New Techniques to Curtail the Tail Latency in Stream Processing Systems *

Guangxiang Du
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA
gdu3@illinois.edu

Indranil Gupta
University of Illinois at Urbana-Champaign
Urbana, IL, 61801, USA
indy@illinois.edu

ABSTRACT

This paper presents a series of novel techniques for reducing the tail latency in stream processing systems like Apache Storm. Concretely, we present three mechanisms: (1) adaptive timeout coupled with selective replay to catch straggler tuples; (2) shared queues among different tasks of the same operator to reduce overall queueing delay; (3) latency feedback-based load balancing, intended to mitigate heterogenous scenarios. We have implemented these techniques in Apache Storm, and present experimental results using sets of micro-benchmarks as well as two topologies from Yahoo! Inc. Our results show improvement in tail latency up to 72.9%.

Keywords

Apache Storm, Stream Processing Systems, Tail Latency

1. INTRODUCTION

Stream processing systems have become extremely popular in the last few years, and they are being used to process a variety of data in real-time, ranging from social network feeds (to provide trending topics or real-time searches) to processing data from advertisement engines. Stream processing systems that are actively used in industry today include Apache Storm [3], Twitter's Heron [18], Apache Flink [1], Spark Streaming [27], Samza [6], etc.

Due to the real-time nature of these systems, responsiveness of the system is critical. Responsiveness means lowering the latency of processing a tuple of data, i.e., from its input into the system to its results reflecting to users.

Although many approaches have been proposed to reduce the latency, such as traffic-aware task scheduling [9, 25] and elastic scaling of the system [15, 22], etc., they are generally targeted at decreasing the average tuple latency without giving special consideration for the tail. For several applications, tail latency is more critical than average latency–examples include interactive web services, financial trading, and security-related applications.

The causes for tail latency have been well studied [13, 19]. The tail may be prolonged due to a variety of factors: network congestion, high resource utilization, interference, heterogeneity, highly variable I/O blocking, etc. Tail latency has received attention in areas like Web search engines [13, 17], high capacity data-stores [23] and datacenter networks [7, 24]. However, tail latency has received little attention in stream processing systems.

In this paper, we present three novel techniques to reduce the tail latency of stream processing systems. The high level ideas of our techniques bear similarity with some existing work, such as executive execution [8, 12], work stealing [10], yet those approaches cannot be applied to stream processing systems directly. Our first approach sets timeouts for tuples in an adaptive manner to catch the stragglers, and then replays tuples automatically. The latency of the fastest incarnation of a tuple is considered the tuple's latency (more details in Section 3.1). Our second technique seeks to merge input queues of several tasks of the same operator inside each worker process–this leverages well-known queueing theory results [5] (more details in Section 3.2). Our third technique targets heterogeneous scenarios, where we implement a novel latency-based load balancing scheme that is not as expensive as the power of two choices, but uses a similar intuition to gradually adjust load and achieve latency balance (more details in Section 3.3).

We implemented these three techniques into Apache Storm. Our experimental results with sets of micro-benchmarks as well as two topologies from Yahoo! Inc. show that we lower the 90^{th} latency by 2.2%-56%, the 99^{th} latency by 17.5%-60.8%, and the 99.9^{th} latency by 13.3%-72.9%.

2. SYSTEM MODEL

Before diving into our design, we outline our model for a stream processing job. This model is generally applicable to a variety of systems including Storm [3], Flink [1], Samza [6], etc.

1. A job is called a *topology*. It is a DAG (Directed Acyclic graph) of *operators*–in Storm, source operators are also referred to as spouts, non-source operators are referred to as bolts. We assume operators are *stateless*, as this covers a wide variety of applications, e.g., Storm assumes statelessness. Popular operator kinds include filter, transform, join, etc.

*This work was supported in part by the following grants: NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft.

DCC'16, July 25-28, 2016, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/2955193.2955206

2. Data flows along the edges of the DAG in the form of discrete units called *tuples*.
3. Tuples originate from the root nodes of the DAG, and output tuples exit out of the sink nodes.
4. Each operator is split into multiple tasks (as specified by the user).
5. The tuples arriving at a given operator are programmatically split across its tasks, using a *grouping mechanism*. Popular choices in Storm are shuffle grouping, fields grouping, and all grouping. In shuffle grouping, arriving tuples are spread randomly or round-robin across the operator's tasks. Shuffle grouping is the most popular among these choices, since: 1) it is useful in load balancing, 2) many operators are stateless (e.g., filter, transform), and these prefer shuffle grouping (in comparison, fields grouping is used by counters, all grouping is used for joins). As a result, in this paper we focus on DAGs that only use shuffle grouping.
6. At each machine, one or more *worker processes* are run. Each worker process is assigned some tasks from one topology.

