ShadowSync: Latency Long Tail caused by Hidden Synchronization in Real-time LSM-tree based Stream Processing Systems

Shungeng Zhang*
Augusta University
Augusta, USA
szhang2@augusta.edu

Qingyang Wang
Louisiana State University
Baton Rouge, USA
qwang26@lsu.edu

Yasuhiko Kanemasa
Fujitsu Limited
Kawasaki, Japan
kanemasa@fujitsu.com

Julius Michaelis
Fujitsu Limited
Kawasaki, Japan
michaelis@fujitsu.com

Jianshu Liu
Louisiana State University
Baton Rouge, USA
jliu96@lsu.edu

Calton Pu
Georgia Institute of Technology
Atlanta, USA
calton.pu@cc.gatech.edu

Abstract

Mission-critical, real-time, continuous stream processing applications that interact with the real world have stringent latency requirements. For example, e-commerce websites like Amazon improve their marketing strategy by performing real-time advertising based on customers' behavior, and latency long tail can cause significant revenue loss. Recent work [39] showed a positive correlation between latency long tail and variance in the execution time of synchronous invocation chains (critical paths) in microservices benchmarks. This paper shows that asynchronous, very short but intense resource demands (called millibottlenecks) outside of critical paths can also cause significant latency long tail.

Using a traffic analysis stream processing application benchmark, we evaluated the impact of asynchronous workload bursts generated by a multi-layer data structure called LSM-tree (log-structured merge-tree) for continuous checkpointing. Outside of the critical path, LSM-tree relies on maintenance operations (e.g., flushing/compaction during a checkpoint) to reorganize LSM-tree in memory and on disk to keep data access latency short. Although asynchronous, such recurrent maintenance operations can cause frequent millibottlenecks, particularly when they overlap, a problem we call *ShadowSync*. For scheduling and statistical reasons, significant latency long tail can arise from ShadowSync caused by asynchronous recurrent operations. Our experimental results show that with typical settings of benchmark components such as RocksDB, ShadowSync can prolong request message latency by up to 2 seconds. We show effective mitigation methods can alleviate both scheduled and statistical ShadowSync reducing the latency long tail to less than 20% of the original at the 99.9th percentile.

CCS Concepts: • **General and reference → Performance**; **Experimentation**; • **Computer systems organization → Data flow architectures**.

*Also with Louisiana State University.

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3565251

Keywords: stream processing, synchronization, performance instability

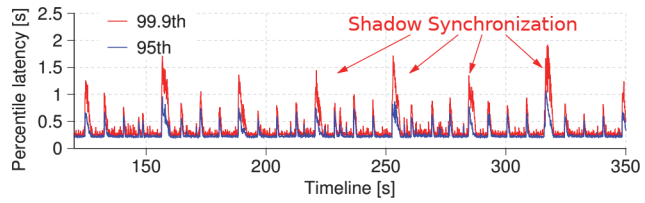
ACM Reference Format:
Shungeng Zhang, Qingyang Wang, Yasuhiko Kanemasa, Julius Michaelis, Jianshu Liu, and Calton Pu. 2022. ShadowSync: Latency Long Tail caused by Hidden Synchronization in Real-time LSM-tree based Stream Processing Systems. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3528535.3565251

1 Introduction

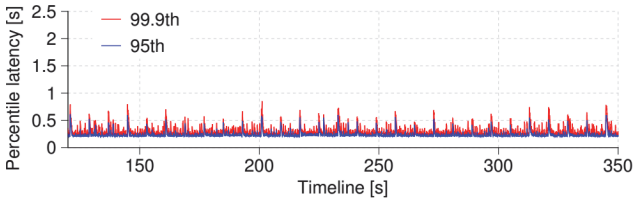
Mission-critical, real-time, continuous stream processing applications such as real-time advertising [33], real-time alarming [57], and real-time object recognition [7] have stringent latency requirements, often dictated by real-world interactions [48, 58]. The ongoing cross-industry event P99 Conf [43] especially focuses on the tail latency due to its practical significance in real-world applications. As a concrete scenario from the Amazon Go checkout-free supermarket [47], as the customer walks through the store, all the actions and purchase behaviors that compose the customer experience are captured in real-time as event streams, which are in turn analyzed and acted upon using real-time stream processing to provide customized advertisement and recommendation services. Latency delays of such real-time applications at the magnitude of seconds can cause serious customer dissatisfaction and lead to significant revenue loss [19].

Recent work [39] has studied the variance in the critical path (synchronous invocation chain of composite microservices) correlated to latency long tail. In contrast to, and complementing their approach, we focus on latency long tail caused by asynchronous services *outside* of the critical path. Although these asynchronous services are running in parallel to the critical path, they can become significant sources of latency long tail problems through the formation of millibottlenecks [38, 45, 50] caused by overlapping asynchronous services, a phenomenon we call *ShadowSync*.

To demonstrate and evaluate the impact of millibottlenecks and associated latency long tail created by asynchronous services, we use a real-time traffic jam ranking program [58] as a representative example (Section 3). The program produces a stream of vehicle data, with components such as Flink [13] for message processing and RocksDB [9] for data management under continuous checkpointing (a key feature to guarantee high fault tolerance and fast recovery). To manage large state data and to support its incremental backup,



(a) Without our mitigation methods.



(b) With our mitigation methods.

Figure 1. We found that *ShadowSync* significantly contributes to the latency long tail problem of large-scale event-stream processing applications, even though the system is facing a constant external workload and the overall system utilization is low to moderate.

RocksDB adopts LSM-tree (log-structured merge-tree), a multi-layer data manager, with level 0 in memory, and less frequently accessed data migrated to higher levels in a file system. A *flush* operation copies data from memory (level 0) to disk (level 1), and a *compaction* operation consolidates data scattered among higher levels into a compact representation to reduce access latency. A major design goal of the LSM-tree is to minimize the critical path of memory data access, by offloading non-critical data to disk through flushing, and reorganization through compaction.

Although the critical path is the first order of concern in real-time stream processing systems, our study shows that asynchronous services, even though offloaded to the outside of the critical path, are far more than second-order effects. For example, our analysis shows for both scheduling and statistical reasons, the asynchronous services (e.g., flushing and compaction during continuous checkpointing) often overlap in time, forming *ShadowSync* and significant latency long tail problems. Under typical checkpointing configurations, Figure 1 shows the latency spikes generated by the scheduling of flushing (triggered every 8sec) and compaction (triggered every 32sec). The ShadowSync (overlap) will occur at the lowest common multiplier of any fixed triggering periods, even if independently specified. Our results create a non-negligible caveat into the presumed scale-out capabilities of LSM-tree-based key-value stores: latency long tail outside of the critical path. For example, our experiments scale out to 128 stage instances, each of them with periodic/recurrent asynchronous services prone to these overlaps.

The main contribution of this paper is a detailed experimental study of *ShadowSync*, confirming millibottlenecks and latency long tail in a real-time continuous stream processing benchmark. The ShadowSync phenomena are due to overlapping transient asynchronous operations from different components in the system. Our study covers CPU millibottlenecks from two kinds of ShadowSync: (1) overlaps created by the scheduling of asynchronous services at fixed periods (Section 3.2), and (2) statistical overlaps when the number of concurrent instances reaches sufficiently high levels

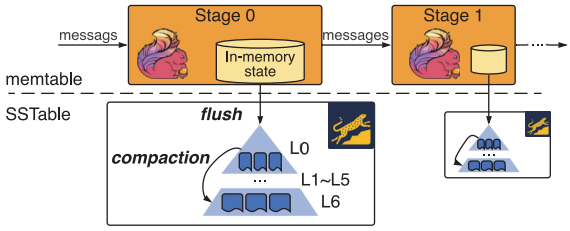


Figure 2. RocksDB implementation of LSM-tree.

(Section 3.3). Although the CPU millibottlenecks only last for a fraction of a second, they prevent the processing of other components for a sufficiently long time to create a latency long tail.

The experimental confirmation of asynchronous services such as ShadowSync as sources of latency long tail is a significant discovery. Up to now, it has been commonly assumed that offloading work from critical paths to asynchronous services would enable increased performance through scaling out, which is the common way implemented in the latest stream processing systems offloading continuous checkpointing tasks to data stores like RocksDB. Our study shows that simply shifting work from a critical path to asynchronous services may improve throughput by scaling out, but there is no free lunch: The new asynchronous services may create new sources of latency long tail. Additional care must be taken to remove or reduce ShadowSync caused by asynchronous services.

Our second contribution consists of effective mitigation methods to address the latency long tail issues caused by ShadowSync. The mitigation methods are designed to significantly alleviate the hidden interference among the apparently independent asynchronous services in two ways. First, to remove ShadowSync created by scheduling, the first set of mitigation algorithms use appropriate waiting to avoid overlapped execution of heavy-weight operations such as flushing and compaction. Second, to reduce statistical overlaps created by many concurrent service executions, we control carefully the amount of concurrency through the judicious allocation of soft resources such as flushing and compaction threads. Our evaluation results show that these mitigation methods reduce 99.9*th* percentile latency to less than 20% of the original benchmark application and even reduce the 95*th* to less than 50% of the original.

The rest of the paper is organized as follows. Section 2 shows a periodic/recurrent latency long tail problem caused by *ShadowSync*. Section 3 introduces two types of ShadowSync. Section 4 and 5 introduce and evaluate the proposed mitigation methods, respectively. Section 6 discusses ShadowSync with different sources or in different stream processing pipelines. Section 7 summarizes related work, and Section 8 concludes the paper.

