NEW TECHNIQUES TO LOWER THE TAIL LATENCY IN STREAM PROCESSING SYSTEMS

BY

GUANGXIANG DU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Indranil Gupta

ABSTRACT

Over the past decade, the demand for real time processing of huge amount of streaming data has emerged and grown rapidly. Apache Storm, Apache Flink, Samza and many other stream processing frameworks have been proposed and implemented to meet this need. Although lots of effort has been made to reduce the average latency of stream processing systems, how to shorten their tail latency has received little attention.

This thesis 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 in the range of 2%-72.9%.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Indranil Gupta, for his guidance, advice, patience and support for my thesis research project. Without his assistance and invaluable suggestions on improving writing skills, this thesis would not have been possible.

I would like to thank Le Xu, for sharing two topologies layout from Yahoo! Inc. used in her Stela project as well as her help and advice.

I also want to extend my sincere gratitude to Qi Wang and Hongwei Wang, my teammates for CS 525 course project. My thesis research project is based on the work of our course project in Advanced Distributed Systems. Therefore, I want to thank them for inspiration, research suggestions and collaboration.

Last of all, I would like to thank my parents and my friends, for their love, encouragement and help throughout the journey.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

Stream processing systems have become extremely popular in the last few years [1], 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 [2], Twitter's Heron [3], Apache Flink [4], Spark Streaming [5], Samza [6], S4 [7], 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 (e.g., on admin dashboards).

Although many approaches have been proposed to reduce the latency, such as traffic-aware [8, 9], resource-aware [10, 11], priority-based [12] task scheduling as well as elastic scaling of the system [13, 14, 15, 16], etc., they generally target at decreasing the average tuple latency without giving special consideration for the tail. However, for applications that require consistently high performance, like interactive web service, financial trading or security-related applications, tail latency is more critical than average latency.

The causes for tail latency have been well studied [17, 18, 19, 20]. The tail may be prolonged due to a variety of factors: network congestion, high resource utilization, interference, heterogeneity, highly variable I/O blocking, power saving mode of the hosts and etc. Tail latency has received attention in areas like Web search engines [21, 22, 23, 24], high capacity data-stores [25, 26] and datacenter networks [27, 28, 29, 30, 31]. However, tail latency has received little attention in stream processing systems.

In this thesis, we present three novel techniques to reduce the tail latency of stream processing systems. The high level ideas of our techniques share bare similarity with some existing work, such as execulative execution [32, 33], work stealing [34, 35], 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 [36] (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 the 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%.

1.2 Contributions of this Thesis

The contributions of the thesis are:

(i) We show why traditional methods such as blind replication and the power of two choices in load balancing, may not be appropriate for the problem we focus on. Then we propose three techniques: (1) Adaptive Timeout Strategy coupled with selective replay; (2) Improved Concurreny for worker process by merging input queue for tasks of the same operator; and (3) Latency feedback-based Load Balance.

(ii) We implement these three techniques on Apache Storm, one of the most popular stream processing systems.

(iii) We perform evaluation on our three techniques and compare them with the Storm default implementation. Our techniques achieve improvement in tail latency in the range of 2%-74%.

CHAPTER 2

SYSTEM MODEL

In this chapter, we outline our system model for a stream processing job. This model is generally applicable to a variety of systems including Storm [2], Flink [4], Samza [6], etc.

(i) 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.

(ii) Data flows along the edges of the DAG in the form of discrete units called *tuples.*

(iii) Tuples originate from the root nodes of the DAG, and output tuples exit out of the sink nodes.

(iv) Each operator is split into multiple tasks (as specified by the user).

(v) The tuples arriving at a given operator can be programmatically split across the tasks, using a *grouping mechanism.* Popular choices in

Apache Storm include shuffle grouping, fields grouping, and all grouping.

Shuffle grouping: tuples arriving at an operator are spread in a random and round-robin way across its constituent tasks.

Fields grouping: tuples are partitioned by the fields specified in the grouping. For instance, in the WordCount topology, the stream is grouped by “word” field, then tuples with the same “word” field will be sent to the same task.

All grouping: all tuples are replicated at all the operator’s tasks.

Stateless DAGs predominantly use shuffle grouping, e.g., filter, transform operators consume shuffle grouping streams. As a result, the rest of the thesis assumes shuffle grouping at all operators.

(vi) At each machine, one or more *worker processes* are run. Each worker process is assigned some tasks from one topology.

CHAPTER 3

DESIGN

In this chapter, we describe the three techniques we use to curtail tail latency in a stream processing application: 1) Adaptive Timeout Strategy, 2) Improved Concurrency Model 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 source operator. In fact, not only is it a linear path of operators, but also a linear path of *tasks*. As shown in Figure 3.1 a given linear path of operator can have multiple linear paths 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.