3. DESIGN

We now describe the three techniques we use to curtail tail latency in a stream processing application: 1) Adaptive Timeout Strategy, 2) Improved Concurrency for Worker Process, and 3) Latency-based Load Balancing.

3.1 Adaptive Timeout Strategy

Consider a topology where operators use only shuffle grouping. By default, each tuple flows through a linear path of operators, starting from some spout. In fact, not only is it a linear path of operators, but a linear path of *tasks*. This increases the effect of congestion on that tuple–if any task on that linear path is congested, the tuple will be delayed.

However, we observed that it is possible that only some linear paths of tasks on a linear path of operators are congested, while other paths are not. This raises the possibility of replicating multiple instances of a tuple at or near the source operator and letting them choose potentially different paths. The latency of the fastest instance is then the given tuple's latency.

Replication, while popular in several systems [24], increases load significantly. Even if we were to replicate each tuple only once at the source operator, this would double the workload, and would not scale for systems that have a 50% utilization.

As a result, we propose to use an adaptive timeout strategy to *selectively* replay tuples that have not finished within the timeout. Though similar ideas have been proposed to address tail latency elsewhere, e.g., large scale web search [13], we are the first to apply it to distributed stream processing, to the best of our knowledge.

Our technique, shown in Algorithm 1, continuously collects the statistics of tuple latency, and periodically adjusts the timeout value based on latency distribution of recent issued tuples. Intuitively, we decide how aggressively (or not) to set the timeout value, based on how long the tail has been in the last adjustment period. This period parameter cannot be too low (O(ms)) because that would entail too much overhead, but it cannot be too high either (O(min)) because the system would not be responsive. Therefore we set it to O(sec), specifically 1 sec. For example, shown in Figure 1, at moment t_i, Algorithm 1 takes the statistics collected during period P_{i-1} as input to compute the timeout value for period P_i.

Algorithm 1 Adaptive Timeout Adjustment

```
1: procedure ADAPTIVE–TIMEOUT–ADJUSTMENT
2:     for adjustment period do
3:         Sort the array of collected tuple latencies
4:         Get the 90^th, 95^th, 99^th and 99.9^th latency
5:         if 99^th latency > 2 × 90^th latency then
6:             Timeout = 90^th latency
7:         else if 99.9^th latency > 2 × 95^th latency then
8:             Timeout = 95^th latency
9:         else
10:            Timeout = 99.9^th latency
11:        end if
12:        Clear the array of collected tuple latencies
13:    end for
14: end procedure
```

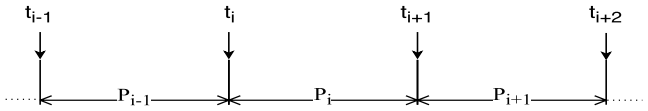

Figure 1: Time Diagram for Adaptive Timeout Strategy

The key idea of the algorithm is to measure the gaps between the 90^{th}, 95^{th}, 99^{th}, 99.9^{th} percentile latencies. If the tail is very long, we set the timeout aggressively. For instance, if the 99^{th} percentile latency is relatively high compared to the 90^{th} (line 5), then we set the timeout aggressively to be low. Otherwise, we set the timeout to a high value, to avoid unnecessary replay of tuples.

Only the spout tasks replay the straggler tuples that miss the timeout. This means that tuples are not duplicated at non-source operators.

3.2 Improved Concurrency For Worker Process

By default in today's systems (Storm [3], Flink [1]) each task has an independent queue to buffer incoming tuples. Our second technique to improve tail latency applies when a worker process contains more than one task from the same operator (and in the same topology). Then we can improve the latency by *merging* the input queues for these tasks. A task, whenever free, then opportunistically grabs the next available tuple from the shared input queue. The approach is illustrated in Figure 2. Since tasks subscribe shuffle-grouping streams (Section 2), this keeps the execution correct.

