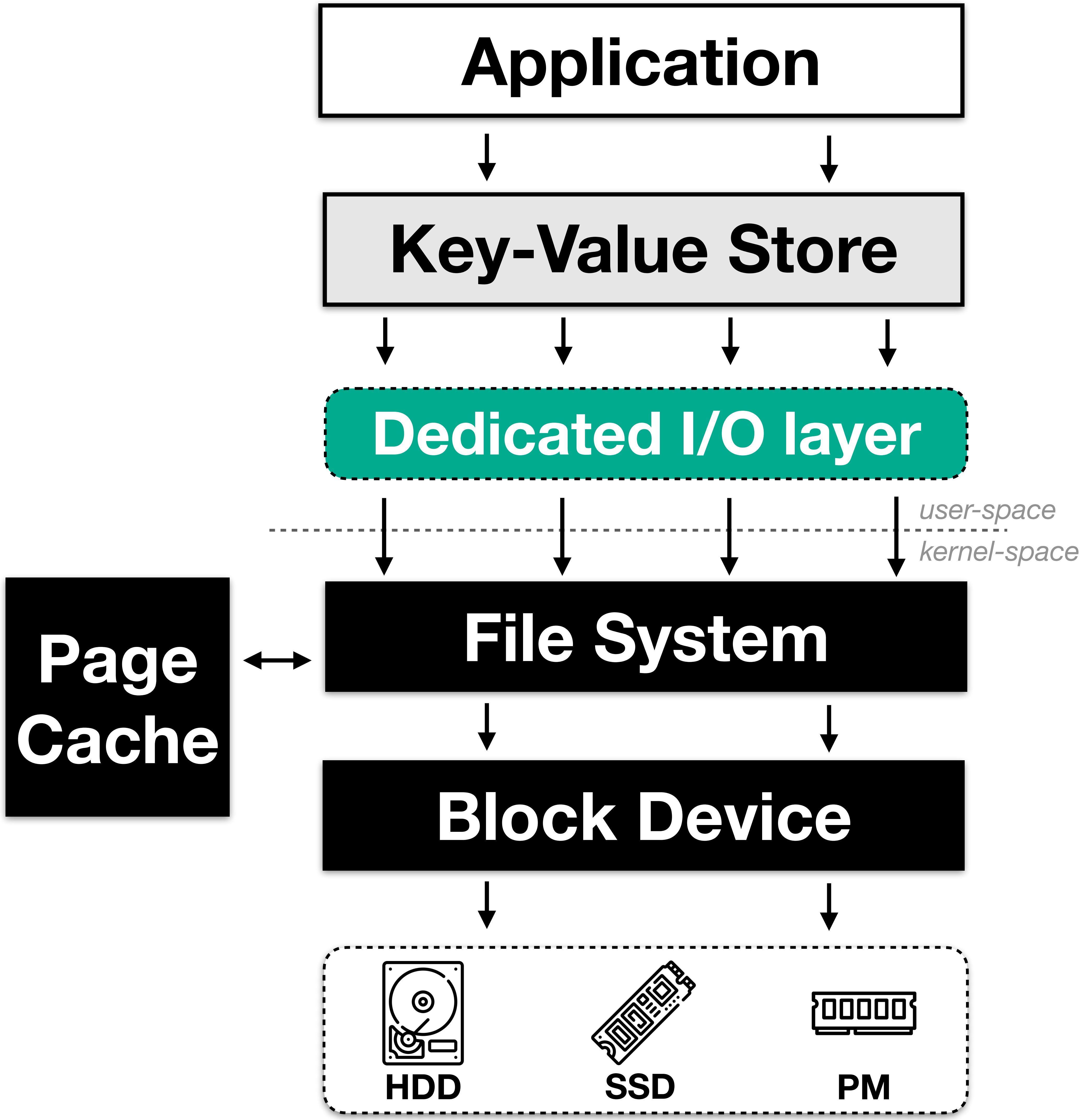
- Propagating context to kernel requires breaking user-to-kernel and kernel-internal APIs
- Kernel-level development is more restricted and error-prone
- Optimizations would be ineffective under kernel-bypass storage stacks (*e.g.*, SPDK, PMDK)



Challenge #3

Actuate at user-level

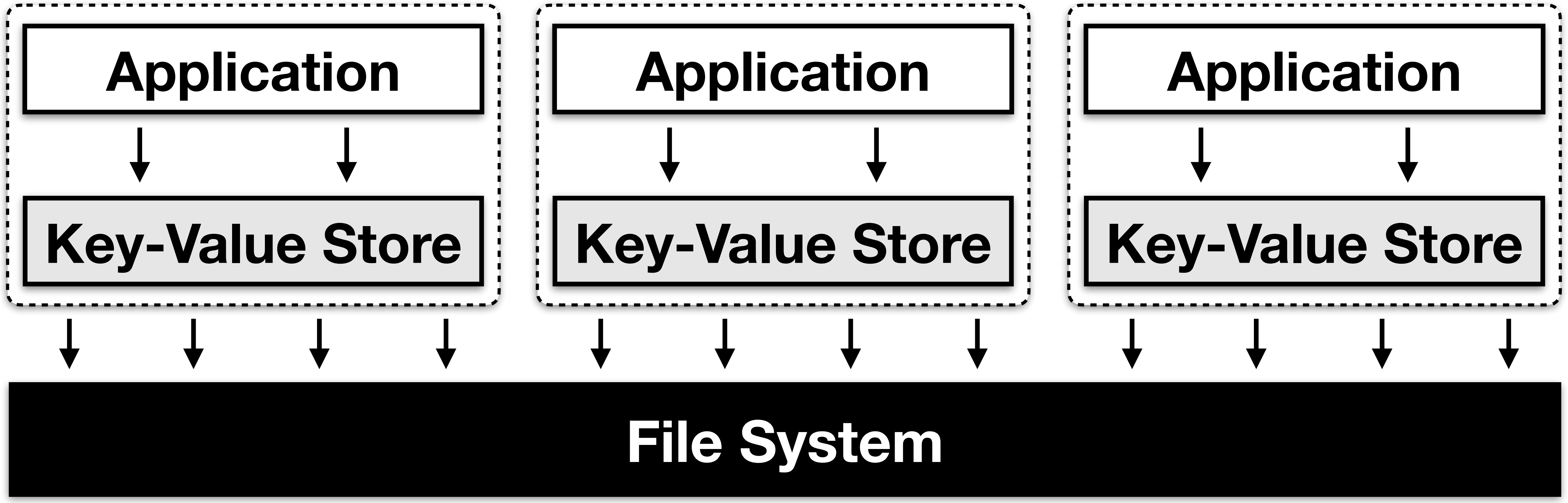
- Optimizations should be implemented at a dedicated user-level layer
- Promote portability across different systems and layers
- Ease information propagation throughout I/O layers



