DISS. ETH No. 25928

Managing and understanding distributed stream processing

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

MORITZ HOFFMANN

MSc ETH in Computer Science, ETH Zurich

born on January 3rd, 1986

citizen of the Federal Republic of Germany

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Prof. Dr. Peter Pietzuch (Imperial College London), co-examiner
Dr. Frank McSherry (ETH Zurich), co-examiner

2019

Abstract

Present-day computing systems have to deal with a continuous growth of data rate and volume. Processing these workloads should introduce as little latency as possible. Today's stream processors promise to handle large volumes of data while providing low-latency query results. In practice however, their computational model, variations of the workload, and the lack of tools for programmers can lead to situations where the latency increases significantly.

The reason for this lies in the design of today's stream processing systems. Specifically, stream processors do not supply meaningful information for debugging root causes of latency problems. Additionally, they have inadequate controllers to automate resource management based on workload properties and requirements of the computation. Lastly, their reconfiguration mechanisms are not compatible with low-latency query processing and cause a significant latency increase while reconfiguring. Solving these problems is crucial to enable users, programmers and operators to maximize the effectiveness of stream processing.

To solve these problems, we present three complementing solutions. We first propose a method to identify factors influencing query latency, using detailed internal measurements combined with knowledge of the computational model of the stream processor. Then, we use system-intrinsic measurements to design and implement an automatic scaling controller for scale-out distributed stream processors. It makes fast and accurate scaling decisions with minimum delay. Lastly, we propose a scaling mechanism that reduces downtime during reconfigurations by orders of magnitude. The mechanism achieves this by interleaving fine-grained configuration updates and data processing.

The solutions presented in this thesis help designers of stream processors to better optimize for low-latency processing, and users to increase their query performance by providing better metrics and automating operational aspects. We think this is an important step towards efficient stream processing.

Zusammenfassung

Die Datenmenge und die Datenfrequenz, mit der aktuelle Computersysteme umgehen müssen, nimmt ständig zu, während zugleich Abfragen auf diese Datenströme mit geringster Latenz bearbeitet werden sollen. Moderne Stream-Prozessoren versprechen zwar, große Datenvolumen mit geringer Abfragelatenz bearbeiten zu können. In der Realität führen die internen Strukturen der Datenverarbeitung, Schwankungen der Arbeitslast und ein Mangel an Hilfsmitteln für Programmierer jedoch zu Situationen, in denen die Abfragelatenz signifikant ansteigt.

Der Grund dafür liegt im Design moderner Stream-Prozessoren. Diese stellen keine aussagekräftigen Informationen über den Status der Abfragen sowie den Ursprung von Latenzproblemen zur Verfügung. Ihnen fehlen Kontrollmechanismen, die eine automatisierte Ressourcenzuordnung auf Grundlage der Arbeitslast und der Abfrageeigenschaften möglich machen könnten. Auch bieten sie keine Möglichkeiten, Konfigurationsänderungen ohne eine deutliche Erhöhung der Latenz anzuwenden.

Um diese Probleme zu beheben, stellen wir drei sich ergänzende Lösungen vor. Als erstes zeigen wir eine Methode, die es unter Nutzung der internen Struktur des Stream-Prozessors und mit Hilfe detaillierter Messungen ermöglicht, genau zu identifizieren, wo Latenz entsteht. Auf der Basis von systemspezifischen Messungen entwickeln wir zweitens eine automatische Steuerung zur Skalierung verteilter Stream-Prozessoren, die schnell und präzise skaliert, und dabei die Abfragelatenz möglichst wenig erhöht. Schließlich stellen wir einen Mechanismus vor, der Konfigurationsänderungen in kleine Teile aufspaltet und während der Berechnung von Abfragen anwendet, anstatt der üblichen Methode, bei der die Berechnung angehalten und neu gestartet werden muss. Dieser Ansatz reduziert die Ausfallzeit während der Änderung von Konfigurationen stark.

Die in dieser Arbeit präsentierten Lösungen helfen Entwicklern von Stream-Prozessoren, diese auf geringe Latenz zu optimieren, und erlauben Nutzern, die Latenz ihrer Anfragen durch bessere Messdaten und Automatisierung zu optimieren. Wir sind überzeugt, dass sie wichtige Beiträge zur Steigerung der Effizienz von Stream-Prozessoren darstellen.

Acknowledgments

The work presented in this dissertation would not have been possible without the collaboration and interactions with wonderful friends and colleagues.

I would like to thank my advisor Timothy Roscoe for always supporting me, believing in me, and guiding me during the course of my doctoral studies. I would also like to thank Gustavo Alonso for co-advising my dissertation. Thanks to you, Peter Pietzuch, for taking part in my committee and the valuable feedback that you have given me. Thank you, Frank McSherry, for hours of interesting discussions, brain-storming sessions and your positive attitude to turn interesting ideas into successful research projects.

During my internship I had the opportunity to collaborate with many incredibly smart collegues, which helped me to shape the structure of my dissertation: Adrian L. Shaw, Alexander Richardson, Chris Dalton, Dejan Milojicic, Geoffrey Ndu, Paolo Faraboschi, and Robert N. M. Watson.

A special thanks goes to all friends and collaborators from the Systems Group and the Strymon project: Andrea, Claude, Desi, Gerd, Ghislain, Ingo, John, Lefteris, Lukas, Matthew, Michal, Pratanu, Pravin, Renato, Reto, Roni, Sebastian, Simon, Stefan, Vasia, and Zaheer. You all made it a pleasure to be part of the group during the last five years.

Also thanks to all my friends for bearing with me during the time of writing, and especially to Florian for designing the great cover page.

For many great things in my life, I would like to thank my parents Brigitte and Hartmut. Your support, motivation and courage helped me through all my studies and allowed me to pursue my path. Finally, special thanks to you, Marina, for standing by my side and your patience during difficult times.

Contents

1
Introduction

The continuously growing rate and volume of available data is one of the major problems of present-day computing. This calls for systems that are able to process and analyze stream-like data with minimal latency. Systems have to be easy to understand and manage in order to provide latency guarantees. In this dissertation we explore the problem of system observability and manageability, and propose new and improved techniques that help stream processors to handle large amounts of data and limit processing latency.

Many data-driven applications have to provide timely results on large volumes of data, collected from various distributed sources. The data can be seen as a continuous stream or flow of data because it is only revealed over time. Typical applications include collecting and processing of data obtained from environmental sensors to identify fluctuations over time and warn when critical situations occur; analyzing user behavior on the Internet in real-time to track trends; and complex analytics on financial transactions for fraud detection. Many of the use cases will become more important in the future, which also means that the amount of data will continue to increase.

Stream processing in software is a paradigm to answer queries over continually arriving data. Such systems are supposed to provide results based on standing queries which are maintained with low latency while ingesting a high throughput of events. Many queries can be represented as dataflows, a graph-based query representation that enables simple and efficient parallel processing. For this reason, stream processors can be scaled from single machines to large clusters, depending on the computation's resource requirements and user's latency targets. Stream processors provide a good basis for data-driven applications that rely on high-throughput and low-latency data processing.

The data we want to process generally has varying and potentially unpredictable performance characteristics. It is generated by external sources which

can produce data at arbitrary times and often in bursts. For example, a large transaction on the stock market can generate many more follow-up transactions, or trending websites attract many users in short time. This leads to a workload where bursts of data and long-lasting load variations are the norm rather than the exception. A stream processing system has to handle such fluctuations gracefully while continually providing its service.

Today's stream processing solutions often fall short on providing continual service while adhering to their expected latency performance. For example, Spark Streaming's computational structure requires a barrier between scheduling operators, which causes periods of resource under-utilization because fast workers have to wait for slow workers to complete their task. Situations occur where a stream processor fails to achieve optimal resource utilization and is unable to provide simultaneous low-latency and high-throughput processing. The computation of results can be delayed if the amount of new data increases too much. In the worst case, the system becomes unstable or has to drop data to keep running. Both outcomes are not desirable because results could be delayed, unreliable, or at worst incorrect.

The first step to determine the causes for such undesirable effects is to identify where the problems occur. For this, it is necessary to have readily accessible and easily understandable metrics at hand. The level of detail depends on what problem needs to be analyzed and ranges from detailed operator scheduling information and operator dependency information to specific aggregated metrics, for example, how much data is processed *while an operator is running*. Current stream processors do not report metrics in a way that allows operators to properly analyze and address unstable configurations. Metrics are often dependent on a stream processor's execution model and therefore require a deep understanding of the system. Also, they are usually aggregated and lack a precise description on how to interpret them. Even minor changes in the definition and interpretation of reported metrics can give much better insight into the reasons where and why stream processors fail to deliver their expected performance.

Better metrics combined with system-level scheduling and dependency information enable us to better understand the interaction of the components in stream processors. They help human operators to improve the configuration parameters and permit automated controllers to generate better configurations to tune performance characteristics. Additionally, they help programmers to combine components in a more efficient manner, potentially improving processing latency and throughput. Deriving good parameters for a stream processor

is a repeating task as workload performance characteristics change over time and the stream processor may need to be reconfigured to match the workload. Therefore, we need up-to-date metrics that reflect the current state of the processor to be able to adapt its configuration to meet latency and throughput targets even if the workload changes.

1.1 Structure of the dissertation

In this thesis, we introduce techniques to improve understandability and simplify managing stream processors in an environment of volatile workloads without introducing undesired latency problems.

- Firstly, we propose a method to utilize and refine information available within the stream processing system and extend it with new information to gain a better understanding of the system's performance.
- Secondly, we use specific system-intrinsic measurements to implement an automatic controller for scaling-out distributed stream processors.
- Thirdly, we study the problem of reconfiguring distributed stream processors and develop a new mechanism to deploy configuration updates with minimal latency impact.

It is our hypothesis that these methods and tools help to bring a stream processor's capability closer to the needs of users. For these reasons, we will aim at improving the state-of-the-art of stream processing from the following three angles.

Gain better insight: Existing stream processors lack meaningful diagnostic information to be able to determine root-causes for performance problems. The first step toward a better understanding of stream processing systems is to characterize precisely how their workload and queries interact. Especially in long-running distributed systems end-to-end latency is caused by manifold interactions between numerous components. In this thesis we analyze current problems of distributed system performance debugging and introduce a new technique to derive performance indicators allowing a precise breakdown of factors affecting end-to-end latency. The technique combines an analysis of the actual processing time with a component dependency evaluation using critical path analysis. This dissertation introduces the *critical-participation* metric, which adapts critical path analysis to long-running computations. The metric gives analysts an unprecedented understanding of the latency hotspots within

a stream processor and can be used for improving implementations, adjusting query plans, and balancing resource assignments.

Automate control: Today's stream processing systems expose various parameters to adjust their performance, for example the number of worker threads, the operators assigned to workers or the sizes of queue buffers. Specifically, they can distribute work across many machines and assign operators to remote workers. Manually managing the distribution of a stream processor requires a lot of knowledge and experience, and suffers from a slow reaction time. Nevertheless, current stream processors have inadequate controllers to automatically manage resources based on the workload properties and service-level objectives. Existing techniques to automatically scale-out stream processors across machines have a slow convergence, require many adjustments and are often based on inadequate heuristics. We also found that scaling-out existing stream processors revealed high query latency due to the lack of efficient reconfiguration mechanisms.

To address this problem, we present an automated controller to show that a distributed stream processor can be automatically scaled-out based on external requirements, detailed internal measurements, and knowledge of the computation's structure.

Reconfigure seamlessly: The reconfiguration mechanisms offered by current stream processors are not compatible with service-level objectives and cause a significant latency increase while applying a new configuration. During scaling or under volatile workloads, reconfigurations of stream processors ideally should not affect query latency. Many current reconfiguration mechanisms have a great latency impact or large resource requirements and thus are only suitable for infrequent reconfigurations. Because of this, they tend to be used as a last resort. It would be highly desirable to reconfigure stream processors frequently and fluidly as their workload varies.

We propose a mechanism that handles configuration updates as data and interleaves reconfigurations with computation by splitting the reconfigurations into fine-grained updates. The mechanism avoids synchronization by building on system-provided coordination primitives. This enables reconfigurations of running stream processors delivering peak latencies that are orders of magnitude smaller than existing techniques, permitting virtually seamless fine-grained and frequent control over stream processing systems.

To begin, we introduce basic concepts and show a historic perspective of stream processing systems, followed by an overview of related work, in Chapter 2.

Then, we analyze the outlined problems in the order presented above. In Chapter 3, we employ critical-path analysis to introduce a novel technique to analyze latency bottlenecks in distributed stream processors. In Chapter 4, we describe a controller for distributed stream processors which automatically determines resource requirements to fulfill service-level objectives. In Chapter 5, we present a mechanism to apply configuration updates on a stream processor while maintaining minimal latency. Lastly, in Chapter 6, we conclude the thesis with a discussion of learnings and future work.

1.2 Notes on collaborative work

This thesis includes results from collaborations with Zaheer Chothia, Desislava Dimitrova, Matthew Forshaw, Vasiliki Kalavri, Andrea Lattuada, John Liagouris, Frank McSherry, Timothy Roscoe, and Sebastian Wicki. Chapter 3 is based on *SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows* [Hof+18]. Chapter 4 is based on *Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows* [Kal+18]. Chapter 5 is based on *Megaphone: Latency-conscious state migration for distributed streaming dataflows* [Hof+19].

