A Case for Dynamically Programmable Storage Background Tasks

Ricardo Macedo, Alberto Faria, João Paulo, José Pereira
INESC TEC & University of Minho

38th IEEE International Symposium on Reliable Distributed Systems Workshops
1st Workshop on Distributed and Reliable Storage Systems
Lyon, France October 1st 2019

Motivation and Background

Modern storage infrastructures feature **long** and **complex I/O paths**

- Composed by several layers, such as *hypervisors*, *schedulers*, *databases*, and *file systems*

Layers employ **independent optimizations** to serve applications

- **Partial visibility** of the infrastructure **inhibits** optimal system-wide **performance**
- High levels of **I/O interference** and **performance degradation**
- Degradation amplifies when **concurrent I/O services** compete for shared resources

Motivation and Background

Background tasks are predefined I/O tasks that can rapidly **overload shared resources**

- Compaction, checkpointing, and replication
- Introduce significant **I/O interference** and **workload burstiness**
- Processed in best-effort manner to minimize interference with foreground workflows

The **decision** of **when** and **how** to execute such operations is taken by the **layer** itself, regardless of the overall load on the infrastructure

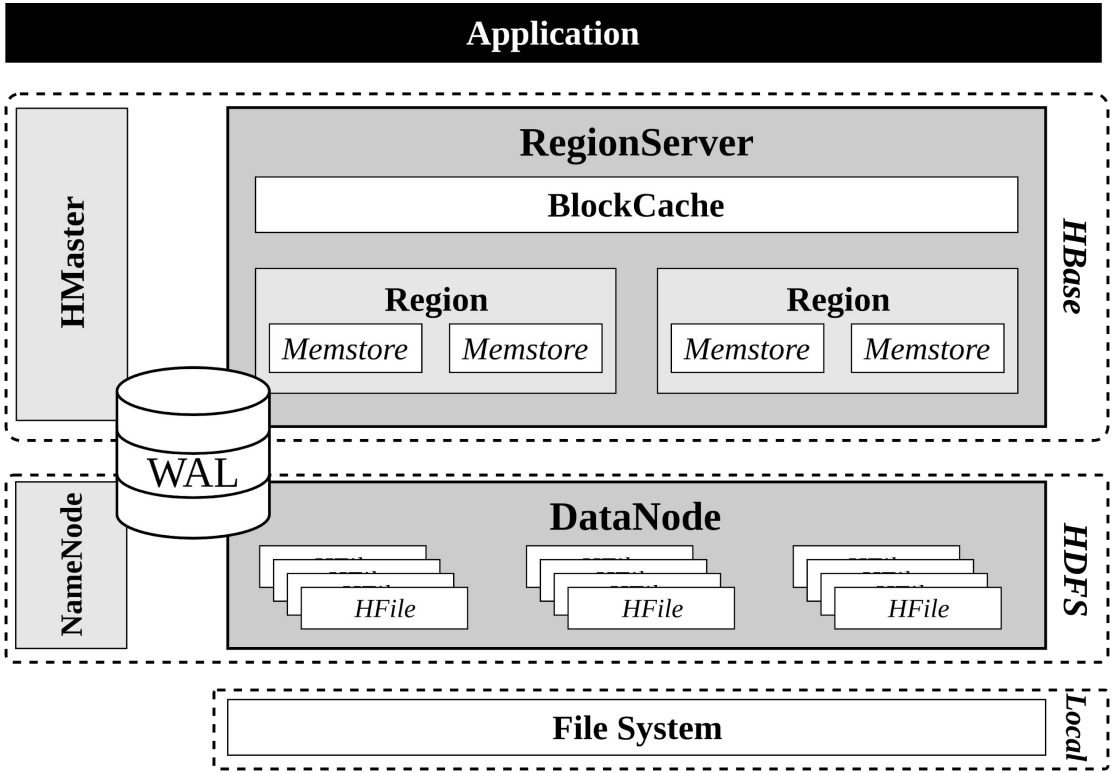
To achieve **optimal holistic performance**, storage background tasks should be **dynamically programmable** and their execution handled in **end-to-end** fashion.

Contributions

- Extensive evaluation of the impact of storage background tasks
  - Impact of **compaction** processes under **HBase**
  - Impact of **checkpointing** processes under **PostgreSQL**
  - Analysis of **mean** and **tail latencies** performance

- Design of a **programmable** storage system to achieve **optimal holistic performance**