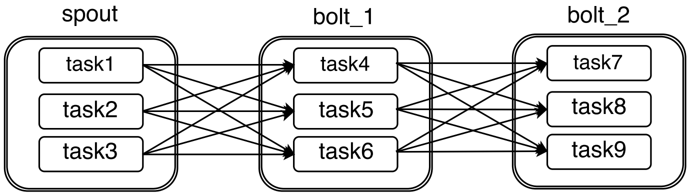


Figure 3.1: A Simple Three-stage Topology Connected Through Shuffle-grouping Streams. Any Tuple From a Spout Task has 9 Possible Source-Sink Paths.

The latency of the fastest instance is then the given tuple's latency.

Replication, while proven effective in several systems [28, 29, 30], 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 been fully processed within the timeout. Though similar ideas have been proposed to address tail latency elsewhere, e.g., large scale web search [21], 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 (set to 1 sec in our implementation). For example, shown in Figure 3.2, at moment t_i, Algorithm 1 takes the statistics collected during interval I_{i-1} as input to compute the timeout value for interval I_i.

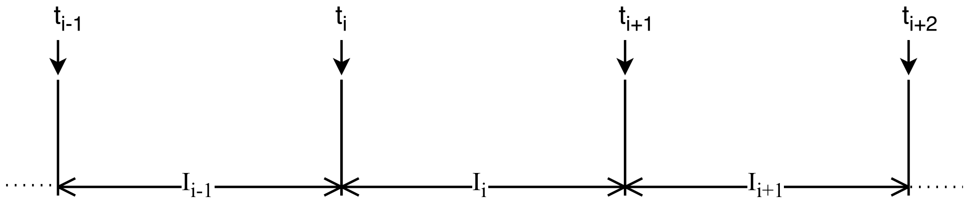


Figure 3.2: Time Diagram for Adaptive Timeout Strategy

Algorithm 1 Adaptive Timeout Adjustment

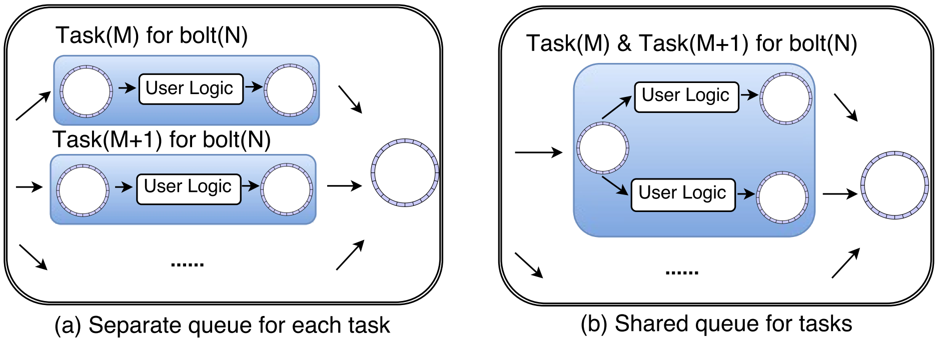
```
 1: procedure ADAPTIVE–TIMEOUT–ADJUSTMENT
 2:     for adjustment period do
 3:         Sort the array of collected tuple latencies
 4:         Get the $90^{th}$, $95^{th}$, $99^{th}$ and $99.9^{th}$ latency
 5:         if $99^{th}$ latency $> 2 \times 90^{th}$ latency then
 6:             $Timeout = 90^{th}$ latency
 7:         else if $99.9^{th}$ latency $> 2 \times 95^{th}$ 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
```

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 [2], Flink [4]) 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 3.3. Since we assume shuffle grouping stream at the operator (Chapter 2), this keeps the execution correct.



(a) Separate queue for each task

(b) Shared queue for tasks

Figure 3.3: Modified Concurrency Model of Worker Process. (a) Default (original): Each task has an independent input buffer. (b) Modified: Different tasks, inside the same worker process, of the same operator (whose input only consists of shuffle grouping streams) share the input buffer.

In an M/M/c queue model, 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 [36]:

$$\rho = \frac{\lambda}{c\mu} \tag{3.1}$$

$$Qt_{avg} = \frac{\frac{(\frac{(c\rho)^c}{c!})(\frac{1}{1-\rho})}{\Sigma_{k=0}^{c-1}\frac{(c\rho)^k}{k!}+(\frac{(c\rho)^c}{c!})(\frac{1}{1-\rho})}}{c\mu - \lambda} \tag{3.2}$$

Figure 3.4 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.