Challenge #4

Partial visibility

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

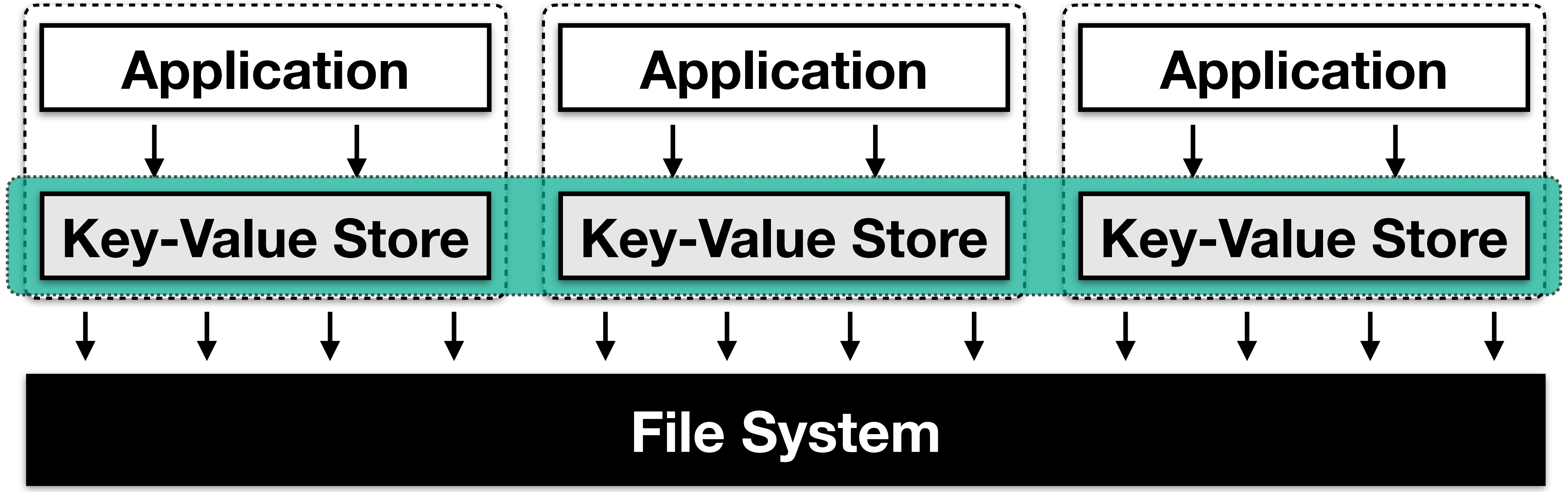


Note: the storage backend can either be local (*e.g.*, ext4, xfs) or distributed (*e.g.*, Lustre, GPFS)

Challenge #4

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

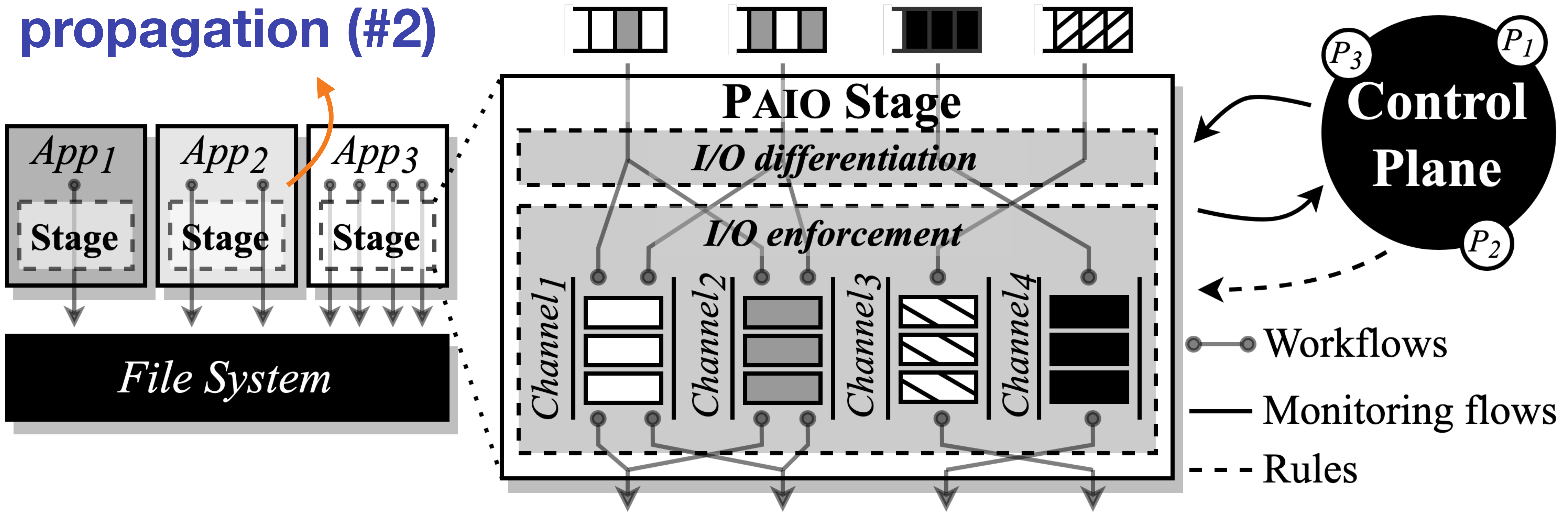
- **User-level** framework for building **portable** and **generally applicable** optimizations
- Adopts ideas from **Software-Defined Storage** [6]
  - I/O optimizations are implemented ***outside*** applications as **data plane stages**
  - **Stages** are controlled through a **control plane** for 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

Information propagation (#2)

Global visibility (#4)



User-level (#3)

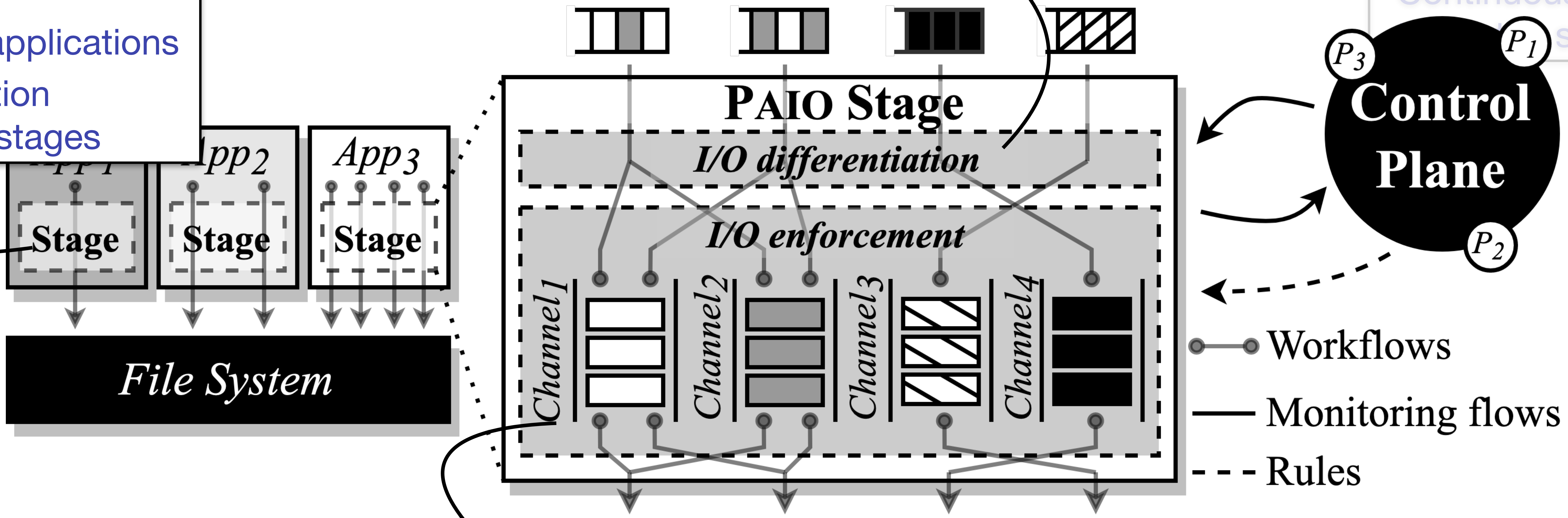
Dedicated I/O layer (#1)

PAIO design

Differentiation module
- Classifies and differentiates requests
- Context propagation
- Different levels of differentiation

Control interface
- Enables the control plane to orchestrate the stage
- Continuously monitors and fine-tunes the stage

Stage interface
- Access point for applications
- Forwards application requests to PAIO stages



