Massively-Parallel Stream Processing under QoS Constraints with Nephele

Björn Lohrmann
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
bjoern.lohrmann@tu-berlin.de

Daniel Warneke
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
daniel.warneke@tu-berlin.de

Odej Kao
Technische Universität Berlin
Einsteinufer 17
10587 Berlin
Germany
odej.kao@tu-berlin.de

ABSTRACT

Today, a growing number of commodity devices, like mobile phones or smart meters, is equipped with rich sensors and capable of producing continuous data streams. The sheer amount of these devices and the resulting overall data volumes of the streams raise new challenges with respect to the scalability of existing stream processing systems.

At the same time, massively-parallel data processing systems like MapReduce have proven that they scale to large numbers of nodes and efficiently organize data transfers between them. Many of these systems also provide streaming capabilities. However, unlike traditional stream processors, these systems have disregarded QoS requirements of prospective stream processing applications so far.

In this paper we address this gap. First, we analyze common design principles of today's parallel data processing frameworks and identify those principles that provide degrees of freedom in trading off the QoS goals latency and throughput. Second, we propose a scheme which allows these frameworks to detect violations of user-defined latency constraints and optimize the job execution without manual interaction in order to meet these constraints while keeping the throughput as high as possible. As a proof of concept, we implemented our approach for our parallel data processing framework Nephele and evaluated its effectiveness through a comparison with Hadoop Online.

For a multimedia streaming application we can demonstrate an improved processing latency by factor of at least 15 while preserving high data throughput when needed.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Parallel databases; C.2.4 [**Distributed Systems**]: Distributed applications

HPDC'12, June 18–22, 2012, Delft, The Netherlands.

General Terms

Experimentation, Measurement, Performance

Keywords

Large-Scale Data Processing, Stream Processing, Quality of Service, MapReduce

1. INTRODUCTION

In the course of the last decade, science and the IT industry have witnessed an unparalleled increase of data. While the traditional way of creating data on the Internet allowed companies to lazily crawl websites or related data sources, store the data on massive arrays of hard disks, and process it in a batch-style fashion, recent hardware developments for mobile and embedded devices together with ubiquitous networking have also drawn attention to *streamed data*.

Streamed data can originate from various different sources. Every modern smartphone is equipped with a variety of sensors, capable of producing rich media streams of video, audio, and possibly GPS data. Moreover, the number of deployed sensor networks is steadily increasing, enabling innovations in several fields of life, for example energy consumption, traffic regulation, or e-health. However, a crucial prerequisite to leverage those innovations is the ability to process and analyze a large number of individual data streams in a near-real-time manner. As motivation, we would like to illustrate two emerging scenarios:

- **Live Media Streaming:** Today, virtually all smart phones can produce live video streams. Several websites like Livestream[1] or Ustream[2] have already responded to that development, offering their users to produce and broadcast live media content to a large audience in a way that has been reserved to major television networks before. Recently, we have seen first steps towards this "citizen journalism" during the political incidents in the Middle East or the "Occupy Wall Street" movement. However, at the moment, the capabilities of those live broadcasting services are limited to media transcoding and simple picture overlays. Although the content of two different streams may overlap to a great extent (for example because the people filming the scene are standing close to each other),

[1] `http://www.livestream.com/`
[2] `http://www.ustream.tv/`

they are currently processed completely independent of each other. In contrast to that, future services might also offer to automatically *aggregate* and *relate* streams from different sources, thereby creating a more complete picture and eventually better coverage for the viewers.

- **Energy informatics:** Smart meters are currently being deployed in growing numbers at consumer homes by power utilities. Smart meters are networked devices that monitor a household's power consumption and report it back to the power utility. On the utility's side, having such near-real-time data about power consumption is a key aspect of managing fluctuations in the power grid's load. Such fluctuations are introduced not only by consumers but also by the increasing, long-term integration of renewable energy sources. Data analytics applications that are hooked into the live meter data stream can be used for many operational aspects such as monitoring the grid infrastructure for equipment limits, initiating autonomous control actions to deal with component failures, voltage sags/spikes, and forecasting power usage. Especially in the case of autonomous control actions, the freshness of the data that is being acted upon is of paramount importance.

Opportunities to harvest the new data sources in the various domains are plentiful. However, the sheer amount of incoming data that must be processed online also raises scalability concerns with regard to existing solutions. As opposed to systems working with batch-style workloads, stream processing systems must often meet particular Quality of Service (QoS) goals, otherwise the quality of the processing output degrades or the output becomes worthless at all. Existing stream processors [1, 2] have put much emphasis on meeting provided QoS goals of applications, though often at the expense of scalability or a loss of generality [17].

In terms of scalability and programming generality, the predominant workhorses for data-intensive workloads at the moment are massively-parallel data processing frameworks like MapReduce [12] or Dryad [14]. By design, these systems scale to large numbers of compute nodes and are capable of efficiently transferring large amounts of data between them. Many of the newer systems [8, 11, 14, 16, 18] also allow to assemble complex parallel data flow graphs and to construct pipelines between the individual parts of the flow. Therefore, these systems generally are also suitable for streaming applications. However, so far they have concentrated on few streaming application, like online aggregation or "early out" computations [11], and have not considered QoS goals.

This paper attempts to bridge that gap. We have analyzed a series of open-source frameworks for parallel data processing and highlight common design principles they share to achieve scalability and high data throughput. We show how some aspects of these design principles can be used to trade off the QoS goals latency and throughput in a fine-grained per-task manner and propose a scheme to automatically do so during the job execution based on user-defined latency constraints. Starting from the assumption that high data throughput is desired, our scheme monitors potential latency constraint violations at runtime and can then gradually applies two techniques, *adaptive output buffer sizing* and *dynamic task chaining*, to met the constraints while maintaining high throughput as far as possible. As a proof of concept, we implemented the scheme for our data processing framework Nephele and evaluated their effectiveness through a comparison with Hadoop Online.

The rest of this paper is structured as follows: In Section 2 we examine the common design principles of today's massively-parallel data processing frameworks and discuss the implications of meeting the aforementioned QoS constraints. Section 3 presents our scheme to dynamically adapt to the user-defined latency constraints, whereas Section 4 contains an experimental evaluation. Section 5 provides a brief overview of current stream and parallel data processors. Finally, we conclude our paper in Section 6.

2. MASSIVELY-PARALLEL DATA PROCESSING AND STREAMED DATA

In recent years, a variety of frameworks for massively-parallel data analysis has emerged [8, 11, 12, 14, 16, 18]. Many of them are open-source software. Having analyzed their internal structure, we found they often follow similar design principles to achieve scalability and high throughput.

This section highlights those principals and discuss their implications on stream processing under QoS constraints.

2.1 Design Principles of Parallel Data Processing Frameworks

Frameworks for parallel data processing typically follow a master-worker pattern. The master node receives jobs from the user, splits them into sets of individual tasks, and schedules those tasks to run on the available worker nodes.

The structure of those jobs can usually be described by a graph with vertices representing the job's individual tasks and the edges denoting communication channels between them. For example, from a high-level perspective, the graph representation of a typical MapReduce job would consist of a set of Map vertices connected via edges to a set of Reduce vertices. Some frameworks have generalized the MapReduce model to arbitrary directed acyclic graphs (DAGs) [8, 14, 18], some even allow graph structures containing loops [16].

However, independent of the concrete graph model used to describe jobs for the respective framework, the way both the vertices and edges translate to system resources at runtime is surprisingly similar among all of these systems.