2 Real-world Exemplar of the ShadowSync Problem

Both the throughput and latency of a message in real-time stream processing systems are often limited by the critical path of that message. As an example, recent work [39] showed a correlation between the variance of critical path execution and latency long tail. Quite often, efforts are made to reduce the critical path to improve request latency, through the specialization of common cases and offloading heavy work (e.g., I/O) to asynchronous services. A good example is the *Log-Structured Merge-Tree* (LSM-tree) [31]

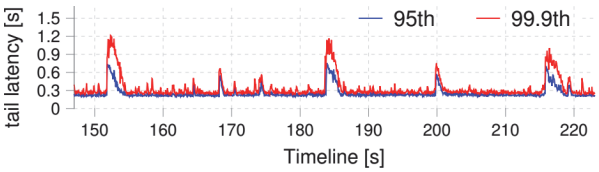


Figure 3. Illustrative latency spikes in timeline analysis.

implementation of key-value stores. The main requests are served by data in memory via a fast critical path, and storage operations involving disk I/O are shifted to asynchronous services.

Our benchmark adopts 1) RocksDB (Figure 2), a state-of-the-art LSM-based key-value store widely-used by industry practitioners like Facebook [6, 32], Uber [35], and Netflix [4]. 2) Flink, a mainstream messaging platform widely adopted by real-time stream processing applications such as real-time analytics services (e.g., Amazon Kinesis Data Analytics [21, 44]) and real-time alarming (e.g., eBay [11] and Capital One [12] monitoring platforms). In RocksDB, the fast execution path uses in-memory data access to reduce latency. Write operations append new data to an in-memory log file called *memtable*. Periodically, the *flushing* operation moves memtable content to files called SSTables (Sorted Sequence Table). To improve average access latency and throughput, SSTables are organized hierarchically in levels. Periodically, the SSTables are reorganized to reflect the access patterns in an asynchronously triggered operation called *compaction*, to remove duplicated data items and deleted ones.

Both flushing and compaction involve intense CPU overhead and disk I/O operations over a short period of time (from a fraction of a second to 1.5sec in our experiments). To improve performance scalability, flushing and compaction are designed to work independently. However, as we will explain in more detail in the next section, flushing and compaction often overlap in their execution, in a situation we call *ShadowSync*. During a ShadowSync, system resources (e.g., CPU) are fully occupied in a phenomenon called millibottleneck [50], leading to significant queuing and sometimes dropped message packets. As a result, ShadowSync often produces latency long tail, even though they are asynchronous services and outside of the critical path.

Figure 3 illustrates the ShadowSync phenomena during an experiment (borrowed from Section 3). The X-axis shows the timeline of the experiment, from 150sec to 220sec. It shows the latency of messages grouped within windows of 50ms intervals. For most intervals, the message end-to-end latency is between 0.2sec and 0.4sec, but there are three spikes (at 152sec, 184sec, and 216sec) when the messages take more than 1sec to return. These spikes correspond to ShadowSync, when periodic flushing operations (at 16sec intervals) overlap with periodic compaction operations (at 32sec intervals). ShadowSync will be explained in detail in the next section.

3 Experimental Study of ShadowSync

In this section, we describe the results from evaluations of a real-time stream processing benchmark under continuous checkpointing. The experiments show two types of ShadowSync: *scheduled* and *statistical*. Section 3.2 describes findings on scheduled ShadowSync, when the scheduling of flushing and compaction operations from the same stage instances overlap, causing millibottlenecks and latency long tail. Section 3.3 describes the statistical ShadowSync among the flushing and compaction operations from independent instances across stages, which also causes latency long tail.

3.1 Experimental Environment

We use a real-world, real-time streaming platform "Dracena", which is a distributed event stream processing platform designed by Fujitsu to collocate various IoT services [37, 58]. Dracena consists of a combination of open-source utility software components, including (1) Apache Flink [13] as a distributed stream message processing engine, (2) RocksDB as a state management backend involving checkpoint activities (flush and compaction) in Flink worker nodes, (3) Apache Kafka [16] as a persistent message queue for its input/output, and (4) HDFS as reliable backup storage for RocksDB data.

On top of Dracena, we run a real-time traffic jam ranking benchmark application for connected cars on the roads in the Metropolitan Tokyo area. Figure 4(a) depicts the pipeline of our multi-stage streaming dataflow, including three types of objects: cars in *s0*, streets in *s1*, and traffic jam ranking in *s2*. Concretely, each car object in stage *s0* receives an event message every second from a workload generator, where the message data is synthetic data inspired by real car sensor data from our industry partner. The message includes car-ID, speed, position coordinate, etc. After updating, the car object sends its real-time state to a street object (corresponding to a physical street) in stage *s1* that the car is located on at that moment. The street object will calculate the degree of traffic jam based on the number of cars on it. Then the street will send the information to the traffic jam ranking object in stage *s2*, which aggregates and ranks the traffic jam from all streets in the city as the final output.

We conducted our experiments in the public CloudLab cluster [36]. Figure 4(b) shows the deployment of our benchmark application on Flink's worker nodes and the flow of data messages among them. In our experiments, **stage *s0*, *s1*, and *s2* execute with 64, 64, and 1 instance(s), respectively**. We specify the number of cars (e.g., 10k cars) in the streets to control the intensity of the workload. Figure 4(c) outlines the choices of the hardware specification in our experiments. The embedded RocksDB for the state management of Flink is deployed in in-memory `tmpfs` to eliminate the interference caused by disk I/O (e.g., write stalls [30]), and the state files managed by RocksDB will be asynchronously backed up (after each checkpoint) to the remote HDFS for persistence purpose. Such a hybrid configuration (i.e., `tmpfs` plus HDFS) can achieve both low tail latency and high scalability based on industry experience [58]. For comparison, we also show experimental results using NVMe SSDs at the end of the evaluation (see Section 5.3).

RocksDB implements the flushing operation as a straight dump of a memtable buffer to a single SSTable file. The flushing operation is initiated automatically if the memtable fills up, or it can be initiated through a checkpoint operation of Flink. As the number of SSTable files grows at each level, the compaction operation is scheduled automatically. By default, four SSTables trigger compaction at L0 (i.e., level 0).

3.2 Scheduled ShadowSync Flushing and Compaction within the Same Stage

We show our first case study of the ShadowSync problem, where the scheduling of flushing and compaction operations from the same stage instances overlap during checkpoints in a stream processing

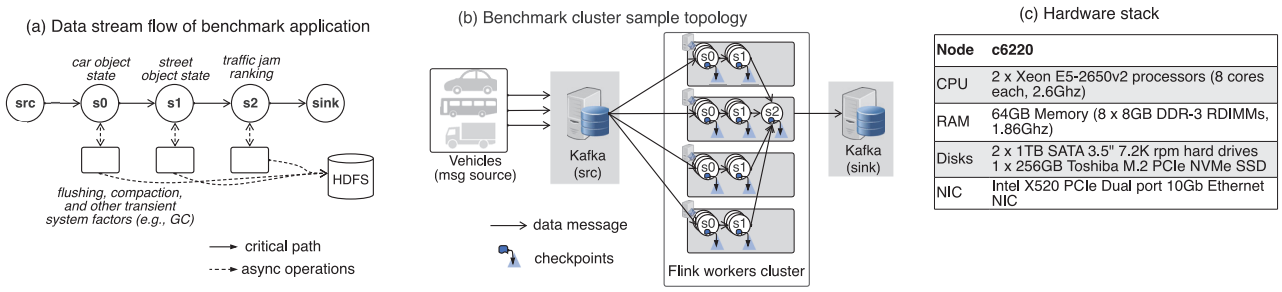


Figure 4. Details of the experimental setup.

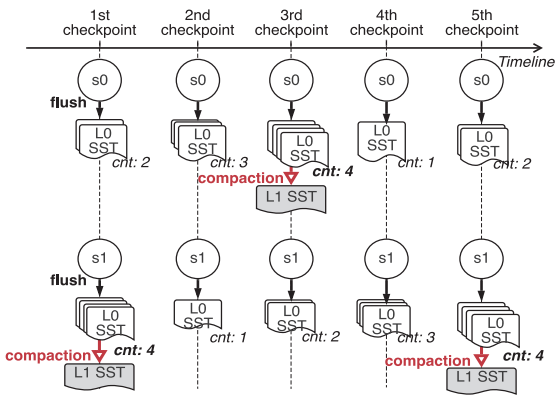


Figure 5. Illustration of scheduled *ShadowSync* flushing and compaction within the same stage over five continuous checkpoints. The top shows an *s0* instance while the bottom shows an *s1* instance. Compaction is triggered once the number of accumulated L0 SSTables (the counter increases by 1 for every flushing) reaches 4. The initial counter value may differ in different stages.

system, causing millibottlenecks and latency long tail. This study is to answer two questions as raised in the motivation section:

- **Q1**: What is the main reason for flushing and compaction operations to overlap and form *ShadowSync*?
- **Q2**: What is the impact of *ShadowSync* on the latency long tail of a stream processing application?