2
Background and motivation

To begin, we introduce some fundamental concepts for distributed stream processing, which we will use throughout this thesis. Then, we take a step back and present a critical history of stream processing systems. We show the evolution of techniques and their improvements and shortcomings, thus giving an overview of the context in which our work is placed. We highlight the problems in existing systems to motivate the solutions we are presenting in this thesis. Afterwards we give an overview of specific related work for the individual problems discussed in this thesis. The texts of Cugola and Margara [CM12] are excellent starting points for further reading on the history and capabilities of stream processors.

2.1 Stream processing concepts

Informally speaking, a stream is a set of data that is only revealed over time. To give some examples: A distributed sensor network might send a stream of measurement values, users browse the Internet and generate a stream of interactions, or users send tweets. Each measurement or interaction occurs at a point in time and carries a measured value or other data, and the data is made available over time. Also, the data is produced in a distributed fashion, meaning there may not exist a sequential order of elements. Our definitions are inspired by the formalism used by Stephens [Ste97].

Definition 1 (Stream)**.** *A stream s is a function from time T to sets of data D:*

$$s : T \to D$$

We call two streams *equal* if and only if they yield the same data at equal points of time. Let S and R be two streams, and $a, b \in T$. Then,

$$a = b \rightarrow S(a) = R(b).$$

We are interested in transforming streams to produce new insights and results. For this reason, we introduce the concept of queries to describe the processing of streams. A query is defined by a user and expresses how to transform data from existing input streams to formulate new output streams. It is a standing description of how data from input streams is converted into output streams. A query can consume at least one input stream and produce at least one output stream.

A simple example is a *word-counting* query. This query reads complete sentences or pieces of text and outputs the number of word occurrences over time. To produce this result, it needs to split text into words and then group equal words to calculate the number of occurrences. Thus, the data domain of its input stream are pieces of text and the output stream's data consists of sets of (word, count)-pairs. Word-count is a simple example for stream processing. Of course, stream processing can be used for more complex queries, such as incremental graph computations and incremental data management.

Definition 2 (Query)**.** *A query q is a function from m input streams to n output streams. Each stream has its own data type. All streams have the same time domain T.*

$$q\colon\ (T \rightarrow D_1, \ldots, T \rightarrow D_m) \rightarrow \left(T \rightarrow D'_1, \ldots, T \rightarrow D'_n\right).$$

- $S^{in} = (T \rightarrow D_1, \ldots, T \rightarrow D_m)$ *is the ordered set of input streams.*
- $S^{out} = \left(T \rightarrow D'_1, \ldots, T \rightarrow D'_n\right)$ *is the ordered set of output streams.*

We observe the queries can be composed to bigger units if a query's set of output streams corresponds to another query's set of input streams. This property allows us to combine queries and decompose them into smaller units, which we will explore in the next section. Due to the same reason, we use the term *operator* as a synonym for *query*.

2.2 Expressing queries as dataflows

In this thesis, we build on existing stream processors that use a dataflow model to implement queries. Examples of stream processors following this model

are *Apache Flink* [Car+15], *Apache Heron* [Heron], or *Timely Dataflow* [McS]. The dataflow abstraction uses a decomposition of queries into operators to process data and streams to forward data between operators. The model has several interesting properties, of which one is simple parallelization due to the data parallel nature of operators.

A stream processor executes queries by mapping them to *workers*. In a computer, a worker can be a CPU core, machine, or accelerator such as GPUs and FPGAs. We assume a worker is an execution unit that executes operators sequentially. Importantly, a worker does not offer parallel execution of operators but can execute different operators sequentially. We assume that at any point in time, a worker can only execute a single query. So, how can we parallelize query processing?

To solve the problem of assigning a query to multiple parallel workers, a common approach is to split the query into smaller queries. For this reason we formalize a decomposition of queries into *dataflows*, which are a *graph of queries*.

Definition 3 (Dataflow)**.** *A dataflow is a directed graph $G = (Q, S)$ of queries Q and streams S.*

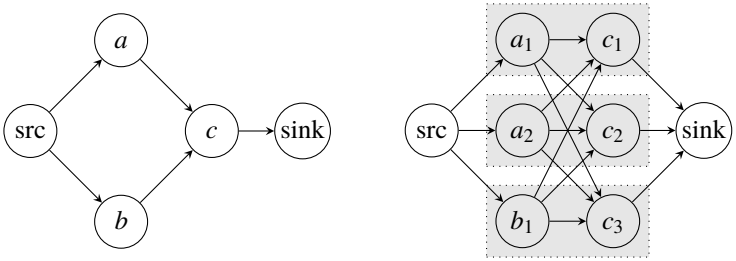
- $S^{source} \subseteq S$ *is the set of source streams.*
- $S^{sink} \subseteq S$ *is the set of sink streams.*

The decomposition of a query q with input streams S^{in} and output streams S^{out} has the property that the dataflow's source and sink streams are equivalent to the in- and out-streams of q: $S^{\text{in}} \equiv S^{\text{source}}$ and $S^{\text{out}} \equiv S^{\text{sink}}$.

The queries in a dataflow can be scheduled individually by the stream processor on several workers. Queries in a dataflow have well-defined dependencies between each other in the form of streams. This allows for parallel processing as we assume no other data is shared between operators.

Figure 2.1 illustrates a logical graph and its corresponding physical graph for a dataflow with a source, a sink, and three operators. Operators a, b, and c execute with two, one and three physical instances.

The dataflow model enables different models of parallelism. Independent operators can be executed concurrently, which is commonly called task parallelism. Subsequent operators in the same dataflow graph can be executed concurrently, called pipeline parallelism. The downstream operator can read the upstream's output while the upstream operator already computes new data. Parallel instances of the same operator can compute results on shards of the



(a) Logical dataflow graph (b) Physical dataflow graph

Figure 2.1: Logical (a) and physical (b) dataflow graphs. The logical dataflow has a source, a sink and three operators in between. The source sends data to both a and b. Their results are combined by operator c, which computes the final result. The gray boxes in (b) show an example assignment of operators to workers.

input data, which is called data parallelism. These models of parallelism allow stream processors to distribute computations over large clusters of machines, laying the ground for low-latency and high-throughput query processing.

Decomposition of queries into dataflows fits well to today's computer architectures. Stonebraker [Sto86] introduced the term of a *shared-nothing architecture*. In this architecture, databases are divided into independent components that can only communicate with messages. No other form of information sharing, for example using shared memory or disks, is permitted. Today, this model can be expanded to distributed computer architectures, such as clusters and rack-scale systems, where processing units are linked by networks. The dataflow model naturally follows a similar goal. Queries are divided into independent components with clear constraints for their execution. All data is exchanged in the form of streams. The dataflow model obviously finds its parallel in distributed shared-nothing architectures.

We formally introduced streams and queries, which are the foundation of stream processing. Queries can be decomposed into dataflows, well-matched to today's distributed computer architectures and allowing different levels of parallel execution. Modern stream processors use this to scale dataflow computations to large clusters with many workers. Next, we analyze the history of stream processing, leading to current state-of-the-art systems and techniques.

2.3 A critical history of stream processing

Computer systems have always been used to process large-scale inputs since the beginning of the era.[1] Initially, their use-cases were dealt with individually, as the concept of data streams had not yet evolved, the volume of data was relatively low, latency requirements were not strict, and computer systems tended to run specialized software.

The idea to interpret data processing as a continuous stream of data has been around since the 1960s [Ste97; Bur75]. The term *stream processing* saw increasing use starting in the 1980s when new semantics for stream processing were introduced, both as a way to express programs [Ste97; WA85] and to process data, with computers becoming ever-more wide-spread, fueling the need for generalized approaches to processing problems. General paradigms of dealing with large amounts of data prevailing at the time began to be adapted to stream-related problems. Relational databases were used for low-latency queries on transactional data, and tape-oriented systems for large amounts of data that had to be processed at a high throughput,

Around this time, many computer systems became networked, which instantly increased both the amount of accessible data and the data generated by larger-scale systems. These data needed to be processed, either to gain momentary insights, or for storage and data retrieval, depending on each use case. The growth of distributed resources also required new application architectures to manage them.

Stream processing can be analyzed from various angles. Traditionally, many use-cases requiring low-latency data processing relied on *complex event processing*, which is a technique to identify relevant, user-defined patterns in stream-like data. A parallel development is to apply techniques known from relational databases on low-latency stream data processing. In the following discussion we choose to analyze stream processing techniques from a database perspective.

In the remainder of Section 2.3, we will present three generations of stream processors and explain what use-cases they were designed for and what limitations were imposed. The first generation of stream processors covers applications dealing with stream-like data, specialized for specific use-cases with known properties. Such systems were good at what they were designed for

[1] Even one of the first practical applications of digital computing, the decryption of huge amounts of German radio messages in the *Colossus* [Flo83] computers during World War II, can be seen as an early case of stream processing.

but offered little elasticity to accommodate changing requirements, such as workload variations. Second generation stream processors aimed at providing generalized stream processing capabilities, combined with some elasticity. Most research was motivated by applying techniques of relational databases to streams. The generality improved adoption, but limited scalability was still an issue and often solved by forcefully limiting the amount of data by dropping records. Third generation systems aim at solving the latter problem: good scalability can replace load-shedding mechanisms, but it comes at a price of increased complexity.

2.3.1 First generation stream processors

First generation stream processors were designed as generic single-machine applications tailored to their use-cases. They were used to compute queries with a large amount of input data on machines with limited random-access memory significantly smaller than the input data. For example, *Tangram* [JMC89] explored the implications of logic programming for stream processing providing a "database-flow" computation capability, suitable for both traditional database queries as well as time-based (stream) data processing. Other systems like *Tribeca* [Sul96] offered a restricted query interface for specialized applications while still providing the functionality of a stream processor. Tribeca is a system to analyze streams of network traces that do not fit in memory. Instead, data can be played back from tapes for later analysis.

Problem sizes were not very big and hence stream processors were not designed to be distributed across machines. Individual machines did not offer large resources, which meant that latency was not too much of a concern.

First-generation stream processors did influence later stream processors for their query language expressiveness and demonstrated the trade-offs between query complexity and runtime resource requirements. While early stream processors did solve problems for the use-cases they were designed for, they did not gain wide adoption. They did, nevertheless, demonstrate that generic solutions can efficiently solve specific problems.

2.3.2 Second generation stream processors

Research into stream processing systems gained momentum at the beginning of the 2000s, when a variety of new systems appeared. Abadi et al. [Aba+03] argued that traditional relational databases are not well-suited for a variety of

applications because they are designed around the data being at the center of the system. The authors identified a trend from human-generated data towards machine-generated data as the main motivation for a new generation of stream processors. Database systems process queries as they arrive, trying to answer them as soon as possible, but do not offer well-integrated means to express standing queries where a client should be notified whenever a result changes. Hence, handling real-time updates and associated actions is not well-supported and requires a fundamentally different system architecture. By this time, structured query language (SQL) had become the de-facto standard of querying databases, but stream processing systems lacked a similar standardized interface.

Second generation stream processing systems aimed at closing the gap between database semantics common to database management systems (DBMSs) with stream-semantics, queries over a history of data, and better support for triggering actions as data changed. In addition to this, new use-cases emerged. Smaller and more energy-efficient computers enabled a wide deployment of sensors for data collection. Also, stock markets became more-and-more computerized, requiring more advanced computer systems to analyze data and make decisions in short amounts of time.

To provide versatile semantics for advanced use-cases, second-generation stream processors introduced the notion of *temporal windows*. Windows provide an abstract to perform operations on input data and state based on time. For example, data can be aggregated per window of a defined duration. This enables different computations and also allows stream processors to safely discard data after results have been computed and a window is over. Windowing is discussed in further detail in *Aurora* [Aba+03].

Aurora [Aba+03] is a database with support for continuous queries on data generated by machines in addition to human-initiated transactions. It supports continuous queries where the input data is processed only once, database-style queries that require persistence of data, and ad-hoc queries that can access a limited amount of history data. In overload situations, it focuses on maintaining the output of results by dropping data, but this potentially gives imprecise and incorrect results and is generally undesirable in stream processing systems. It trades-off correctness against resilience to workload fluctuations. Aurora has a centralized storage manager and scheduler, both of which limit its scalability to only scale-up within a single machine but prevents scale-out across multiple machines. Its query language is similar to SQL, however with adapted semantics. For example, a join operator can only cover a limited amount of time, which

does not represent general join processing. A reason is that converting queries from a SQL-like language to actual dataflows requires an optimizer, which did not yet exist at this time (and is currently not a fully solved problem). Here, Aurora again trades off performance and liveness against generality of streaming computations.

Aurora introduced a new approach for the semantics of a stream processing system, but knowingly left many parts of the system design as future work. The same authors presented *Borealis* [Aba+05] in 2005, which extends Aurora's model to a distributed stream processor to extend the model to scale-out. Also, the authors recognized that data tends to arrive out-of-order and needs to be either serialized before results can be computed, or results need to be changed at a later point in time. Borealis uses a data model where data represents an insertion (new data), a deletion, or a revision of some existing data and provides operators that can handle each over a limited history. The key conflict is to provide results as early as possible while still being able to change them as new data arrives. Like Aurora, Borealis lacks an abstract notion of time to decouple processing from event time.