In an M/M/c queue model [5], let λ represent the queue's input rate, μ be the server's service rate, and c be the number of servers for the queue. Two quantities are important: the queue's utilization ρ (proportion of time the servers are occupied, with $\rho < 1$ required for the queue to be stable), and average queueing time Qt_{avg} (time items spend in the queue before they are served). They are derived as follows from [5]:

$$\rho = \frac{\lambda}{c\mu} \tag{1}$$

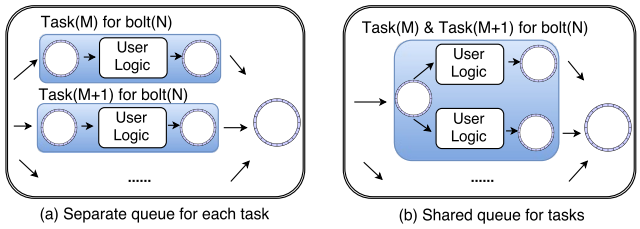

Figure 2: Modified Concurrency of worker process. (a) Default (original): Each task has an independent input queue. (b) Modified: Different tasks, inside the same worker process, of the same operator (which only consumes shuffle-grouping streams) share the input queue.

$$Qt_{avg} = \frac{\frac{(\frac{(c\rho)^c}{c!})(\frac{1}{1-\rho})}{\Sigma_{k=0}^{c-1}\frac{(c\rho)^k}{k!}+(\frac{(c\rho)^c}{c!})(\frac{1}{1-\rho})}}{c\mu-\lambda} \quad (2)$$

Figure 3 plots the variation of Qt_{avg} with c and ρ. It shows that for a given queue utilization, increasing the number of servers for a queue will lead to lower queueing time.

The decrease in queueing time is larger under higher queue utilization. This means this technique especially works well under high queue utilization.

While the M/M/c model assumed Poisson arrivals, Section 5 evaluates our technique under realistic input patterns. The synchronization across threads incurred in accessing the shared queue entails some overhead. This overhead is small, and our experiments in Section 5.1.2 show that the overall benefit outweighs this overhead.

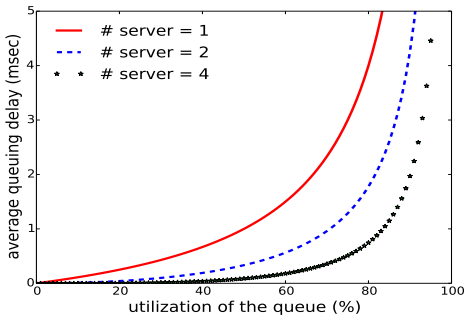

Figure 3: Average Queueing Time vs Utilization as number of servers is scaled up: From M/M/c queueing theory model.

3.3 Latency-based Load Balancing

Many stream processing systems run in heterogeneous conditions, e.g., the machines (or VMs) may be heterogeneous, the task assignment may be heterogeneous (machines have different number of tasks), etc. In these scenarios, some tasks may be faster than other tasks within the same operator. Partitioning the incoming stream of tuples uniformly across tasks thus exacerbates the tail latency.

The power of two choices in randomized load balancing [20] suggests that letting an incoming tuple choose the least loaded of two randomly chosen downstream tasks reduces latency (and can in fact perform close to optimal). However, this technique inherently relies on incoming tuples having up-to-date load information about the selected tasks. In fact we implemented the power of 2 choices approach and found that stale information leads to heavy oscillation. In a stream processing system with fast moving tuples being processed in milliseconds, continuously monitoring load statistics incurs prohibitive control traffic. This makes the power of two choices hard to implement as-is.

We propose a new technique called latency-based load balancing. The key idea is to collect statistics only periodically, and to *pair up* slow and fast tasks allowing the fast part of the pair to steal work. Our algorithm differs from the power of two choice mainly in two aspects: (1) periodic statistics collection; (2) smooth load adjustment among tasks to suppress load oscillation. Our algorithm is shown in Algorithm 2.