Each task vertex of the overall job typically translates to either a separate process or a separate thread at runtime. Considering the large number of CPUs (or CPU cores) these frameworks must scale up to, this is a reasonable design decision. By assigning each task to a different thread/process, those tasks can be executed independently and utilize a separate CPU core. Moreover, it gives the underlying operating system various degrees of freedom in scheduling the tasks among the individual CPU cores. For example, if a task cannot fully utilize its assigned CPU resources or is waiting for an I/O operation to complete, the operating system can assign the idle CPU time to a different thread/process.

The communication model of massively-parallel data processing systems typically follows a producer-consumer pattern. Tasks can produce a sequence of *data items* which are then passed to and consumed by their successor tasks according to the edges of the job's graph representation. The way the data items are physically transported from one task to the other depends on the concrete framework. In the most lightweight case, two tasks are represented as two different

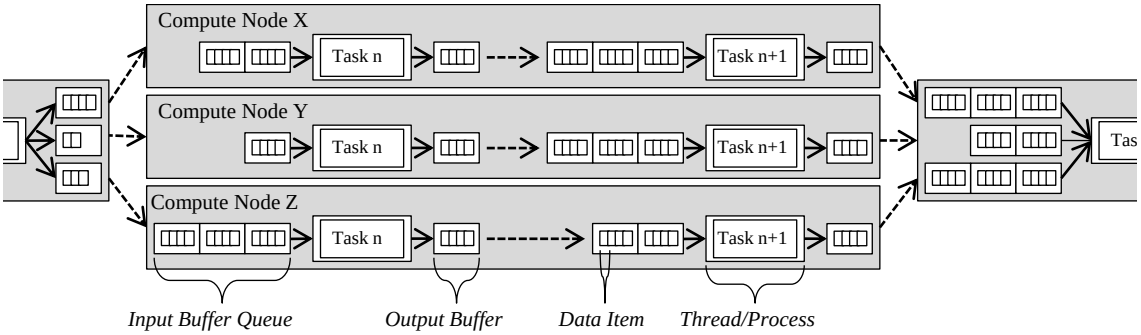


Figure 1: Typical processing pattern of frameworks for massively-parallel data analysis

threads running inside the same operating system process and can use shared memory to exchange data. If tasks are mapped to different processes, possibly running on different worker nodes, the data items are typically exchanged through files or a network connection.

However, since all of these frameworks have been designed for data-intensive workloads and hence strive for high data throughput, they attempt to minimize the transfer overhead per data item. As a result, these frameworks try to avoid shipping individual data items from one task to the other. As illustrated in Figure 1, the data items produced by a task are typically collected in a larger *output buffer*. Once its capacity limit has been reached, the entire buffer is shipped to the receiving task and in many cases placed in its *input buffer queue*, waiting to be consumed.

2.2 Implications for QoS-Constrained Streaming Applications

Having highlighted some basic design principles of today's parallel data processing frameworks, we now discuss which aspects of those principles provide degrees of freedom in trading off the different QoS goals latency and throughput.

2.2.1 The Role of the Output Buffer

As explained previously, most frameworks for parallel data processing introduce distinct output buffers to minimize the transfer overhead per data item and improve the data item throughput, i.e. the average number of items that can be shipped from one task to the other in a given time interval.

For the vast majority of data processing frameworks we have analyzed in the scope of our research, the output buffer size could be set on a system level, i.e. all jobs of the respective framework instance were forced to use the same output buffer sizes. Some frameworks also allowed to set the output buffer size per job, for example Apache Hadoop[3]. Typical sizes of these output buffers range from several MB to 4 or 8 KB, depending on the focus of the framework.

While output buffers play an important role in achieving high data throughput, they also make it hard to optimize jobs for current parallel data processors towards the QoS goal latency. Since an output buffer is typically not shipped until it has reached its capacity limit, the latency an individual data item experiences depends on the system load.

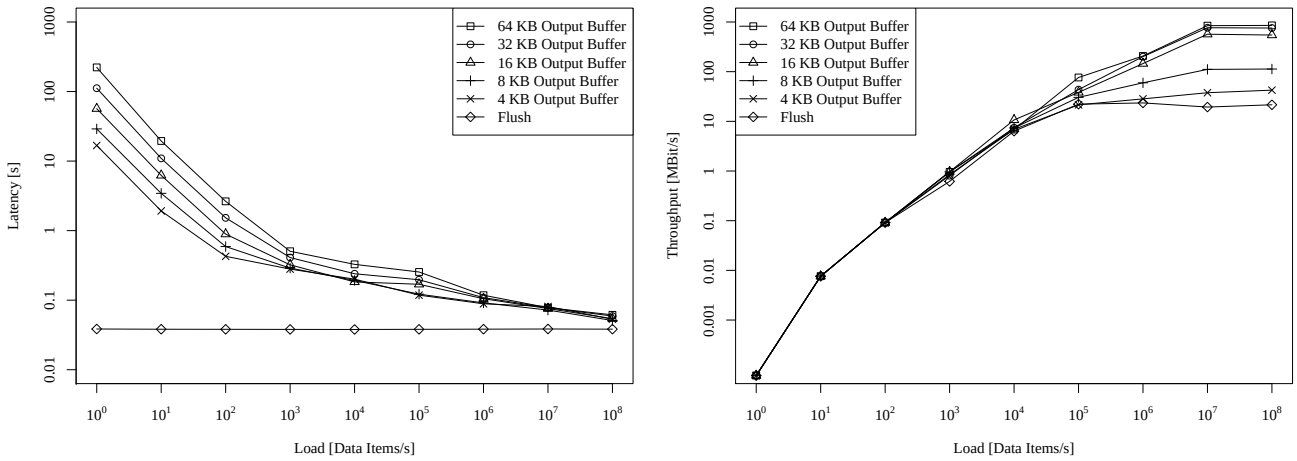
In order to illustrate this effect, we created a small sample job consisting of two tasks, a sender task and a receiver task. The sender created data items of 128 bytes length at a fixed rate n and wrote them to an output buffer of a fixed size. Once an output buffer had reached its capacity limit, it was sent to the receiver through a TCP connection. We ran the job several times. Between each run, we varied the output buffer size.

The results of this initial experiment are depicted in Figure 2. As illustrated in Figure 2(a), the average latency from the creation of a data item at the sender until its arrival at the receiver depends heavily on the creation rate and the size of the output buffer. With only one created data item per second and an output buffer size of 64 KB, it takes more than 222 seconds on an average before an item arrives at the receiver. At low data creation rates, the size of the output buffer has a significant effect on the latency. The more the data creation rate increases, the more the latency converges towards a lower bound. At a rate of 10^8 created items per second, we measured an average data item latency of approximately 50 milliseconds (ms), independent of the output buffer size.

As a baseline experiment, we also executed separate runs of the sample job which involved flushing incomplete output buffers. Flushing forced the system to transfer the output buffer to the receiver after each written data item. As a result, the average data item latency was uniformly 38 ms, independent of the data creation rate.

Figure 2(b) shows the effects of the different data creation rates and output buffer sizes on the throughput of the sample job. While the QoS objective latency suggests using small output buffers or even flushing incomplete buffers, these actions show a detrimental effect when high data throughput is desired. As depicted in Figure 2(b), the data item throughput that could be achieved grew with the size of the output buffer. With relatively big output buffers of 64 or 32 KB in size, we were able to fully saturate the 1 GBit/s network link between the sender and the receiver, given a sufficiently high data creation rate. However, the small output buffers failed to achieve a reasonable data item throughput. In the most extreme case, i.e. flushing the output buffer after every written data item, we were unable to attain a data item throughput of more than 10 MBit/s. The reason for this is the disproportionally high transfer overhead per data item (output buffer meta data, memory management, thread synchronization) that massively-parallel data processing frameworks in general are not designed for. Similar behavior is known from the TCP networking layer, where the Nagle algorithm can be deactivated (TCP_NODELAY option) to improve connection latency.