To answer the first question (**Q1**), we start our analysis by figuring out how and when the flushing and compaction activities are involved during checkpoints in a stream processing system. Concretely, the checkpointing mechanism in Flink is coordinated and pipelined for a consistent global state of the distributed streaming data flow [5]. As a state backend of Flink, RocksDB maintains the runtime states (e.g., car and street objects) for all stages in memory (i.e., named `memtable`). At the beginning of each checkpoint, each stage instance in Flink *flushes* its corresponding `memtables` in RocksDB into filesystem (we use in-memory `tmpfs` in our experiments to avoid disk I/O) as L0 SSTable files. Once the accumulated SSTable files of a stage instance reach a preset compaction threshold (default is 4 in RocksDB), the *compaction* activity will be triggered, merging accumulated SSTables to the next level (from L0 to L6, see Figure 2). Figure 5 illustrates that a compaction activity for both a *s0* and a *s1* instance will be periodically triggered every 4 continuous flushing activities. Since each stage typically has tens to hundreds of stage instances for parallelism (64 for both *s0* and *s1* in our experiments [1]), at every $4th$ checkpoint, the triggered compaction activities from the same stage instances will overlap with all the triggering flushing activities, causing the ShadowSync problem between flushing and compaction.

Table 1. Statistics of flushing and compaction activities during the same period in Figure 3.

Checkpoints	**1st CP*** at 152sec		2nd CP at 168sec		**3rd CP** at 184sec		4th CP at 200sec		**5th CP** at 216sec	
stage	s0	**s1**	s0	s1	**s0**	s1	s0	s1	s0	**s1**
# of flush for the CP	64	**64**	64	64	**64**	64	64	64	64	**64**
avg flush time [ms]	33	87	31	106	29	63	29	70	27	58
# of compaction for each CP	0	**64**	0	0	**64**	0	0	0	0	**64**
avg compaction time [ms]	\	10	\	\	392	\	\	\	\	291
total compaction input size [MB]	\	590	\	\	2029	\	\	\	\	589

*CP = Checkpoint

Table 1 shows the statistics of flushing and compaction activities for both stage *s0* and *s1* over five checkpoints during the same period (i.e., 150~220sec) as in Figure 3. This table shows that the flushing and compaction activities of all instances synchronize at the $1st$ and $5th$ checkpoints for stage *s1* while the $3rd$ checkpoint for stage *s0*. What's interesting is that the three times of synchronization match well in time with the three latency spikes (at 152sec, 184sec, and 216sec) as shown in Figure 3, suggesting the causal relationship.

To confirm the above causal relationship and also answer **Q2**, Figure 6 shows the point-in-time analysis of Flink worker node CPU utilization, message queue along with flushing and compaction concurrency during the same period as in Figure 3. This figure illustrates that the synchronization at 152sec, 184sec, and 216sec between flushing and compaction activities (see Figure 6(c) and 6(d)) cause the CPU utilization of the Flink worker nodes to be 100% as shown in Figure 6(a), resulting in high message queues in all stages during the same periods as shown in Figure 6(b). Thus, the high message queues cause the three latency spikes as shown in Figure 3.

There still is one mystery in Figure 6: While the starting times of flushing and compaction concurrency spikes (e.g., at 184sec) in ShadowSync are almost the same, the concurrency spikes last much longer than the flushing spikes. This is an important discovery because each concurrency spike has the same duration of the CPU saturation of the Flink worker nodes, suggesting a significant performance impact.

To figure this out, we further zoom in the period 183~186sec (i.e., the $3rd$ checkpoint), shown in Figure 7, in which we show the start

[1] 64 corresponds to the total number of CPU cores in our 4 Flink nodes.

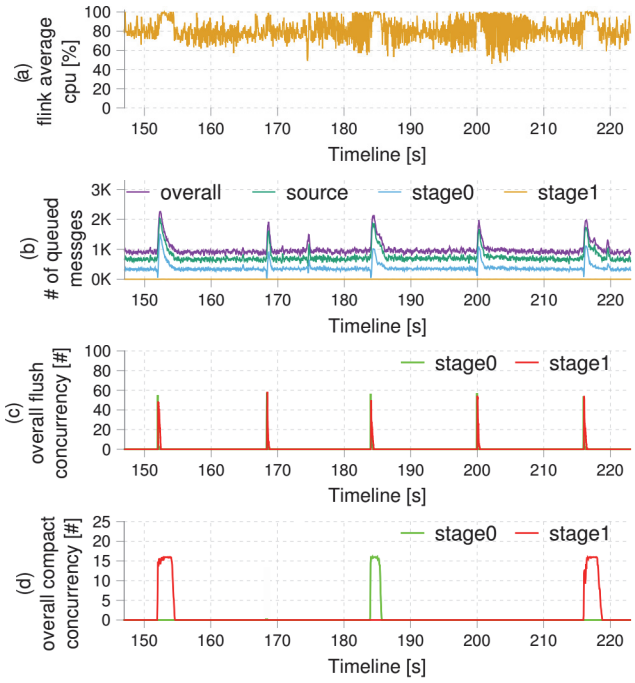


Figure 6. ShadowSync of flushing and compaction activities in RocksDB during the same period in Figure 3, leading to large latency spikes (e.g., period 151~153sec, 184~186sec, and 217~220sec).

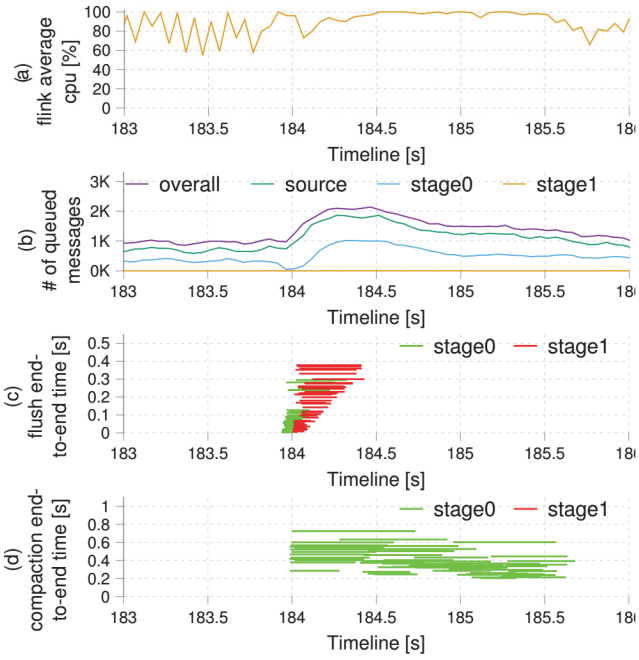


Figure 7. Zoom in analysis of ShadowSync of flushing and compaction activities in RocksDB during the period 183~186sec in Figure 6(c) and 6(d). Each thin line segment in (c)(d) means the start, the end, and the duration of either a flushing or compaction activity from a specific stage instance, and the longer of each line segment the higher its position in (c)(d).

and end timestamps of each flush and compaction activity (we have 64 instances for both *s0* and *s1*) in one checkpoint. Each thin line segment in Figure 7(c) and 7(d) shows the start, the end, and the duration of either a flushing or compaction activity from a specific instance. Figure 7(d) clearly shows that the 64 compaction activities for *s0* instances upon the current checkpoint last much longer than their corresponding flush activities (totally 128 for both *s0* and *s1*) in Figure 7(c). This is because flush activities in RocksDB are "stop-the-world" (thus not contend for CPU resources with other threads) and all happen in memory, they finish fast. On the contrary, the limited number of compaction threads in RocksDB (totally 16 by default) need to process a large amount of CPU-intensive compaction activities (totally 64) while contending for CPU resources with other Flink worker threads, leading to prolonged CPU saturation as shown in Figure 6(a). This discovery also suggests that a smart flush/compaction threads allocation strategy is needed to resolve/mitigate the ShadowSync problem, which we will discuss more in Section 4.2.2.

3.3 Statistical ShadowSync flushing and compaction across Stages

The second case of ShadowSync involves flushing and compaction activities across different Flink stages, which we call *Statistical* ShadowSync. In the previous section, we show that the bursts of compaction activities from *s0* and *s1* occur at different checkpoints during a 4-checkpoint cycle (see Figure 6(d)). However, our experiments show that the periodic *ShadowSync* of flushing and compaction activities across different stages can also occur given a slightly different system initial condition (e.g., checkpoint interval changes), leading to more severe CPU contention among flushing and compaction activities and thus higher latency spikes.

We show similar experimental results as in the previous section during a 2-min runtime in Figure 8, with only one change: the Flink checkpoint interval is reduced from 16sec to 8sec for faster failure recovery. Our results show even higher latency spikes upon the occurrences of ShadowSync, with a 4-checkpoint cycle in Figure 8(a) (e.g., three small latency spikes and one large latency spike in a 32-second cycle). Unlike the latency cycle in Section 3.2, where bursts of compaction activities for *s0* and *s1* stage instances are separated into different checkpoints (see Figure 6(d)), majority of compaction activities in this new experiment overlap in the same checkpoint period in a cycle as shown in Figure 8(d) (at 254sec, 286sec, 318sec, 350sec). Such an overlap leads to a more severe CPU contention among flushing and compaction activities across different stages, leading to even higher latency spikes (e.g., over 2sec) than those in Figure 3.