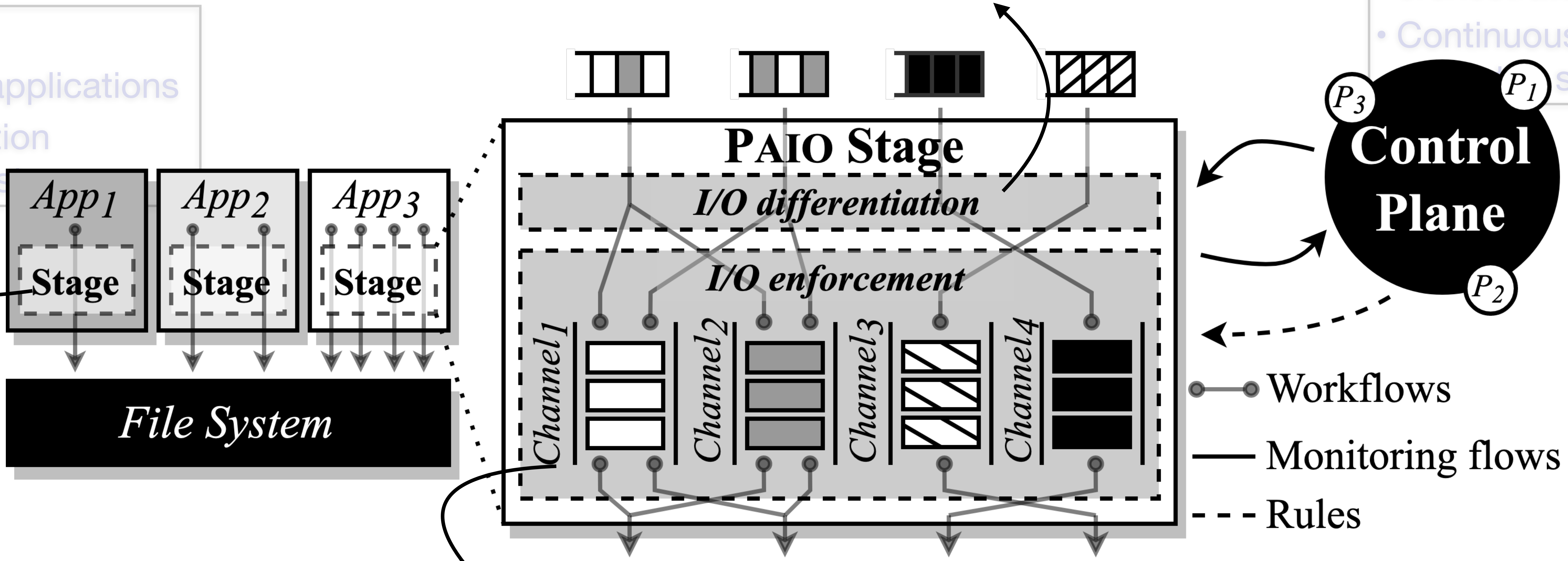
Enforcement module
- Applies the actual I/O mechanisms over requests
- Organized in Channels and Enforcement objects

PAIO design

Differentiation module
- Classifies and differentiates requests
- Context propagation
- Different levels of differentiation

Stage interface
- Access point for applications
- Forwards application requests to PAIO s

Control interface
- Enables the control plane to orchestrate the stage
- Continuously monitors and fine- stage



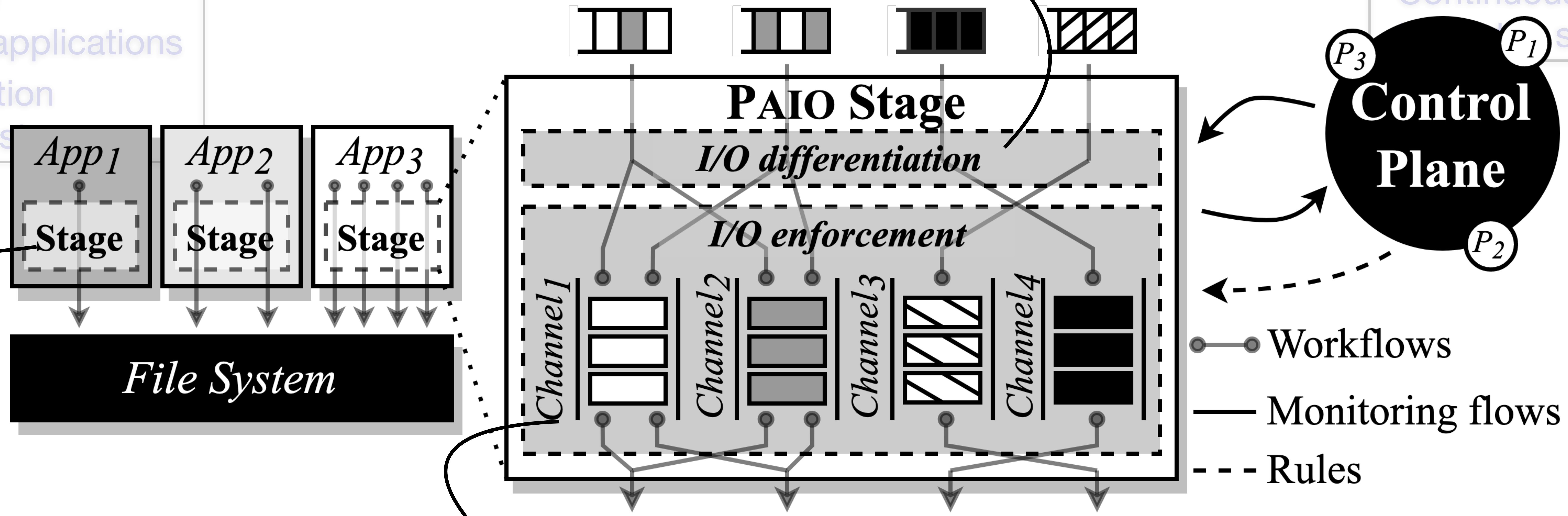
Enforcement module
- Applies the actual I/O mechanisms over requests
- Organized in Channels and Enforcement objects

PAIO design

Differentiation module
- Classifies and differentiates requests
- Context propagation
- Different levels of differentiation

Stage interface
- Access point for applications
- Forwards application requests to PAIO s

Control interface
- Enables the control plane to orchestrate the stage
- Continuously monitors and fine- stage



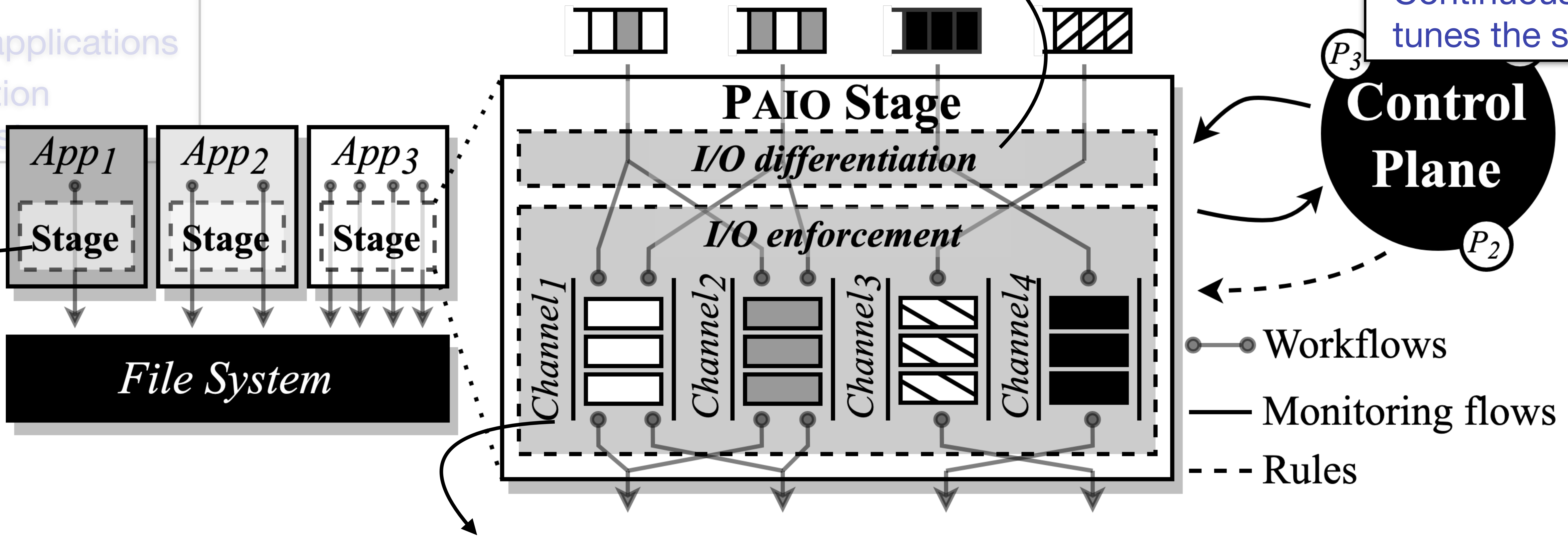
Enforcement module
- Applies the actual I/O mechanisms over requests
- Organized in Channels and Enforcement objects