[3] `http://hadoop.apache.org/`



(a) Average data item latency

(b) Average data item throughput

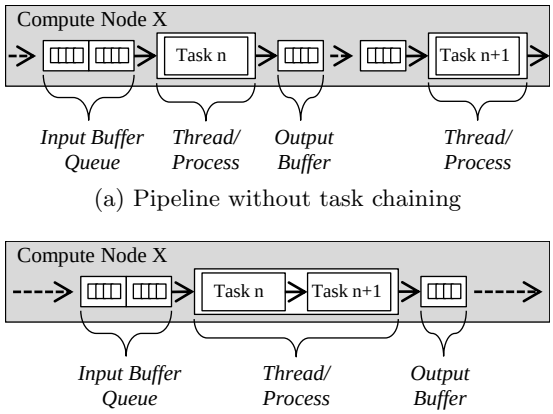
Figure 2: The effect of different output buffer sizes on data item latency and throughput

In sum, the sample job highlights an interesting trade-off that exists in current data processing frameworks with respect to the output buffer size. While jobs with low latency demands benefit from small output buffers, the classic data-intensive workloads still require relatively large output buffers in order to achieve high data throughput. This trade-off puts the user in charge of configuring a reasonable output buffer size for his job and assumes that (a) the used processing framework allows him to specify the output buffer size on a per-job basis, (b) he can estimate the expected load his job will experience, and (c) the expected load does not change over time. In practice, however, at least one of those three assumptions often does not hold. One might also argue that there is no single reasonable output buffer size for an entire job as the job consists of different tasks that produce varying data item sizes at varying rates, so that any chosen fixed output buffer size can only result in acceptable latencies for a fraction of the tasks but not for all of them.

2.2.2 The Role of the Thread/Process Model

Current frameworks for parallel data processing typically map different tasks to different operating system processes or at least different threads. While this facilitates natural scalability and load balancing between different CPUs or CPU cores, it also raises the communication overhead between tasks. In the most lightweight case, where different tasks are mapped to different threads within the same process and communication is performed via shared memory, the communication overhead typically only consists of thread synchronization, scheduling, and managing cache consistency issues. However, when the communicating tasks are mapped to different processes or even worker nodes, passing data items between them additionally involves serialization/deserialization and, depending on the way the data is physically exchanged, writing the serialized data to the network/file system and reading it back again.

Depending on the complexity of the tasks, the communication overhead can account for a significant fraction of the overall processing time. If the tasks themselves are lightweight, but the data items are rather large and complex to serialize/deserialize (as in case of a filter operation on a nested XML structure [4]), the overhead can limit the throughput and impose a considerable processing latency.



(a) Pipeline without task chaining

(b) Pipeline with task chaining

Figure 3: Different execution models with and without task chaining

As illustrated in Figure 3, a common approach to address this form of communication overhead is to chain lightweight tasks together and execute them in a single thread/process. The most popular example in the area of parallel data processing is probably the chained map functions from Apache Hadoop. However, a similar idea was also described earlier as rewriting a program to its "normal form" by Aldinucci and Danelutto [3] in the context of stream parallel skeletons.

Before starting a Hadoop job, a user can specify a series of map functions to be chained. Hadoop will then execute these functions in a single process. Chaining tasks often also eliminates the need for separate output buffers. For example, in case of Hadoop's chained map functions, the user code of the next map function in the processing chain can be directly invoked on the previous map function's output. Depending on the semantics of the concatenated tasks, chaining may also render the serialization/deserialization between tasks superfluous. If the chained tasks are stateless (as typically

expected from map functions in Hadoop), it is safe to pass the data items from one task to the other by reference.

With regard to stream processing, chaining tasks offers an interesting approach to reduce processing latency and increase throughput at the same time. However, similar to the output buffer size, there might also be an important trade-off, especially when the job's workload is unknown in advance or is likely to change over time.

In its current form, task chaining is performed at compile time, so once the job is running, all chained tasks are bound to a single execution thread. In situations with low load, this might be beneficial since communication overhead is decreased and potential throughput and latency goals can be met more easily. However, when the load increases in the course of the job processing, the static chaining prevents the underlying operating system from distributing the tasks across several CPU cores. As a result, task chaining can also be disadvantageous if (a) the complexity of the chained tasks is unknown in advance or (b) the workload the streaming job has to handle is unknown or changes over time.

3. AUTOMATIC QOS-OPTIMIZATION FOR STREAMING APPLICATIONS

Currently, it is the user of a particular framework who must estimate the effects of the configured buffer size and thread/process model on a a job's latency and throughput characteristics in a cumbersome and inaccurate manner.

In the following, we propose an extension to parallel data processing frameworks which spares the user this hassle. Starting from the assumption that high throughput continues to be the predominant QoS goal in parallel data processing, our extension lets users add latency constraints to their job specifications. Based on these constraints, it continuously monitors the job execution and detects violations of the provided latency constraints *at runtime*. Our extension can then selectively trade high data throughput for a lower processing latency using two distinct strategies, *adaptive output buffer sizing* and *dynamic task chaining*.

As a proof of concept, we implemented this extension as part of our massively-parallel data processing framework Nephele [18] which runs data analysis jobs based on DAGs. However, based on the common principles identified in the previous section, we argue that similar strategies are applicable to other frameworks as well.

3.1 Specifying Latency Constraints

For the remainder of the paper, we will assume a DAG $G = (V_G, E_G)$ as the underlying structure of a job. At runtime each vertex $v \in V_G$ is a a task containing user code. The directed edge $e = (v_1, v_2) \in E_G$ is a channel along which the task v_1 can send data items of arbitrary size to task v_2.

In order to specify latency constraints, a user must be aware how much latency his application can tolerate in order to still be useful. With his knowledge from the application domain a user should then identify latency critical sequences of tasks and channels within the DAG for which he can express required upper latency bounds in the form of *constraints*. These constraints are part of the job description and provide information to the framework about where optimizations are necessary.

In the following, we will formally distinguish between task, channel, and sequence latency, based on which latency constraints can then be expressed.

3.1.1 Task Latency

Given a task v_i, an incoming channel $e_{in} = (v_x, v_i)$ and an outgoing channel $e_{out} = (v_i, v_y)$, we shall define the *task latency* $tl(d, v_i, v_x \rightarrow v_y)$ as the time difference between a data item d entering the user code of v_i via the channel e_{in} and the next data item exiting the user code via e_{out}.

This definition has several implications. First, task latency is undefined on source and sink tasks as these task types lack incoming and, respectively, outgoing channels. Task latencies can be infinite if the task never emits for certain in/out channel combinations. Moreover, task latency can vary significantly between subsequent items, for example, if the task reads two items but emits only one item after it has read the last one of the two. In this case the first item will have experienced a higher task latency than the second one.

3.1.2 Channel Latency

Given two tasks $v_i, v_j \in V$ connected via channel $e = (v_i, v_j) \in E_G$, we define the *channel latency* $cl(d, e)$ as the time difference between the data item d exiting the user code of v_i and entering the user code of v_j. The channel latency may also vary significantly between data items on the same channel due to differences in item size, output buffer utilization, network congestion, and the length of the input queues that need to be transited on the way to the receiving task.