The reason of overlap of compaction activities across different stages (*s0* and *s1*) is illustrated in Figure 9. Recall from Section 3.2, compaction activities for all stage instances in Flink occur when the counter for their corresponding L0 SSTables reaches a threshold (by default 4). In this case, the counting of SSTables from different stage instances may synchronize in time if the initial counter values for all stage instances are the same and every stage instance keeps the same pace to increase and reset its counter value. This is the case in Figure 9, where the instances of both stage *s0* and *s1* (64 for each stage) start with the same initial L0 SSTables counter value and follow the same cycle (from 1 to 4), resulting in the

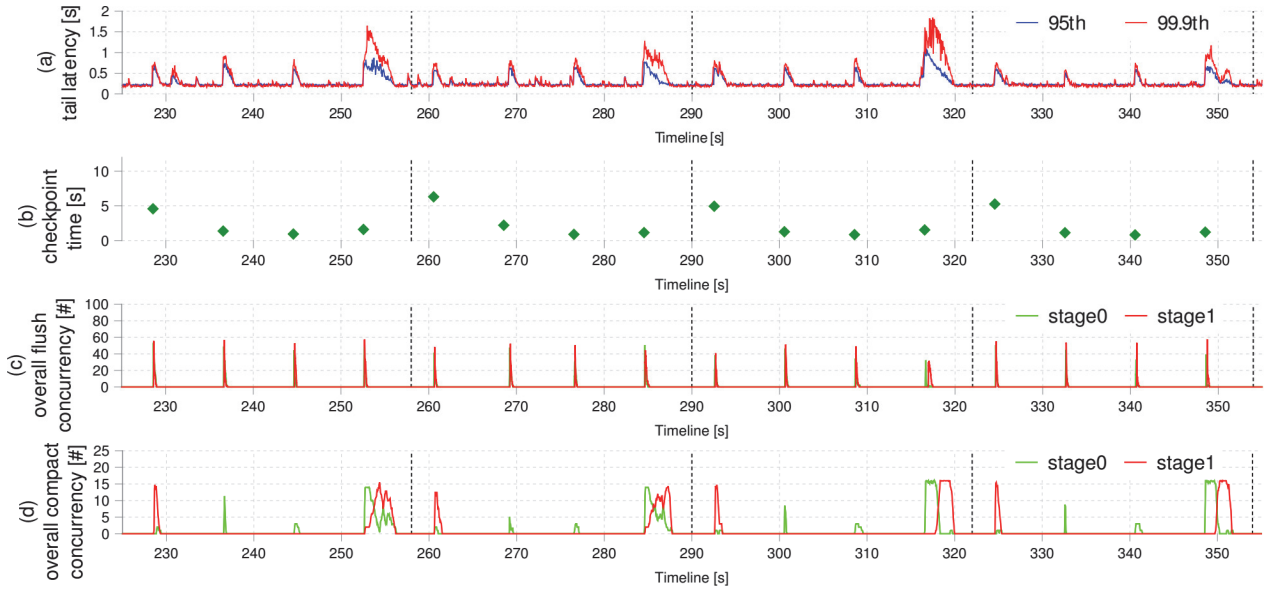


Figure 8. The synchronization of bursts caused by compaction activities from different stages leads to high latency spikes.

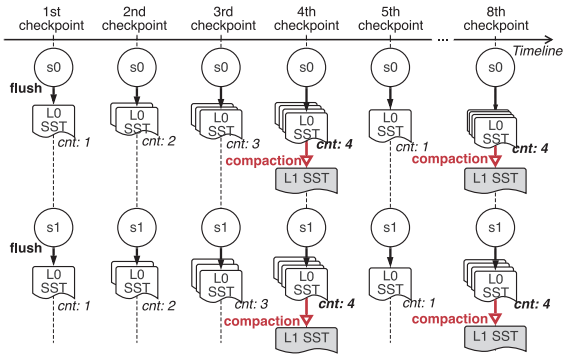


Figure 9. Illustration of statistical *ShadowSync* flushing and compaction across stages over five continuous checkpoints. The top shows an *s0* instance and the bottom shows an *s1* instance. Compaction activities for both an *s0* and *s1* instance could sync in time due to the same initial L0 SSTables counter value and the same cycle (from 1 to 4) of such two instances, causing the *ShadowSync* problem and even greater latency long tail in Figure 8(a).

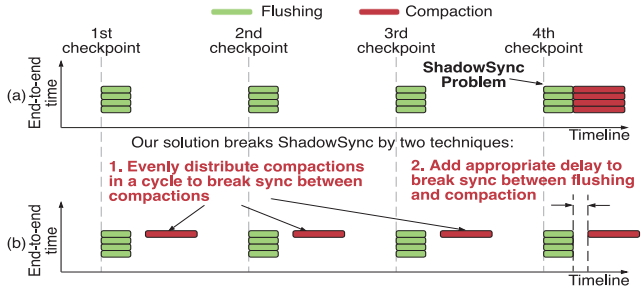


Figure 10. Main idea of mitigation methods to mitigate *ShadowSync* by delaying and rearranging compaction activities across checkpoints in one cycle.

synchronization of compaction activities from these two stages. This explains the significant overlap of the compaction activities from both stage *s0* and *s1* at every 4th checkpoint in a 4-checkpoint cycle as shown in Figure 8(d). When all compaction activities from these stage instances are overlapped, the worst synchronization situation happens, leading to a severe CPU contention problem and high tail latency spikes (see Figure 8(a)).

We note that the initial L0 SSTable counter value for different stage instances could change when system settings or workload conditions change, making the overlap of compaction activities across different stage instances difficult to predict. This is because flushing in Flink can occur in two ways: 1) flushing upon a checkpoint (the typical way), and 2) flushing when the memtable of a stage instance becomes full. The second way of flushing often occurs during the initialization period of our benchmark application, in which all the objects in different stage instances are initialized before the runtime experiment starts. During this initialization period, different stage instances (*s0* and *s1* in our case) may experience different frequencies of flushing due to their high update ratio and varied object size, thus the counter values of L0 SSTable across different stage instances can be different after the initialization stage. For example, Figure 8(d) shows occasional compaction activities from both stage *s0* and *s1* in the first 3 checkpoints in the 4-checkpoint cycle; the case study of Section 3.2 even shows completely out-of-sync compaction activities from stage *s0* and *s1* (see Figure 6(d)).

4 Mitigation Methods

So far we have studied two types of the *ShadowSync* problem of LSM-based checkpointing in real-time stream processing applications: 1) the *scheduled* ShadowSync flushing and compaction within the same stage instances (Sections 3.2), 2) the *statistical* ShadowSync flushing and compaction across different stage instances (Section 3.3). In this section, we introduce effective methods to mitigate the latency long tail problem caused by ShadowSync. Our mitigation methods follow two basic principles:

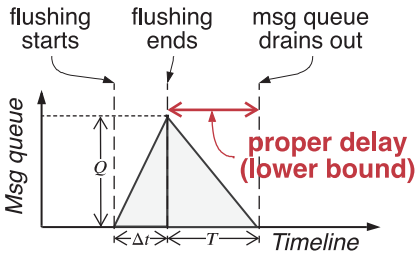


Figure 11. Estimation of appropriate delay between flushing and compaction activities based on the drain-out of message queue triggered by flushing.

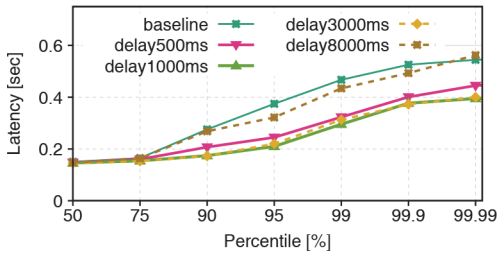


Figure 12. Impact of the delay between flushing and compaction activities on end-to-end tail latency. The delay 1000ms/3000ms achieves the lowest latency.

1. Desynchronize scheduled/statistical overlap of flushing and compaction activities as much as possible;
2. Reduce the overlap probability by reducing the duration of each flushing and compaction activity.

Concretely, the first principle can be met by adding randomization to the triggering condition of compaction (Section 4.1). Our choice of a randomized algorithm in contrast to deterministic solutions (e.g., priority-based scheduling) is based on the global nature of checkpointing. A checkpoint command initiates hundreds (or more) of stage instances, which may have similar checkpoint behavior and cause resource consumption overlap. While deterministic solutions typically would rely on assumptions (e.g., guaranteed mismatched run-time allocations), a randomized algorithm can always achieve the goal of avoiding coincidences. The second principle can be met by choosing the appropriate thread pool size for flushing and compaction accordingly (Section 4.2).

4.1 Scheduled Desynchronization by Randomizing Intervals between Flushing and Compaction

The fundamental reason for *ShadowSync* is that hundreds of stage instances along the streaming dataflow follow the same procedure to do flushing and compaction. As a result, the ShadowSync of flushing and compaction will likely occur under two conditions: (1) the initial condition (e.g., the initial number of SSTable files) of a large number of stage instances is the same; (2) the periodic checkpointing command from the Flink Master triggers the immediate flushing of all stage instances [2]. The first condition enables the SSTable counter of each stage instance to sync in time, thus a large number of compaction activities could happen simultaneously (overlap). The second condition will make sure that the simultaneous compaction activities overlap with hundreds of triggering flushing activities. Unfortunately, both conditions are likely to be held in real systems. Figure 10(a) illustrates the *ShadowSync* when both conditions are met, making it a serious performance problem for real-time stream processing.