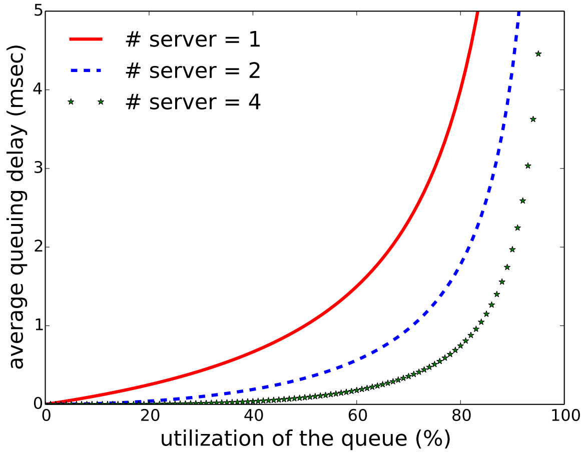


Figure 3.4: Average Queueing Time vs Utilization as number of servers is scaled up: From M/M/c queueing theory model.

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, Chapter 5 evaluates our technique under realistic input patterns.

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 [37, 38, 39] 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 a stream processing system with fast moving tuples being processed in microseconds, 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.

```
procedure LATENCY-BASED–LOAD–BALANCING
    for monitor period do
        for $task_j$ in $D_i$ do
            Collect $T_j$
            $Wt_j \leftarrow \alpha \times T_j + (1 - \alpha) \times Wt_j$
        end for
        Sort tasks in $D_i$ based on $Wt$ in ascending order and store them in array $A[\,]$
        for k in $\{0, 1, 2 \ldots |A|/2\}$ do
            if $\frac{A[|A|-1-k].Wt}{A[k].Wt} > thres$ then
                $A[|A| - 1 - k]$.traffic- -
                $A[k]$.traffic+ +
            else
                break
            end if
        end for
    end for
end procedure
```

The algorithm is activated once per monitoring period (set to 5 sec in our implementation) at each task, except at the sinks. 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.

CHAPTER 4

IMPLEMENTATION

In this chapter, we will discuss our implementation of three techniques we described in Chapter 3 on top of Apache Storm. Apache Storm [2] is one of the most popular open-source, highly scalable, fault-tolerant distributed stream processing framework used in industry for realtime processing today. We implemented the three techniques based on Apache Storm version 0.10.0.

4.1 Adaptive Timeout Strategy

Storm has a built-in mechanism to guarantee message processing [40] and provide at-least once semantics so that if a tuple has not been completely processed within timeout, it will get replayed. Every spout task stores records of the pending tuples, i.e., not completely processed tuples, that it issues. When a tuple is fullly processed, its associated acker task will notify its originating spout task to erase its record. If a tuple's record remains in a spout task longer than timeout, the spout task regards that tuple as failed and retransmits it. 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 replayed from spout (already in flight instances will flow through the topology).

4.2 Improved Concurrency For Worker Process

Storm has an intermediate abstraction between a worker process and a task, called an *executor* [41]. Their relationships are illustrated in Figure 4.1.

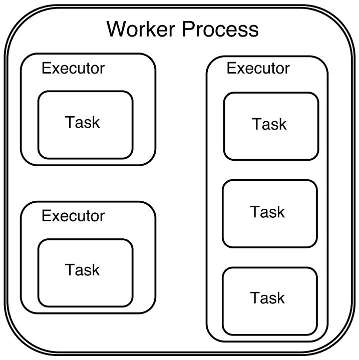


Figure 4.1: Relationship of Worker Process, Executor, Task

Each *Worker Process* is a JVM process. A machine (or VM) may run one or more worker processes for one or more topologies. A worker process runs executors for a specific topology.

Each *Executor* is a processing thread within a worker process. 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.

Each *Task* is the actual data processing logic.

Therefore, 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 bolt, they will share a queue in our system.

4.3 Latency-based Load Balancing

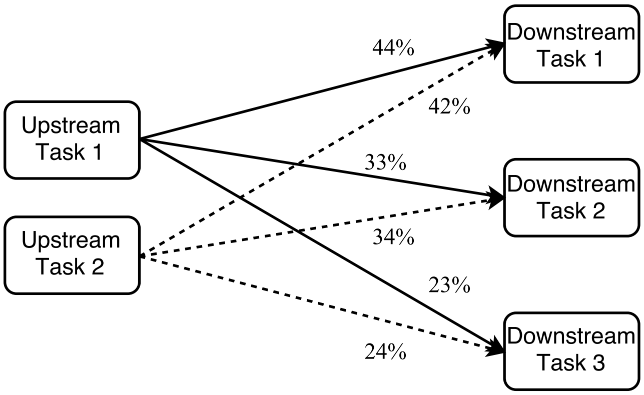


Figure 4.2: Asynchronous, Independent Load Adjustment of Upstream Tasks. The upstream tasks perform load adjustment at different moments, and they make slightly different decisions. 44%, 33% and 23% of upstream task1's outgoing traffic are fed into downstream task1, 2, 3 respectively. 42%, 34% and 24% of upstream task2's outgoing traffic are fed into downstream task1, 2, 3 respectively.