3.1.3 Sequence Latency

Sequences are series of of connected tasks and channels and thus should be used to identify the parts of the DAG for which the application has latency requirements. Let us assume a sequence $S = (s_1, \ldots, s_n)$, $n \geq 1$ of connected tasks and channels. The first element of the sequence is allowed to be either a task or a channel. For example, if s_2 is a task, then s_1 needs to be an incoming and s_3 an outgoing channel of the task. If a data item d enters the sequence S, we can define the *sequence latency* $sl(d, S)$ that the item d experiences as $sl^*(d, S, 1)$ where

$$sl^*(d, S, i) = \begin{cases} l(d, s_i) + sl^*(s_i(d), S, i+1) & \text{if } i < n \\ l(d, s_i) & \text{if } i = n \end{cases}$$

If s_i is a task, then $l(d, s_i)$ is equal to the task latency $tl(d, s_i, v_x \rightarrow v_y)$ and $s_i(d)$ is the next data item produced by s_i to be shipped via the channel (s_i, v_y). If s_i is a channel, then $l(d, s_i)$ is the channel latency $cl(d, s_i)$ and $s_i(d) = d$.

3.1.4 Latency Constraints

When the user has identified latency critical sequences within the DAG, he can then express the maximum tolerable latency on these sequences as a set of latency constraints $C = \{c_1, \ldots, c_n\}$ to be attached to the job description. Each constraint $c_i = (S_i, l_{S_i}, t)$ defines a desired upper latency limit l_{S_i} for the arithmetic mean of the sequence latency $sl(d, S_i)$ over all the data items $d \in D_t$ that enter the sequence S_i during any time span of t time units:

$$\frac{\sum_{d \in D_t} sl(d, S_i)}{|D_t|} \leq l_{S_i} \quad (1)$$

Note that such a constraint does not specify a hard upper latency bound for each single data item but only a "statistical" upper bound over the items running through the workflow during the given time span. While hard upper bounds may be desirable, we doubt that meaningful hard upper bounds can be achieved in most real-world setups of massively-parallel data processing frameworks.

3.2 Measuring Workflow Latency

In order to make informed decisions where to apply optimizations to a running workflow we designed and implemented means of sampling and estimating the latency of a sequence. The master node that has global knowledge about the defined latency constraints will instruct the worker nodes about where they have to perform latency measurements. For the elements (task or channel) of each constrained sequence, latencies will be measured on the respective worker node once during a configured time interval, the measurement interval. This scheme can quickly produce high numbers of measurements with rising numbers of tasks and channels. For this reason we locally preaggregate measurement data on the worker nodes and ship one message once every measurement interval from the workers to the master. Each message contains the following data:

1. An estimation of the average *channel latency* for each locally incoming channel (i.e. it is an incoming channel on the worker node) of the constrained sequences. The average latency of a channel is estimated using *tagged* data items. A tag is a small piece of data that contains a creation timestamp and a channel identifier and it is added when a data item exits the user code of the channel's sender task and is evaluated just before the data item enters the user code of the channel's receiver task. The receiving worker node will then add the measured latency to its aggregated measurement data. The tagging frequency is chosen in such a way that we have one tagged data item during each measurement interval if there are any data flowing through the channel. If the sending and receiving tasks are executed on different worker nodes, clock synchronization is required.

2. The average *output buffer lifetime* for each channel of the constrained sequences, which is the average time it took for output buffers to be filled. If no output buffer was filled on the channel during the measurement interval, this is indicated as such in the message.

3. An estimation of the average *task latency* for each task of the constrained sequences. Task latencies are measured in an analogous way to channels, but here we do not require tags. Once every measurement interval, a task will note the difference in system time between a data item entering the user code and the next data item leaving it on the channels specified in the constrained sequences. Again, the measurement frequency is chosen in a way that we have one latency measurement during each measurement interval if there are any data flowing through the channel.

Let us assume a constrained sequence $S = (e_1, v_1, e_2)$. Tags will be added to the data items entering channel e_1 once every measurement interval. Just before a tagged data item enters the user code of v_1, the tag is removed from the data item and the difference between the tag's timestamp and the current system time is added to the locally aggregated measurement data. Let us assume a latency measurement is required for the task v_1 as well. In this case, just before handing the data item to the task, the current system time is stored in the task environment. The next time the task outputs a data item to be sent to e_2 the difference between the current system time and the stored timestamp is again added to the locally aggregated measurement data. Before handing the produced data item to the channel e_2, the worker node may choose to tag it, depending on whether we still need a latency measurement for this channel. Once every measurement interval the worker nodes flush their aggregated measurement data to the master node.

The master node stores the measurement data it receives from the worker nodes. For each constraint $(S_i, l_{S_i}, t) \in C$, it will keep all latency measurement data concerning the elements of S_i that are fresher than t time units and discard all older measurement data. Then, for each element of S_i, it will compute a running average over the measurement values and add the results up to an estimation of the left side of Equation 1. The accuracy of this estimation depends mainly on the chosen measurement interval.

The aforementioned *output buffer lifetime* measurements are subjected to the same running average procedure. To the running average of the *output buffer lifetime* of channel e over the past t time units we shall refer as $oblt(e, t)$. Note that the time individual data items spend in output buffers is already contained in the channel latencies, hence we do not need the output buffer lifetime to estimate sequence latencies. It does however play the role of an indicator, when trying to locate channels where the output buffer sizes can be optimized (see Section 3.3).

The measurement overhead is quite low as only one message from each of the workers to the master is required during a measurement interval. Even for large numbers of nodes, the collected data can be easily held in main memory by the master node. If necessary, the number of messages can be reduced by increasing the measurement interval.

3.3 Reacting to Latency Constraint Violations

Based on the measurement data as described in Section 3.2, the master node can identify those sequences of the DAG that violate their constraint and initiate countermeasures to improve latency. It will apply countermeasures until the constraint has been met or the necessary preconditions for applying countermeasures are not met anymore. In this case it will report the failed optimization attempt to the user who then has to either change the job or revise the constraints.

Given a DAG $G = (V_G, E_G)$, a sequence $S = (s_1, \ldots, s_n)$, and a violated latency constraint (S, l_S, t), the master node attempts to eliminate the effect of improperly sized output buffers by adjusting the buffer sizes for each channel in S individually and apply dynamic task chaining to reduce latencies further. Buffer size adjustment is an iterative process which may increase or decrease buffer sizes at multiple channels, depending on the measured latencies. Note that after each run of the buffer adjustment procedure the master node waits until all latency measurement values based on the old buffer sizes have been flushed out. The conditions and procedures for changing buffer sizes and dynamic task chaining are outlined in the following sections.

3.3.1 Adaptive Output Buffer Sizing

For each channel e in S the master node compares the average *output buffer latency* $obl(e,t)$ that data items on this channel experience to the running average of the channel latency. The average output buffer latency of a data item is estimated as $obl(e,t) = \frac{oblt(e,t)}{2}$, where $oblt(e,t)$ is the running average of the output buffer lifetime (see Section 3.2). If $obl(e,t)$ supersedes both a certain minimum threshold (for example 5 ms) and the task latency of the channel's source task, the master node sets the new output buffer size $obs^*(e)$ to

$$obs^*(e) = max(\epsilon, obs(e) \times r^{obl(e,t)}) \quad (2)$$

where $\epsilon > 0$ is an absolute lower limit on the buffer size, $obs(e)$ is the current output buffer size, and $0 < r < 1$. We chose $r = 0.98$ and $\epsilon = 200$ bytes as a default. This approach might reduce the output buffer size so much that most records do not fit inside the output buffer anymore, which is detrimental to both throughput and latency. Hence, if $obl(e) \approx 0$, we will increase the output buffer size to

$$obs^*(e) = min(\omega, s \times obs(e)) \quad (3)$$

where $\omega > 0$ is an upper bound for the buffer size and $s > 1$. For our prototype we chose $s = 1.1$.