Algorithm 2 Latency-based Load Balancing

Notations:
D_i: $task_i$'s set of downstream tasks.
Wt_i: $task_i$'s aged latency. α: aging rate.
T_i: $task_i$'s average latency measured in last period.
thres: tolerance for difference among tasks' latencies within the same operator.

```
1: procedure LATENCY-BASED–LOAD–BALANCING
2:     for monitor period do
3:         for task_j in D_i do
4:             Collect T_j
5:             Wt_j ← α × T_j + (1 − α) × Wt_j
6:         end for
7:         Sort tasks in D_i based on Wt in ascending order
   and store them in array A[]
8:         for k in {0, 1, 2 ... |A|/2} do
9:             if A[|A|−1−k].Wt / A[k].Wt > thres then
10:                A[|A| − 1 − k].traffic- -
11:                A[k].traffic+ +
12:            else
13:                break
14:            end if
15:        end for
16:    end for
17: end procedure
```

The algorithm is activated once per monitoring period at each task, except at the sinks. For the same reasons outlined in Section 3.1, we set this period parameter to be O(sec), specifically 5 sec. We believe that the system is not sensitive to this value changing slightly. Each task collects latency statistics from its own immediate downstream tasks and uses Wt as criteria to sort the tasks from fastest to slowest. The tasks are then divided into two groups of equal size: faster tasks and slower tasks. Each slower task is paired up with a faster task. For each pair, the algorithm balances load by shifting one unit of traffic (1% of the upstream task's outgoing traffic in our implementation) from the slower task to the faster task at a time. The effect of the algorithm is that faster tasks steal load from slower tasks, and thus latencies of tasks converge over a period of time. Further, this algorithm requires only periodic collection of data, and thus has lower overhead than continuous load collection.

4. IMPLEMENTATION

We implemented and incorporated the three techniques described in the previous section into Apache Storm v 0.10.0.

Adaptive Timeout Strategy: Storm has a built-in mechanism to guarantee message processing [3] and provide atleast once semantics so that if a tuple has not been completely processed within timeout, it will get replayed. The timeout is typically specified by user and fixed once the topology is deployed. We embedded our adaptive timeout adjustment algorithm into this mechanism such that the system adaptively adjusts the timeout to catch only the straggler tuples. Another modification is also required: if any instances of a tuple, which has a unique identifier, finishes, no new instances of that tuple will be emitted from spout (already in flight instances will flow through the topology).

Improved Concurrency For Worker Process: Storm has an intermediate abstraction between a worker process and a task, called an *executor* [3]. Each executor runs one (by default) or more tasks of the same operator in a topology. It is the executor (not the task) that has an independent input queue. So, we apply our technique across executors instead of tasks. Each Storm worker has a data structure to record the metadata of the topology, including which operators the executors belong to, and which streams operators subscribe to. During the initialization phase of worker processes, if two or more executors are found to belong to the same operator, they will share a queue in our system.

Latency-based Load Balancing: Each task performs the load adjustment for its downstream tasks periodically. Different tasks perform the adjustment independently without synchronization. Note that this asynchrony means that upstream tasks may have slightly inconsistent views of statistics of downstream tasks. But this inconsistency does not affect correctness, i.e., the aggregate effect over multiple upstream tasks is close to the case if they had consistent measurements. In our implementation, the basic unit of traffic adjustment is 1% of the upstream task's outgoing traffic. The aging parameter (α), tolerance threshold ($thres$) and monitor period in Algorithm 2 are set to 0.5, 1.2 and 5 sec respectively.

5. EXPERIMENTAL RESULTS

We performed experiments on the Google Compute Engine [4]. Our default experimental settings are in Table 1. Each VM by default runs one worker process. We evaluate our three individual techniques separately on sets of micro-benchmarks (Section 5.1) as well as two topologies from Yahoo! (Section 5.2).

Table 1: Storm Cluster Configuration.

VM Node	Machine configuration	Role in the cluster
1 VM	n1-standard-1 (1 vCPU, 3.75GB memory)	Zookeeper & Nimbus
5 VMs	n1-standard-2 (2 vCPUs, 7.5GB memory)	Worker Node

5.1 Micro-benchmark Experiments

5.1.1 Adaptive Timeout Strategy

We compared the tail latency as well as the cost of different approaches: adaptive timeout strategy and different levels of blind tuple replication (0%, 20%, 50% and 100%). x% of replication means that a randomly chosen x% of tuples are issued with two instances at the spout task and the latency of the faster one is regarded as the tuple's latency. The benchmark is a "Exclamation Topology" (a 4-operator linear topology connected by shuffle grouping stream) from Storm example topologies, where each operator has 5 tasks. The experimental results are summarized in Table 2. The cost column represents increased workload compared with the default method, i.e., 0% replication.