We apply two techniques to address the two pre-conditions of *ShadowSync*, respectively, shown in Figure 10(b): first, to avoid the compaction activities from a large number of stage instances being triggered by flushing at the same time, we randomize the default triggering threshold (i.e., four SSTables) for each stage instance to trigger compaction. As a result, the originally scheduled compaction activities at one checkpoint will spread across multiple checkpoints, depending on the range of the assigned random number. To best mitigate the *ShadowSync* problem, we should evenly spread all compaction activities through all checkpoints as shown in Figure 10(b). Since by default every 4 checkpoints form a cycle (see Figure 8), we add an extra random integer $\alpha \in [0, 4)$ on top of the original cycle, where α obeys a uniform distribution. In this case, the new compaction threshold will be $4 + \alpha$, thus compaction activities of all stage instances are likely to be evenly distributed across the checkpoints during runtime.

Second, to avoid/mitigate overlap involving both flushing and compaction (Section 3.2), we are inspired by [41] and insert an appropriate delay between each cluster of flushing and compaction activities upon a checkpoint [3]. We note that flushing impacts the tail latency performance of the target system by stopping message processing in Flink due to its "stop-the-world" feature, which causes messages to queue. Figure 11 illustrates the message queue building up and draining out along with the flushing activities upon a checkpoint. Assume the system receives an input message rate λ and the duration for flushing activities is Δt (the message queue building-up time). We can calculate the queued messages Q in the system during flushing activities as:

$$Q = \lambda \times \Delta t \tag{1}$$

Once flushing activities are done, the system can process all queued messages to its full capability, thus the queued messages in the system will start to drain. The queue drain period can be approximated as follows:

$$T = \frac{Q}{C} = \frac{\lambda \times \Delta t}{C} \tag{2}$$

where C is the full processing capability of the system. To avoid the deterioration of queued messages in the system due to the compound queuing effect caused by the overlap of flushing and compaction activities, we should postpone the compaction activities for an appropriate delay T until all queued messages (caused by flushing) are drained out. In our experiments, we are able to monitor the input message rate λ, the duration for each cluster of flushing activities Δt, and the message processing capability C of the system, and calculate a reasonable delay time T, which is between 800ms to 1sec in our experiments.

Figure 12 shows the end-to-end system latency as we increase the delay of compaction in RocksDB from 100ms to 8000ms (detailed experimental setup in Section 5.1). We note the calculated drain-out time T is between 800ms to 1sec. The figure shows that the system achieves the best tail latency performance when the delay is set to 1000ms, and tail latency performance almost keeps the same

[2] Due to the high checkpoint frequency during runtime, flushing caused by memtables of a stage instance becoming full rarely occurs between consecutive checkpoints.

[3] [41] focuses single RocksDB instance study while our research targets the interaction between hundreds of RocksDB instances and Flink.

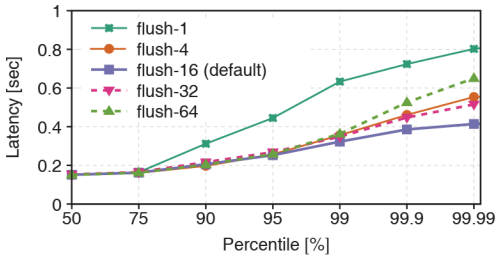


Figure 13. Impact of number flush threads on the intensity of internal bursts. The best flush threads allocation (16) matches the number of CPU cores per Flink worker node.

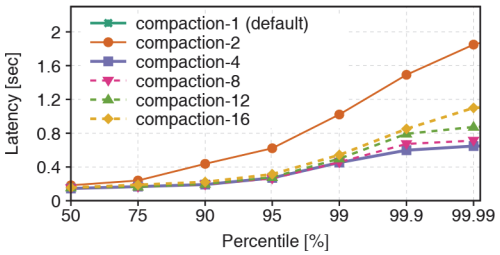


Figure 14. Impact of number compaction threads on the intensity of internal bursts. The best compaction threads allocation is 4 in our environment.

when the delay increases to 3000ms. This is because too short a delay (e.g., <1000ms) is not enough for the message queue to be drained out before compaction activities (see Figure 11). On the other hand, too long a delay (e.g., 8000ms) will cause the flushing of the current checkpoint overlaps with the compaction from the previous checkpoint since the checkpoint interval is 8sec in our experiments, still causing ShadowSync.

4.2 Statistical Desynchronization through Appropriate Soft Resource Allocations

In this section, we show that appropriate soft resource allocations can mitigate the statistical *ShadowSync* of flushing and compaction activities across different stage instances. Through extensive experiments, we show that the system tail latency is sensitive to the allocation of flushing and compaction threads. While soft resource allocation does not address the ShadowSync of flushing and compaction directly, an appropriate allocation can reduce the length of each flushing and compaction activity, thus reducing the probability of ShadowSync of a large amount of flushing and compaction activities upon a checkpoint.

4.2.1 Choosing Appropriate Number of Flushing Threads.

We note flushing in RocksDB will lock the in-memory state (memtable) of each stage instance and prevent further state update operations, blocking the normal message processing in the critical path of the streaming dataflow in Flink. A "rule-of-thumb" approach for the flushing thread pool allocation is to set its size the same as the underlying CPU cores. For example, we should configure 16 flushing threads on each Flink worker node, which has two Octa-core processors (Figure 4(c)). In this case, flushing could fully utilize the underlying CPU cores to decrease the overall flushing time upon each checkpoint while avoiding unnecessary locking overhead introduced by over-allocation flushing thread [52].

We validate our "rule-of-thumb" approach for flushing thread pool allocation in Figure 13. In this evaluation, we gradually increase the number of flushing threads in RocksDB from 1 to 64. The figure shows that the *flush-16* case indeed achieves the best tail latency performance, suggesting the reduced impact of flushing activities on system performance; either under- or over-allocation leads to deteriorated tail latency. In the following experiments, we will set the number of flushing threads to 16 in RocksDB.

4.2.2 Choosing Appropriate Number of Compaction Threads.

Unlike the "stop-the-world" flushing in RocksDB, compaction is inherently an asynchronous process and will not block message processing in Flink. Even though compaction activities in RocksDB occur outside the critical path of the message processing, they could contend for the shared CPU resources [2] with message processing threads in Flink.

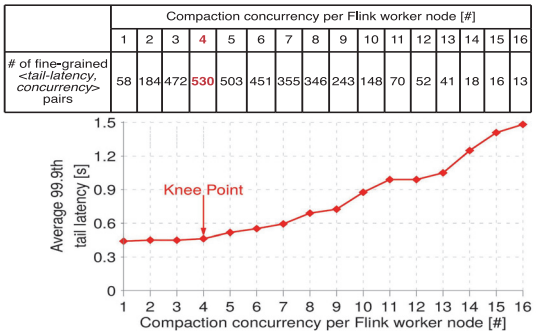


Figure 15. Our approach finds appropriate compaction thread pool allocation (Knee Point) through a fine-grained statistical analysis (i.e., Kneedle [42]) between the system tail latency and the real-time compaction threads concurrency.

Figure 14 shows the impact of different sizes of compaction thread pool on the system tail latency performance. In this set of experiments, we gradually increase the compaction threads in RocksDB from 1 to 16 (the tail latency for *compaction-1* reaches several minutes, thus we omit the results here). The figure shows that the system achieves the best tail latency performance when the number of compaction threads is 4. This is because a too-large allocation brings a severe CPU contention problem while a too-small allocation will prolong the compaction process due to insufficient parallelism to do the compaction job for tens to hundreds of stage instances.

Unlike the allocation of flushing threads, the appropriate allocation of compaction threads is non-trivial since the brute-force approach by manually trying all possible allocations is time-consuming. To fast infer an appropriate compaction threads allocation, we adopt a fine-grained statistic analysis inspired by Kneedle [42], which enables us to infer a reasonable allocation during system runtime. Specifically, we measure the fine-grained system tail latency and real-time compaction threads concurrency over continuous 50ms time windows and plot the correlation between these two metrics as shown in Figure 15. Since the real-time compaction threads concurrency changes fast over different 50ms time windows, we can quickly plot the correlation of the system tail latency with a wide range of compaction threads concurrency (e.g., from 1 to 16 in Figure 15). We then find the knee point of the correlation curve through Kneedle to get the optimal concurrency

setting. For example, Figure 15 shows that the knee point of the curve is when the concurrency of compaction threads is 4, and any concurrency beyond 4 will result in increased system tail latency. Thus, we recommend 4 as an appropriate compaction threads allocation per Flink worker node. Such a recommendation through our fine-grained statistic analysis also matches with the recommendation derived from our previous brute-force approach shown in Figure 14.

5 Experimental Evaluation

In this section, we show a more detailed evaluation of the effectiveness of our proposed mitigation methods using two representative streaming benchmark applications: real-time traffic jam ranking in Flink [13] and real-time word count in Kafka Streams [16].

5.1 Case 1: Real-time Traffic Jam Ranking in Flink

In this case study, we conduct experimental evaluations on a real-time traffic jam ranking application in Flink to verify that our mitigation methods can effectively alleviate *ShadowSync* problems and reduce the latency long tail.

The real-time traffic jam ranking application (see Figure 4) is for connected cars running on the roads in the Tokyo metropolitan area. Our trace-based workload generator reads the real-world traffic data [58] and sends event messages to the backend stream processing system every second. Each event message contains the location information and basic sensor data (e.g., car-ID and speed) of the corresponding physical car, and the event message size is about 6kB on average. In our experiments, the message rate in all cases keeps 60K messages/second, under which the Flink worker node CPU usage is about 75%. More detailed experimental setup is in Section 3.1.