Each task performs the load adjustment for its downstream tasks periodically. Different tasks perform the adjustment independently without synchronization, as shown in Figure 4.2. Note that this asynchrony means that upstream tasks may have slightly inconsistent views of statistics of downstream tasks. However, 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.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we present evaluation of our three techniques. We performed experiments on the Google Compute Engine [42]. Our default experimental settings are in Table 5.1. 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 5.1: Storm Cluster Configuration.

VM Node	Machine configuration	Role
1 VM (Master node)	n1-standard-1 (1 vCPU, 3.75GB memory)	Zookeeper [43] & Nimbus
5 VMs (storm1-storm5)	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.

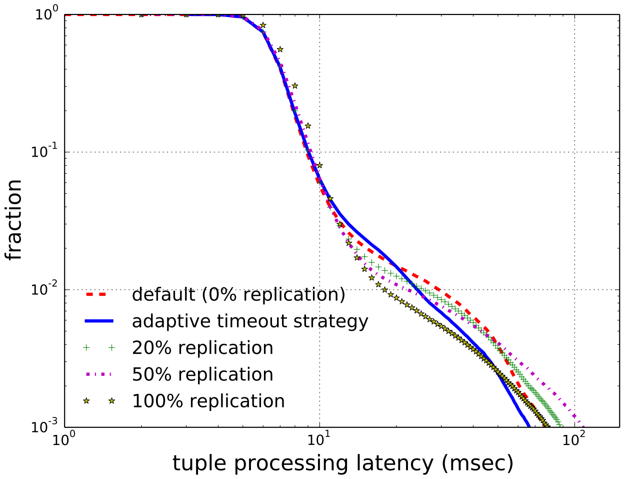


Figure 5.1: Latency Comparison between Adaptive Timeout Strategy and Replication Approach (spout-only replication).

Table 5.2: 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 experimental results are shown in Figure 5.1, which is a complementary cumulative distribution function (CDF). The (x,y) point on the plot means that y fraction of tuples experience a latency of at least x ms. The same results are also summarized in Table 5.2. The cost column represents increased workload compared with the default method, i.e., 0% replication.

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 are plotted in Figure 5.2 and Figure 5.3. It shows 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.

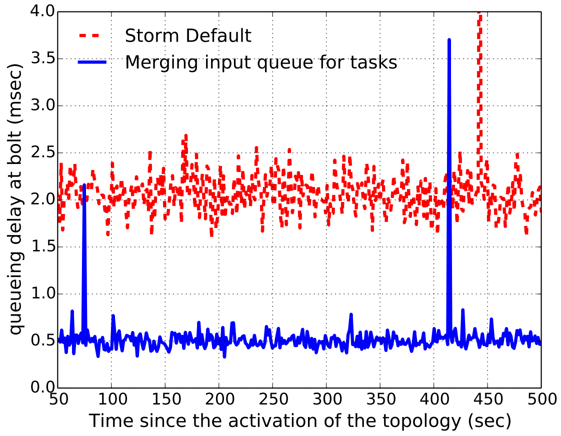


Figure 5.2: Merging input queues among executors reduces queueing delay.

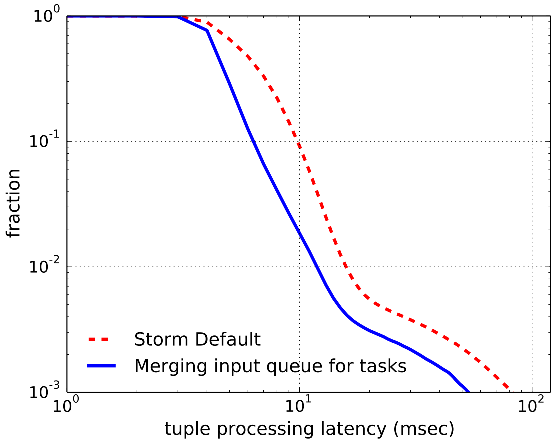


Figure 5.3: Latency Achieved by Improved Concurrency Model for Storm Worker.

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.

As shown in Figure 5.4 and Figure 5.5, the latency-based load balancing technique shifts load from slower tasks to faster tasks to achieve latency balance among tasks from the same operator.

The experimental results are summaried in Table 5.3. 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.