PAIO design

Stage interface
- Access point for applications
- Forwards application requests to PAIO s

Differentiation module
- Classifies and differentiates requests
- Context propagation
- Different levels of differentiation

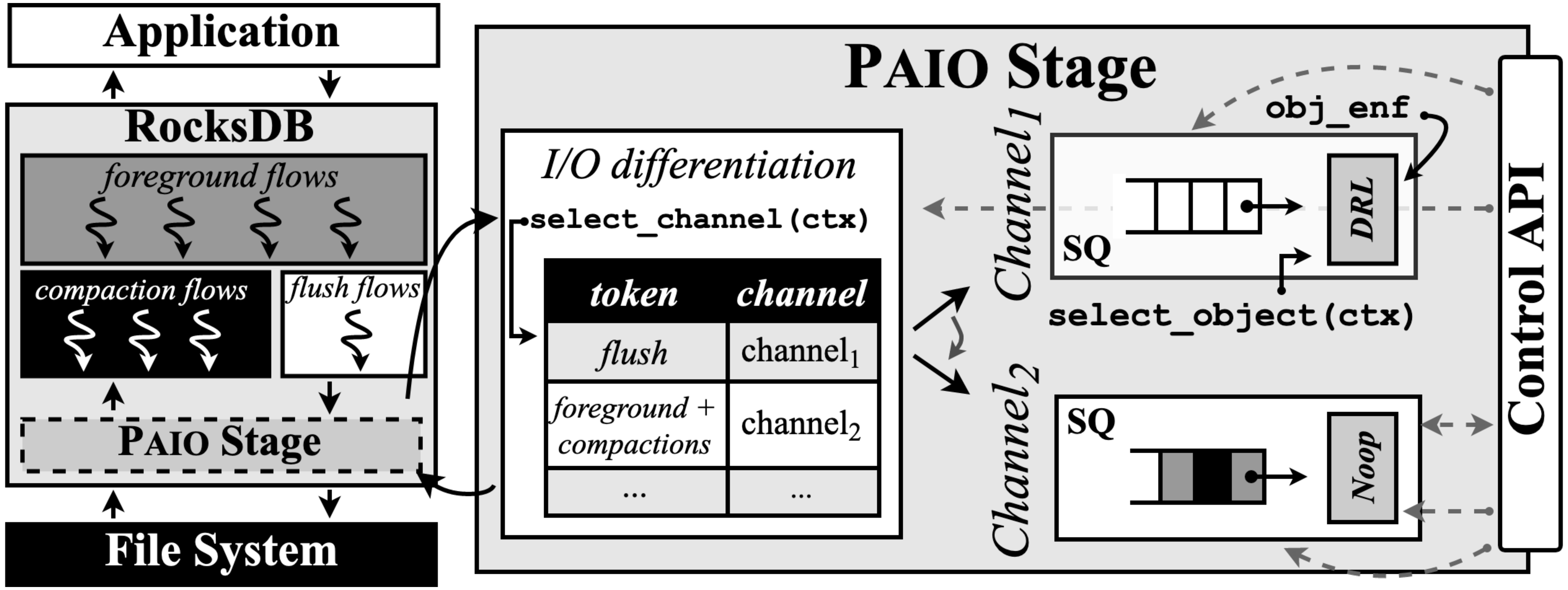
Control interface
- Enables the control plane to orchestrate the stage
- Continuously monitors and fine-tunes the stage



Enforcement module
- Applies the actual I/O mechanisms over requests
- Organized in Channels and Enforcement objects