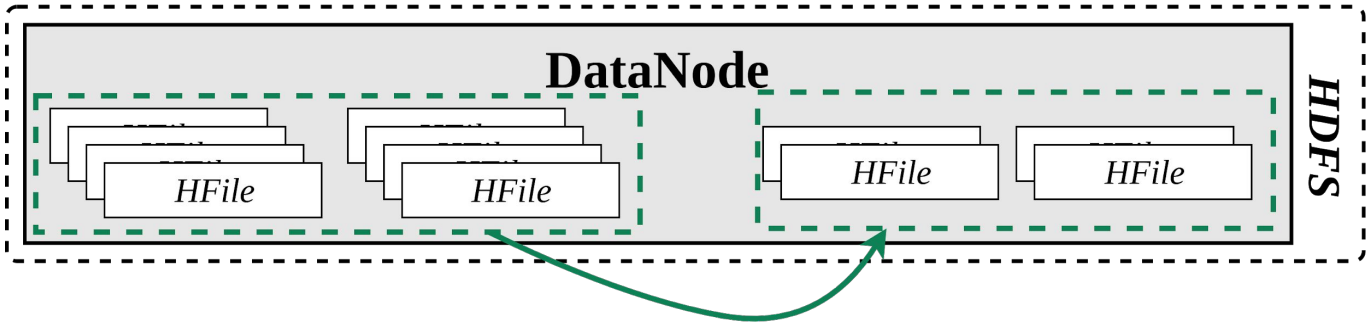
Case Study: HBase



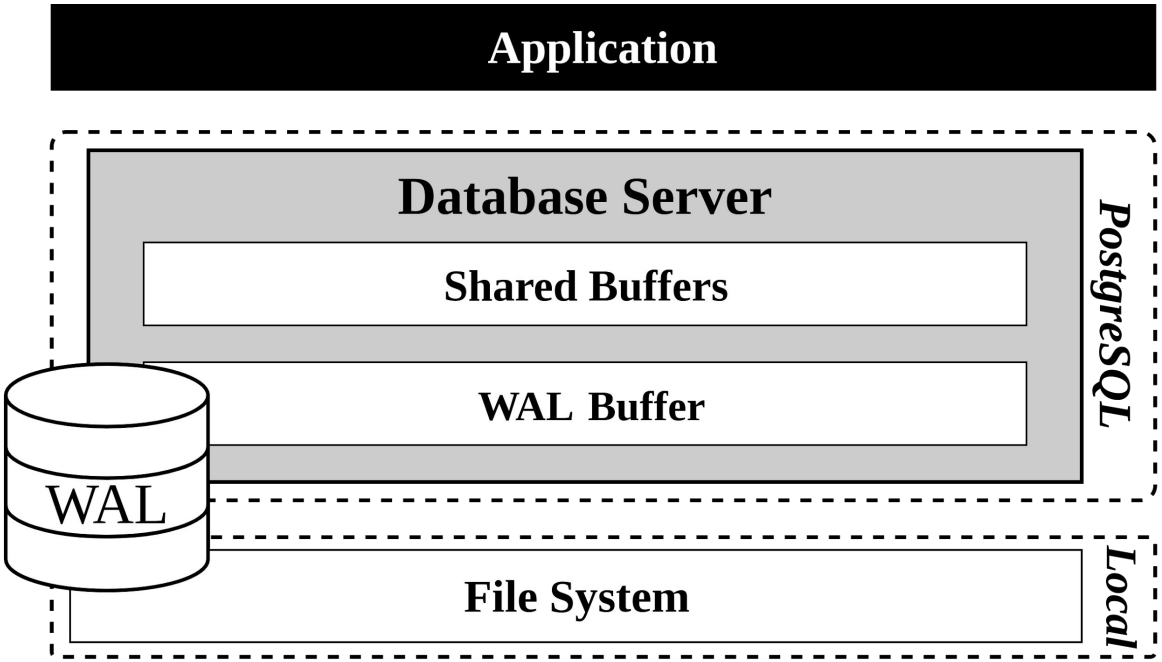
Case Study: HBase

LSM-based design leads to the execution of **background compactions**

- Merges several small-sized HFiles into fewer larger ones
- While improving read performance, compactions introduce I/O interference and burstiness



Case Study: PostgreSQL



Case Study: PostgreSQL

To truncate the log and allow **fast recovery**, PostgreSQL performs **checkpoints**

- Flush dirty data pages in *shared buffers* to disk
- Executed either when the **WAL** file is about to **exceed a certain size** or upon a **timeout**
- **Checkpoint completion target** adjusts the **throughput** at which checkpoints are made
- **Lower bound** leads to **I/O burstiness** and **higher bound** to **longer recovery** times

Methodology

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

Methodology

Testbed

- HBase 2.0.5 pseudo-distributed mode, backed by HDFS 2.9.2
- PostgreSQL 11.3 backed by an ext4 file system

Workloads*

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

* Workloads previously used in [22]: *"MeT: Workload aware elasticity for NoSQL"*