3.3.2 Dynamic Task Chaining

Task chaining pulls certain tasks into the same thread, thus eliminating the need for queues and handing over data items between these tasks. In order to be able to chain a series of tasks $v_1, \ldots, v_n$ within the constrained sequence S they need to fulfill the following conditions:

- They all run as separate threads with the same process on the worker node, which excludes any already chained tasks.
- The sum of the CPU utilizations of the task threads is lower than the capacity of one CPU core or a fraction thereof, for example 90% of a core. How such profiling information can be obtained has been described in [7].
- They form a path through the DAG, i.e. each pair $v_i, v_{i+1} \in V_G$ is connected by a channel $e = (v_i, v_{i+1}) \in E_G$ in the DAG.
- None of the tasks has more than one incoming and more than one outgoing channel, with the exception of the first task v_1 which is allowed to have multiple incoming channels and the last task v_n which is allowed to have multiple outgoing channels.

The master node looks for the longest chainable series of tasks within the sequence. If it finds one, it instructs the worker node to chain the respective tasks. When chaining a series of tasks the worker node needs to take care of the input queues between them. There are two principal ways of doing this. The first one is to simply drop the existing input queues between these tasks. Whether this is acceptable or not depends on the nature of the workflow, for example in a video stream scenario it is usually acceptable to drop some frames. The second one is to halt the first task v_1 in the series and wait until the input queues between all of the subsequent tasks $v_2, \ldots, v_n$ in the chain have been drained. This will temporarily increase the latency due to a growing input queue of v_1 that needs to be reduced after the chain has been established.

3.4 Relation to Fault Tolerance

In large clusters of compute nodes, individual nodes are likely to fail [12]. Therefore, it is important to point out how our proposed techniques to trade off high throughput against low latency at runtime affect the fault tolerance capabilities of current parallel data processing frameworks.

As these data processors mostly execute arbitrary black-box user code, currently the predominant approach to guard against execution failures is referred to as log-based rollback-recovery in literature [13]. Besides sending the output buffers with the individual data items from the producing to the consuming task, the parallel processing frameworks additionally materialize these output buffers to a (distributed) file system. As a result, if a task or an entire worker node crashes, the data can be re-read from the file system and fed back into the re-started tasks. The fault tolerance in Nephele is also realized that way.

Our two proposed optimizations affect this type of fault tolerance mechanism in different ways: Our first approach, the adaptive output buffer sizing, is completely transparent to a possible data materialization because it does not change the framework's internal processing chain for output buffers but simply the size of these buffers. Therefore, if the parallel processing framework wrote output buffers to disk before the application of our optimization, it will continue to do so even if adaptive output buffer sizing is in operation.

For our second optimization, the dynamic task chaining, the situation is different. With dynamic task chaining activated, the data items passed from one task to the other no longer flow through the framework's internal processing chain. Instead, the task chaining deliberately bypasses this processing chain to avoid serialization/deserialization overhead and reduce latency. Possible materialization points may therefore be incomplete and useless for a recovery.

We addressed this problem by introducing an additional annotation to the Nephele job description which prevents our system from applying dynamic task chaining on particular parts of the DAG. This way our streaming extension might lose one option to respond to violations of a provided latency goal, however, we are able to guarantee that Nephele's fault tolerance capabilities remain fully intact.

4. EVALUATION

Having presented both the adaptive output buffer sizing and the dynamic task chaining for Nephele, we will now evaluate their impact based on an example job. To put the measured data into perspective, we also implemented the example job for another parallel data processing framework with streaming capabilities, namely Hadoop Online[4].

We chose Hadoop Online as a baseline for comparison for three reasons: First, Hadoop Online is open-source software and was thus available for evaluation. Second, among all large-scale data processing frameworks with streaming capabilities, we think Hadoop Online currently enjoys the most popularity in the scientific community, which also makes it an interesting subject for comparison. Finally, in their research paper, the authors describe the continuous query

[4] `http://code.google.com/p/hop/`

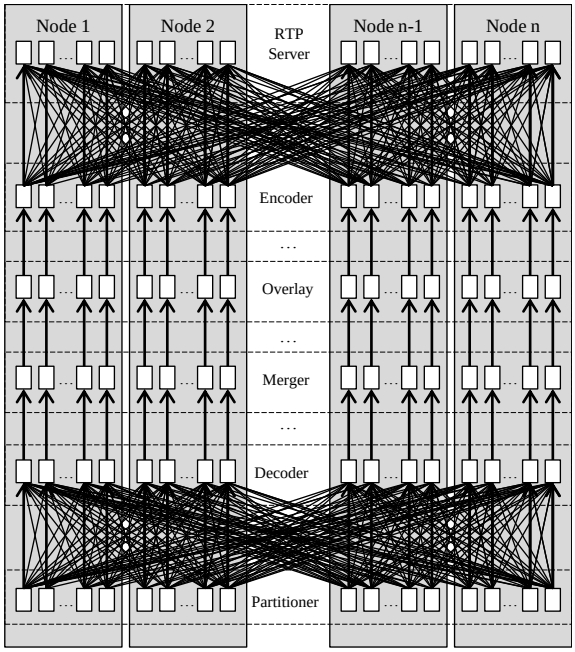


Figure 4: Structure of the Nephele job

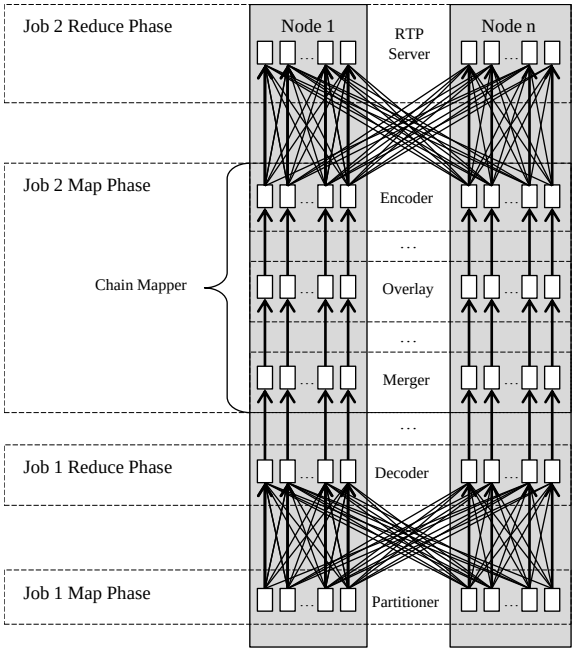


Figure 5: Structure of the Hadoop Online job

feature of their system to allow for near-real-time analysis of data streams [11]. However, they do not provide any numbers on the actually achievable processing latency. Our experiments therefore also shed light on this question.

4.1 Job Description

The job we use for the evaluation is motivated by the "citizen journalism" use case described in the introduction. We consider a web platform which offers its users to broadcast incoming video streams to a larger audience. However, instead of simple video transcoding which is done by existing video streaming platforms, our system additionally groups related video streams, merges them to a single stream, and augments the stream with additional information, such as Twitter feeds or other social network content. The idea is to provide the audience of the merged stream with a broader view of a situation by automatically aggregating related information from various sources.