TelegraphCQ [Cha+03] is another second-generation stream processing system with a focus on query optimization for unpredictable workloads. It uses the concept of an *Eddy* to dynamically switch between implementations and decouples transient data storage from operators to ensure correctness. TelegraphCQ's implementation is based on Postgres, a relational database, and the authors note that many components also apply to stream processors. The authors point out that data exchanged between streaming operators are handled as persistent but temporary relations, but they also note this introduces a performance bottleneck and prevents scale-out.

At roughly the same time, *STREAM* [Ara+03] introduced the continuous query language (CQL), a new query language to express standing queries on streams. The language was aligned to SQL but added important constructs to handle different kinds of streams, which are insertion streams, difference streams or fully materialized result sets. The query language can describe time-based queries, which is an important difference compared to existing SQL. Like Aurora and TelegraphCQ, STREAM's implementation lacks scale-out functionality. STREAM uses load shedding to handle overload situations but tries to approximate results based on statistical models if data is dropped. Its scheduling policies are geared towards maintaining throughput and system stability, potentially at the price of higher latency.

Second-generation stream processors offered better support for expressing stream computations with a query language and optimizing query plans, while focusing on maintaining throughput. Their processing latency was determined by the complexity of their internal queuing networks. They showed a clear improvement on expressing continuous queries but their architecture limited system scalability, which nevertheless is required to gracefully handle workload fluctuations. To overcome this limitation, systems had mechanisms to trade-off availability versus correctness by dropping data under high load.

2.3.3 MapReduce: large-scale computations

In 2004, Google published *MapReduce* [DG04], a programming model for processing large data sets on large clusters. Its core idea is to parallelize data processing with a local map phase followed by a global shuffle and reduce phase, with the shuffle phase serving as a global barrier. MapReduce had a surprising effect on large-scale cluster computing because with little effort large amounts of data could be processed, at a very high throughput. This also lead to a *third generation of stream processing systems*. Although MapReduce is not a stream processor, people started using its batch-processing semantics to approximate stream processing over batches of data. The data has to be stored on disk before it can be processed. This approach enabled systems to provide a high throughput, although at the cost of high and unpredictable latency. *Hadoop* [Hadoop] is an open-source implementation of MapReduce, and offers stream processing in terms of batch processing.

MapReduce's model significantly influenced the architecture of future stream processing systems as it presented an approach for scale-out. However, its computational model is too simple for any stream processing application. Queries have to be in the form of a map-and-reduce phase but only have an implicit and restricted notion of time (when a batch of data ends, some time has probably elapsed). Computations are scheduled by a centralized controller, which is not able to prioritize tasks based on their latency requirements, there is a global barrier between subsequent data-dependent operations, and computations are pre-planned and cannot contain data-dependent alternatives. To summarize, MapReduce's model is a good inspiration to distribute workloads across many machines but is itself not suitable for low-latency workloads and data-dependent computations. However, it demonstrated that data processing can be scaled out, which opened a path forward for modern stream processing systems that adapt to workload variations while processing all data to produce correct results.

2.3.4 Third-generation stream processors

Third-generation stream processors combine stream processing with large-scale cluster computing and aim for correct and low-latency results. While previous systems were able to deliver low-latency results, their limited scalability required mechanisms to control a stream processor's workload. Commonly, this was achieved by trading-off precision with latency, for example by dropping data in overload situations. Combining stream-processing semantics with large scale cluster computations and guarantees to process all data opened new interesting problems, some of which are not yet completely solved. In this thesis, we describe some problems in detail and explain our solutions. Before that, we present third-generation stream processors and their properties.

DryadLINQ [Yu+08] is a system and programming model for large-scale distributed programs. It compiles queries on large datasets into a dataflow representation to enable efficient parallel execution. Queries can be expressed in .NET and LINQ, a query language designed to to be integrated directly in the source code of programs. The queries can be much more complex than what MapReduce offers. The authors criticize limitations of SQL, such as fixed data types and limited support for iterative computations. DryadLINQ supports iterative computations on static data and claims to optimize queries both statically and dynamically while the system is running.

Similarly, *Spark* [Zah+12] is a cluster-computing model that extends the model of MapReduce to provide a common abstraction for different computations. In contrast to MapReduce, it does not require replication of intermediate results but rather remembers how a result can be computed. In the event of a fault Spark can determine which datasets need to be reconstructed and schedules the required operators. Its programming abstractions are defined in term of *resilient distributed datasets*. This abstraction helps to avoid some latency overheads of MapReduce but still leads to a relatively high latency as data between operators needs to be saved before a downstream operator can be scheduled. Spark lacks an abstract notion of time and instead assumes that processing of batches corresponds to advances of time. It therefore cannot easily decouple processing time from event time. Like DryadLINQ, Spark is designed to process existing and complete data. Spark has a centralized scheduler which distributes work packages and awaits their completion. Its fault-tolerance mechanism requires Spark's scheduler to only schedule new work once previous work has been fully completed. Work is parallelized across partitions of data, but natural variations can cause partitions to be differently

sized and hence processing time can vary, which limits the scheduler's ability to assign new tasks to the processing latency of the slowest worker.[2]

Discretized streams [Zah+13] is a layer on Spark. It maps Spark's execution model to stream processing, including basic support for iterative computations. It reduces the batch size so that each batch represents a time range of data of configurable length. Its notion of time is thus tied to the availability of batches of data, which limits the expressiveness.

SEEP [Fer+13] is a fault-tolerant stream processor supporting scale-out based on its state-management technique. It shows that partitioning state and co-locating it with upstream operators is beneficial for both fault-tolerance as well as scaling-out. Compared to MapReduce, SEEP assigns operators to specific workers and maintains the operator state locally on that worker. For fault-tolerance, the state is periodically checkpointed to upstream operators. This approach leads to good query latency because data access for stateful operators is cheap. Latency increases, though, during checkpointing and scaling operations because affected operators are momentarily unable to serve requests. SEEP offers good latency during steady state but introduces momentarily high latency during reconfigurations.

Storm [Tos+14] and its newer, API-compatible sibling *Heron* [Kul+15] are stream processors that scale out to large clusters while providing high throughput with good latency. They allow programmers to define queries as dataflow graphs, which are mapped to physical resources using a cluster scheduler. Fault-tolerance can be provided by specialized operator implementations that can persist their state when the system decides to take a checkpoint. Both implementations can only track progress based on system time and do not have a notion of a different, possibly data-dependent, time domain. This can limit compatible use-cases and strongly ties processing speed to a notion of time. Later stream processing systems address this problem by separating the time of processing of data from the time inherent to the data itself.

Third generation stream processors show how the dataflow and stream processing paradigm can be mapped to scale-out architectures. A scale-out architecture requires new approaches for handling per-operator state as it implies that query execution is distributed across machines. Due to the properties of scale-out architectures, systems have to cope with fluctuations of processing capacity and failures on top of the general stream processing workload.

[2] Spark's authors are aware of the scheduling problem and published a solution with *Drizzle* [Ven+17], allowing the scheduler to distribute tasks eagerly.

While these systems achieve good throughput, latency can be high. For systems based on MapReduce, this applies during the whole execution. None of the presented batch processing-based systems optimizes for low query latency. This opened research on optimizing systems both for latency and throughput while providing a rich set of operators.

The processing latency of a stream processor depends on three factors: the latency of the operators, the length of the queues between operators and the batching of data. MapReduce's approach is to create large batches of data, which are by default 64 MiB. Processing large amounts of data in one operation leads to very good throughput as data is readily available. However, to reduce latency we are interested in processing data as soon as possible after it is available. Waiting for larger batches of data leads to higher latency. The other extreme, processing single data elements as they arrive, leads to good latency but has very low throughput because the processing overhead is high. Instead, many new systems adopt a micro-batching approach where small batches of data are processed individually, typically based on how much data has arrived since the last time the input was drained. Micro-batching can lead to both good latency and high throughput.

Nephele [WK09] and its fork *Apache Flink* [Car+15] are distributed stream processors that claim to provide both low-latency stream processing and batch jobs using one model. They use micro-batching to increase throughput while maintaining relatively low latency. In Flink, a computation is represented as a directed acyclic graph. Still, its watermark-based progress tracking technique permits a limited set of cyclic computations. Flink provides detailed aggregate performance indicators. As we will show later, these indicators are useful in diagnosing some performance problems but are not adequate for detecting problems arising from the dependency graph of operators.

MillWheel [Aki+13] is a framework for a fault-tolerant stream processor. It provides a low-watermark mechanism similar to Flink that allows operators to determine if all data before a certain time has been received. MillWheel's fault-tolerance is achieved by externalizing all operator states to a fault-tolerant storage system. The authors assume that computations can be large and span many machines in a shared cloud-based configuration, which implies that machines hosting operators can crash or be preempted. For this reason, a fast restart mechanism can be achieved by externalizing all state such that a new instance can start processing immediately after the old disappears. However, such a setup suffers from relatively high processing latency as all state reads and updates need to be committed to an external storage system.

Naiad [Mur+13] introduced a general-purpose dataflow system to provide both low-latency stream processing and high-throughput batch processing semantics. We use Naiad's execution model *timely dataflow* as a basis for large parts of this dissertation. It uses logical time to provide iterative computations interleaved with processing of new data, and data is processed in the form of micro-batches. Query dataflows are expressed as a directed graph that can have cycles to support iterative computations while also processing new data. To enable low-latency query processing the execution model is based on fine-grained operator scheduling on top of operating system thread scheduling. Also, by multiplexing operators on workers it achieves a better scalability with large dataflows consisting of many operators. Nevertheless, this comes with a price: fine-grained batching of data combined with many operator instances and custom scheduling increases the complexity of the system dramatically. From our experience it takes a lot of knowledge and effort to precisely understand the behavior of a computation and to derive actions from observations. Naiad's increased complexity demonstrates the need for powerful tools for operators and developers to understand and manage the system well.

New generation stream processors show that by better scheduling of operators both throughput- and latency-critical query-processing can be provided inside the same system. However, they do not yet provide the necessary structure and tools to make full use of their capabilities.

This short overview of the different approaches to stream processing over time and its current state demonstrates the need for better techniques to analyze stream processors' latency problems, mechanisms to update their configuration, and controllers to determine resource requirements automatically. In this thesis, we introduce approaches to fill these deficits.

Next, we give an overview of specific related work and techniques for the individual chapters of this thesis. Many of this thesis' contributions are implemented and evaluated on Timely Dataflow. We briefly review these concepts in Section 2.4 for completeness, as they are necessary to understand the following sections. In Section 2.5, we present related work for analyzing system latency bottlenecks, controllers for stream processors and reconfiguration mechanisms, which is relevant for Chapter 3. In Section 2.6, we show work related to scaling controllers for distributed dataflows, which is important for Chapter 4. Afterwards in Section 2.7, we exhibit related material to reconfiguring stream processors, which is required for Chapter 5.

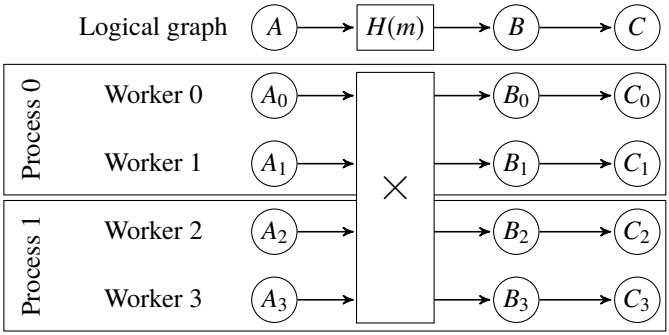


Figure 2.2: Timely Dataflow execution model

2.4 Timely dataflow concepts

A streaming computation in Naiad is expressed as a *timely dataflow*:[3] a directed (possibly cyclic) graph where nodes represent stateful operators and edges represent data streams between operators. Each data record in a *timely dataflow* bears a *logical timestamp*, and operators maintain or possibly advance the timestamps of each record. Example timestamps include integers representing milliseconds or transaction identifiers, but in general can be any set of opaque values for which a partial order is defined. The Timely Dataflow system tracks the existence of timestamps, and reports that processed timestamps no longer exist in the dataflow, which indicates the forward progress of a streaming computation.

A *timely dataflow* is executed by multiple *workers* (threads) belonging to one or more OS processes, which may reside in one or more machines of a networked cluster. Workers communicate with each other by exchanging messages over data channels (shared-nothing paradigm) as shown in Figure 2.2. Each worker has a local copy of the entire Timely Dataflow dataflow graph and executes all operators in this graph on (disjoint) partitions of the dataflow's input data. Each worker repeatedly executes dataflow operators concurrent with other workers, sending and receiving data across data exchange channels. Due to this asynchronous execution model, the presence of concurrent "in-flight" timestamps is the rule rather than the exception.

[3] We use *timely dataflow* to reference the model and Timely Dataflow to describe the system [McS].