Results publicly available at https://rgmacedo.github.io/drss19-website/

Methodology

HBase deployments

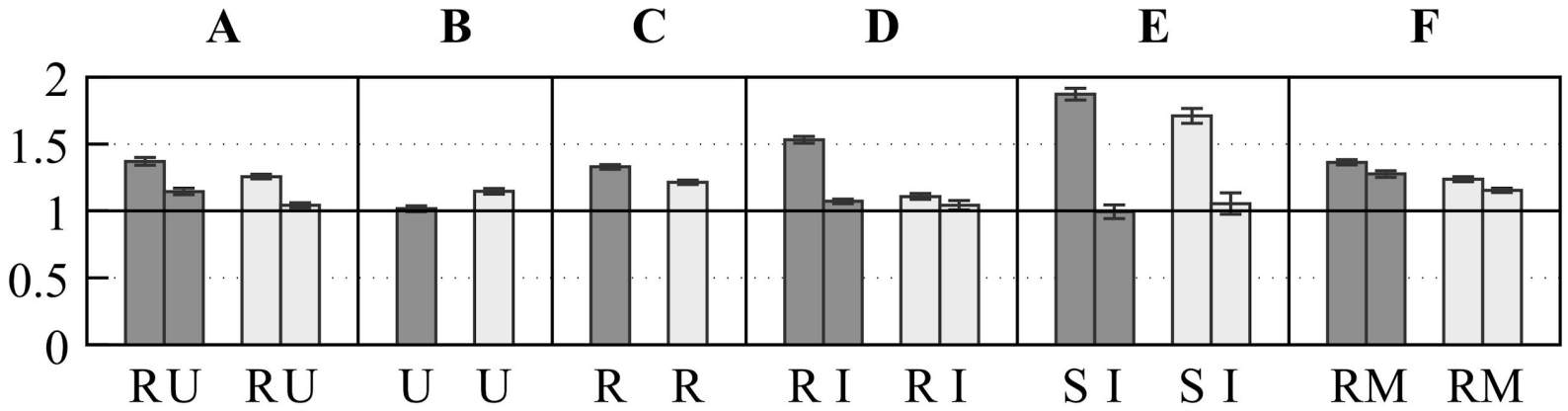
- **With compaction effects.** Execution phase immediately after loading phase
- **Without compaction effects.** Execution phase after a waiting period

PostgreSQL deployments

- 6 configurations with varying **WAL size** and **checkpoint completion target**
- 128MiB and 1024MiB for **maximum WAL size** parameter
- 0.1, 0.5, and 0.9 for **checkpoint completion target** parameter