In the following we will describe the structure of the job, first for Nephele and afterwards for Hadoop Online.

4.1.1 Structure of the Nephele Job

Figure 4 depicts the structure of the Nephele evaluation job. The job consists of six distinct types of tasks. Each type of task is executed with a degree of parallelism of m, spread evenly across n compute nodes.

The first tasks are of type *Partitioner*. Each *Partitioner* task acts as a TCP/IP server for incoming video feeds, receives H.264 encoded video streams, assigns them to a group of streams and forwards the video stream data to the *Decoder* task responsible for streams of the assigned group. In the context of this evaluation job, we group video streams by a simple attribute which we expect to be attached to the stream as meta data, such as GPS coordinates. More sophisticated approaches to detect video stream correlations are possible but beyond the scope of our evaluation.

The *Decoder* tasks are in charge of decompressing the encoded video packets into distinct frames which can then be manipulated later in the workflow. For the decoding process, we rely on the xuggle library[5].

Following the *Decoder*, the next type of tasks in the processing pipeline are the *Merger* tasks. *Merger* tasks consume frames from grouped video streams and merge the respective set of frames to a single output frame. In our implementation the merge step simply consists of tiling the individual input frames in the output frame.

After having merged the grouped input frames, the *Merger* tasks send their output frames to the next task type in the pipeline, the *Overlay* tasks. An *Overlay* task augments the merged frames with information from additional related sources. For the evaluation, we designed each *Overlay* task to draw a marquee of Twitter feeds inside the video stream, which are picked based on locations close to the GPS coordinates attached to the video stream.

The output frames of the *Overlay* tasks are encoded back into the H.264 format by a set of *Encoder* tasks and then passed on to tasks of type *RTP Server*. These tasks represent the sink of the streams in our workflow. Each task of this type passes the incoming video streams on to an RTP server which then offers the video to an interested audience.

4.1.2 Structure of the Hadoop Online Job

For Hadoop Online, the example job exhibits a similar structure as for Nephele, however, the six distinct tasks have been distributed among the map and reduce functions of two individual MapReduce jobs. During the experiments on Hadoop Online, we executed the exact same task code as for Nephele apart from some additional wrapper classes we had to write in order to achieve interface compatibility.

As illustrated in Figure 5 we inserted the initial *Partitioner* task into the map function of the first MapRe-

[5] `http://www.xuggle.com/`

duce job. Following the continuous query example from the Hadoop Online website, the task basically "hijacks" the map slot with an infinite loop and waits for incoming H.264 encoded video streams. Upon the reception of the stream packet, the packet is put out with a new key, such that all video streams within the same group will arrive at the same parallel instance of the reducer. The reducer function then accommodates the previously described *Decoder* task. As in the Nephele job, the *Decoder* task decompresses the encoded video packets into individual frames.

The second MapReduce job starts with the three tasks *Merger*, *Overlay*, and *Encoder* in the map phase. Following our experiences with the computational complexity of these tasks from our initial Nephele experiments, we decided to use a Hadoop chain mapper and execute all of these three tasks consecutively within a single map process. Finally, in the reduce phase of the second MapReduce job, we placed the task *RTP Server*. The *RTP Server* tasks again represented the sink of our data streams.

In comparison to the classic Hadoop, the evaluation job exploits two distinct features of the Hadoop Online prototype, i.e. the support for continuous queries and the ability to express dependencies between different MapReduce jobs. The continuous query feature allows to stream data from the mapper directly to the reducer. The reducer then runs a moving window over the received data. We set the window size to 100 ms during the experiments. For smaller window sizes, we experienced no significant effect on the latency.

4.2 Experimental Setup

We executed our evaluation job on a cluster of $n = 10$ commodity servers. Each server was equipped with two Intel Xeon E5430 2.66 GHz CPUs (four cores per CPU) and 32 GB RAM. The nodes were connected via regular Gigabit Ethernet links and ran Linux (kernel version 2.6.39). Each node ran a KVM virtual machine with eight cores. Inside the virtual machines we used Linux (kernel version 2.6.38) and Java 1.6.0.26 to run Nephele's worker component. Additionally, each virtual machine launched a Network Time Protocol (NTP) daemon to maintain clock synchronization among the workers. During the entire experiment, the measured clock skew was below 2 ms among the machines.

Each worker node ran eight tasks of type *Decoder*, *Merger*, *Overlay* and *RTP Server*, respectively. The number of incoming video streams was fixed for each experiment and they were evenly distributed over the *Partitioner* tasks. We always grouped and subsequently merged four streams into one aggregated video stream. Each video stream had a resolution of 320×240 pixels and was H.264 encoded. The initial output buffer size was 32 KB. Unless noted otherwise, all tasks had a degree of parallelism of $m = 80$.

Those experiments that were conducted on Nephele with latency constraints in place, specified one constraint $c = (S, l, t)$ for each possible sequence

$$S = (e_1, v_D, e_2, v_M, e_3, v_O, e_4, v_E, e_5) \quad (4)$$

where v_D, v_M, v_O, v_E are tasks of type *Decoder*, *Merger*, *Overlay* and *Encoder* respectively. All constraints specified the same upper latency bound $l = 300$ ms over the data items within the past $t = 5$ seconds. The measurement interval on the worker nodes was set to 1 second so that the running averages on the master node were computed over sets of five measurement values. Due to their regular nature,

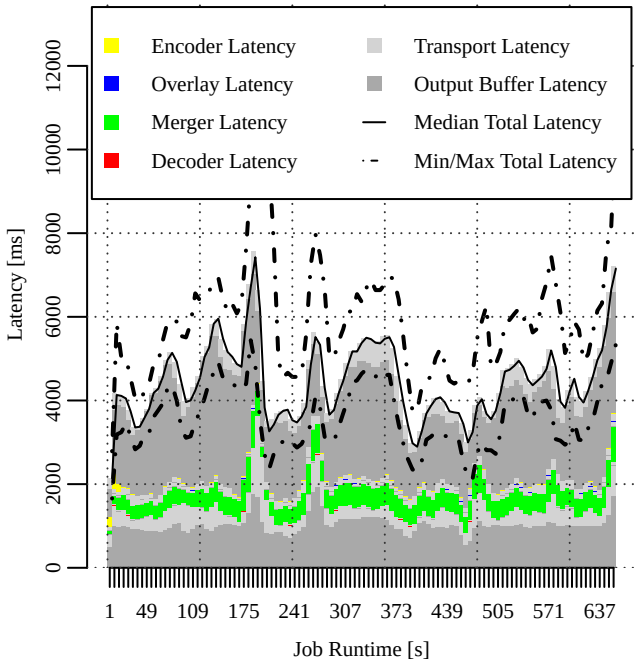


Figure 6: Latency w/o optimizations (320 video streams, degree of parallelism $m = 80$, 32 KB fixed output buffer size)

the resulting 80^4 constraints could be efficiently represented and managed using framework data structures.

4.3 Experimental Results

We evaluated our approach on the Nephele framework with the job described in Section 4.1.1 in three scenarios which are (1) without any kind of latency optimizations (2) with adaptive output buffer sizing and (3) with adaptive output buffer sizing as well as dynamic task chaining. As a baseline for comparison with other frameworks we evaluated the Hadoop Online Job described in Section 4.1.2 on the same testbed.