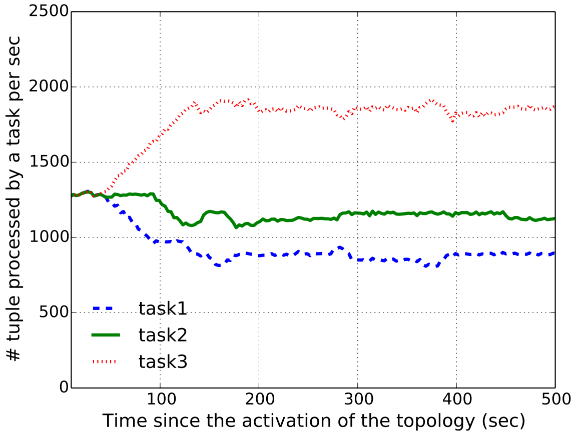


Figure 5.4: Load of different tasks from the same bolt changes over time. Faster tasks attract more load.

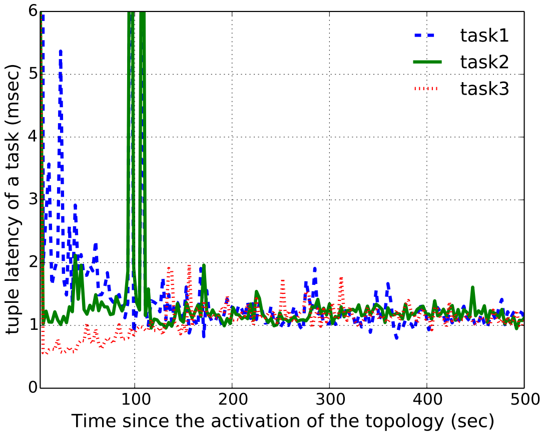


Figure 5.5: Latency of different tasks from the same bolt changes over time. Latencis of different tasks converge.

Table 5.3: 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

With the benchmark used in Section 5.1.2, we vary the tasks's input queue utilization (ρ) and observe its effect on the adaptive timeout strategy and the improved concurrency model, shown in Figure 5.6. For the adaptive timeout strategy, its improvement on tail latency first increases and then decreases as tasks' input queue utilization rises. For the improved concurrency model, there is a positive correlation between tasks' input queue utilization and its improvement on tail latency, as we showed in Section 3.2. Overall, we see that the adaptive timeout strategy is preferable over the improved concurrency model when the utilization of bolt task's input queue is low (<58%, under our experiment settings).

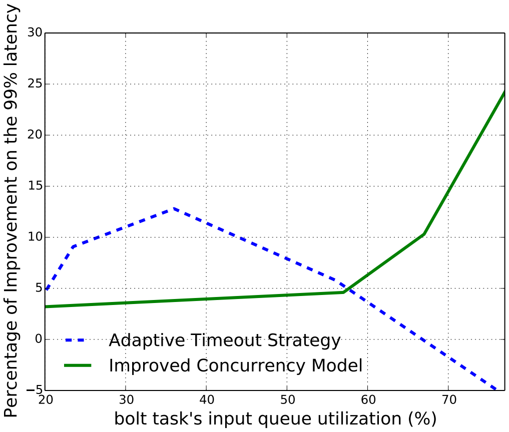


Figure 5.6: Effect of tasks's input queue utilization on the Adaptive Timeout Strategy and the Improved Concurrency Model.

With the benchmark used in scenario-c of Section 5.1.3, we vary the system workload and observe its effect on the latency-based load balance and the adaptive timeout strategy, shown in Figure 5.7. 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).

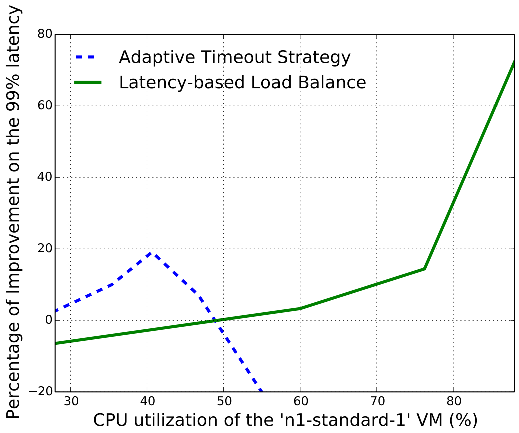


Figure 5.7: Effect of workload on the Latency-based Load Balance and the Adaptive Timeout Strategy.