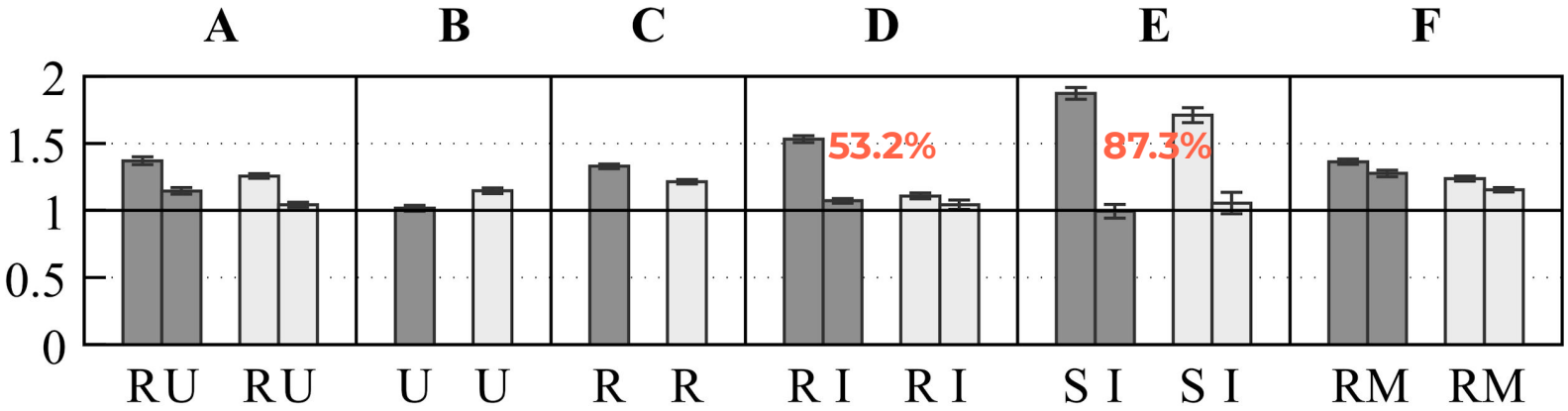
HBase Compactions

Mean latency results



HBase Compactions

Mean latency results

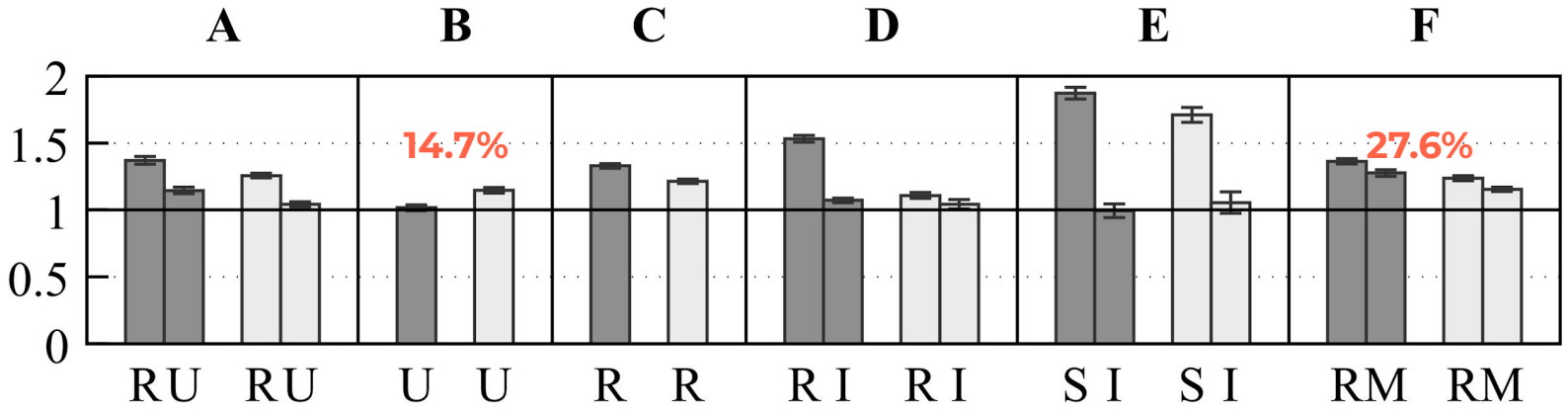


Inherent **dependency over files** being compacted results in **high I/O interference** over read-oriented requests

Read-oriented operations exhibit an **overhead** of at most **955.2%** at the 99th percentile latency

HBase Compactions

Mean latency results



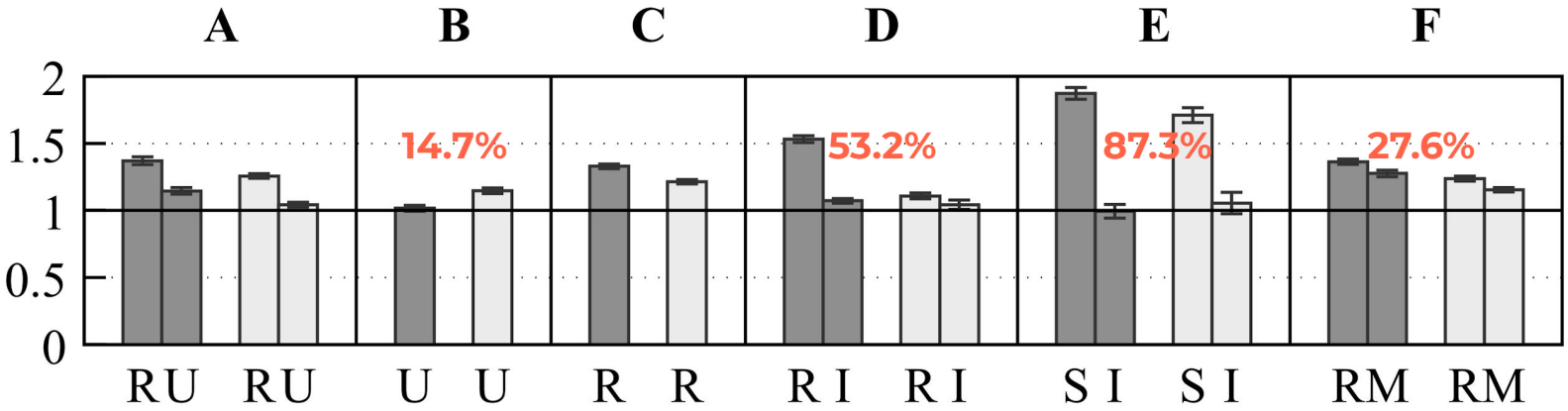
Write operations are **sequentially written** to the WAL and then persisted at the Memstore

Experienced compactions do not impose major disk overload and I/O interference

Performance degradations of at most **40%** at the 99th percentile latency for write operations

HBase Compactions

Mean latency results



CPU utilization remains mostly unaltered

Read and write **disk throughput** experience an **increase** of **33** and **15MiB/s**, respectively

Default **throttling policy limits compaction throughput** to reduce I/O interference