4.3.1 Latency without Optimizations

First, we ran the Nephele job with constraints in place but prevented the master node from applying any optimizations. Figure 6 summarizes the measurement data received by the master. As described in Section 3.2, the master node maintains running averages of the measured latencies of each task and channel. Each sub-bar displays the arithmetic mean over the running averages for tasks/channels of the same type. For the plot, each channel latency is split up into mean output buffer latency (dark gray) and mean transport latency (light gray), which is the remainder of the channel latency after subtracting output buffer latency. Hence, the total height of each bar is the sum of the arithmetic means of all task/channel latencies and gives an impression of the current overall workflow latency. The solid and dot-dashed lines provide information about the distribution of measured sequence latencies (min, max, and median).

The total workflow latency fluctuated between 3.5 and 7.5 seconds. The figure clearly shows that output buffer and channel latencies massively dominated the total workflow latency, so much in fact that most task latencies are hardly

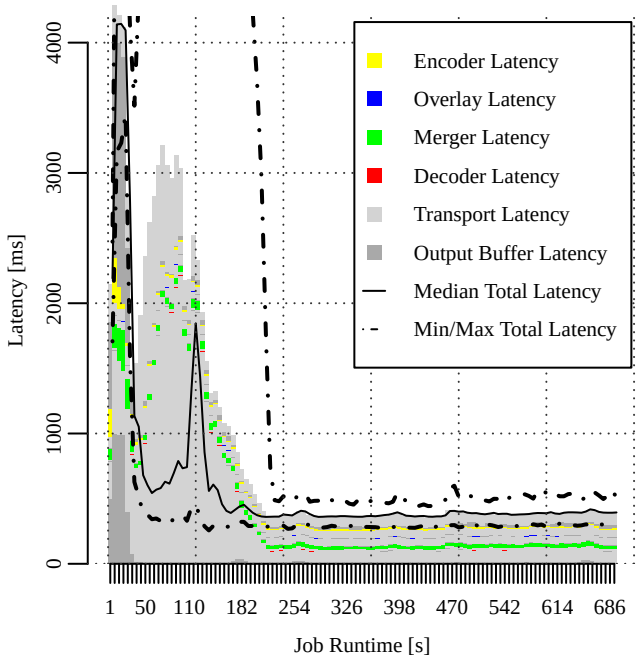


Figure 7: Latency with adaptive buffer sizing (320 video streams, degree of parallelism $m = 80$, 32 KB initial output buffer size)

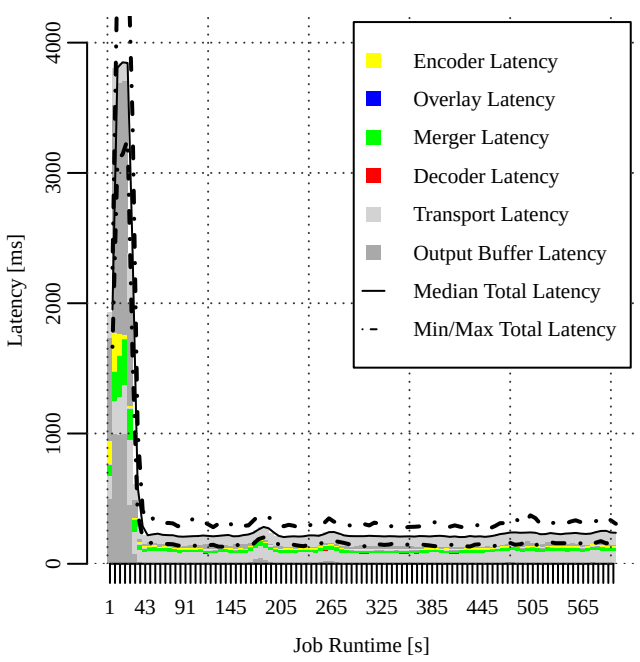


Figure 8: Latency with adaptive buffer sizing and task chaining (320 video streams, degree of parallelism $m = 80$, 32 KB initial output buffer size)

visible at all. The main reason for this is the output buffer size of 32 KB which was too large for the compressed video stream packets between *Partitioner* and *Decoder* tasks, as well as *Encoder* and *RTP Server* tasks. These buffers sometimes took longer than 1 second to be filled and when they were placed into the input queue of a *Decoder* they would take a while to processed. The situation was even worse between the *Encoder* and *RTP Server* tasks as the number of streams had been reduced by four and thus it took even longer to fill a 32 KB buffer. Between the *Decoder* and *Encoder* tasks the channel latencies were much lower since the initial buffer size was a better fit for the decompressed images.

Another consequence of the buffer size were large variations in total workflow latency that stemmed from the fact that task threads such as the *Decoder* could not fully utilize their CPU time because they fluctuated between idling due to input starvation and full CPU utilization once a buffer had arrived.

The anomalous task latency of the *Merger* task stemmed from the way we measure task latencies and limitations of our frame merging implementation. Frames that needed to be grouped always arrived in different buffers. With large buffers arriving at a slow rate the *Merger* task did not always have images from all grouped streams available and would not produce any merged frames. This caused the framework to measure high task latencies (see Section 3.1.1).

4.3.2 Latency with Adaptive Output Buffer Sizing

Figure 7 shows the results when using only adaptive buffer sizing to meet latency constraints. The structure of the plot is identical to Figure 6 which is described in Section 4.3.1.

Our approach to adaptive buffer sizing quickly reduced the buffer sizes on the channels between *Partitioner* and *Decoder* tasks, as well as *Encoder* and *RTP server* tasks. The effect of this is clearly visible in the diagram, with an initial workflow latency of 4 seconds that is reduced to 400 ms on average and 500 ms in the worst case. The latency constraint of 300 ms has not been met, however we attained a latency improvement of one order of magnitude compared to the unoptimized Nephele job.

The convergence phase at the beginning of the job during which buffer sizes were decreased took approx. 4 minutes. There are several reasons for this phenomenon. First, as the master node started with output buffers whose lifetime was often larger than the measurement interval there often was not enough measurement data for the master to act upon during this phase. In this case it waited until enough measurement data were available before checking for constraint violations. Second, after each output buffer size change the master node waits until all old measurements for the respective channel have been flushed out before revisiting the violated constraint, which took at least 5 seconds each time.

4.3.3 Latency with Adaptive Output Buffer Sizing and Task Chaining

Figure 8 shows the results when using adaptive buffer sizing and dynamic task chaining. The latency constraints were identical to those in Section 4.3.2 and the structure of the plot is again identical to Figure 6.

Our task chaining approach chose to chain the *Decoder*, *Merger*, *Overlay* and *Encoder* tasks because the sum of their CPU utilizations did not fully saturate one CPU core.