As Timely Dataflow workers execute, they communicate the numbers of logical timestamps they produce and consume to all other workers. This information allows each worker to determine the possibility that any dataflow operator may yet see any given timestamp in its input. The Timely Dataflow workers present this information to operators in the form of a *frontier*:

Definition 4 (Frontier)**.** *A **frontier** F is a set of logical timestamps such that*

1. *no element of F is strictly greater than another element of F,*
2. *all timestamps on messages that may still be received are greater than or equal to some element of F.*

In many simple settings a frontier is analogous to a *low watermark* in streaming systems like Flink, which indicates the single smallest timestamp that may still be received. In *timely dataflow* a frontier must be set-valued rather than a single timestamp because timestamps may be only partially ordered.

Operators in *timely dataflow* may retain *capabilities* that allow the operator to produce output records with a given timestamp. All received messages come bearing such a capability for their timestamp. Each operator can choose to drop capabilities, or downgrade them to later timestamps. The Timely Dataflow system tracks capabilities held by operators, and only advances downstream frontiers as these capabilities advance.

Timely dataflow frontiers are the main mechanism for coordination between otherwise asynchronous workers. The frontiers indicate when we can be certain that all messages of a certain timestamp have been received, and it is now safe to take any action that needed to await their arrival. Importantly, frontier information is entirely passive and does not interrupt the system execution; it is up to operators to observe the frontier and determine if there is some work that cannot yet be performed. This enables very fine-grained coordination, without system-level intervention. Further technical details of progress tracking in *timely dataflow* can be found in *Naiad* [Mur+13] and in the work of Abadi et al. [Aba+13].

2.5 Related work: Distributed systems performance analysis

A goal of this thesis is to improve the understanding of stream processors in order to derive good metrics and trace performance problems to their cause.

There exists abundant literature on performance analysis, characterization, and debugging of distributed systems, and stream processors are a special instance of distributed systems. We distinguish three main areas of related work.

Critical path analysis Critical path analysis (CPA) is a technique to find the longest dependency path in an execution trace because this path will determine the end-to-end latency and hence improving operations that are on the critical path can result in performance improvements. Nevertheless, we know of no prior work to perform online critical path analysis for long running computations, or across a broad range of execution models. Yang et al. [YM88] first applied CPA to distributed and parallel applications, defined the program activity graph (PAG), gave a distributed algorithm for CPA, and showed its benefits over traditional profiling. The PAG represents activations of operators and messages between them as a graph over time. CPA and related techniques have since been used to analyze distributed programs like message passing interface (MPI) applications [Sch05; Böh+12] and web services [Cho+14], in all cases using offline traces. Schulz [Sch05] proposes a tool that intercepts MPI library calls, gathers traces on each worker locally during execution, combines the traces after program termination to construct the program activity graph, and computes and visualizes the critical path. The novelty of the approach lies in the fact that each parallel process creates a local execution subgraph during execution. These subgraphs are stored to disk after termination and the PAG construction and CP computation are performed offline. The method is not applicable to long-running jobs. The author presents results for up to 128 workers and 300×10^3 nodes (full)/1000 nodes (collapsed) but does not mention how long the analysis takes.

Böhme et al. [Böh+12] combine critical path analysis with traditional performance profiles to uncover load imbalance problems and characterize resource consumption of MPI applications. Their work uses critical path profiles and imbalance indicators to reveal imbalance problems. All analyses are performed offline by replaying execution traces. The CPA is integrated as an extension to an existing performance analysis tool. Böhme et al. not only compute the critical path but also attempt to understand the impact of changes to the critical path and to execution. They argue that the critical path is hard to obtain online and that scalable extraction is challenging.

Chow et al. [Cho+14] present the Mystery Machine, an end-to-end performance analysis tool for web services. The system uses log samples to generate

and validate a dependency graph model of execution segments among different components of the Facebook service infrastructure stack. The goal of the system is to determine the factors that affect end-to-end latency of requests to Facebook servers and suggest optimizations that would possibly lower that latency. The solution targets heterogeneous distributed systems with diverse executable components and software stacks. A major goal is to use existing logging infrastructure and automatically extract an execution model from already available traces. The traces are gathered in a central location and the analysis is performed offline as a parallel Hadoop job. The Mystery Machine uses critical path analysis to find bottlenecks. It creates a dependency graph that resembles the program activity graph. This graph is initially fully connected and dependencies are gradually removed by using explicit information from the traces. The system computes critical paths on the traces and aggregates the results to find how often an activity is on the critical path or what percentage of the critical path an activity type accounts for. The Mystery Machine also calculates slack for each segment and identifies outlying requests.

Algorithms to compute the k longest (near-critical) paths in a computation are given by Alexander et al. [Ale+98]. The authors also describe how to build a PAG for operating system-level trace points.

The first *online* method for computing critical path profiles seems to have been introduced by Hollingsworth [Hol96], where performance traces are piggybacked on data messages exchanged by processes at runtime. However, the proposed algorithm is too expensive to construct the full PAG and is thus limited to a small number of user-selected activities. A nice feature of Hollingsworth's [Hol96] work is combining online CPA with dynamic instrumentation to selectively enable trace points on demand. Saidi et al. [Sai+08] extend the analysis of Hollingsworth [Hol96] to the full software stack, and Dooley and Kalé [DK10] use this information for adaptive scheduling. *Sonata* [Guo+13] pinpoints critical outliers using a correlation techniques. It supports offline analysis of MapReduce jobs through identifying correlations between tasks, resources and job phases but its approach only applies to batch processors where the execution of tasks is synchronized between workers.

Dataflow performance analysis Ousterhout et al. [Ous+15] employ *blocked time* analysis to dataflow, a "what-if" approach quantifying performance improvement assuming a resource is infinitely fast. They instrument Spark workers to measure the time a thread is blocked on network or disk, and use this to

diagnose stragglers and simulate the potential improvement from optimizing I/O usage. Blocked time analysis is performed offline and assumes staged batch execution. It can only identify bottlenecks due to network and disk i/o and does not provide insights into the interdependence of parallel tasks and operators. Bedini et al. [Bed+13] preset an alternative approach based on the *actor model* rather than CPA for *Storm* [Tos+14]. *HiTune* [Dai+11] and *Theia* [Gar+12] focus on Hadoop profiling; in particular, on cluster resource utilization and task progress monitoring.

Distributed systems profiling A comprehensive overview of prior work in distributed profiling is presented by Zhao et al. [Zha+16], which also introduces *Stitch*, a tool for profiling multi-level software stacks using traces. Stitch requires no domain knowledge of the reference system, but its *flow reconstruction principle* assumes logged events are sufficient to reconstruct the execution flow. *VScope* [Wan+12] targets online anomaly detection and root-cause analysis in large clusters and is designed to take advantage of the existing logging pipelines in data centers. Finally, we note that capturing dependencies between activities in dataflows is similar to *causal profiling* in *Coz* [CB15], which does not focus on distributed dataflows. *Coz* determines dependencies non-intrusively in existing binaries without instrumentation using repeated executions in a sandboxed environment.

2.6 Related work: Scaling controllers for distributed stream processors

A scaling controller makes two kinds of decisions. First, it detects symptoms of over- or under-provisioning (for example backpressure) and decides whether to make a change. Detection is often straightforward and addressed by conventional monitoring tools. Second, the controller must identify the causes of symptoms (e.g., a bottlenecked or idle operator) and propose a scaling action. The second decision is challenging, involving performance analysis and prediction. Streaming systems supporting a form of automatic dynamic scaling (e.g., *Google Cloud Dataflow* [KD16; Aki+15], *Heron* [Kul+15; Flo+17], *Pravega* [Des17], *Spark Streaming* [Zah+13], and *IBM System S* [Ged+14]) and research prototypes (e.g., *SEEP* [Fer+13] and *StreamCloud* [Gul+12]) focus

on the first decision and either ignore the second or provide speculative, often ad-hoc solutions for it.

A good scaling controller should provide the SASO properties [Hel+04] familiar from control theory: *stability* (not oscillating between different configurations), *accuracy* (finding the optimal configuration for the given workload), short *settling times* to reach the optimal configuration, and no *overshoot*.

Speculative scaling decisions that do not provide these properties can lead to unstable configurations for streaming systems. They lead to temporary over- or under-provisioning, and the resulting sub-optimal resource utilization incurs unnecessary costs. Oscillations can in turn degrade performance due to frequent scaling actions. Speculative scaling can be slow to converge, resulting in service level objective (SLO) violations or load shedding. Instead, a scaling controller should use a non-speculative approach to fulfill the SASO properties.

Designing a scaling controller with SASO properties is non-trivial, and existing dynamic scaling techniques for stream processing do not achieve SASO. Here, we summarize existing approaches, and then examine why they frequently lead to inaccurate, unstable, and slow scaling decisions.

Many stream processors [Zah+13; Car+17; Tos+14; Kul+15; Aki+13; WT15] have elastic runtimes and allow job reconfiguration by migrating or externalizing state, but the majority relies entirely on manual intervention for both symptom detection and scaling actions.

Table 2.1 summarizes those systems that do provide some form of automatic scaling. For more details also see the work by Assunção, Veith, and Buyya [AVB17]. We categorize them by the *metrics* used for symptom detection, the *policy* logic for deciding when to scale, the type of *scaling action* which defines which operators to scale and by how much, and their optimization *objective* (i.e. a latency or throughput SLO). The metrics column shows different varieties of metrics the systems use to base their decisions on. We distinguish between *observed rates*, which are simple aggregated rates, and *true rates*, which can be correlated with other metrics. Both are explained in more detail in Section 4.1. DS2, in the last row of the table, is our solution.

We identify two areas in which current systems fall short of the controller properties we would like to see: first, the metrics used do not provide enough information to make fast and accurate decisions as to how to rescale the system, and second, the policies used for scaling (and the models they are based on) are often simplistic and rule-based.

Table 2.1: Overview of automatic scaling policies in distributed dataflow systems.

System	Metrics	Policy	Scaling Action	Objective
Borealis [Aba+05]	CPU, network slack, queue sizes	Rule-based	Load shedding	Latency, throughput
StreamCloud [Gul+12]	Average CPU, observed rates	Threshold-based	Speculative, multi-operator	Throughput
SEEP [Fer+13]	User/system CPU time	Threshold-based	Speculative, single-operator	Latency, throughput
IBM Streams [Ged+14]	Congestion, observed rates	Threshold-based, blacklisting	Speculative, single-operator	Throughput
FUGU+ [Hei+14a]	CPU, processing time	Threshold-based	Speculative, single-operator	Latency
Nephele [LJK15]	Mean task latency, service time, interarrival time, channel latency	Queuing theory model	Predictive, multi-operator	Latency
DRS [Fu+17]	Service time, interarrival time	Queuing theory model	Predictive, multi-operator	Latency
Stela [XPG16]	Observed rates	Threshold-based	Speculative, single-operator	Throughput
Spark Streaming [SparkDy]	Pending tasks	Threshold-based	Speculative, multi-operator	Throughput
Google Dataflow [AD16]	CPU, backlog, observed rates	Heuristics	Speculative, multi-operator	Latency, throughput
Dhalion [Flo+17]	Backpressure, queue sizes, observed rates	Rule-based, blacklisting	Speculative, single-operator	Throughput
Pravega [Des17]	Observed rates	Rule-based	Speculative, single-operator	Throughput
DS2 (Chapter 4)	True processing and output rates	Dataflow model	Predictive, multi-operator	Throughput

Limited metrics Most systems rely on coarse-grained *externally observed* metrics to detect suboptimal scaling. These metrics include CPU utilization, throughput measurements, queue sizes, and others. CPU and memory utilization can be inadequate metrics for streaming applications, particularly in cloud environments due to multi-tenancy and performance interference [Ram+16]. *StreamCloud* [Gul+12] and *SEEP* [Fer+13] try to mitigate the problem by separating user time and system time, but preemption can make these metrics misleading: high CPU usage by a task running on the same physical machine as a dataflow operator can trigger incorrect scale-ups (false positives) or prevent correct scale-downs (false negatives), for example. *Google Cloud Dataflow* [KD16] uses CPU utilization only for scale-down decisions but could still suffer from false negatives. The CPU utilization is also unsuitable for systems like Timely Dataflow [McS; Mur+13], which continuously check for more work.

These metrics also imply continuous threshold tuning, a cumbersome and error-prone process. Incorrect scaling decisions can often arise from slightly misconfigured thresholds, even on fine-grained metrics [Flo+17]. *Dhalion* [Flo+17] and *IBM System S* [Ged+14] also use backpressure and congestion to identify bottlenecks. These signals are only helpful where a bottleneck exists. If the dataflow is using resources unnecessarily, such metrics will not trigger reconfigurations. Moreover, in under-provisioned dataflows, backpressure will only detect a single bottleneck; for this reason and to minimize the effects of incorrect decisions [Sch+09; Flo+17], each scaling action only configures one operator, which increases convergence time.