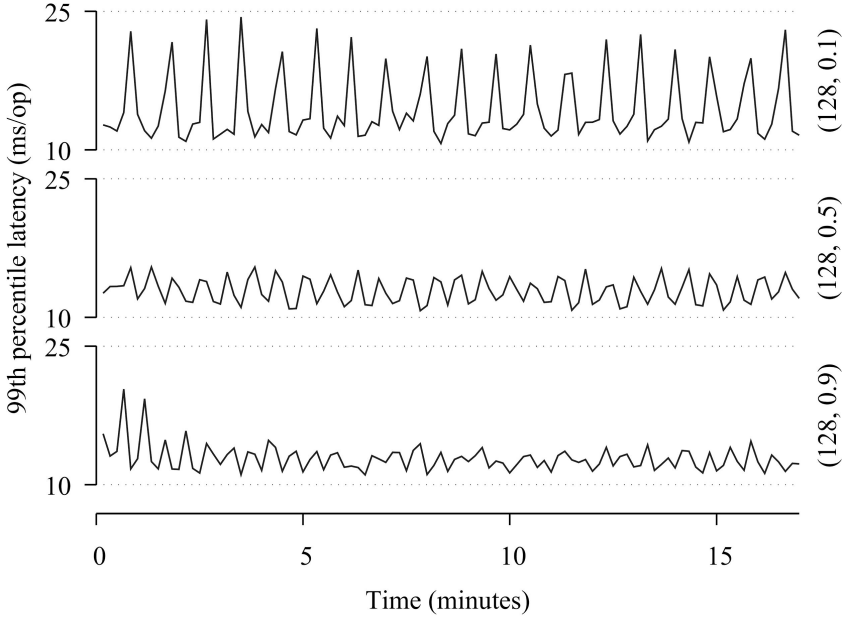
PostgreSQL Checkpointing

Tail latency results

At 99th percentile latency, 0.1, 0.5, and 0.9 completion target configurations experience an **overhead** of at most **61.9%**, **33.2%**, and **33.3%**, respectively

Lower values result in **I/O burstiness**, inhibiting QoS provisioning and sustained performance

Larger values **throttle** WAL write **performance**, occupying disk bandwidth for longer periods



99th percentile latency variation for the update operation for each PostgreSQL configuration, under *Workload A* with 1 thread

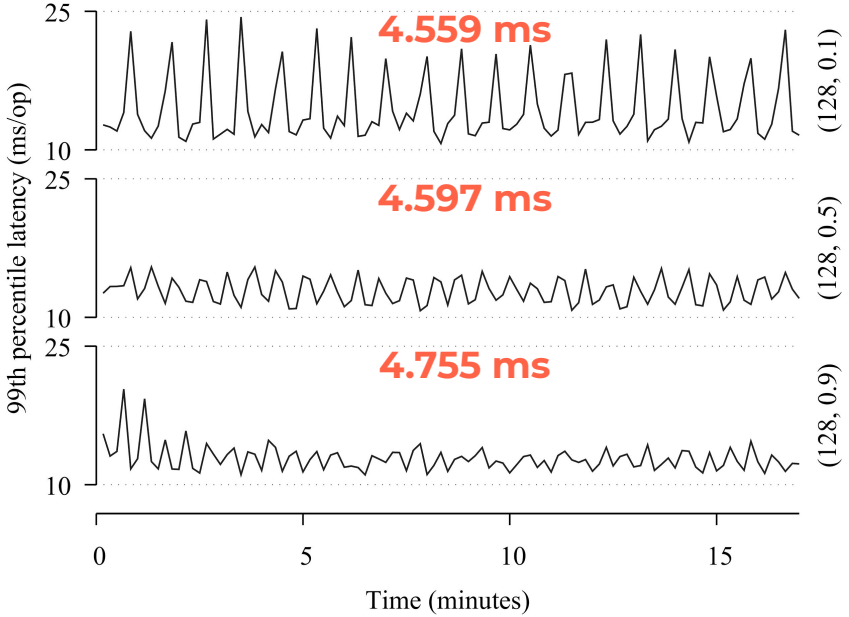
PostgreSQL Checkpointing

Tail latency results

At 99th percentile latency, 0.1, 0.5, and 0.9 completion target configurations experience an **overhead** of at most **61.9%**, **33.2%**, and **33.3%**, respectively

Lower values result in **I/O burstiness**, inhibiting QoS provisioning and sustained performance

Larger values **throttle** WAL write **performance**, occupying disk bandwidth for longer periods



99th percentile latency variation for the update operation for each PostgreSQL configuration, under *Workload A* with 1 thread

Discussion

Compaction and checkpointing heavily **impact** foreground tasks performance

HBase **throttles** compaction throughput

- Does not provide the building block to dynamically adapt such settings
- Applications experience compaction effects for longer periods of time

PostgreSQL **cannot dynamically adjust** checkpointing activity to the overall load of the infrastructure

Storage tasks should be **dynamically programmable** to achieve **optimal holistic performance**