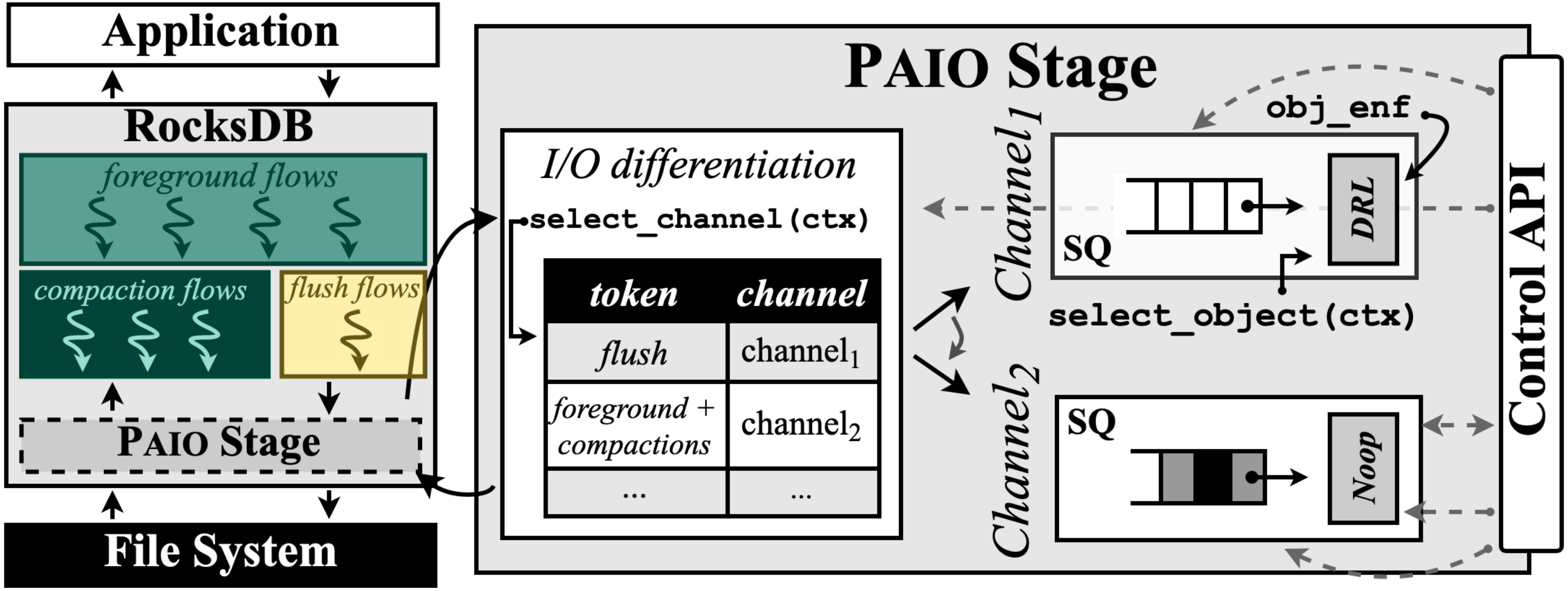
PAIO design

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



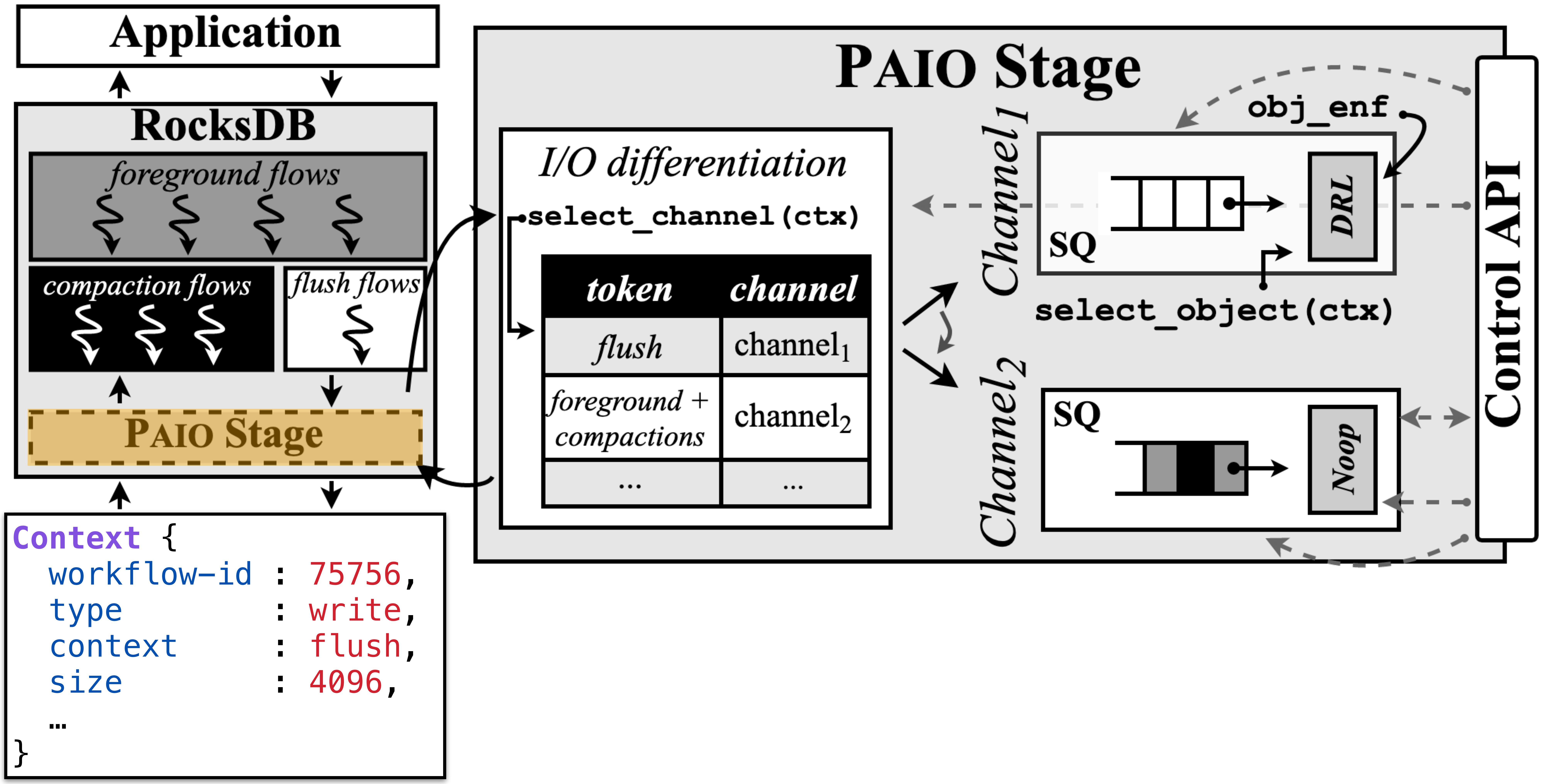
Policy: *limit the rate of RocksDB's flush operations to X MiB/s*

I/O differentiation

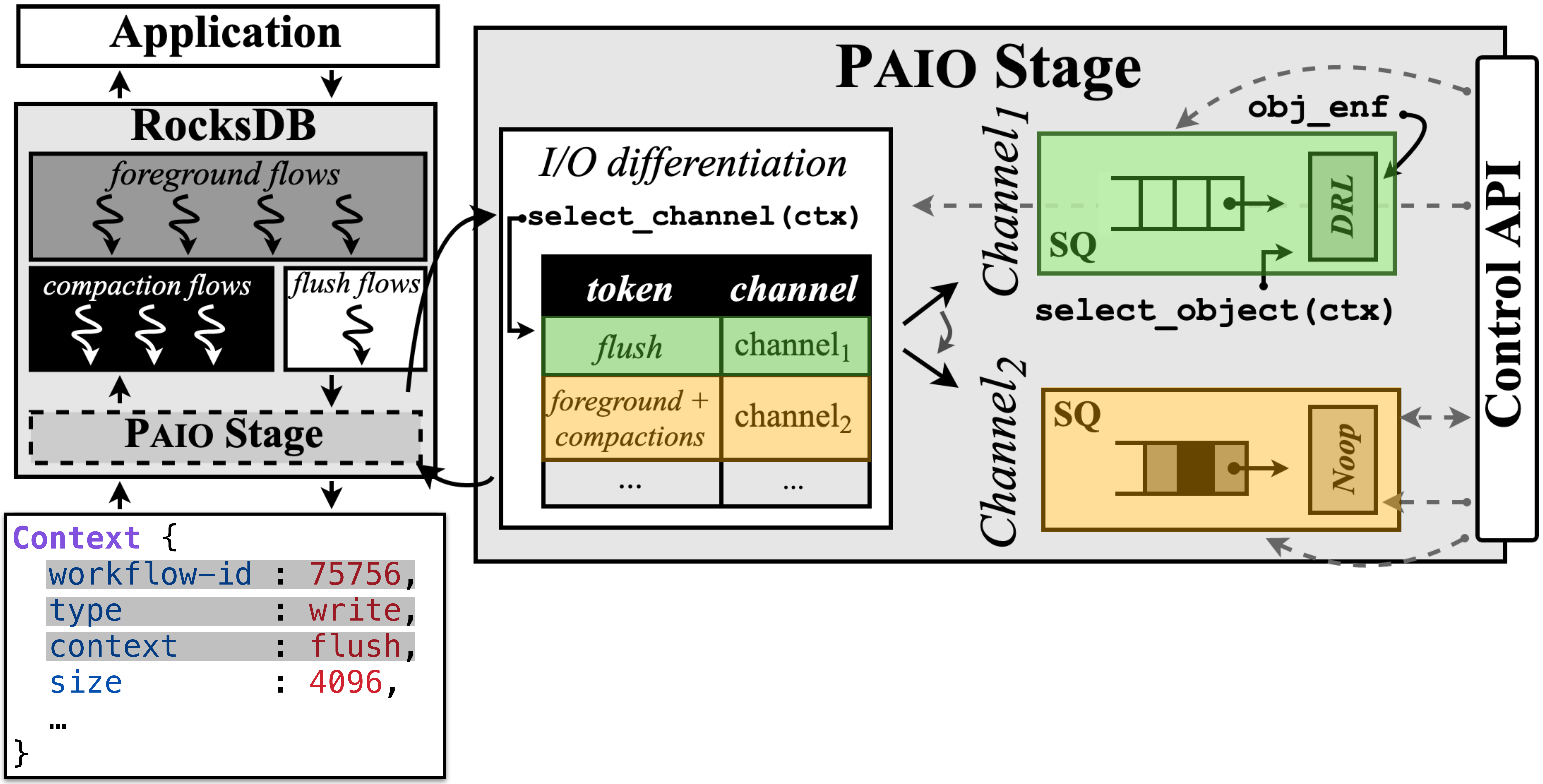


Identify the origin of POSIX operations (i.e., **foreground**, **compaction**, or **flush** operations)