Simplistic performance models The scaling policy is generally expressed in simple rules, using predefined thresholds and conditions, e.g., *CPU utilization* $>$ *80% and backpressure* $\implies$ *scale up*. This results in simplistic performance models with poor predictive accuracy, which are unable to consider the structure of the dataflow graph or computational dependencies among operators. We note the exceptions of recent work by Lohrmann, Janacik, and Kao [LJK15] based on *Nephele* [WK09] and *DRS* [Fu+17], which use queuing theory models. Nevertheless, both systems show poor prediction quality in some cases, while Nephele also seems to suffer from temporary over-provisioning and slow convergence.

Since the controller cannot accurately estimate how much to scale an operator, scaling actions are mostly *speculative*. The system applies pessimistic

strategies which introduce only small changes to the number of provisioned resources [Fer+13; Ged+14] and most policies configure a single operator at a time. This delays convergence to a steady state significantly, as all steps of the scaling process are repeated many times: SLO monitoring, decision making, state migration, and redeployment. Floratou et al. [Flo+17] show that Heron needs almost an hour to reach a steady state that can handle the input rate after the first time backpressure is observed.

More aggressive strategies try to apply configurations and blacklist them if they degrade performance. Schneider et al. [Sch+09] describe an approach that allows arbitrary scaling steps but requires a user-defined function to calculate the new number of instances, whereas Spark supports exponential increases in resources [SparkDy]. *StreamCloud* [Gul+12] tries to estimate the optimal number of virtual machines in a single step, but uses very coarse-grained scaling on a subgraph of the dataflow topology. Google Cloud Dataflow is the only system we know with fully automatic scaling per operator, although details of the model used have not been disclosed.

2.7 Related work: Applying configuration updates

Distributed stream processing jobs are long-running dataflows that continually ingest data from sources with dynamic rates and must produce timely results under variable workload conditions [PokGO; Twi].

To satisfy latency and availability requirements, modern stream processors support *consistent online reconfiguration*, in which they update parts of a dataflow computation without disrupting its execution or affecting its correctness. Such reconfiguration is required during **rescaling** to handle increased input rates or reduce operational costs [Fer+13; Flo+17], to provide **performance isolation** across different dataflows by dynamically scheduling queries to available workers, to allow **code updates** to fix bugs or improve business logic [Arm+18; Car+17], and to enable **runtime optimizations** like execution plan switching [ZRH04] and straggler and skew mitigation [Fan+17].

Streaming dataflow operators are often stateful, partitioned across workers by key, and their reconfiguration requires *state migration*: intermediate results and data structures must be moved from one set of workers to another, often across a network. Existing state migration mechanisms for stream processors either pause and resume parts of the dataflow (as in Flink [Car+15], Dhalion [Flo+17], and SEEP [Fer+13]) or launch new dataflows alongside the old configuration

(as for example in ChronoStream [WT15] and Gloss [Raj+18]). In both cases state moves "all-at-once", with either high latency or resource usage during the migration.

The topic of state migration has been extensively studied for distributed databases [Bar+12; Das+11; Elm+11; Elm+15]. Notably, Squall [Elm+15] uses transactions to multiplex fine-grained state migration with data processing. These techniques are appealing in spirit, but use mechanisms (transactions, locking) not available in high-throughput stream processors and are not directly applicable without significant performance degradation.

A distributed dataflow computation runs as a physical execution plan which maps operators to provisioned compute resources (or workers). The execution plan is a directed graph whose vertices are operator instances (each on a specific worker) and edges are data channels (within and across workers). Operators can be stateless (e.g., filter, map) or stateful (e.g., windows, rolling aggregates). State is commonly partitioned by key across operator instances so that computations can be executed in a data-parallel manner. At each point in time of a computation, each worker (with its associated operator instances) is responsible for a set of keys and their associated state.

State migration is the process of changing the assignment of keys to workers and redistributing the respective state accordingly. A good state migration technique should be *non-disruptive* (minimal increase in response latency during migration), *short-lived* (migration completes within a short period of time), and *resource-efficient* (minimal additional resources required during the migration).

We present an overview of existing state migration strategies in distributed streaming systems and identify their limitations. We then review live state migration methods adopted by database systems and provide an introduction into Megaphone's approach to bring such migration techniques to streaming dataflows, which we present in Chapter 5.

2.7.1 State migration in streaming systems

Distributed stream processors, including research prototypes and production-ready systems, use one of the following three state migration strategies.

Stop-and-restart A straightforward way to realize state migration is to temporarily stop program execution, safely transfer state when no computation

is being performed, and restart the job once state redistribution is complete. This approach is most commonly enabled by leveraging existing fault-tolerance mechanisms in the system, such as global state checkpoints. It is adopted by Spark Streaming with *Discretized streams* [Zah+13], *Structured Streaming* [Arm+18], and *Apache Flink* [Car+17].

Partial pause-and-resume In many reconfiguration scenarios only one or a small number of operators need to migrate state, and halting the entire dataflow is usually unnecessary. An optimization first introduced in *Flux* [Sha+02] and later adopted in variations by *SEEP* [Fer+13], *IBM Streams* [IBMSt], *StreamCloud* [Gul+12], *Chi* [Mai+18], and *FUGU* [Hei+14b], pauses the computation only for the affected dataflow subgraph. Operators not participating in the migration continue without interruption. This approach can use fault-tolerance checkpoints for state migration [Fer+13; Mai+18] or state can be directly migrated between operators [Gul+12; Hei+14b].

Dataflow Replication To minimize performance penalties, some systems replicate the whole dataflow or subgraphs of it and execute the old and new configurations in parallel until migration is complete. *ChronoStream* [WT15] concurrently executes two or more computation slices and can migrate an arbitrary set of keys between instances of a single dataflow operator. *Gloss* [Raj+18] follows a similar approach and gathers operator state during a migration in a centralized controller using an asynchronous protocol.

Current systems fall short of implementing state migration in a non-disruptive and cost-efficient manner. Existing stream processors migrate state *all-at-once*, but differ in whether they pause the existing computation or start a concurrent computation. Strategies that pause the computation can cause high latency spikes, especially when the state to be moved is large. On the other hand, dataflow replication techniques reduce the interruption, but at the cost of high resource requirements and required support for input duplication and output de-duplication. For example, for ChronoStream to move from a configuration with x instances to a new one with y instances, $x + y$ instances are required during the migration.

2.7.2 Live migration in database systems

Database systems have implemented optimizations that explicitly target limitations we have identified in the previous section, namely unavailability and resource requirements. Even though streaming dataflow systems differ significantly from databases in terms of data organization, workload characteristics, latency requirements, and runtime execution, the fundamental challenges of state migration are common in both setups.

Albatross [Das+11] adopts virtual machine live migration techniques and is further optimized by Barker et al. [Bar+12] with a dynamic throttling mechanism, which adapts the data transfer rate during migration so that tenants in the source node can always meet their SLOs. *ProRea* [SCM13] combines push-based migration of hot pages with pull-based migration of cold pages. *Zephyr* [Elm+11] proposes a technique for live migration in shared-nothing transactional databases which introduces no system downtime and does not disrupt service for non-migrating tenants.

The most sophisticated approach for relational databases is *Squall* [Elm+15], which interleaves state migration with transaction processing by partly using transaction mechanisms to effect the migration. Squall introduces a number of interesting optimizations, such as pre-fetching and splitting reconfigurations to avoid contention on a single partition. In the course of a migration, if migrating records are needed for processing but not yet available, Squall introduces a transaction to acquire the records (completing their migration). This introduces latency along the critical path, and the transaction locking mechanisms can impede throughput, but the system is neither paused nor replicated.

To the best of our knowledge, no stream processor implements such a fine-grained migration technique.

2.7.3 Live migration for streaming dataflows

Next, we present our solutions to the problems outlined previously. In the following chapter, we introduce Snailtrail, a system to analyze distributed systems for latency problems. In Chapter 4, we present DS2, a scaling controller that automatically and accurately determines the required level of parallelism per dataflow to meet its SLOs. Lastly, in Chapter 5 we show how fine-grained configuration updates can be interleaved with query processing to reduce the latency impact of reconfigurations by orders of magnitude, using Megaphone.

3
Snailtrail: online critical path analysis for distributed dataflows

This chapter is based on the paper SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows *[Hof+18].*

Modern stream processors provide a complex execution environment for distributed dataflows. They scale dataflows across many nodes and schedule operators based on data availability and time constraints. The underlying execution environment, namely the operating system and networks, add complexity which can manifest in hard-to-understand runtime behavior. Understanding the runtime behavior is necessary, though, to achieve low-latency query processing. To address the problem of system understandability we present a generalization of critical path analysis (CPA) to online performance characterization of long-running, distributed dataflow computations.

Existing tools which aggregate performance information from servers and software components into visual analysis and statistics [Nag; Sac+03] can be useful in showing what each part of the system is doing at any point in time, but are less helpful in explaining which components in a complex distributed system need improvement to reduce end-to-end latency. On the other hand, tools which capture detailed individual traces through a system, such as *Splunk* [Car12] and *VMware vRealize Log Insight* [LogI], can isolate specific instances of performance loss, but lack a "big picture" view of what really matters to performance over a long (possibly continuous) computation on a varying workload.

In this chapter we show that the design space for useful performance analysis of so-called "big data" systems is much richer than currently available tools would suggest.

Critical path analysis is a proven technique for gaining insight into the performance of a set of interacting processes [YM88], and we review the basic idea in Section 3.1. However, CPA is not directly applicable to long-running and streaming computations for two reasons. Firstly, it requires a complete execution trace to exist before analysis can start. In modern systems, such a trace may be very large or, in the case of stream processing, unbounded. Secondly, in a continuous computation, there exist many critical paths (as we show later on), which also change over time, and there is no established methodology for choosing one of them to gain an overall understanding of the system. It is therefore important to aggregate the paths both spatially across the distributed computation and temporally for an evolving picture of the system's performance.

According to prior work [ARH94; YDG89], the accuracy of CPA increases with the number of critical paths considered. However, existing approaches require full path materialization in order to aggregate information from multiple critical paths. Thus, they restrict analysis to k critical paths, where k is much smaller than the total number of paths in the trace. In open-ended computations where analysis is performed on trace snapshots and all paths are of equal length, materializing all paths is impractical, especially if the analysis needs to keep up with real time. For instance, in our experiments, the number of paths in a 10 second snapshot of Spark traces is in the order of 10^{21}.

This chapter's first contributions in Section 3.2 are definitions of the *transient critical path*, a modification of the classical critical path applicable to continuous unbounded computations, and *critical participation (CP)*, a metric which captures the importance of an execution activity in the transient critical paths of computation, and which can be used to generate new time-varying performance summaries. The CP metric can be computed *online* and aggregates information from *all* paths in a snapshot without the need to materialize any path.

Our next contribution in Section 3.3 is a model for the execution of distributed dataflow programs sufficiently general to capture the execution (and logging) of commonly-used systems—Spark, Flink, TensorFlow, and Timely Dataflow—and detailed enough for us to define transient critical paths and CP over each of these.

We then show in Section 3.5 an algorithm to compute CP online, in real time, and describe Snailtrail, a system built (itself as a Timely Dataflow program) to do this on traces from the four dataflow systems listed above. In Section 3.7 we evaluate Snailtrail's performance, demonstrate online critical path analysis using all four reference systems with a variety of applications and

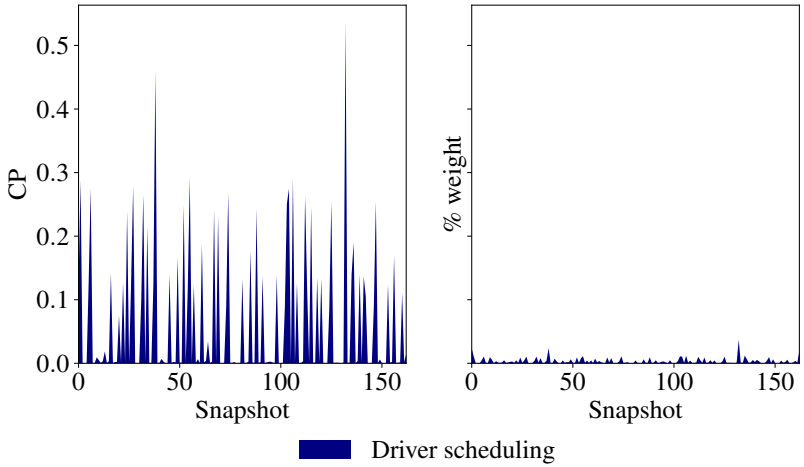


Figure 3.1: CP-based (left) and conventional profiling (right) summaries of Spark's driver activity on BDB [bdb] from [Ous] for 64 s snapshots. Spikes indicate coordination between workers and the driver. Conventional profiling fails to show that Spark's driver is a latency bottleneck whereas Snailtrail with its CP metric clearly indicates a latency problem.

workloads, and show how CP is more informative than existing methods, such as conventional profiling and single-path critical path analysis (Sections 3.7.4 and 3.7.5).

Figure 3.1 gives a flavor of how CP compares with conventional profiling techniques. In Spark, a driver schedules the processing of batches of data to individual workers. Only after all workers finished processing their assigned batch, the driver will schedule a new round of work. This can cause high latency if there is imbalance between the size and processing time of batches or if the driver is slow determining a new set of batches to be processed. A critical path in Spark could, for example, go through the driver's scheduling activity and the slowest of the workers in each round. The key difference in the plot is that our approach highlights activities that contribute significantly to the performance of the system, while discarding processing time that lies outside the critical path.