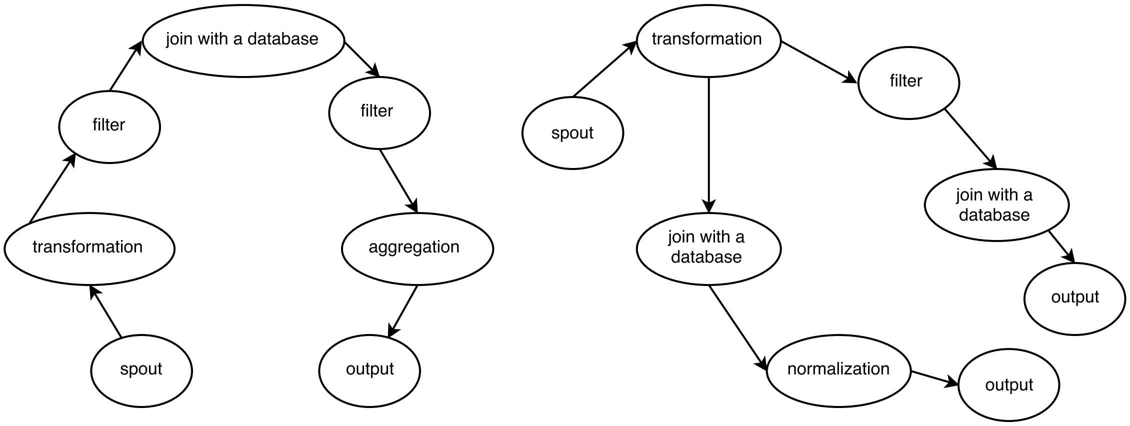
The suggested conditions for applying each technique are summarized in Table 5.4. Since the scopes of each different techniques do not overlap with each other, we recommend using each technique exclusively for its own targeted scenarios.

Table 5.4: Conditions for Applying Techniques

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 Model	$\geq$ 2 tasks of the same operator in a worker process && high utilization for tasks' input queues.
Latency-based Load Balance	heterogeneity within systems causes some tasks to be much slower than others of the same operator.

5.2 Yahoo! Benchmark Experiments

We acquired two Yahoo! topologies (Page Load Topology and Processing Topology) from the Stela project [44]. The layout of these two topologies is shown in Figure 5.8. We evaluate our three techniques and compare them with the Storm default implementation.



(a) Page Load Topology (b) Processing Topology

Figure 5.8: Yahoo! Benchmark topology layout

Our three sets of experiments use Page Load Topology and Processing Topology, yet the settings are different such that each technique is evaluated under their targeted scenarios.

Experiment settings for the adaptive timeout strategy are as follows: Each operator has 5 tasks, evenly distributed among 5 worker processes in round robin manner. Each spout task emits 240 tuples per second.

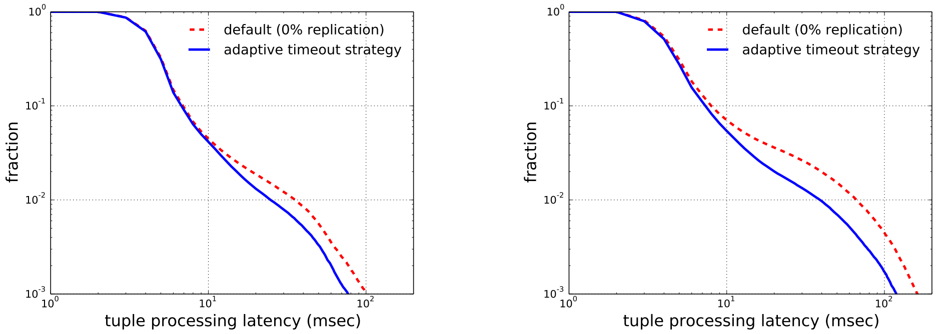
Experiment settings for the improved concurrency model are as follows: The spout and output bolt each have 5 tasks, evenly distributed among 5 worker processes. Other bolts each have 4 tasks and *all tasks of a given operator* are assigned to the same worker process. Each spout task emits 530 tuples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

Experiment settings for the latency-based load balance are as follows: 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 per second.

The latency improvement achieved by our three techniques are shown in Figure 5.9, Figure 5.10 and Figure 5.11. The same result is summarized in Table 5.5.