Table 2: 99^{th}, 99.9^{th} Latency and Cost of Adaptive Timeout Strategy versus the Replication Approach (spout-only replication).

Approach	99^{th} latency (ms)	99.9^{th} latency (ms)	Cost
default	29.2	76.6	—
adaptive timeout	24.1	66.4	2.92%
20% replication	25.5	87.8	20%
50% replication	22.1	107.7	50%
100% replication	17.9	78.1	100%

The adaptive timeout strategy improves the 99^{th} percentile latency and 99.9^{th} latency by 17.5% and 13.3% respectively compared to the default. The adaptive timeout strategy is better than 20% replication not only by providing lower latency but also by incurring less cost. Although 50% and 100% replication achieve lower 99^{th} latency than the adaptive timeout strategy, they incur prohibitively high cost. Thus, the adaptive timeout technique can serve as an effective alternative to replication, especially when the system cannot afford the expense of replicating many tuples.

5.1.2 Improved Concurrency For Worker Process

We use a micro-topology where a spout connects to a bolt through shuffle-grouping stream. This is small but representative of a part of a larger topology. The bolt is configured with 20 executors, so each worker process is assigned with 4 executors. For each executor's input queue, $\lambda = 350$ tuples/s and $\mu = 450$ tuples/s, thus its $\rho = 78\%$. We examine the improvement of merging executors' input queues with respect to the Storm default, i.e., an independent input queue for each executor. Our experimental results show that the average queueing delay drops from 2.07 ms to 0.516 ms and this translates to reduced tail latency. The 90^{th} latency, 99^{th} latency and 99.9^{th} latency are improved by 3.49 ms (35.5%), 3.94 ms (24.9%) and 30.1 ms (36.2%) respectively.

5.1.3 Latency-based Load Balancing

We experiment with three kinds of heterogeneous scenarios: (a) Different Storm workers are assigned different numbers of tasks; (b) Subset of Storm workers are competing for resources with external processes (residing in the same VM); (c) Storm workers are deployed in a cluster of heterogeneous VMs. When the tasks are computation-intensive, the heterogeneity they experience is severe.

The benchmark is a 4-operator linear topology connected by shuffle grouping stream, where each bolt task performs 200,000 arithmetic operations. In detail, the scenarios are:

Scenario-a: The spout has 5 tasks and the bolts have 3

tasks each, thus total number of tasks in the topology is 14. The default Storm scheduler assigns 4 VMs with 3 tasks each and the fifth VM with 2 tasks.

Scenario-b: All operators each have 5 tasks, evenly distributed in 5 worker processes. Within 2 of 5 VMs, we run a data compression program as an external process that causes interference (and thus heterogeneity).

Scenario-c: All operators each have 5 tasks, evenly distributed in 5 worker processes. The topology is deployed in a cluster of 4 'n1-standard-2' VMs and a 'n1-standard-1' VM.

The experimental results are summaried in Table 3. We observe that the 90^{th}, 99^{th}, 99.9^{th} latency are reduced by 2.2%-56%, 21.4%-60.8% and 25%-72.9%, respectively.

Table 3: Latency Achieved by Latency-based Load Balancing under different scenarios. 'D' means the default approach: split traffic evenly across tasks. 'L' means the Latency-based load balancing approach.

Scenario	90^{th} latency (ms)		99^{th} latency (ms)		99.9^{th} latency (ms)	
	D	L	D	L	D	L
(a)	11.15	10.9	29.9	23.5	104.5	57.1
(b)	21.05	9.16	92.9	36.4	204.3	153.1
(c)	9.3	7.67	127.8	62.4	598.6	162

5.1.4 *Conditions for Applying the Techniques*

Figure 4 shows that the adaptive timeout strategy is preferable over the improved concurrency for worker process when the utilization of bolt executor's input queue is low (<58%, under our experiment settings). While this intersection point may differ for different scenarios, this plot shows qualitatively when each technique is preferable.