Figure 16 shows the performance comparison between the original benchmark application and the same application but with our proposed mitigation methods in Section 4 applied. The delay of compaction activities (triggered by flushing) in RocksDB is set to 1sec as an appropriate setting (see Figure 12). Both cases use the default flushing and compaction threads allocation (16 for each) in RocksDB.

Comparing the baseline case in Figure 16(a) with our solution in Figure 16(b), our proposed mitigation methods can effectively reduce large latency spikes. For example, the 99.9th percentile latency could reach over 2sec at 209sec in Figure 16(a), which is caused by the overlap between highly concurrent flushing and compaction activities shown in Figure 16(c) and 16(e), respectively. After applying the desynchronization techniques between flushing and compaction, all of the 99.9th percentile latency spikes are below 0.5sec as shown in Figure 16(b). This is because the 1sec delay between flushing and compaction in RocksDB effectively decreases the frequency of the *ShadowSync* between highly concurrent flushing and compaction activities as shown in Figure 16(d) and 16(f). Figure 16(f) further shows the compaction activities from both stage *s0* and *s1* are evenly distributed across the four consecutive checkpoints, further reducing *ShadowSync* among compaction activities within the same stage or across different stages.

5.2 Case 2: Word Count in Kafka Streams

We present evaluation results of a real-time Word Count application in Kafka Streams [16], which has been widely used in previous research [10, 24, 25]. In this set of experiments, each partition of Kafka producer reads a line from a synthetic workload generator (generating a set of random works about 25K per second) as a sentence and emits it, and each partition of Kafka consumer first splits the sentences into words that are then forwarded to the `group()` function to subsequently count the occurrences of each word. This is a stateful streaming application as counters need to maintain the current count for each word as their internal state. We note that Kafka Streams by default uses RocksDB to maintain the local state on a computing node. We run the WordCount application on a single dedicated node which is equipped with two Octa-core processors (see Figure 4(c)), and set the partition for sentence events (parallelism) to 64 to fully utilize the underlying CPU cores. In our evaluation experiments, the Kafka node average CPU usage is 70%.

Figure 17 shows the tail latency performance comparison between the original word count benchmark application and the one with our mitigation methods applied. The delay of compaction activities (triggered by flushing) in RocksDB is set to 1sec based on our measurement. Both cases use the default flushing and compaction threads allocation (16 for each) in RocksDB for a fair comparison.

Our evaluation results show that our proposed desynchronization techniques can effectively mitigate the latency long tail problem caused by ShadowSync. For example, the 99.9th percentile latency for *solution* is about 0.7sec, which is significantly better than that in the cases of *baseline* (i.e., 1.3sec), indicating the effectiveness of our proposed methods for mitigating latency long tail problem. Figure 18 further show fine-grained (e.g., 50ms granularity) timeline analysis of metrics across different system components for a more detailed comparison. For example, the 99.9th percentile latency in the baseline case in Figure 18(a) could reach 3sec, which is due to the overlap between highly concurrent flushing and compaction activities shown in Figure 18(c) 18(e). On the other hand, all the 99.9th percentile latency in our solution case in Figure 18(b) is below 2sec, the improvement of which is due to the desynchronization among concurrent flushing and compaction activities as shown in Figure 18(d) 18(f).

5.3 Evaluation with NVMe SSDs

In this section, we deploy Flink and Kafka Streams benchmark applications on the nodes that equip NVMe SSDs to contain SSTable files of RocksDB instead of using in-memory tmpfs. We repeat the same experiments as in Section 5.1 and 5.2 and the results show that our mitigation methods are still effective in reducing the latency long tail problem caused by ShadowSync.

Results of Real-time Traffic Jam Ranking in Flink. We validate the effectiveness of our proposed mitigation methods in the Flink benchmark application using NVMe SSDs in Figure 19. This figure shows the tail latency performance comparison of the real-time traffic jam monitoring application in Flink with/without our mitigation methods in NVMe SSDs. For example, Figure 19(a) shows that our system faces the Statistical *ShadowSync* problem (see Section 3.3) across different Flink stages. Compared to our experimental results in Figure 8(a), we note that the tail latency performance of Flink in NVMe SSDs is worse than that in the case of in-memory `tmpfs`. For example, the 99.9th percentile latency of Flink in NVMe SSDs is up to 2.3sec in Figure 19(a), while it is less than 2sec in the case of in-memory `tmpfs` (see Figure 16(a))). This is due to the inevitable heavy disk I/O operations caused by the flushing activities, leading to non-negligible overhead and latency spikes

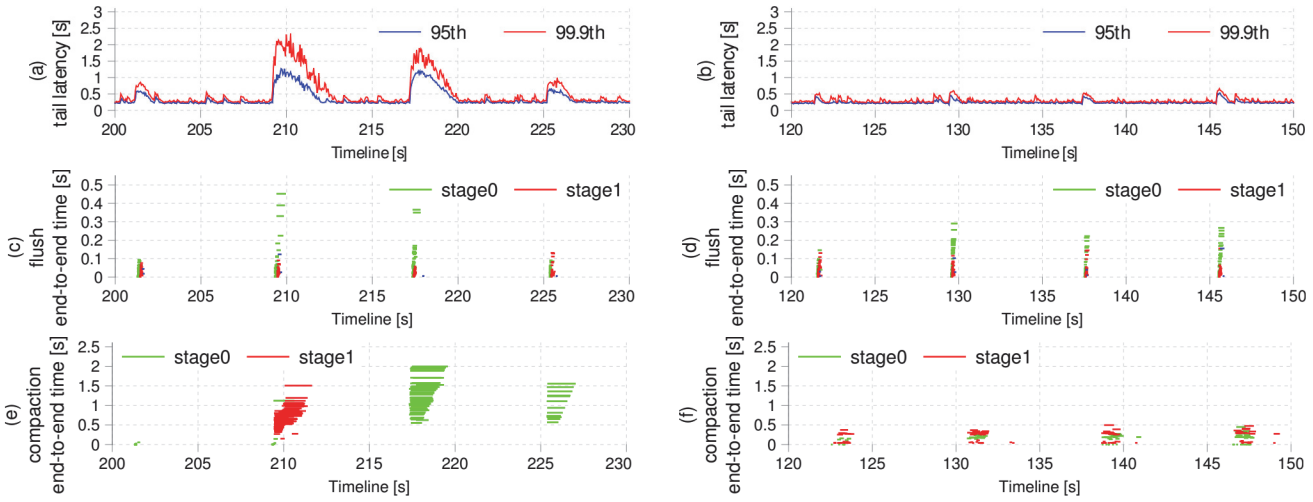


Figure 16. Our proposed mitigation methods can effectively reduce the overlap between concurrent flushing and compaction activities across different stage instances. Figure (a)(c)(e) show the performance metrics of our original benchmark application, while (b)(d)(f) show the corresponding metrics after our proposed mitigation methods with a 1sec delay are applied. We note that in (e)(f), the total number of compaction activities during these four checkpoints are the same, but (f) shows the compaction activities are evenly spread across these four checkpoints, while (e) shows more skewed compaction activities.

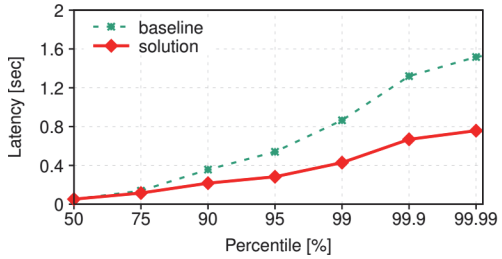


Figure 17. Tail latency comparisons between the original Word Count application in Kafka Streams with our proposed mitigation methods.

(detailed results are omitted due to space limitations). Figure 19(b) shows that our proposed mitigation methods can effectively reduce the latency long tail caused by the Statistical *ShadowSync* problem among highly concurrent flushing and compaction activities across different stages, suggesting the effectiveness of our proposed mitigation methods in alleviating the ShadowSync problem when running on top of SSDs.

Results of Real-time Word Count in Kafka Streams. Figure 20 shows the fine-grain performance comparison between the WordCount benchmark application without/with mitigation methods. Comparing the performance in Figure 20(a) with the baseline in Figure 18(a), the tail latency performance for Kafka Streams in NVMe SSDs is worse than that in the case of in-memory `tmpfs`, which is due to the inevitable heavy disk I/O operations caused by the flushing activities. Our experimental results in Figure 20(b) are consistent with our observations in the previous WordCount experiments using in-memory `tmpfs`, suggesting that our proposed mitigation methods also work well in alleviating the ShadowSync problem when running on top of SSDs.

6 Discussion

ShadowSync in different stream processing pipelines. While ShadowSync is mainly evaluated in our Flink and RocksDB (due to their high popularity) stream processing pipeline, it should also exist in other pipelines such as Storm [46], Spark [15], and Samza [14], which can also configure KV data stores as their backends to support continuous checkpointing for high fault-tolerance and fast recovery (e.g., other KV stores like Cassandra [17] that support incremental checkpointing). This is because modern stream processing pipelines typically involve hundreds or even thousands of processing units (or tasks) that share a limited number of physical machines with fixed CPU cores. In a continuous checkpointing scenario, hundreds (or more) of flushing and compaction activities (one for each stateful processing unit) will be launched upon each checkpoint; they will likely collide with each other by forming either scheduled or statistic ShadowSync as described in Section 3.