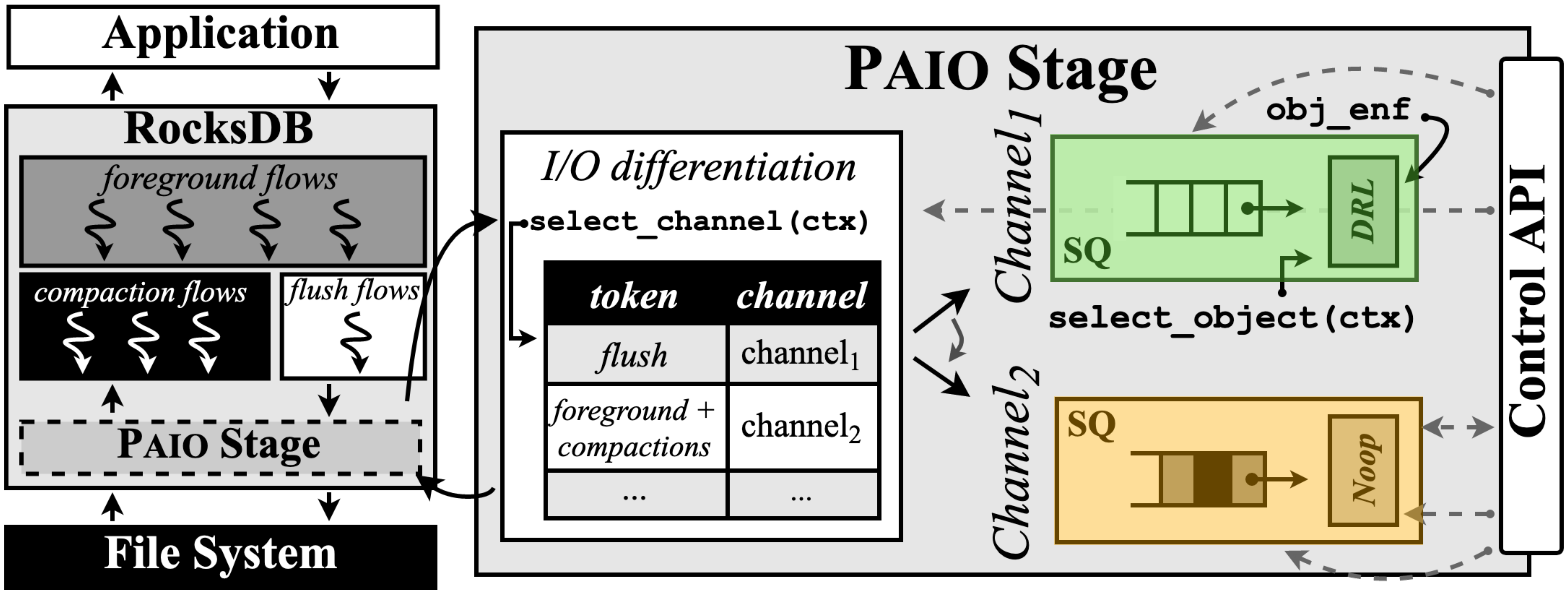
I/O differentiation



I/O differentiation

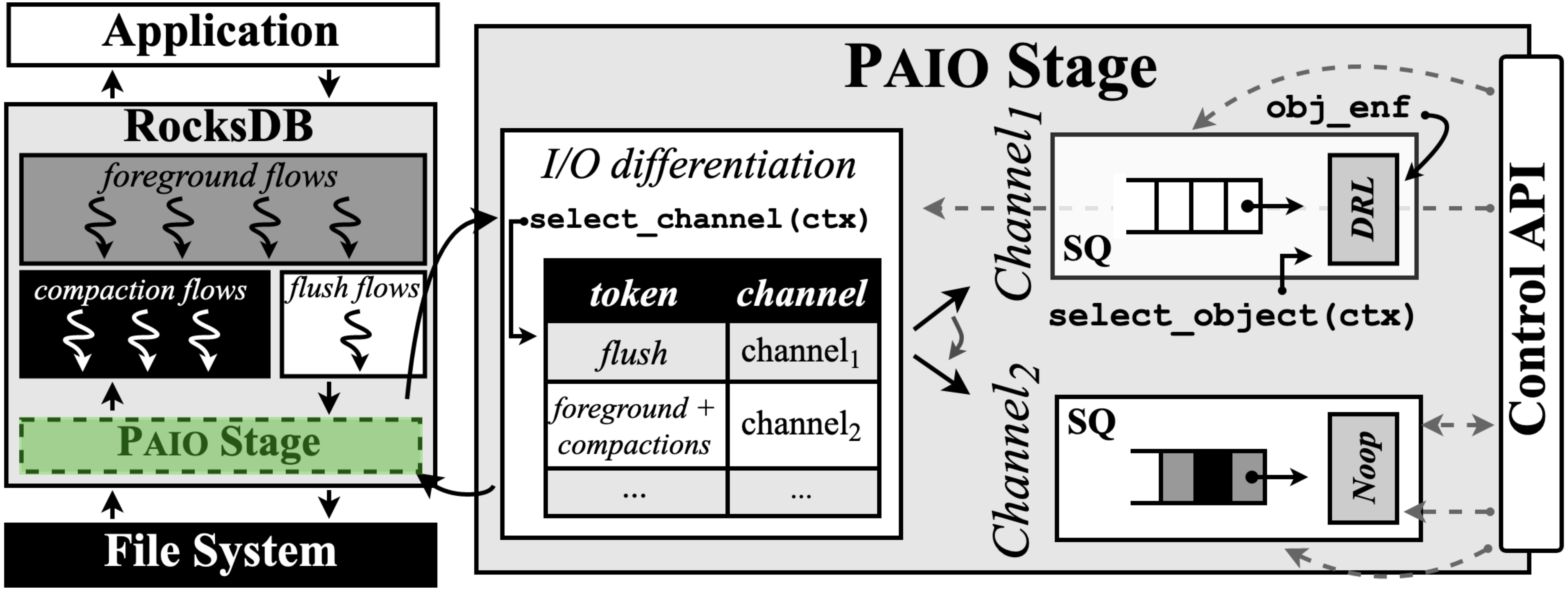


I/O enforcement



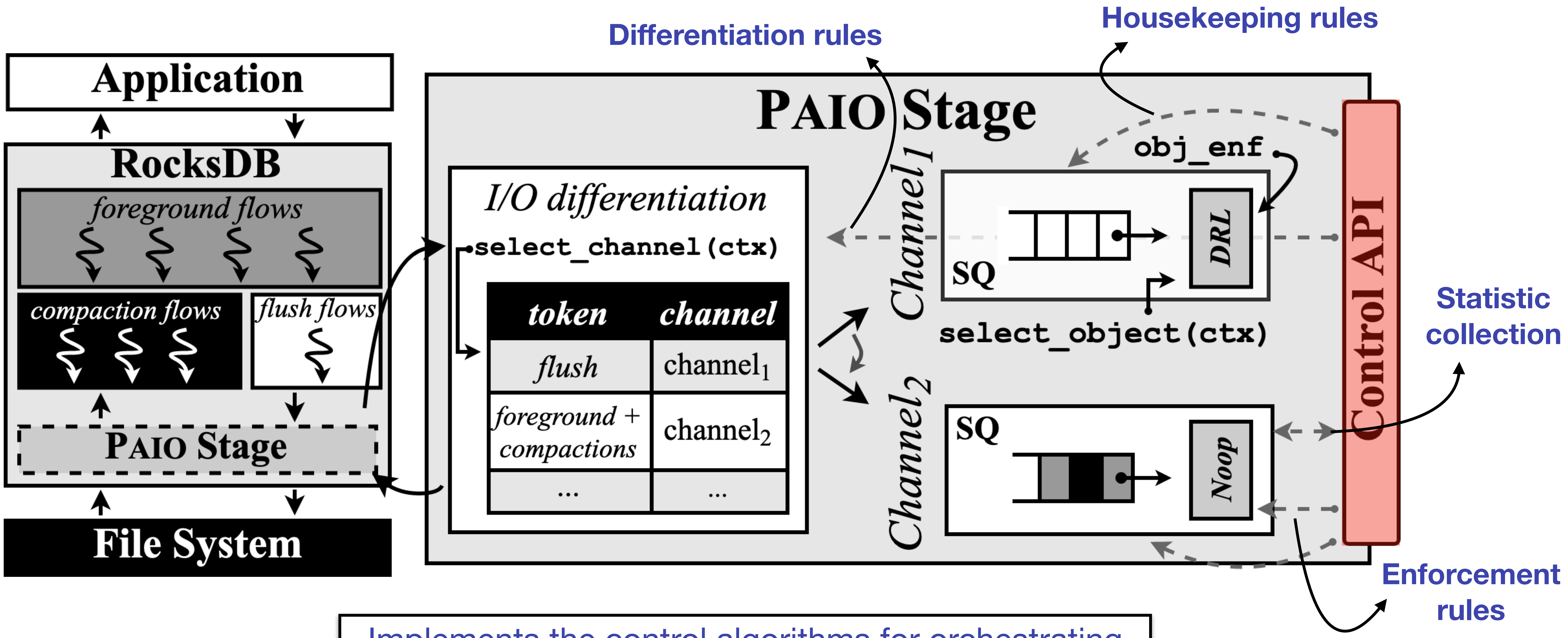
PAIO currently supports **Noop** and **DRL** enforcement objects

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)

Part 3

building storage data planes

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 anc

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

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

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 and sustained tail latency

Mixture workload
50% read 50% write

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

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 still **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