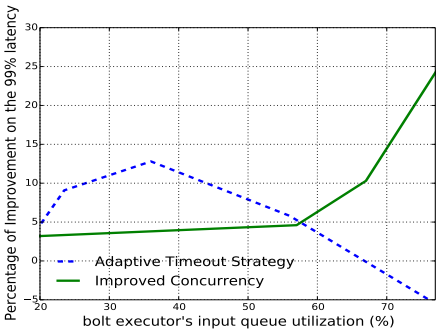

Figure 4: Effect of executor's input queue utilization on the Adaptive Timeout Strategy and the Improved Concurrency for Worker Process.

Figure 5 shows the latency-based load balance works well under high workload when the heterogeneity among different VMs is most prominent. The adaptive timeout strategy achieves improvement on tail latency under moderate or low system workload (<49% CPU utilization, under our experiment settings).

We summarize the recommended guidelines for applying each technique in Table 4. Since the scopes of each different techniques do not overlap with each other, we recommend using each technique exclusively for its own targeted scenarios.

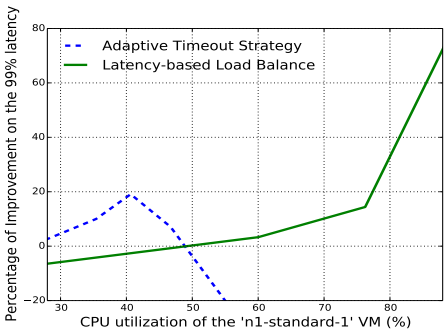

Figure 5: Effect of workload on the Latency-based Load Balance and the Adaptive Timeout Strategy.

Table 4: Conditions for Applying Techniques

Technique Name	Conditions for applying the technique
Adaptive timeout strategy	moderate or low system workload && moderate or low utilization for queues in the systems.
Improved concurrency for worker process	$\geq$ 2 tasks of the same operator in worker process && high utilization for tasks' input queues.
Latency-based load balance	heterogeneity within systems causes some tasks to be much slower than others within the same operator.

5.2 Yahoo! Benchmark Experiments

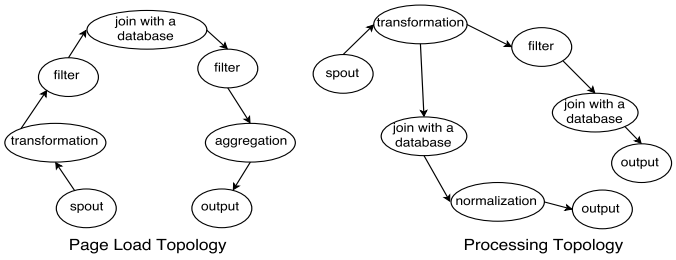

Figure 6: Yahoo! Benchmark topology layout

We evaluate our techniques using two Yahoo! topologies (Page Load Topology and Processing Topology) [26], shown in Figure 6. The settings for different techniques are different such that each technique is evaluated under their targeted scenarios.

For adaptive timeout: Each operator has 5 tasks, evenly distributed among 5 worker processes. Each spout task emits 240 tuples/s.

For improved concurrency: The spout and output bolt each have 5 tasks, evenly distributed among 5 worker processes. Other bolts each have 4 tasks and *all tasks of a given operator* are assigned to the same worker process. Each spout task emits 530 tuples/s, the bolt tasks have 1 ms delay. For each task's input queue, $\lambda = 660$ tuples/s and $\mu = 830$ tuples/s, thus its $\rho = 80\%$.

For the latency-based load balance: Each operator each have 5 tasks, evenly distributed among 5 worker processes. The topologies are deployed in a heterogeneous cluster (4 'n1-standard-2' VMs and a 'n1-standard-1' VM). Each spout task emits 600 tuples/s.

We summarize the results in Table 5: for detailed plots, see [14].

Table 5: Latency Improvement Achieved by our three techniques with two Yahoo! Benckmark Topologies

Name	90^{th} latency	99^{th} latency	99.9^{th} latency
Adaptive timeout strategy	—	28%-40%	24%-26%
Improved concurrency	16%-19%	36%-42%	20%-32%
Latency-based load balance	22%-48%	50%-57%	21%-50%

6. RELATED WORK

A variety of techniques have been proposed to shorten tail latency in different areas.

In networking, reducing tail latency by redundancy has been applied, including issuing multiple DNS queries in parallel to resolve the same name [24], replicating the SYN packets or even the entire flow to avoid uncertainty.

In batch processing systems such as MapReduce [12] and Apache Spark [2], when a job is close to completion, the master schedules backup execution for the straggler tasks, called as speculative execution.