Table 5.5: Latency Improvement Achieved by our three techniques with two Yahoo! Benckmark Topologies

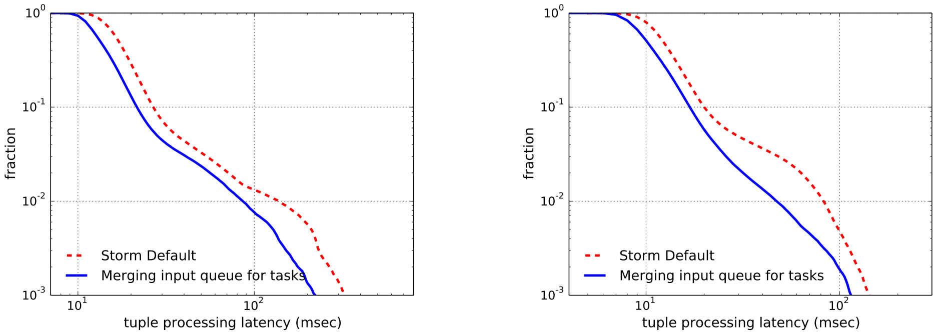
Name	90^{th} latency	99^{th} latency	99.9^{th} latency
Adaptive Timeout Strategy	——	28%-40%	24%-26%
Improved Concurrency Model	16%-19%	36%-42%	20%-32%
Latency-based Load Balance	22%-48%	50%-57%	21%-50%



(a) Page Load Topology

(b) Processing Topology

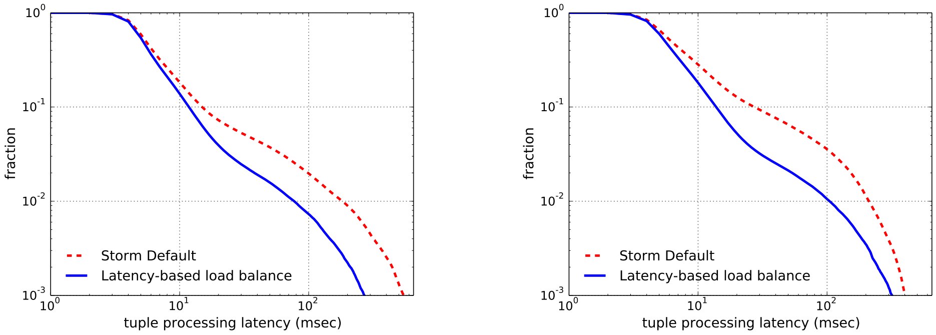
Figure 5.9: Latency Improvement Achieved by Adaptive Timeout Strategy with Yahoo! Benchmark Topology



(a) Page Load Topology

(b) Processing Topology

Figure 5.10: Latency Improvement Achieved by Improved Concurreny Model for Worker Process with Yahoo! Benchmark Topology



(a) Page Load Topology

(b) Processing Topology

Figure 5.11: Latency Improvement Achieved by Latency-based Load Balance with Yahoo! Benchmark Topology

CHAPTER 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 [29, 30], replicating the SYN packets or even the entire flow [28] to avoid uncertainty.

In batch processing systems such as MapReduce [32], Dryad [45] and Apache Spark [46, 47, 48], 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 [21]. 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 via inter-machine communication. Some systems [22, 23] try to predict which requests are long-running requests and then selectively parallelize them.

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 [17]. Raising voltage supply [49, 50, 51] is another option to rein the applications' tail latency, at the cost of increased power consumption.

In shared networked storage, reactive feedback-control based storage scheduling [52] as well as a combination of per-workload priority differentiation and rate limiting [53] has been shown effective.

From the perspective of cloud service provider, a judicious VM scheduling algorithm, such as separating computing intensive and latency sensitive VMs, is beneficial to low tail latency [19, 54].

CHAPTER 7

CONCLUSION

In this thesis, we presented three novel techniques to shorten the tail latency in stream processing systems. The adaptive timeout strategey can reduce 99^{th} and 99.9^{th} latency in a variety of situations at the cost of slightly increased workload. The improved concurrency for worker process and the latency-based load balance improve 90^{th}, 99^{th} and 99.9^{th} latency under high queue utilization and heterogeneity respectively. 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.

Future Work: One direction for future work is enable the stream processing systems to apply these techniques *adaptively*. In other words, the stream processing systems can decide *when* to turn on *which* technique given detailed profiles of workload and resource during runtime. Another direction is to adapt, within a technique, to a set of *optimal* parameters such as period, aging parameter and etc.

REFERENCES

[1] "Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse," http://www.infoq.com/articles/stream-processing-hadoop, last visited: 04/2016.

[2] "Apache Storm," http://storm.apache.org/, last visited: 04/2016.

[3] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[4] "Apache Flink," https://flink.apache.org/, last visited: 04/2016.

[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. USENIX, 2012, pp. 10–10.

[6] "Samza," http://samza.apache.org/, last visited: 04/2016.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. IEEE, 2010, pp. 170–177.

[8] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. ACM, 2013, pp. 207–218.

[9] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-Aware Online Scheduling in Storm," in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 535–544.

[10] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer, "Soda: An optimizing scheduler for large-scale stream-based distributed computer systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware.* Springer-Verlag New York, Inc., 2008, pp. 306–325.

[11] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference.* ACM, 2015, pp. 149–161.