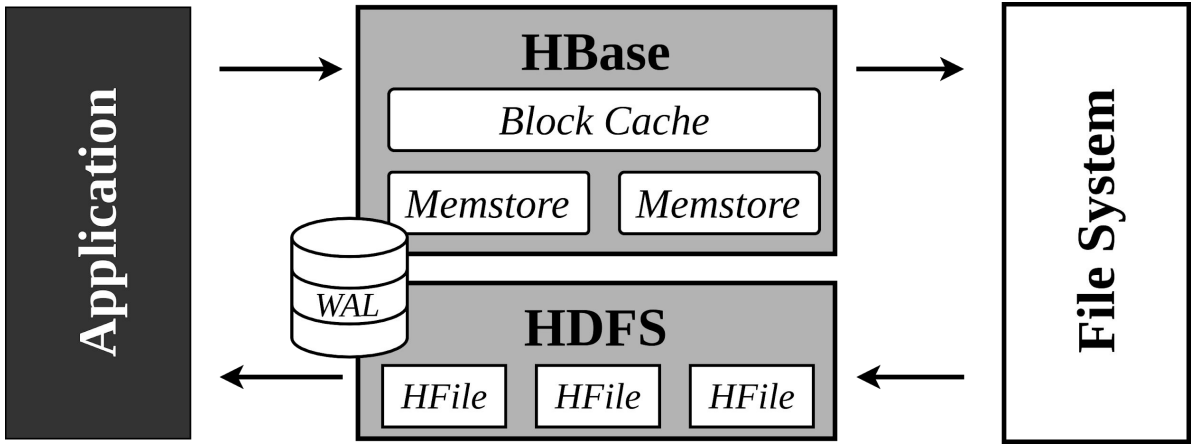
Programmable Storage Background Tasks

Following the **Software-Defined Storage** principles

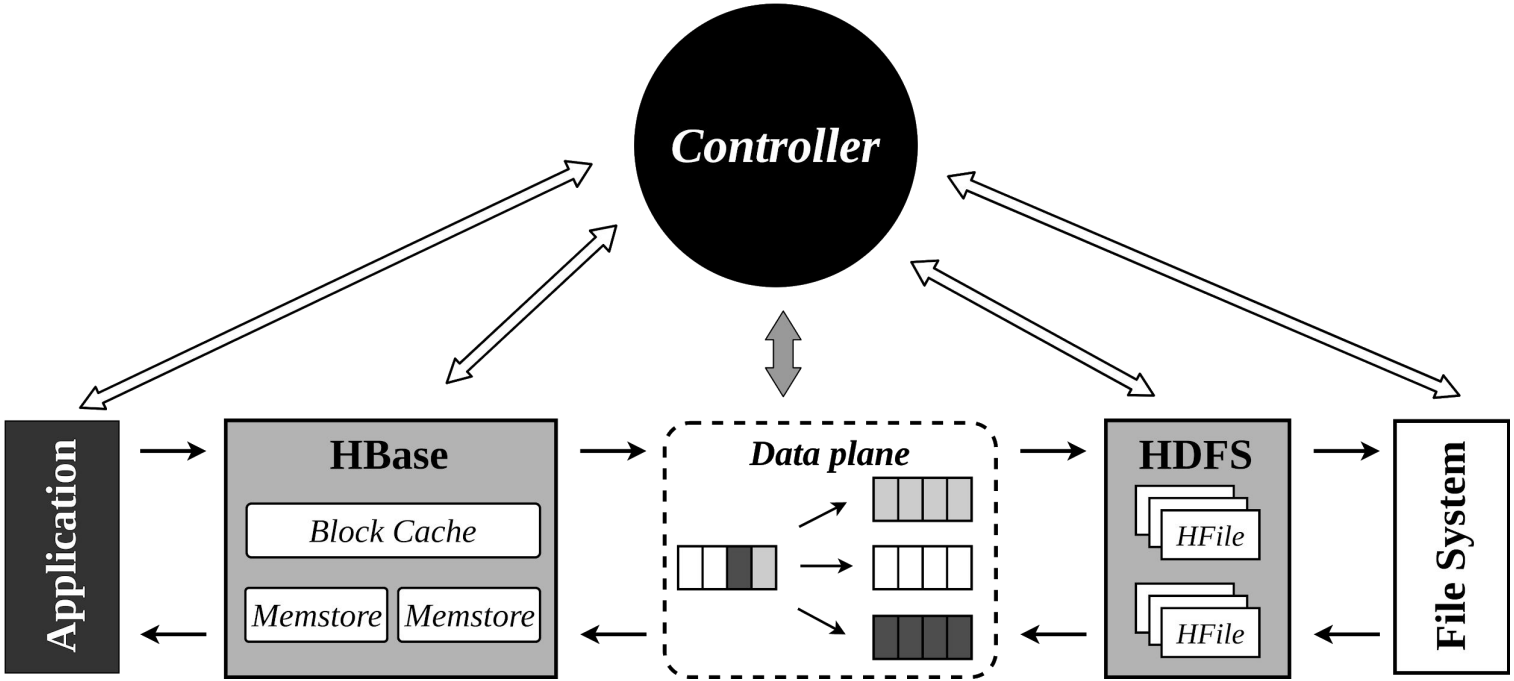
- **Decouple** background **mechanisms** and **policies**
- At layer level, a **data plane dynamically adapts** background activities
- At infrastructure-level, **policy-enabled controller** provides **adaptable end-to-end control**