After the initial calibration phase, the total workflow latency stabilized at an average of around 240 ms and a maximum of approx. 300 ms. This finally met all defined latency

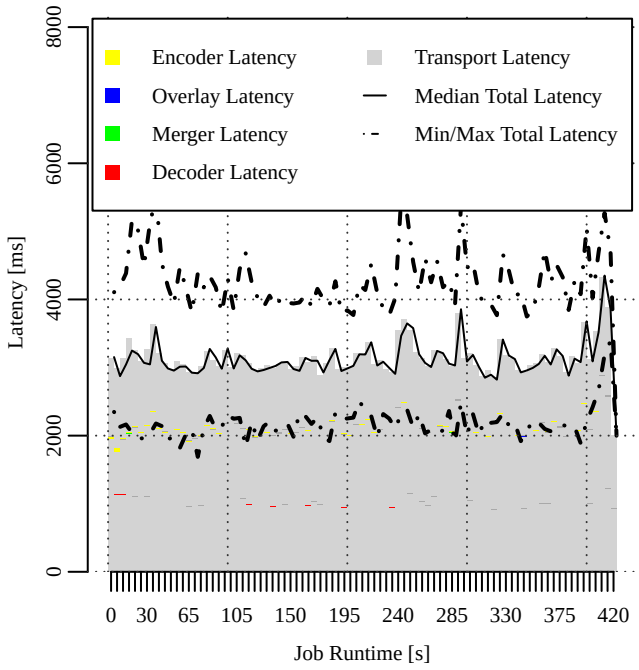


Figure 9: Latency in Hadoop Online (80 video streams, degree of parallelism $m = 10$, 100 ms window size)

constraints, which caused the optimizer to not trigger any further actions. In our case this constituted another 40% improvement in latency compared to not using task chaining and an improvement by a factor of at least 15 compared to the unoptimized Nephele job.

4.3.4 *Latency in Hadoop Online*

Figure 9 shows a bar plot of the task and channel latencies obtained from the experiments with the Hadoop Online prototype. The plot's structure is again identical to Figure 6, however the output buffer latency has been omitted as these measurements are not offered by Hadoop Online.

Similar to the unoptimized Nephele job, the overall processing latency of Hadoop Online was clearly dominated by the channel latencies. Except for the tasks in the chain mapper, each data item experienced an average latency of about one second when being passed on from one task to the next.

Due to limitations in our setup and Hadoop Online we could only deploy one processing pipeline per host. Therefore, we had to reduce the degree of parallelism for the experiment to $m = 10$ and could only process 80 incoming video streams concurrently. A positive effect of this reduction is a significantly lower task latency of the Merger task because, with fewer streams, the task had to wait less often for an entire frame group to be completed.

Apart from the size of the window reducer, we also varied the number of worker nodes n in the range of 2 to 10 as a side experiment. However, we did not observe a significant effect on the channel latency either.

5. RELATED WORK

Over the past decade stream processing has been the subject of vivid research. In terms of scalability, the existing approaches essentially fall into three categories: Centralized, distributed, and massively-parallel stream processors.

Several centralized systems for stream processing have been proposed, such as Aurora [2] and STREAM [5, 15]. Aurora is a DBMS for continuous queries that are constructed by connecting a set of predefined operators to a DAG. The stream processing engine schedules the execution of the operators and uses load shedding, i.e. dropping intermediate tuples to meet QoS goals. At the end points of the graph, user-defined QoS functions are used to specify the desired latency and which tuples can be dropped. STREAM presents additional strategies for applying load-shedding, such as probabilistic exclusion of tuples. While these systems have useful properties such as respecting latency requirements, they run on a single host and do not scale well with rising data rates and numbers of data sources.

Later systems such as Aurora*/Medusa [10] support distributed processing of data streams. An Aurora* system is a set of Aurora nodes that cooperate via an overlay network within the same administrative domain. In Aurora* the nodes can freely relocate load by decentralized, pairwise exchange of Aurora stream operators. Medusa integrates many participants such as several sites running Aurora* systems from different administrative domains into a single federated system. Borealis [1] extends Aurora*/Medusa and introduces, amongst other features, a refined QoS optimization model where the effects of load shedding on QoS can be computed at every point in the data flow. This enables the optimizer to find better strategies for load shedding.

The third category of possible stream processing systems is constituted by massively-parallel data processing systems. In contrast to the previous two categories, these systems have been designed to run on hundreds or even thousands of compute nodes in the first place and to efficiently transfer large data volumes between them. Traditionally, those systems have been used to process finite blocks of data stored on distributed file systems. However, many of the newer systems like Dryad [14], Hyracks [8], CIEL [16], or our Nephele framework [18] allow to assemble complex parallel data flow graphs and to construct pipelines between the individual parts of the flow. Therefore, these parallel data flow systems in general are also suitable for streaming applications. Recently, there have also been efforts to extend MapReduce by streaming capabilities [9, 11]. However, the general focus of these systems has still been high-throughput batch-job execution and QoS aspects have not been considered so far.

The systems S4 [17] and Storm[6] can also be classified as massively-parallel data processing systems, however, they stand out from the other systems as they have been designed for low-latency stream processing from the outset. These systems do not necessarily follow the design principles explain in Section 2.1. For example, Twitter Storm does not use intermediate queues to pass data items from one task to the other. Instead, data items are passed directly between tasks using batch messages on the network level to achieve a good balance between latency and throughput.

None of the systems from the third category has so far offered the capability to express high-level QoS goals as part of the job description and let the system optimize towards these goals independently, as it was common for previous systems from category one and two.

[6] `https://github.com/nathanmarz/storm`

6. CONCLUSION AND FUTURE WORK

Growing numbers of commodity devices are equipped with sensors capable of producing continuous streams of rich sensor data, such as video and audio in the case of mobile phones or power consumption data in the case of smart meters that are being deployed at consumer homes as part of the smart grid initiative. High numbers of such devices will produce large amounts of streamed data that will raise the bar for future stream processing systems both in terms of processing throughput and latency, as some use cases require the data to be processed within a given time span.

In this paper we examined using existing massively-parallel data processing frameworks such as Nephele for this purpose and presented strategies to trade off throughput versus latency to meet latency constraints while keeping the data throughput as high as possible. We showed how our strategies, *adaptive output buffer sizing* and *dynamic task chaining*, can be used to meet user-defined latency constraints for a workflow. We provided a proof-of-concept implementation of our approach and evaluated it using a video streaming use case. We found that our strategies can improve workflow latency by a factor of at least 15 while preserving the required data throughput.

We see the need for future work on this topic in several areas. The Nephele framework is part of a bigger software stack for massively-parallel data analysis developed within the Stratosphere project[7]. Therefore, extending the streaming capabilities to the upper layers of the stack, in particular to the PACT programming model [6], is of future interest. Furthermore, we plan to explore strategies for other QoS goals such as jitter and throughput that exploit the capability of a cloud to elastically scale on demand.

7. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research*, CIDR '05, pages 277–289, 2005.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '99, pages 955–962. IASTED/ACTA, 1999.

[4] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. MapReduce and PACT - comparing data parallel programming models. In *Proc. of the 14th Conference on Database Systems for Business, Technology, and Web*, BTW '11, pages 25–44. GI, 2011.

[5] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30:109–120, Sept. 2001.

[6] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proc. of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130. ACM, 2010.

[7] D. Battré, M. Hovestadt, B. Lohrmann, A. Stanik, and D. Warneke. Detecting bottlenecks in parallel DAG-based data flow programs. In *Proc. of the 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '10, pages 1–10. IEEE, 2010.

[8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162. IEEE, 2011.

[9] Q. Chen, M. Hsu, and H. Zeller. Experience in Continuous analytics as a Service (CaaaS). In *Proc. of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 509–514. ACM, 2011.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research*, CIDR '03, pages 257–268, 2003.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proc. of the 7th USENIX conference on Networked systems design and implementation*, NSDI '10, pages 21–21. USENIX Association, 2010.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[13] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, Mar. 2007.

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *First Biennial Conference on Innovative Data Systems Research*, CIDR '03, pages 245–256, 2003.

[16] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proc. of the 8th USENIX conference on Networked systems design and implementation*, NSDI '11, pages 9–9. USENIX Association, 2011.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177. IEEE, 2010.

[18] D. Warneke and O. Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):985–997, June 2011.

[7] `http://www.stratosphere.eu/`