[12] P. Bellavista, A. Corradi, A. Reale, and N. Ticca, "Priority-based resource scheduling in distributed stream processing systems for big data applications," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC).* IEEE, 2014, pp. 363–370.

[13] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams," in *Proceedings of 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS).*

[14] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proceedings of 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS).* IEEE, 2015, pp. 399–410.

[15] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1447–1463, 2014.

[16] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu, "Elastic scaling of data parallel operators in stream processing," in *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009.* IEEE, 2009, pp. 1–12.

[17] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and Application-level Sources of Tail Latency," in *Proceedings of the ACM Symposium on Cloud Computing.* ACM, 2014, pp. 1–14.

[18] "Google: Taming the Long Latency Tail," http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html, last visited: 04/2016.

[19] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation.* USENIX, 2013, pp. 329–342.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing.* ACM, 2012, p. 9.

[21] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, pp. 74–80, 2013.

[22] M. E. Haque, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley et al., "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2015, pp. 161–175.

[23] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *Proceedings of the 37th International ACM SIGIR Conference on Research & development in Information Retrieval.* ACM, 2014, pp. 253–262.

[24] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *ACM SIGCOMM Computer Communication Review.* ACM, 2013, pp. 159–170.

[25] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation.* USENIX, 2015, pp. 513–527.

[26] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation.* USENIX, 2015, pp. 543–557.

[27] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation.* USENIX, 2012, pp. 19–19.

[28] S. Liu, W. Bai, H. Xu, K. Chen, and Z. Cai, "Repflow on node. js: Cutting tail latency in data center networks at the applications layer." *Computing Research Repository*, 2014.

[29] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, “Low latency via redundancy,” in *Proceedings of the ninth ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2013, pp. 283–294.

[30] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, “More is less: reducing latency via redundancy,” in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 13–18.

[31] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data Center TCP (DCTCP),” in *Proceedings of the ACM SIGCOMM 2010 Conference*. ACM, 2010, pp. 63–74.

[32] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, pp. 107–113, 2008.

[33] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, “Reining in the Outliers in Map-reduce Clusters Using Mantri,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10. USENIX, 2010, pp. 265–278.

[34] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM (JACM)*, pp. 720–748, 1999.

[35] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, “Chaos: Scale-out graph processing from secondary storage,” in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 410–424.

[36] “M/M/c queue,” https://en.wikipedia.org/wiki/M/M/c_queue/, last visited: 04/2016.

[37] M. Mitzenmacher, “The power of two choices in randomized load balancing,” *Parallel and Distributed Systems, IEEE Transactions on*, pp. 1094–1104, 2001.

[38] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich, “Queueing system with selection of the shortest of two queues: An asymptotic approach,” *Problemy Peredachi Informatsii*, pp. 20–34, 1996.

[39] M. Harchol-Balter, M. Crovella, and C. D. Murta, “On choosing a task assignment policy for a distributed server system,” in *Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*. Springer-Verlag, 1998, pp. 231–242.

[40] “Guaranteeing Message Processing of Storm,” http://storm.apache.org/documentation/Guaranteeing-message-processing.html, last visited: 04/2016.

[41] “Understanding the Parallelism of a Storm Topology,” http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html, last visited: 04/2016.

[42] “Google Compute Engine,” https://cloud.google.com/compute/, last visited: 04/2016.

[43] “Zookeeper,” https://zookeeper.apache.org/, last visited: 04/2016.

[44] L. Xu, B. Peng, and I. Gupta, “Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand,” in *IEEE International Conference on Cloud Engineering (IC2E)*, 2016.

[45] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, 2007, pp. 59–72.

[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2012, pp. 2–2.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. USENIX, 2010, pp. 10–10.

[48] “Apache Spark,” http://spark.apache.org/, last visited: 04/2016.

[49] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 271–282.

[50] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, “Tradeoffs between power management and tail latency in warehouse-scale applications,” in *Proceedings of 2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 31–40.

[51] B. Vamanan, H. B. Sohail, J. Hasan, and T. Vijaykumar, "Timetrader: exploiting latency tail to save datacenter energy for online search," in *Proceedings of the 48th International Symposium on Microarchitecture.* ACM, 2015, pp. 585–597.

[52] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling high-level slos on shared storage systems," in *Proceedings of the Third ACM Symposium on Cloud Computing.* ACM, 2012, pp. 14:1–14:14.

[53] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail latency QoS for shared networked storage," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. ACM, 2014, pp. 1–14.

[54] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, "Small is better: Avoiding latency traps in virtualized data centers," in *Proceedings of the 4th Annual Symposium on Cloud Computing.* ACM, 2013, p. 7.