Table 3.1: Notation used throughout Chapter 3

Symbol	Description
$a : [\text{start}, \text{end}]$	Activity a with start and end timestamps
G	Activity graph
$G_{[t_s,t_e]}$	Snapshot of activity graph G in the time interval $[t_s, t_e]$
$\prod_{t_s}^{t_e}(e)$	Projection of edge e on the time interval $[t_s, t_e]$
$v[w]$	Worker id of vertex v
$v[t]$	Timestamp t of vertex v
$e[w]$	Weight w of edge e
$e[p]$	Type p of edge e
$\|\|\mathbf{p}\|\|$	Total weight of edges in path $\mathbf{p}$
E_w	Set of worker activities
E_c	Set of communication activities
$c(e)$	Transient path centrality of edge e
CP_e	Critical participation of edge e

We believe Snailtrail is the first system for online real-time critical path analysis of long-running and streaming computations. We also show in Section 3.7 the potential of such analysis for configuration tuning, straggler mitigation, and online detection of communication imbalance.

3.1 Critical path analysis background

Critical path analysis (CPA) has been successfully applied to high-performance parallel applications like MPI programs [CC15; Sch05], and the basic concepts also apply to the distributed dataflow systems we target in this chapter. In this section we review classical CPA applied to batch computations as a prelude to our extension of CPA to online and continuous computations in the next section. Table 3.1 summarizes the notation we use in this section and the rest of this chapter.

We view distributed computation as executed by individual system *workers* that perform *activities* (e.g., data transformations or communication). The *critical path* is defined as the sequence of activities with the longest duration throughout the execution.

Definition 5 (Activity)**.** *A logical operation performed at any level of the software stack, and associated with two timestamps* $[start, end]$*,* $start \leq end$*, that denote the start and end of its execution with respect to a clock* C*.*

An activity can be either an operation performed by a worker (*worker activity*) or a message transfer between two workers (*communication activity*). Typically, worker activities correspond to the execution of some code, but can also be I/O operations performed by the worker (e.g., reads/writes to/from disk). Communication activities correspond to worker interactions, e.g., data exchange or control information.

Different systems have different abstractions corresponding to workers, for example threads, processes and virtual machines. For consistency, we define a worker as follows.

Definition 6 (Worker)**.** *A logical execution unit that performs an ordered sequence of activities with respect to a clock* C*.*

We require that no two activities a_i and a_j of the same worker $a_i : [\text{start}_i, \text{end}_i]$ and $a_j : [\text{start}_j, \text{end}_j]$ (where $i \neq j$) can overlap in time, i.e. either $\text{end}_i \leq \text{start}_j$ or $\text{start}_i \geq \text{end}_j$. In other words, a single worker can only execute one activity at a time but is not able to execute activities in parallel.

Central to CPA is the program activity graph (PAG), which describes what workers are doing. It is a graph representation of workers, the activities they execute and communication between workers.

Definition 7 (Program activity graph)**.** *A PAG* $G = (V, E)$ *is a connected directed labeled acyclic graph.*

- V *is the set of vertices. A vertex* $v \in V$ *represents an event corresponding to the start or end of an activity. Each vertex* v *has a* timestamp $v[t]$ *and a* worker id $v[w]$*.*
- $E \equiv E_w \cup E_c \subset V \times V$*,* $E_w \cap E_c = \emptyset$*, is the set of directed edges representing activities and communication between workers. An edge* $e = (v_i, v_j) \in E$ *represents an activity* $a : [start, end]$*, where* $v_i[t] = start$ *and* $v_j[t] = end$*. An edge* e *has a type* $e[p]$ *and a weight* $e[w]$ *indicating the activity duration in time units, so that* $e[w] = v_j[t] - v_i[t] = end - start \geq 0$*.*
- *An edge* $e \in E_w$ *denotes a worker activity whereas an edge* $e \in E_c$ *denotes a communication activity.*

The weight of a path in the PAG is the sum of the weights of the activities on the path.

Definition 8 (Path weight)**.** *For a path* $\mathbf{p}$*, the* weight $\|\mathbf{p}\|$ *is the sum of the weight of the components:*

$$\|\mathbf{p}\| = \sum_{e \in \mathbf{p}} e[w].$$

The direction of an edge $e = (v_1, v_2) \in E$ from node $v_1 \in V$ to node $v_2 \in V$ denotes a *happened-before* relationship between the nodes [Lam78]. The critical path is then defined as the *longest path* in the program activity graph.

Definition 9 (Critical path)**.** *For a program activity graph* $G = (V, E)$*, the critical path is a path* $\mathbf{p} \in G$ *such that* $\nexists \mathbf{p}' \in G : \|\mathbf{p}'\| > \|\mathbf{p}\|$.

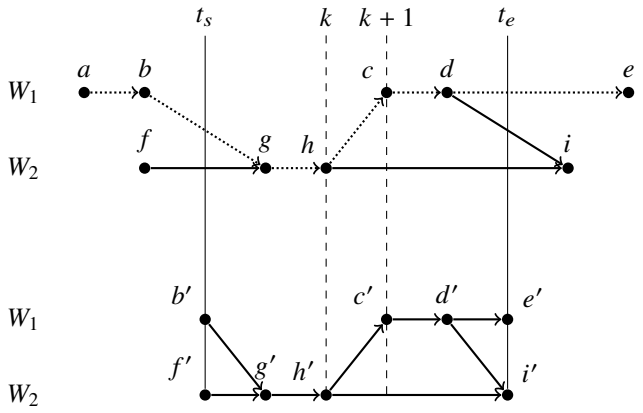
The number of critical paths in a PAG from a batch computation tends to be small as all critical paths have to have maximal length. Due to this they might only cover a small part of the activity graph.

3.2 Online critical path analysis

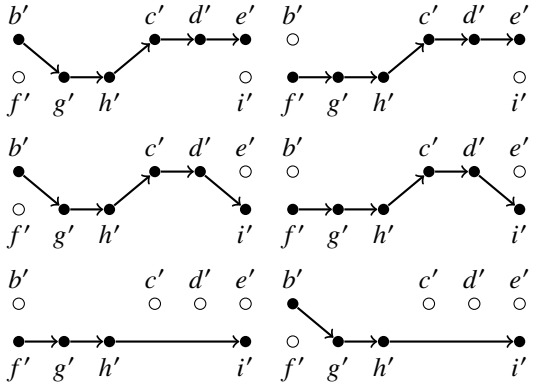
Offline processing in traditional CPA is not feasible for long-running or continuous computations like streaming applications or machine learning model training. In these cases, neither the program activity graph nor the critical path can be defined as in Section 3.1. Instead, we define online CPA on PAG *snapshots*, performing it on user-defined *time windows*: slices of the PAG that contain activities within a specified time interval. This enables not only performance analysis of running applications, but also targeting specific parts of the computation like the model training phase in a TensorFlow program or a specific time window in a Flink stream. To achieve this, we show here how to define a time-based program activity graph snapshot and a *transient critical path* on this graph. We then define the *critical participation* performance metric, and we provide the intuition behind it in Section 3.2.3.

3.2.1 Transient critical paths

To retrieve a snapshot of the PAG, we first assign activities to time windows. For each edge in a graph, we call its corresponding edge in a snapshot an *edge projection*. The edge projection ensures that all activities fit the snapshot's time window.



(a) Program activity timelines of a distributed execution with two workers. The vertical lines divide the timeline into intervals of one time unit. The critical path is highlighted (⋯›) in the top timeline. The bottom timeline shows the PAG snapshot into the time interval $[t_s, t_e]$.



(b) All transient critical paths for the graph snapshot of (a).

Figure 3.2: A program activity graph, its snapshot in the interval $[t_s, t_e]$, and its transient critical paths.

Definition 10 (Edge projection)**.** *Let $e = (v_i, v_j)$ be an edge of an activity graph $G = (V, E)$, where $e \in E$ and $v_i, v_j \in V$. Let also $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a clock C. Let u_s be a copy of v_i with $u_s[t] = t_s$ and u_e a copy of v_j with $u_e[t] = t_e$. The projection of e on $[t_s, t_e]$ is an edge of the same type as e and is defined only whenever $\left[v_i[t], v_j[t]\right]$ overlaps with $[t_s, t_e]$ as follows:*

$$\prod_{t_s}^{t_e}(e) = \left(\arg\max_{[t]}(v_i, u_s),\ \arg\min_{[t]}(v_j, u_e) \right)$$

Activities entirely within the time interval $[t_s, t_e]$ are unchanged by the projection, whereas activities that straddle the boundaries are truncated to fit the interval. We can now define a snapshot as follows.

Definition 11 (PAG snapshot)**.** *Let $G = (V, E)$ be a PAG, and $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a clock C. The snapshot of G in $[t_s, t_e]$ is a directed labeled acyclic graph $G_{[t_s,t_e]} = (V', E')$ that is constructed by projecting all edges of G on $[t_s, t_e]$.*

The snapshot $G_{[t_s,t_e]}$ is that part of the PAG which can be observed in the time window $[t_s, t_e]$. Figure 3.2a shows this applied to the activity timelines of two worker threads, w_1 and w_2, with time flowing left to right. The complete PAG is shown at the top with the critical path in red. Below is the projection of the PAG into the interval $[t_s, t_e]$. The activities straddling the window, e.g., $\prod_{t_s}^{t_e}((b, g)) = (b', g')$, are projected to fit in the snapshot.

The key observation is that we cannot define a single critical path in a PAG snapshot since there exist *multiple* longest paths with the same total weight: $t_e - t_s$. All paths starting at t_s and ending at t_e are *potentially* parts of the evolving global critical path. For this reason, we define the notion of *transient critical path*:

Definition 12 (Transient critical path)**.** *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in the time interval $[t_s, t_e]$. We define the set of paths $\mathcal{P}$ on $G_{[t_s,t_e]}$ as*

$$\mathcal{P} \equiv \left\{ \mathbf{p} \subseteq E \mid \nexists \mathbf{p}' : \|\mathbf{p}'\| > \|\mathbf{p}\| \right\},$$

where $\mathbf{p}$ denotes a path in $G_{[t_s,t_e]}$, and $\|\mathbf{p}\|$ denotes the total weight of all edges in $\mathbf{p}$. Any path $\mathbf{p} \in \mathcal{P}$ is a transient critical path of the activity graph G in the time interval $[t_s, t_e]$.

Figure 3.2b shows all six transient critical paths for the snapshot in Figure 3.2a. Since each could potentially participate in the evolving global critical path, we need a metric that can aggregate information from all paths and rank activities according to their impact on the computation's performance. In offline CPA such a ranking is trivial because there is only one critical path for the entire computation.

Since all transient paths can potentially be part of the evolving global critical path, an activity that appears on many transient paths is more likely to be critical and should be ranked high. In Figure 3.2b, the edge (d', i') appears in two paths, while the edge (g', h') belongs to all six critical paths. The performance metric we define next incorporates this information and ranks activities based on their potential contribution to the global critical path.

3.2.2 Critical participation (CP metric)

For an activity e on a critical path $\mathbf{p}$, we define its *participation* q_e as the ratio of its duration $e[w]$ and the total length $\|\mathbf{p}\|$ of the critical path $\mathbf{p}$.

$$q_e = \frac{e[w]}{\|\mathbf{p}\|}, \qquad 0 \leq q_e \leq 1 \tag{3.1}$$

The participation can be computed for all activities in a single pass over $\mathbf{p}$.

We correspondingly define the average CP of an activity e in a transient critical path as the sum of its participation across all transient critical paths, normalized with the number of transient critical paths.

$$\mathrm{CP}_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N}, \qquad 0 \leq \mathrm{CP}_e \leq 1 \tag{3.2}$$

where q_e^i is the participation of e to the i-th transient critical path, as given by Equation (3.1), and N is the total number of transient critical paths in the graph snapshot.

A straightforward way to compute CP_e is to materialize all N transient paths and compute the participation of each activity in every path. However, path materialization is not viable in an online setting because a single graph snapshot might contain too many paths to maintain. Instead, we exploit the fact that the CP of an activity actually depends on the total number of transient paths this activity belongs to. Hence, we define the *transient path centrality* as follows.

Definition 13 (Transient path centrality)**.** *Let $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N\}$ be the set of N transient paths of snapshot $G_{[t_s,t_e]}$ with weight $\|\mathbf{p}\| = t_e - t_s$. The transient path centrality of an edge $e \in G_{[t_s,t_e]}$ is defined as*

$$c(e) = \sum_{i=1}^{N} c_i(e), \qquad c_i(e) = \begin{cases} 0 \colon e \notin \mathbf{p}_i \\ 1 \colon e \in \mathbf{p}_i. \end{cases}$$

Both Equation (3.2) and Definition 13 define the CP of an activity and the following holds.

$$\mathrm{CP}_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{c(e)}{N} \cdot \frac{e[w]}{\|\mathbf{p}\|} \tag{3.3}$$

Proof. Recall that e is an activity edge in the PAG snapshot, N is the total number of transient critical paths in the snapshot, q_e^i is ratio of the activity's duration to the total duration of the i^{th} transient critical path (the ratio is 0 if the activity edge is not part of the i^{th} path), $0 \leq c(e) \leq N$ is the number of transient critical paths the activity e belongs to, $e[w]$ is the weight of the activity e, i.e. its duration, and $[t_s, t_e]$ is the snapshot window size.