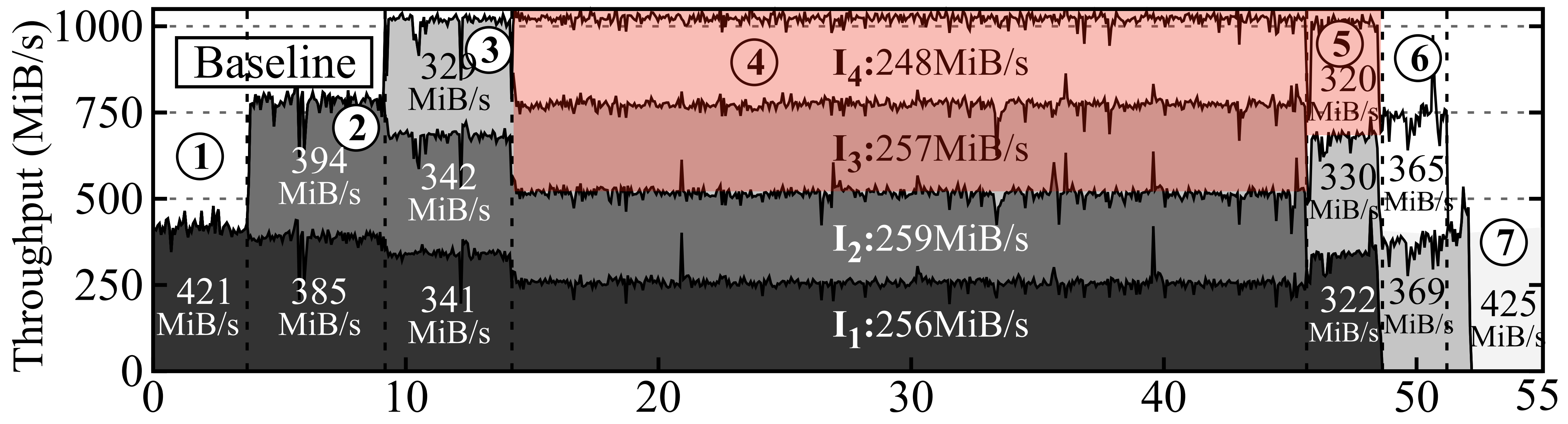
- cgroup's block I/O controller allows static rate limiting read and write operations
- Adjusting the rate requires stopping and restarting jobs
- Cannot leverage from leftover bandwidth

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}

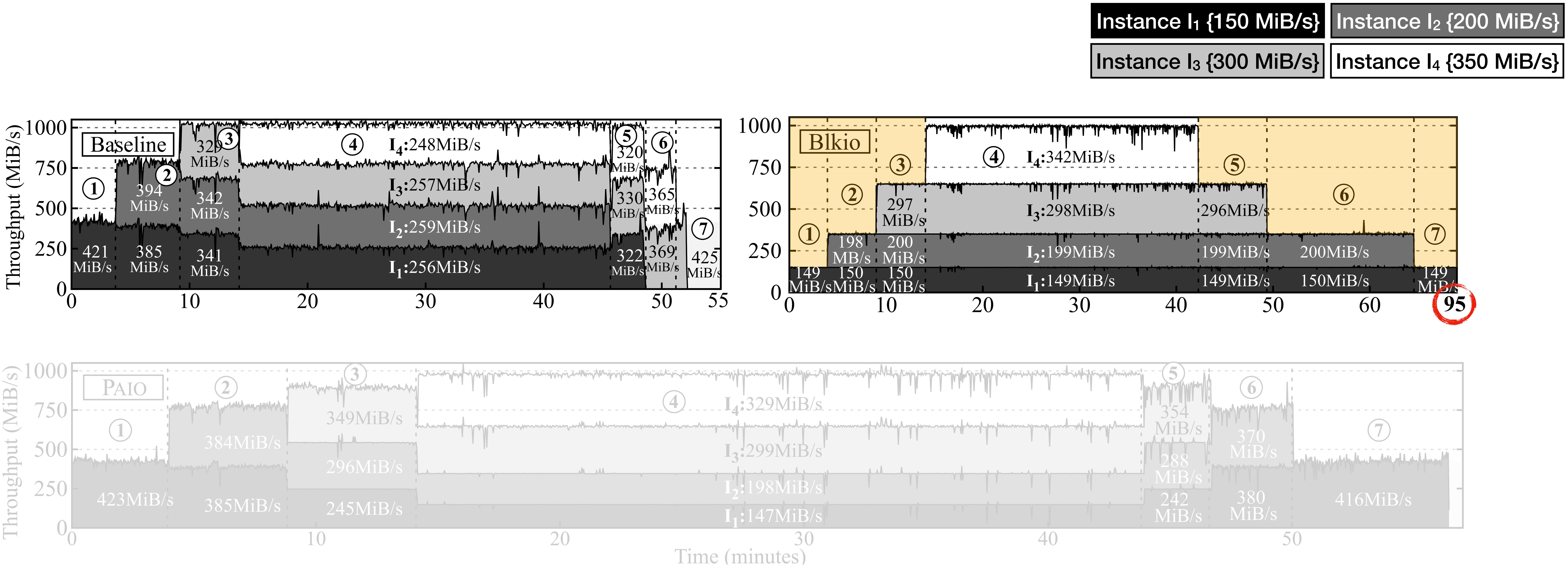


System configuration and workload

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

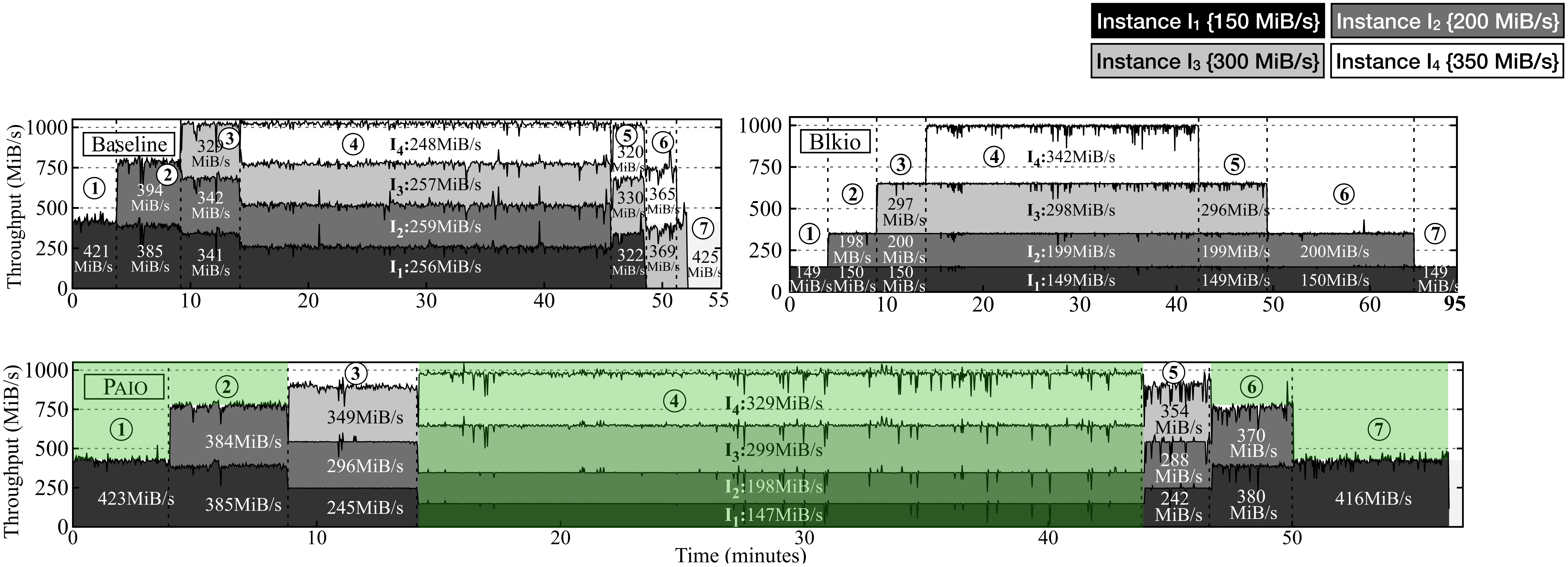
I_3 and I_4 cannot meet their bandwidth targets during 31 and 34 minutes