Minimize I/O variability and interference, and ensure QoS provisioning and resource fairness

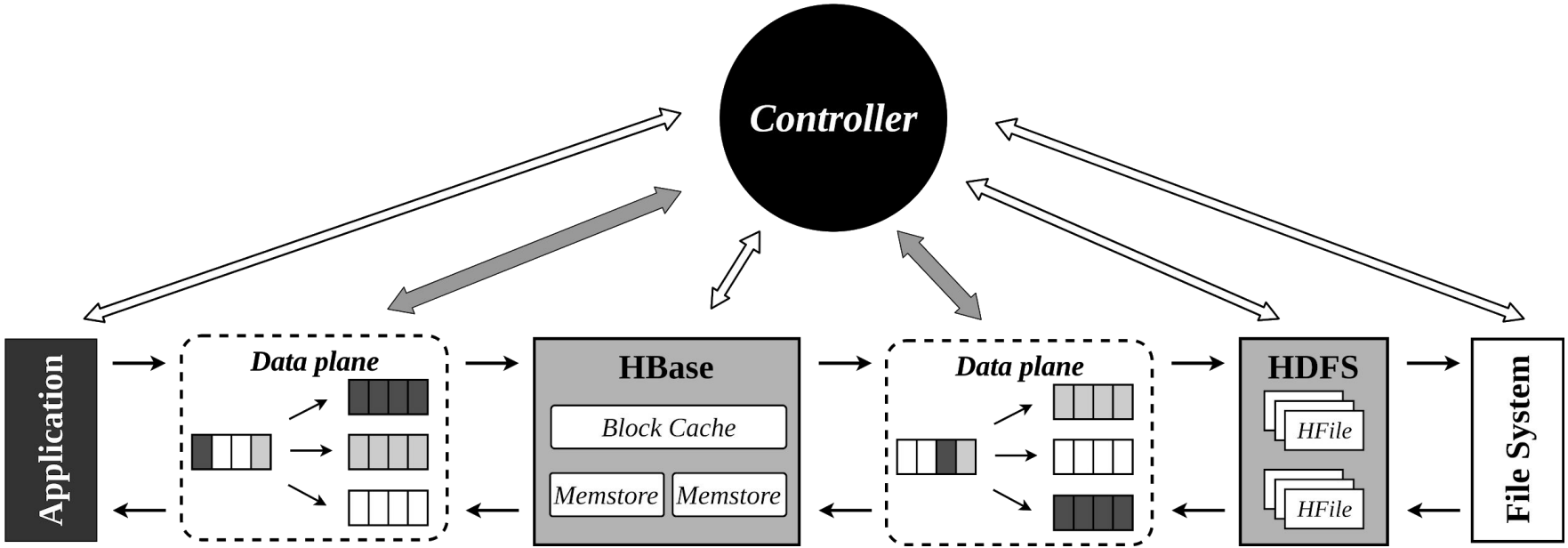
Programmable Storage Background Tasks



Programmable Storage Background Tasks



Programmable Storage Background Tasks



A Case for Dynamically Programmable Storage Background Tasks

Ricardo Macedo, Alberto Faria, João Paulo, José Pereira
INESC TEC & University of Minho

38th IEEE International Symposium on Reliable Distributed Systems Workshops
1st Workshop on Distributed and Reliable Storage Systems
Lyon, France October 1st 2019