Other sources of ShadowSync. While we only show that flushing and compaction can cause ShadowSync, other asynchronous events including garbage collection (GC) in JVM [49], CPU Dynamic Voltage and Frequency Scaling (DVFS) control that adjusts CPU power based on dynamic workload [51], interference from VM collocations [50, 61] could also contribute to the ShadowSync problem in real-time stream processing. For example, we have observed that GCs activities in JVM with a conservative heap size allocation (e.g., 40 GB in our case) are more likely to occur during a flushing period since Flink processes a large amount of Java objects within a very short period of time. Power saving technologies like CPU DVFS control, if enabled, would likely occur more frequently under the dynamic workload and cause transient CPU throttling [51]. Given the highly dynamic nature of these asynchronous events, we believe the ShadowSync problem is more common than reported in this paper, which will be left for our future work.

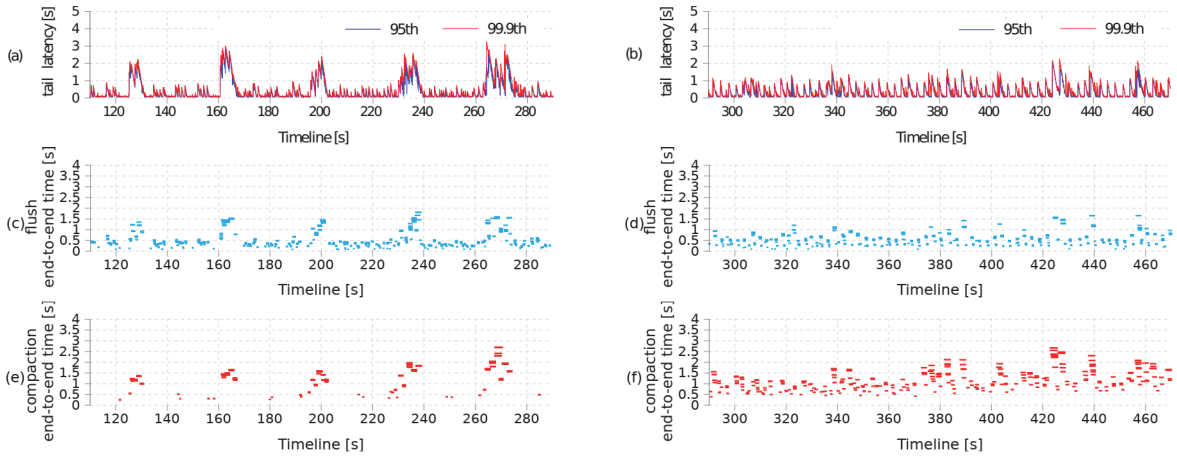
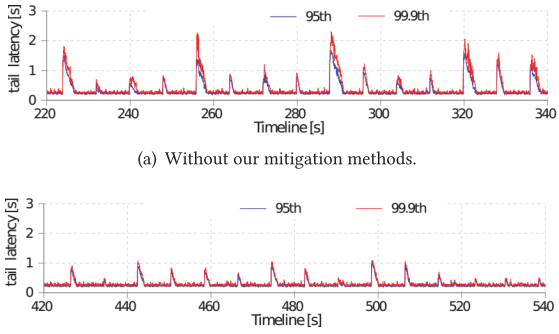


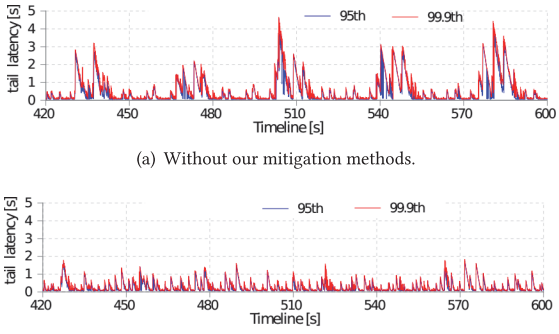
Figure 18. Our proposed mitigation methods can effectively reduce the overlap between highly concurrent flushing and compaction activities within a Kafka Streams Word Count application. Figure (a)(c)(e) show the performance metrics of the *baseline* case in Figure 17, while (b)(d)(f) show the corresponding metrics of the *solution* case in Figure 17 after our proposed mitigation methods are applied.



(a) Without our mitigation methods.

(b) With our mitigation methods.

Figure 19. Our mitigation methods can effectively reduce the latency long tail caused by *ShadowSync* in Flink when RocksDB is deployed on nodes that equip NVMe SSDs.



(a) Without our mitigation methods.

(b) With our mitigation methods.

Figure 20. Our mitigation methods can effectively reduce the latency long tail caused by *ShadowSync* in Kafka Streams when RocksDB is deployed on nodes that equip NVMe SSDs.

7 Related Work

Previous research on solving the latency long tail problem of real-time stream processing systems can be broadly categorized into three classes:

Optimizing checkpoint mechanisms to mitigate the latency long tail in streaming dataflows has been studied before [1, 2, 29, 30, 41, 54]. For example, SILK [3] mitigates latency long tail in LSM-based key-value stores by scheduling I/O bandwidth to internal flushing and compaction operations in a single key-value store instance to alleviate disk I/O contention between them. Incremental checkpointing [8] is a canonical way to avoid large intermittent internal bursts and improve the tail latency performance of large-scale streaming processing systems. These methods can effectively reduce the intensity of the frequent intermittent internal workload bursts, however, the periodic overlapped mode (i.e., *ShadowSync*) caused by flushing and compaction in a stream processing system still exists, which could lead to significant CPU contention and a latency long tail problem.

Identifying root causes of intermittent internal variance that causes millibottlenecks and latency long tail in the cloud [22, 49, 56]. For example, both Microscope [27] and MicroRCA [53] construct a service causal graph to infer the root causes of performance problems in real-time. FIRM [39] leverages fine-grained measurement data and machine-learning methods to adaptively localize components that cause SLO violations and identify the root causes of low-level resource contention in the critical paths. However, none of these approaches has the capability of detecting and locating the variance outside of the critical path, which could become a significant source of the latency long tail problem through the formation of millibottlenecks (see Section 3).

Providing automatic scaling techniques to elasticize resources (e.g., CPU) for cloud applications [18, 23, 24, 40, 59]. For example, autoscaling approaches for distributed dataflows [20, 26, 28, 55, 60] make scaling decisions on the number of underlying processing units (e.g., operator or container) without considering low-level shared-resource interference. FIRM [39] adopts a machine learning approach to abstract a class of resource contention problems. In our study, we conduct fine-grained (e.g., 50ms granularity) timeline analysis of metrics across different system components to help developers/system admins understand the "unexpected" *ShadowSync*

problem between maintenance operations and normal operations, which compliments their study.

To our best knowledge, the development of mitigating frequent millibottlenecks and latency long tail for real-time stream processing applications so far mainly focuses on alleviating the performance variance localized in the critical paths. However, we found millibottlenecks with intense resource requirements outside of critical paths can also cause significant latency long tail problems at moderate average utilization levels. Our work is the first paper to demonstrate the impact of millibottlenecks outside of critical paths on tail latency performance in real-time streaming applications.

8 Conclusion

Mission-critical, real-time, continuous stream processing applications such as real-time analytics and real-time advertising have stringent latency requirements due to their close interactions with the real world. Due to the evolution of real-world large-scale stream processing applications [34], the latency long-tail problem in real-time stream processing systems has received increasing scrutiny. A recent study [39] demonstrated a correlation between execution time variance of critical path and latency long tail in microservices systems. In this paper, our study shows that the variance in critical path latency may be the tip of an iceberg of latency long tail.

Using a benchmark of real-time traffic analytics, asynchronous workload bursts completely outside of the critical path are shown to cause significant latency long tail. These workload bursts are generated by *asynchronous maintenance tasks* inherent to the LSM-tree file structure: flushing in-memory data to disk, and compaction of layered files on disk. The workload bursts create short but intense CPU millibottlenecks, which in turn generate application-level queuing that results in latency long tail. The millibottlenecks and their queuing effects are aggravated by multiple maintenance tasks that overlap in time, a phenomenon called *ShadowSync*.

We show abundant evidence of overlapping ShadowSync phenomena in our experiments. They can be divided into two groups: scheduled (Section 3.2), caused by fixed periods between maintenance operations, and statistical (Section 3.3), where millibottlenecks are created by overlapping flushing and compaction operations from a large number of stages due to scale-out configurations. For each group, we describe and evaluate mitigation methods to avoid and/or reduce ShadowSync (Section 4) through desynchronization techniques. For example, randomizing the period between consecutive flushing (and compaction) operations reduces the probability of overlap for scheduled ShadowSync. Another example is a careful choice of soft resource allocation (e.g., flushing and compaction thread pools) to reduce the number and length of asynchronous operations and therefore reduce their statistical overlap and ShadowSync. Experiments confirm these mitigation methods to be effective in reducing ShadowSync (Section 5).

This improved understanding of ShadowSync in asynchronous workload bursts in real-time stream processing applications and their mitigation can reduce significantly the risks of real-time stream processing applications caused by latency long tail.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their feedback on improving this paper. This research has been partially funded by National Science Foundation by CNS (2000681), RCN (1550379), CRISP (1541074), SaTC (1564097) programs, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies mentioned above.