Per-application bandwidth control



Instances cannot be dynamically provisioned with available disk bandwidth

Per-application bandwidth control

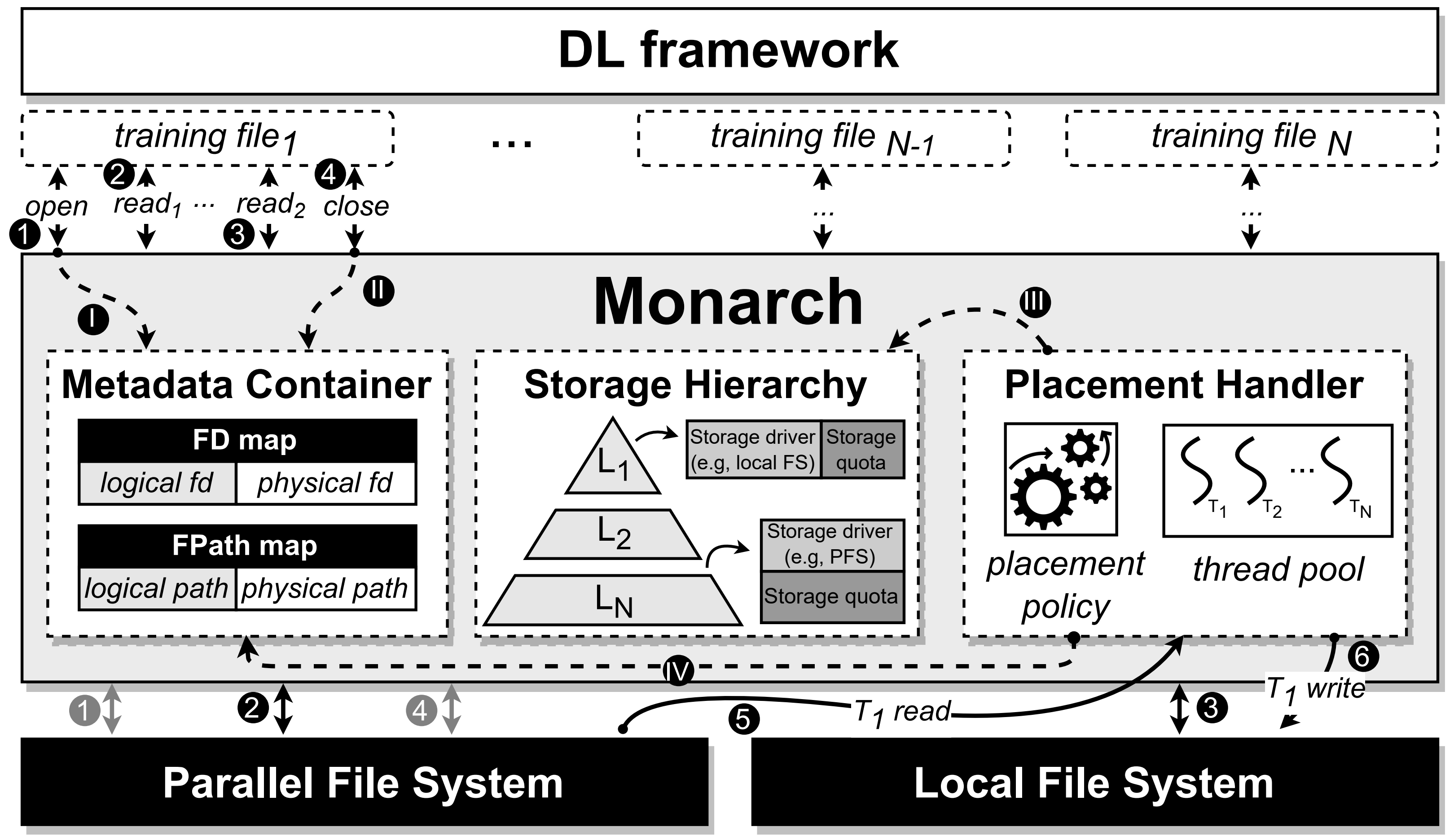


PAIO ensures that policies are met at all times, and whenever leftover bandwidth is available, PAIO shares it across active instances

Storage data planes for deep learning

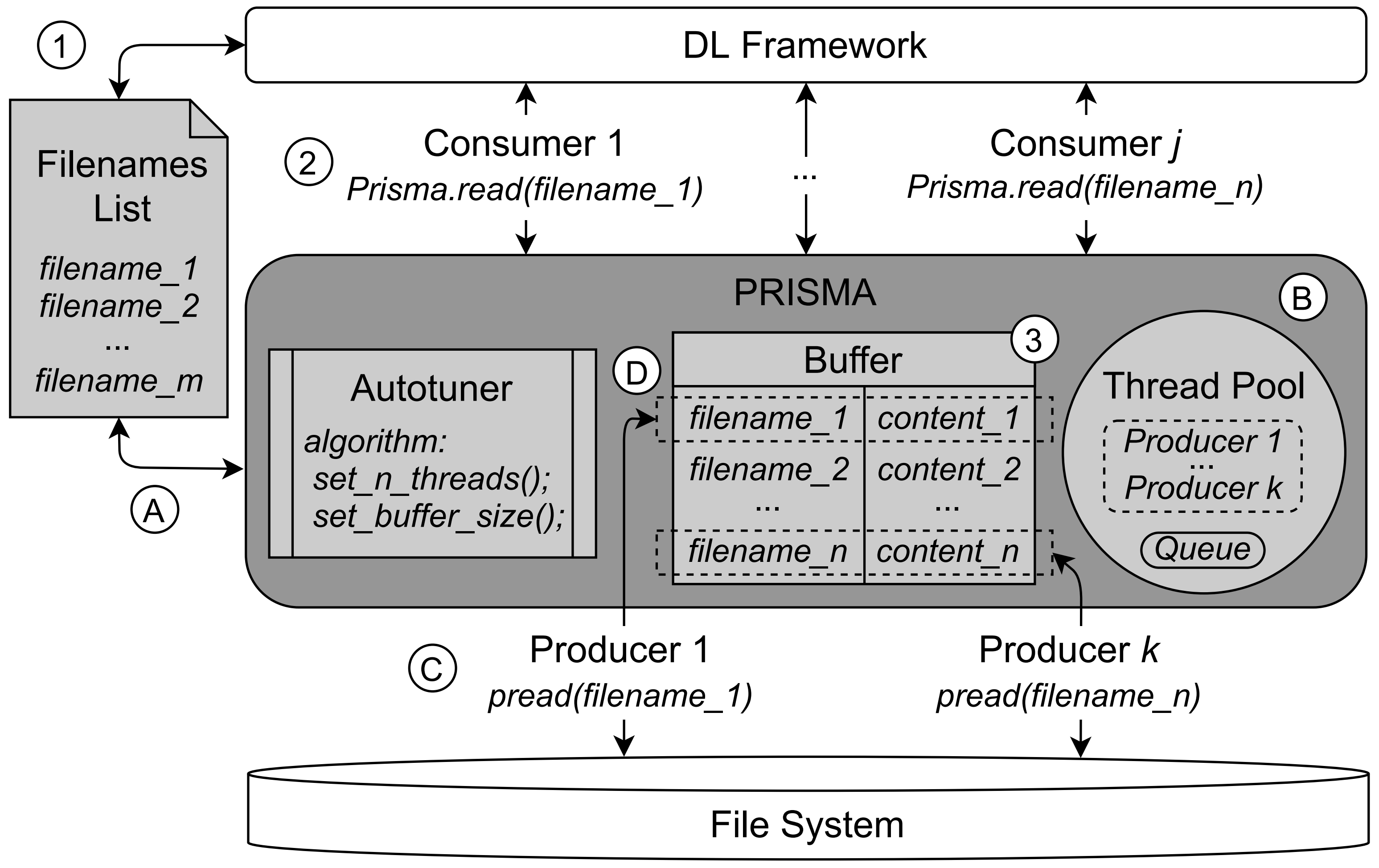
Storage tiering (Monarch)

- 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 …

Q & A

Ricardo Macedo

ricardo.g.macedo@inesctec.pt

github.com/dsrhaslab

dsr-haslab.github.io