In large scale Web services platforms, the concept of hedged requests and tied requests have been proposed [13]. A hedged request means that a secondary request would be sent if the first request has not finished within 95^{th}-percent of expected latency. A tied request is an enhancement of hedged request in that when the first copy of request is scheduled to execute, the second copy would be canceled.

In applications like RPC server, Memcached and Nginx, increasing the number of workers (CPUs) at the server, with parallel workers pulling requests from a shared queue, can improve tail latency [19].

In shared networked storage, a combination of per-workload priority differentiation and rate limiting [28] has been shown effective.

In stream processing systems, several operator-specific optimizations, such as lock-free concurrent data structures [11, 16], for multi-way merge sort/aggregation and low latency handshake algorithm [21] for window join, have been proposed.

7. CONCLUSION

In this paper, we presented three novel techniques to shorten the tail latency in stream processing systems. We also analyze the conditions required for applying each technique. Our implementation on top of Apache Storm shows that these techniques lower the tail latency up to 72.9% compared with the default Storm implementation.

8. REFERENCES

[1] Apache Flink. https://flink.apache.org/, 2016.

[2] Apache Spark. http://spark.apache.org/, 2016.

[3] Apache Storm. http://storm.apache.org/, 2016.

[4] Google Compute Engine. https://cloud.google.com/compute/, 2016.

[5] M/M/c queue. https://en.wikipedia.org/wiki/M/M/c_queue/, 2016.

[6] Samza. http://samza.apache.org/, 2016.

[7] M. Alizadeh et al. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proc. USENIX NSDI*, pages 19–19, 2012.

[8] G. Ananthanarayanan et al. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proc. USENIX OSDI*, pages 265–278, 2010.

[9] L. Aniello et al. Adaptive online scheduling in Storm. In *Proc. ACM DEBS*, pages 207–218, 2013.

[10] R. D. Blumofe et al. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[11] D. Cederman et al. Brief announcement: Concurrent data structures for efficient streaming aggregation. In *Proc. ACM SPAA*, pages 76–78, 2014.

[12] J. Dean et al. MapReduce: simplified data processing on large clusters. *CACM*, pages 107–113, 2008.

[13] J. Dean et al. The tail at scale. *CACM*, 56(2):74–80, 2013.

[14] G. Du. "New Techniques to Lower Tail Latency in Stream Processing Systems", MS Thesis, UIUC, 2016. http://dprg.cs.uiuc.edu/docs/Guangxiang_thesis/DU-THESIS-2016.pdf.

[15] T. Z. J. Fu et al. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *Proc. IEEE ICDCS*, pages 411–420, 2015.

[16] V. Gulisano et al. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In *Proc. IEEE BigData*, pages 144–153, 2015.

[17] M. Jeon et al. Predictive parallelization: Taming tail latencies in web search. In *Proc. ACM SIGIR*, pages 253–262, 2014.

[18] S. Kulkarni et al. Twitter Heron: Stream processing at scale. In *Proc. ACM SIGMOD*, pages 239–250, 2015.

[19] J. Li et al. Tales of the tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proc. ACM SoCC*, pages 1–14, 2014.

[20] M. Mitzenmacher. The power of two choices in randomized load balancing. *Proc. IEEE TPDS*, 12(10):1094–1104, 2001.

[21] P. Roy et al. Low-latency handshake join. *Proc. VLDB Endow.*, 7(9):709–720, 2014.

[22] S. Schneider et al. Elastic scaling of data parallel operators in stream processing. In *Proc. IEEE IPDPS*, pages 1–12, 2009.

[23] L. Suresh et al. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proc. USENIX NSDI*, pages 513–527, 2015.

[24] A. Vulimiri et al. Low latency via redundancy. In *Proc. ACM CoNEXT*, pages 283–294, 2013.

[25] J. Xu et al. T-Storm: Traffic-Aware Online Scheduling in Storm. In *Proc. IEEE ICDCS*, pages 535–544, 2014.

[26] L. Xu et al. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *Proc. IEEE IC2E*, 2016.

[27] M. Zaharia et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proc. USENIX HotCloud*, pages 10–10, 2012.

[28] T. Zhu et al. PriorityMeister: Tail latency QoS for shared networked storage. In *Proc. ACM SoCC*, pages 1–14, 2014.