Without loss of generality, we assume that the transient critical paths $\mathbf{p}_i$ the activity edge e belongs to are numbered from $i = 1$ to $i = c(e)$. Then:

$$\mathrm{CP}_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{\sum_{i=1}^{i=N} \frac{e[w]}{\|\mathbf{p_i}\|}}{N} = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\mathbf{p_i}\|}}{N} + 0 = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\mathbf{p_i}\|}}{N}$$

All transient critical paths in the snapshot have the same duration $\|\mathbf{p_i}\|$, which is equal to the duration of the snapshot $t_e - t_s$. Hence:

$$\mathrm{CP}_e = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} = \frac{c(e) \cdot e[w]}{N \cdot (t_e - t_s)}$$

□

Equation (3.3) indicates that the computation of CP_e can be reduced to the computation of $c(e)$, which requires no path materialization and can be performed in parallel for all edges in $G_{[t_s,t_e]}$. Section 3.5 provides an algorithm for transient path centrality and CP without materialization. Note that we can normalize by the number of paths N and their length $\|\mathbf{p}\|$ because Definition 12 guarantees that all paths have the same length.

We can now compute the transient path centrality and critical participation for the example in Figure 3.2. For instance, $c(d',i') = 2$ and $c(g',h') = 6$. Respectively, since $t_e - t_s = 5$ and $N = 6$, $CP_{(d',i')} = 0.066$ and $CP_{(g',h')} = 0.2$.

The CP of Equation (3.2) can be generalized for activities of a specific type c. Aggregating the CP by different activity types is key to produce meaningful performance summaries. This is explained in more detail in Section 3.6.

$$\sum_{\forall e\colon e[p]=c} \mathrm{CP}_e \tag{3.4}$$

Again, the sum over all activities and activity types is 1.

$$\sum_{\forall c \in G} \sum_{\forall e\colon e[p]=c} \mathrm{CP}_e = 1 \tag{3.5}$$

Intuitively, Equation (3.5) states that the estimated contribution of an activity type, e.g., serialization, to the critical path of the computation is *normalized* over the contribution of all other activity types in the same snapshot.

Proof. Recall that c denotes an activity type, e.g., serialization, and $e[p]$ is the type of the activity edge e in the snapshot $G_{[t_s,t_e]}$. We have:

$$\begin{aligned}
\sum_{\forall c \in G} \sum_{\forall e\colon e[p]=c} \mathrm{CP}_e &= \sum_{\forall e \in G} \mathrm{CP}_e = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=N} q_e^i}{N} \\
&= \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\mathbf{p_i}\|} + 0}{N} = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\mathbf{p_i}\|}}{N} \\
&= \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} \\
&= \frac{N \cdot (t_e - t_s)}{N \cdot (t_e - t_s)} = 1
\end{aligned}$$

This holds since $\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]$ denotes the sum of the weights (durations) of all activity edges that comprise all N transient critical paths in the snapshot, which is equal to $N \cdot (t_e - t_s)$. □

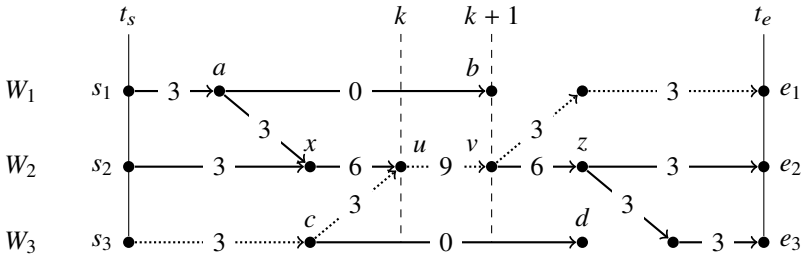


Figure 3.3: A program activity graph snapshot with three workers from time t_s to t_e. The dashed vertical lines demonstrate an interval of one time unit. A randomly chosen critical path is highlighted with dotted edges (⋯→). Edge annotations correspond to the transient path centrality (see Definition 13).

3.2.3 Comparison with existing methods

Figure 3.3 illustrates by example a comparison of CP-based performance analysis with conventional profiling and traditional critical path analysis.

Conventional profiling summaries aggregate the activity duration by type or by worker timeline. Such summaries provide information on how much time a program spends on a certain activity type (e.g., serialization) or a worker spends executing an activity type as compared to other workers. Since conventional profiling summaries rely solely on activity duration and do not capture execution dependencies, they cannot reveal bottlenecks and execution barriers. Conventional profiling in the execution of Figure 3.3 would rank activities (a, b) and (c, d) high since they both have a duration of 3 time units, larger than all other activities. However, optimizing those activities cannot produce a performance benefit for the parallel computation as both activities are followed by a waiting state (denoted with a gap).

On the other hand, CPA captures execution dependencies and can accurately pinpoint activities that influence performance. However, traditional CPA is not directly applicable in a continuous computation as the critical path is not known by just inspecting a snapshot of the execution traces. In a snapshot like the one of Figure 3.3, all paths starting at s_i and finishing at e_i have equal length in time units, thus traditional CPA would choose one of them at random. We have highlighted such a path in Figure 3.3. Although this randomly selected

path does not contain the activities (a, b) and (c, d), whose optimization would certainly not improve the latency of the computation, it misses several important activities, such as (x, u) and (v, z), whose optimization would do so.

The CP metric overcomes the limitations of both conventional profiling and traditional CPA by ranking activities based on their potential contribution to the evolving critical path of the computation, which in turn reflects potential benefits from optimization.

Given a snapshot and no knowledge of the execution timelines outside of it, any path between the s_i and e_i points in Figure 3.3 is *equally probable* to be part of the critical path. CP is a *fairer* metric compared to existing methods in that it aggregates an activity's contribution over *all* transient critical paths and normalizes by the number of paths and the activity's duration. The more paths an activity contributes to, the higher the probability it is a part of the evolving critical path and, hence, the higher its CP metric is. In Figure 3.3, activities (a, b) and (c, d) do not contribute to any path and thus have zero transient path centrality and CP values. On the other hand, activities (x, u), (u, v), and (v, z) will be ranked as top-three by CP, since they participate in six, nine, and six transient critical paths respectively.

In Section 3.7.4 we empirically compare CP-based performance summaries to conventional profiling and traditional CPA, and demonstrate how the results of the latter can be misleading. Further, in Section 3.7.5, we show how CP can detect and help optimize execution bottlenecks like the one represented by activity (u, v) in Figure 3.3, which is executed by a single worker (W_2) and belongs to all nine transient critical paths.

3.3 Applicability to dataflow systems

Here we show the applicability of Snailtrail to a range of modern dataflow systems. Additionally we provide details on the model assumptions and the instrumentation requirements.

Spark, Flink, TensorFlow, and Timely Dataflow are superficially different, but actually similar with regard to CPA: they all execute dataflow programs expressed as directed graphs whose vertices are operators (e.g., map, reduce) and whose edges denote data dependencies. During runtime, a physical dataflow graph is executed by one or more workers, which can be threads or processes in a machine or a cluster. Each worker executes its assigned part of the physical graph and processes partitions of the input data in parallel with other workers.

3.3.1 Activity types

We define a small set of *activity types* we use to classify both the activity of a worker at any given point in time, and communication of data between workers and operators. We consider the following types of *worker* activities:

Data Processing The worker is processing data in an operator. Often the data has a unique identifier. We interpret low-level (de)compression operations as data processing.

Scheduling Deciding which operator a worker will execute. In Spark and Flink, scheduling is done by special workers (the Driver and the JobManager).

Barrier Processing The worker is processing information which coordinates the computation (e.g., distributed progress tracking in Timely Dataflow or watermarks in Flink).

Buffer Management The worker is managing buffers between operators (e.g., Flink's FIFO queues) or buffering data moving to/from disk (e.g., Spark). The activity may include copying data into/out of buffers, locking, recycling buffers (e.g., Flink) and dynamically allocating them (e.g., Timely Dataflow).

Serialization Data is converted to a serial format, an operation common to all dataflow systems when messages are sent between processes and machines, and require a different representation than in memory.

Waiting The worker is waiting on some message (data or control) from another worker, and is therefore either *spinning* (as in Timely Dataflow) or *blocked* on an RPC (as in TensorFlow). Waiting in our model is always a consequence of other, concurrent, activities, and so is a key element of critical path analysis: a worker does not produce anything *useful* while waiting, and so *waiting activities can never be on the critical path*. We will explain the construction of waiting activities in Section 3.4 in situations when the system cannot reliably determine them.

I/O The worker is waiting on an external (uninstrumented) system, for example Spark waiting for HDFS, or Flink spilling a large state to disk. I/O activities have no special meaning, but capture cases where performance of the reference system is limited by an external system.

Unknown Anything else: gaps in trace records and any worker activity not captured by the instrumentation. A large number of unknown activities usually indicates inadequate instrumentation.

In contrast to worker-local activities, interactions between workers are modeled as communication activities, which capture either *application data ex-*

change over communication channels, or *control messages* conveying metadata about worker state or progress. Control messages can be exchanged between pairs of workers as in Timely Dataflow, or through a master as in Spark and Flink.

3.3.2 Instrumenting specific systems

We applied our approach to Spark, TensorFlow, Flink, and Timely Dataflow, mapping each to our taxonomy of activities. In some cases we used existing instrumentation, whereas in others we added our own.

Timely Dataflow [Mur+13; McS] required us to add explicit instrumentation, and was the first system we addressed (in part because Snailtrail is written as a Timely Dataflow computation.) Timely Dataflow's progress tracking corresponds to our "barrier" activity, discrete (de-)serialization is performed on both data records and control messages, and Timely Dataflow's cooperative scheduling means that any otherwise unclassified worker activity corresponds to "scheduling".

Apache Flink [Car+15] adopts (unlike Timely Dataflow) a master-slave architecture for coordination. We treat Flink's `JobManager`, `TaskManager`s, and `Task`s all as workers, and Flink's runtime has clear activities corresponding to buffer management and serialization. Scheduling is performed in the `JobManager`, barrier processing corresponds to the watermark mechanism, and control messages correspond to communication between the `JobManager` and `TaskManager`s. Data message activities are non-trivial to identify in Flink, since data records are batched and exchanged between TaskManagers not Tasks. We added our own instrumentation to record send and receive events as occuring "end-to-end", i.e. when Tasks place messages into or remove messages from buffers. This does not accurately represent the performance of the underlying network, but does mean that data message edges reflect the time between a record being produced and consumed by workers.

TensorFlow [Aba+16] has its own instrumentation based on "Timeline" objects, which we reuse unchanged. While enough to generate meaningful results, it also shows how even a well-considered logging system can easily omit information vital for sophisticated performance analysis.

Spark [Zah+12] also has its native instrumentation which we use to model both the Spark driver and executors as workers. The logs provide information on the lineage of resilient distributed datasets (RDDs) facilitating construction of the PAG. Since executor scheduling is not instrumented, we assume greedily

that a task is started on the most recently used thread, which aligns with Spark's observed behavior.

Many systems provide an interface to publish and extract metrics information. Snailtrail requires a specific set of information to construct the PAG, and if the native instrumentation has sufficient detail, a simple translation of the data format will suffice. Otherwise, the same mechanism can be used to extract additional information by introducing additional trace points, or a new mechanism is required. From our experience, many systems aggregate their performance metrics, which is not compatible with Snailtrail's approach and requires modifications of the system-under-test.

3.3.3 Model assumptions

We support both synchronous and asynchronous execution in shared-nothing and shared-memory architectures. Most dataflow systems use asynchronous computations on shared-nothing clusters, but sometimes synchronous computation is supported (e.g., in TensorFlow), and system workers can share state (e.g., in Timely Dataflow). Specifically, our model is consistent with respect to critical path analysis under two assumptions.

Assumption 1 (Message-based interaction)**.** *Every interaction between operators in the dataflow must occur via message exchange, even if executed by the same worker.*

Note this assumption does *not* preclude shared-memory systems. Operators in the reference dataflow may share state as long as any modification to this state is appropriately instrumented to trigger a 'virtual' message exchange between the workers sharing that state. We use this approach in instrumenting shared state in Timely Dataflow, for example. From the perspective of Snailtrail, a message can serve both as data exchange and activation.

Assumption 2 (Waiting state termination)**.** *Every waiting activity in a worker's timeline is terminated by an incoming message, either from the same or a different worker.*

In other words, a worker in a waiting state cannot start performing activities unprompted without receiving a message. In the activity graph, a waiting edge's end node must correspond to that of a communication activity, i.e. a receive event.

3.3.4 Instrumentation requirements

An activity may consist of sub-operations spanning multiple levels of the stack from user code to operating system and network protocols. A given system can be instrumented at different levels of granularity, depending on the use-case: a multi-layered activity tracking approach enables more detailed performance analysis but introduces higher overhead.