References

[1] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-Value Datastores. *Proc. VLDB Endow.* 8, 8 (apr 2015), 850–861. https://doi.org/10.14778/2757807.2757810

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 363–375.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[4] Netflix Technology Blog. 2016. *Application data caching using SSDs: The Moneta project: Next generation EVCache for better cost optimization.* Retrieved May 22, 2022 from https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef

[5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[6] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1087–1098. https://doi.org/10.1145/2882903.2904441

[7] Xie Chen, Yu Wu, Zhenghao Wang, Shujie Liu, and Jinyu Li. 2021. Developing real-time streaming transformer transducer for speech recognition on large-scale dataset. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (Toronto, ON, Canada). IEEE, 5904–5908.

[8] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2471–2486. https://doi.org/10.1145/3318464.3389723

[9] Facebook. 2021. *RocksDB.* Retrieved Feb 14, 2021 from https://rocksdb.org/

[10] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[11] Flink Forward. 2018. eBay Monitoring Platform Preprocessing and Alerting on Flink | Flink Forward | 9-10 April 2018 | San Francisco. Retrieved August 22, 2022 from https://sf-2018.flink-forward.org/index.html%3Fp=4168.html

[12] Flink Forward. 2018. Finding Bad Acorns | Flink Forward | 9-10 April 2018 | San Francisco. Retrieved August 22, 2022 from https://sf-2018.flink-forward.org/index.html%3Fp=4078.html

[13] The Apache Software Foundation. 2021. *Apache Flink®: Stateful Computations over Data Streams.* Retrieved Feb 14, 2021 from https://flink.apache.org/

[14] The Apache Software Foundation. 2021. *Apache Samza.* Retrieved Feb 14, 2021 from http://samza.apache.org/

[15] The Apache Software Foundation. 2021. *Dynamic resource allocation in spark.* Retrieved Feb 14, 2021 from https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

[16] The Apache Software Foundation. 2021. *Kafka.* Retrieved Feb 14, 2021 from https://kafka.apache.org/0102/documentation/streams/

[17] The Apache Software Foundation. 2022. Apache Cassandra | Apache Cassandra Documentation. Retrieved August 22, 2022 from https://cassandra.apache.org/_/index.html

[18] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3338–3352.

[19] Moojan Ghafurian, David Reitter, and Frank E. Ritter. 2020. Countdown Timer Speed: A Trade-off between Delay Duration Perception and Recall. *ACM Trans. Comput.-Hum. Interact.* 27, 2, Article 11 (mar 2020), 25 pages. https://doi.org/10.1145/3380961

[20] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (Dallas, TX, USA). IEEE, 1994–2004.

[21] Mike Gualtieri. 2021. *The Forrester Wave™: Streaming Analytics, Q2 2021.* Retrieved May 22, 2022 from https://www.forrester.com/report/the-forrester-wave-streaming-analytics-q2-2021/RES161547

[22] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-Hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 469–478. https://doi.org/10.1145/3038912.3052649

[23] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2020. WASP: Wide-Area Adaptive Stream Processing. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 221–235. https://doi.org/10.1145/3423211.3425668

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 783–798.

[25] Faria Kalim, Thomas Cooper, Huijun Wu, Yao Li, Ning Wang, Neng Lu, Maosong Fu, Xiaoyao Qian, Hao Luo, Da Cheng, Yaliang Wang, Fred Dai, Mainak Ghosh, and Beinan Wang. 2019. Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. Macao, China, 1886–1897. https://doi.org/10.1109/ICDE.2019.00204

[26] Sobhan Omranian Khorasani, Jan S. Rellermeyer, and Dick Epema. 2019. Self-Adaptive Executors for Big Data Processing. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 176–188. https://doi.org/10.1145/3361525.3361545

[27] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*. 3–20.

[28] Pinchao Liu, Dilma Da Silva, and Liting Hu. 2021. DART: A Scalable and Adaptive Edge Stream Processing Engine. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 239–252. https://www.usenix.org/conference/atc21/presentation/liu

[29] Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. 2020. FP4S: Fragment-based Parallel State Recovery for Stateful Stream Applications. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (New Orleans, LA, USA). IEEE, 1102–1111.

[30] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-Based Storage Systems. *Proc. VLDB Endow.* 13, 4 (dec 2019), 449–462. https://doi.org/10.14778/3372716.3372719

[31] Chen Luo and Michael J. Carey. 2020. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jan 2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[32] Mark Marchukov. 2017. *LogDevice: a distributed data store for logs.* Retrieved May 22, 2022 from https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/

[33] Hamid Nasiri, Saeed Nasehi, and Maziar Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Journal of Big Data* 6, 1 (2019), 1–24.

[34] Cao Duc Nguyen. 2020. *A Design Analysis of Cloud-based Microservices Architecture at Netflix.* Retrieved May 6, 2021 from https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f

[35] Xu Ning and Maxim Fateev. 2016. *Cherami: Uber Engineering's durable and scalable task queue in Go.* Retrieved May 22, 2022 from https://eng.uber.com/cherami-message-queue-system/

[36] The University of Utah. 2021. *CloudLab.* Retrieved Feb 14, 2021 from https://www.cloudlab.us/

[37] Takafumi Onishi, Julius Michaelis, and Yasuhiko Kanemasa. 2020. Recovery-conscious adaptive watermark generation for time-order event stream processing. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)* (Sydney, NSW, Australia). IEEE, 66–78.

[38] Calton Pu, Joshua Kimball, Chien-An Lai, Tao Zhu, Jack Li, Junhee Park, Qingyang Wang, Deepal Jayasinghe, Pengcheng Xiong, Simon Malkowski, et al. 2017. The millibottleneck theory of performance bugs, and its experimental verification. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (Atlanta, GA, USA). IEEE, 1919–1926.

[39] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 46, 21 pages.

[40] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Comput. Surv.* 51, 4, Article 73 (jul 2018), 33 pages. https://doi.org/10.1145/3148149

[41] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[42] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops (ICDCSW '11)*. IEEE Computer Society, USA, 166–171. https://doi.org/10.1109/ICDCSW.2011.20

[43] Scylla. 2022. *P99 CONF: the event for developers who care about high-performance, low-latency applications.* Retrieved May 16, 2022 from https://www.p99conf.io/

[44] Amazon Web Services. 2022. *Amazon Kinesis Data Analytics: Gain actionable insights from streaming data with serverless, fully managed Apache Flink.* Retrieved May 22, 2022 from https://aws.amazon.com/kinesis/data-analytics/

[45] Huasong Shan, Qingyang Wang, and Qiben Yan. 2018. Very Short Intermittent DDoS Attacks in an Unsaturated System. https://doi.org/10.1007/978-3-319-78813-5_3

[46] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[47] Selima Triki. 2022. The real-time data revolution. https://digazu.com/2022/04/07/the-real-time-data-revolution/

[48] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3132747.3132750

[49] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (Philadelphia, PA, USA). IEEE, 31–40.

[50] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. 2014. Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/trios14/technical-sessions/presentation/wang

[51] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien An Lai, Masazumi Matsubara, and Calton Pu. 2013. Impact of DVFS on N-Tier Application Performance. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (Farmington, Pennsylvania) *(TRIOS '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2524211.2524220

[52] Qingyang Wang, Shungeng Zhang, Yasuhiko Kanemasa, Calton Pu, Balaji Palanisamy, Lilian Harada, and Motoyuki Kawaba. 2019. Optimizing N-Tier Application Scalability in the Cloud: A Study of Soft Resource Allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 4, 2, Article 10 (jun 2019), 27 pages. https://doi.org/10.1145/3326120

[53] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium* (Budapest, Hungary). IEEE, 1–9.

[54] Hailu Xu, Pinchao Liu, Susana Cruz-Diaz, Dilma Da Silva, and Liting Hu. 2020. SR3: Customizable Recovery for Stateful Stream Processing Systems. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 251–264. https://doi.org/10.1145/3423211.3425681

[55] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)* (Berlin, Germany). IEEE, 22–31.

[56] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 389–405. https://www.usenix.org/conference/nsdi21/presentation/xu

[57] Wei D. Xu, Matthew J. Burns, Frédéric Cherqui, and Tim D. Fletcher. 2021. Enhancing stormwater control measures using real-time control technology: a review. *Urban Water Journal* 18, 2 (2021), 101–114. https://doi.org/10.1080/1573062X.2020.1857797 arXiv:https://doi.org/10.1080/1573062X.2020.1857797

[58] Hisatoshi Yamaoka, Kota Itakura, Eiichi Takahashi, Gaku Nakagawa, Julius Michaelis, Yasuhiko Kanemasa, Miwa Ueki, Tatsuro Matsumoto, Riichiro Take, Sayuri Tanie, and Daigo Inoue. 2019. Dracena: A Real-Time IoT Service Platform Based on Flexible Composition of Data Streams. In *2019 IEEE/SICE International Symposium on System Integration (SII)*. Paris, France, 596–601. https://doi.org/10.1109/SII.2019.8700465

[59] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th International Conference on Distributed*

Computing Systems (ICDCS) (Dallas, TX, USA). IEEE, 122–132.

[60] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)* (Milan, Italy). IEEE, 68–75.

[61] Shungeng Zhang, Huasong Shan, Qingyang Wang, Jianshu Liu, Qiben Yan, and Jinpeng Wei. 2019. Tail amplification in n-tier systems: a study of transient cross-resource contention attacks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (Dallas, TX, USA). IEEE, 1527–1538.