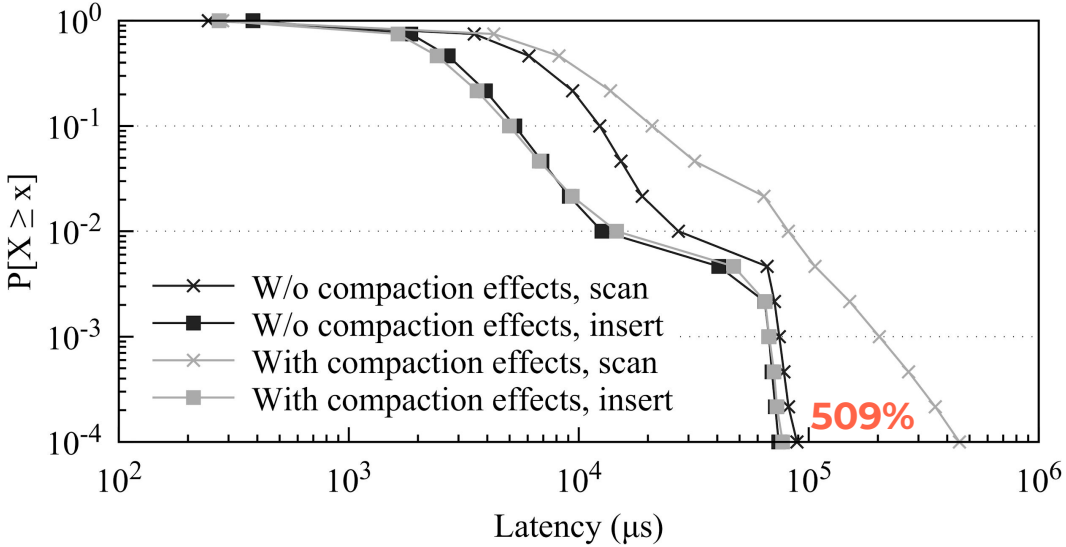
HBase Compactions

Tail latency results

Performance degradations of at most **40%** at the 99th percentile latency for write operations

Read-oriented operations exhibit an **overhead** of at most **955.2%**

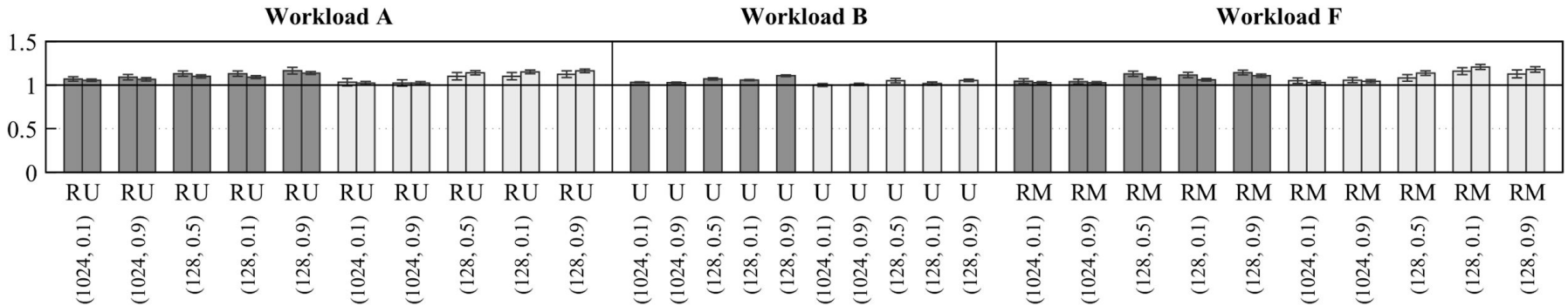
Read-modify-write experiences a **281% overhead**



Complementary cumulative distribution function (CCDF) for the latency of *Workload E* with 10 threads

PostgreSQL Checkpointing

Mean latency results



Read and write operations are equally exposed to performance variations

128MiB WAL size **degrades performance** of at most **20.7%**, **13.9%**, and **17.8%** under 0.1, 0.5, and 0.9 checkpoint completion target

Case Study: HBase

Highly available NoSQL database, made of *RegionServers* and an *HMaster*

- Tables are **horizontally partitioned** by row key ranges into *Regions*
- Writes are first persisted in a **WAL** and then written to a *Memstore* (write-oriented cache)
- Reads **hierarchically traverse** the data store, accessing the *Block Cache*, *Memstore*, and *HFiles*
- Generation of different *HFiles* per *Memstore* leads **read amplification**

Case Study: HBase

LSM-based design leads to the execution of **background compactions**

- **Minor compaction.** Merges several small-sized HFiles into fewer larger ones
- **Major compaction.** Merges all HFiles into a single larger one, removing deleted entries
- While improving read performance, compactions introduce I/O interference and burstiness

Case Study: PostgreSQL

Relational database that handles requests from multiple applications

- Writes are first mapped to a *shared buffer* and then written to a *WAL buffer*
- On commit, changes are **sequentially written** to a **WAL file** on disk
- Reads **hierarchically traverse** the database, accessing the *shared buffer*, *OS cache*, and *disk*

Methodology

Testbed

- HBase 2.0.5 pseudo-distributed mode, backed by HDFS 2.9.2
- PostgreSQL 11.3 backed by an ext4 file system

Workloads*

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

* Workloads previously used in [22]: *"MeT: Workload aware elasticity for NoSQL"*

Execution scenario

- Single and multi-threaded
- Loading phase with 12.5M records (≈16GiB)
- Runs executed 10M operations or when 17 minutes had elapsed
- System state was recreated after each run

Results publicly available at https://rgmacedo.github.io/drss19-website/