We allow this choice, but require that any instrumentation of the reference system satisfy two properties, without which the transient critical paths are ill-defined. The first states that any event having prior events must be caused by an activity earlier in time, i.e. any unexpected event in $(t_s, t_e]$ indicates insufficient instrumentation:

Property 1 (Minimum in-degree)**.** *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of activity graph G in time interval $[t_s, t_e]$. Additionally let*

$$V_s \equiv \left\{ v \in V \mid \nexists v' \in V : \; v'[t] < v[t] \right\}$$

be a set of vertices in $G_{[t_s,t_e]}$. A vertex $v \in V \setminus V_s$ has in-degree of at least one.

The second states that at no point do all system workers perform waiting activities while no communication activity is occurring. Such behavior would imply deadlock, and so any such points in the activity graph of a non-blocked computation indicates insufficient instrumentation.

Property 2 (Communication existence)**.** *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in $[t_s, t_e]$, and $\tau \in [t_s, t_e]$ be a point in time. Let*

$$S \equiv \left\{ e = (v_i, v_j) \in E_w \subseteq E \mid e[p] = \textit{Waiting},\ v_i[t] \;\leq\; \tau \;\leq\; v_j[t] \right\}.$$

If $|S| = N_\tau$, where N_τ is the number of active workers of the reference system at time τ, then $\exists e' = (v_k, v_m) \in E_c \subseteq E$ for which $v_k[t] \leq \tau \leq v_m[t]$.

These two properties can also checked efficiently online to inform users when the ingested activity logs are incomplete. For example, instrumentation (or associated log preprocessing) can guarantee that no waiting activities are created as long as the corresponding communication activity that caused the waiting activity to end has not been observed.

3.4 Program activity graph construction

A full setup of Snailtrail consists of a system under test, a log data preprocessor and the analytical components of Snailtrail. Here, we describe the assumptions Snailtrail makes on the raw input trace to convert it into an activity graph.

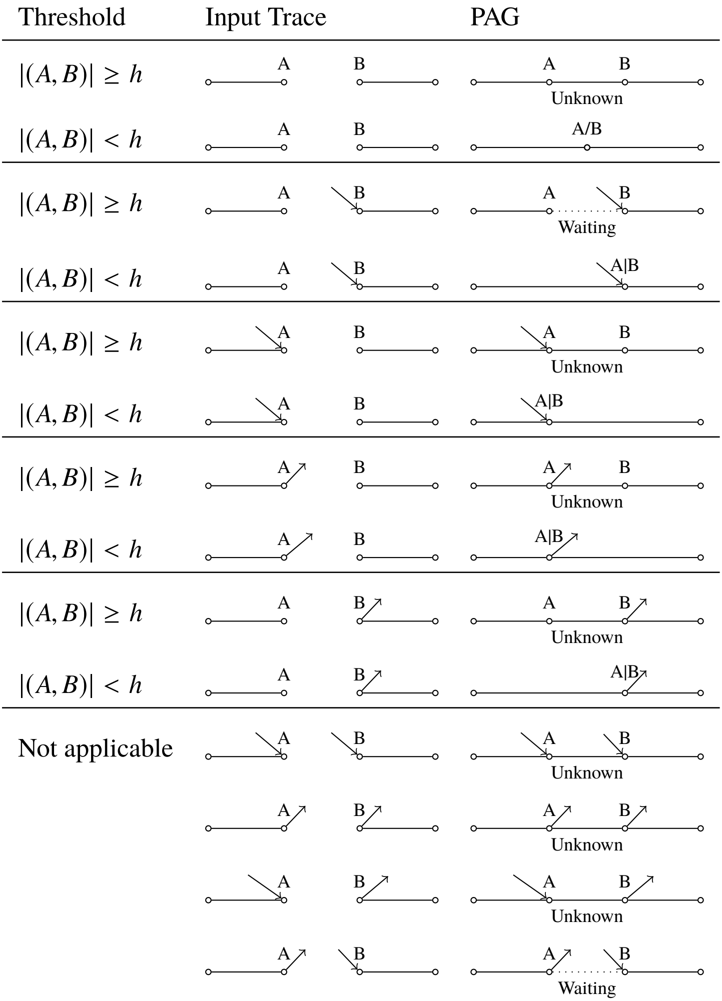
A raw event is produced by the log preprocessor, which is specific to each system under test. The raw event has a type (start/end, send/receive), a timestamp, a worker identifier, an optional remote worker identifier if it is a communication event, a correlation identifier that is used to correlate messages logged by sender and receiver workers as well as an activity and operator type. Each such raw event corresponds to a node in the program activity graph. The node is identified by a worker and a timestamp. Edges are constructed between the nodes using the meta information in the raw events (e.g., activity type) and the assumptions we describe in this section.

The PAG construction is performed in two phases. First, we generate the nodes that correspond to activities logged by each worker, which results in a different timeline per worker. Then, communication activities between worker activities are added. To generate a connected PAG, Snailtrail fills in any gaps that are not identified as *waiting state* in a worker's timeline by aligning neighboring activities or adding *unknown* activities.

Unknown activities are especially useful in practical deployments where log data is also missing due to insufficient instrumentation, transient failures in the upstream logging pipeline, or network delays preventing data for a graph snapshot reaching Snailtrail before it calculates the CP. However, adding unknown activities can introduce significant overhead since such activities can explode the number of activities in the activity graph. Consequently, we instead collapse any gaps smaller than a user-defined threshold (in the experiments of this paper, 100 ms) and create unknown activities for the remaining ones. Nevertheless, it is important to distinguish waiting and unknown activities because the former determine what parts of the PAG can be on a critical path.

The heuristics we apply are shown in Table 3.2. These heuristics are used to determine if a gap between events is to be classified as an unknown or waiting activity. It provides a case distinction based on the duration of the gap and the events before and after the gap. We use a threshold h to determine how to handle gaps of different duration. The result is the PAG that Snailtrail uses to compute the CP metric.

Table 3.2: PAG construction rules in Snailtrail. Gaps smaller than the threshold h are closed by aligning the two sides of the gap. If either of them indicates a message send or receive event, the other is moved. Consecutive messages are either bridged with a waiting activity if a message send is followed by a receive event, or an unknown activity.



The thresholding of adjacent events depends on the kind of activities. If a gap occurs between a worker-local activity and a message and this gap is smaller than the threshold h, we always shift the worker-local activity and leave the node representing a message send or receive as-is. In case there is a gap between two messages we never merge the nodes, i.e. the threshold does not apply here. Instead, we always add either an unknown or a waiting edge.

Dealing with incomplete instrumentation in Spark and TensorFlow In TensorFlow, data communication activities take place between data processing activities on different threads and we recover them the same way Timeline does by using input dependencies to identify communicating parties. Communication duration is computed as the time interval between the source activity ending and the destination activity starting, but note that this masks the effect of buffer management and results in overestimates for message transfer time.

We see many cases in TensorFlow consisting of a message send, followed by a subsequent, later receive on the same worker. TensorFlow supplies no information about what activity was happening between these two events. Initially, we classified this activity as "unknown," but this led to an explosion in the number of potentially critical paths in the system. Instead, we assume that the worker *has* to be in a waiting activity just before the receive, which means that this interval cannot be part of any critical path, and so we classify the entire interval as "waiting". Inspection of the code suggests this is always the case, but it is possible for code to process data while periodically polling for new messages. Yet there are still many gaps in worker timelines which we classify as edges of unknown type. More detailed instrumentation would avoid this problem.

In Spark we infer waiting activities between tasks being sent by Spark's driver to an executor and the executor dispatching them. As with TensorFlow, more careful instrumentation would obviate the need for this, but the result is still completely accurate for critical path determination.

In the case of all-to-all shuffles, Spark's centralized task allocation by the driver allows us to infer control messages and buffering activity surrounding the all-to-all message exchange, which means we can even analyze published Spark traces which lack RDD lineage information, such as the ones published by Ousterhout [Ous].

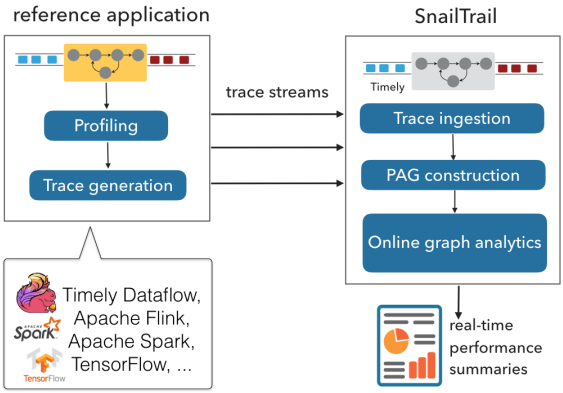


Figure 3.4: Snailtrail system overview. Snailtrail ingests a trace stream from a reference application, constructs the PAG, and computes the critical participation. It outputs CP-based performance summaries.

3.5 Snailtrail system implementation

The CP-computation is implemented in Snailtrail, itself a data-parallel streaming application written in Rust using Timely Dataflow. Figure 3.4 shows an overview of the system. It reads streams of activity traces via sockets, files, or message queues from a reference application and outputs a stream of performance summaries. Snailtrail operates in four pipeline stages: it ingests logs, slices the stream(s) into windows $[t_s, t_e]$ and constructs PAG snapshots, computes the CP of the snapshots, and finally outputs the summaries we show in Section 3.6.

Traces are sent to Snailtrail which ingests a stream S of performance events corresponding to vertices in the activity graph. The snapshots are constructed using Algorithm 1, which we will describe now.

First, Snailtrail extracts from S the events in the time window $[t_s, t_e]$ (line 1). These are then grouped by the worker that recorded them (line 2). Each group corresponds to a worker timeline in Figure 3.2a. Then Snailtrail sorts the events in each timeline by time (line 6), and scans each timeline in turn to create the set of edges E_w (line 8) that correspond to typed worker activities as described in Section 3.3.1. Meanwhile, communication activities are partially

initialized based on a send- and receive-end at each worker (line 9). Then in line 10 partial edges are grouped by the attributes $(w_{\text{id}}^{\text{src}}, w_{\text{id}}^{\text{dst}}, c_{\text{id}})$; note that $w_{\text{id}}^{\text{src}}$ is the sender worker id, $w_{\text{id}}^{\text{dst}}$ is receiver id, and c_{id} is generated to uniquely identify a message. These pairs of partial edges are concatenated to create the final communication edges in E_c, and the output is the union of sets E_w and E_c (line 11). Messages activities that span the snapshot interval miss a corresponding send or receive event. For these message we insert an artificial send/receive node at the corresponding worker to still be able to reason about them. Note that both send and receive events carry enough information to determine all information required for the activity to be constructed.

Input : A stream S of logs and a window $[t_s, t_e]$;
Output : The graph snapshot $G_{[t_s,t_e]}$;

1 Let $S_{[t_s,t_e]}$ be the logged events from S in $[t_s, t_e]$;
2 Group events in $S_{[t_s,t_e]}$ by worker;
3 Let $E_w = \emptyset$ be a set of worker-local activities;
4 Let $E_h = \emptyset$ be a set of communication half edges;
5 **for** *each worker timeline* S_w *in* $S_{[t_s,t_e]}$ *in parallel* **do**
6 Sort S_w by time;
7 **for** *each event* e *in* S_w **do**
8 $E_w = E_w \cup e$ if e is a worker-local activity;
9 $E_h = E_h \cup e$ if e is a send or receive event;
10 Group half edges in E_h by $\left(w_{\text{id}}^{\text{src}}, w_{\text{id}}^{\text{dst}}, c_{\text{id}}\right)$;
11 Create the set E_c of edges for communication activities based on grouped half edges;
12 **return** $E_w \cup E_c$

Algorithm 1: Graph snapshot construction

Algorithm 1 requires two shuffles of the incoming log stream: one on worker id (before line 2), and a second on the triple $\left(w_{\text{id}}^{\text{src}}, w_{\text{id}}^{\text{dst}}, c_{\text{id}}\right)$ (before line 10). The most expensive step is sorting the timeline (line 6), requiring $O\left(|T| \cdot \log |T|\right)$ time, where $|T|$ is the number of events in the timeline. Parallelism is limited by the number of workers in the reference system, which are usually many more than assigned to Snailtrail and the density of the graph. We emphasize that edges in the PAG represent real *happened-before* dependencies given by the instrumentation.

Input : An activity graph snapshot $G_{[t_s,t_e]} = (V, E)$;
Output : A set $S = \{(e, \text{CP}) \mid e \in G_{[t_s,t_e]}\}$ of CP values;

1 Let $V_s \equiv \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$; // `start nodes`
2 Let $V_e \equiv \{v \in V \mid \nexists v' \in V : v'[t] > v[t]\}$; // `end nodes`
// `Perform both traversals in parallel`
3 Traverse $G_{[t_s,t_e]}$ starting from V_s, and count the total number of times each edge is visited, let c_1;
4 Traverse $G_{[t_s,t_e]}$ backwards, starting from V_e, and count the total number of times each edge is visited, let c_2;
5 $S = \emptyset$;
6 **for** *each edge* $e \in E$ **do**
7 $$S = S \cup \left\{ \left(e, \frac{c_1 \cdot c_2 \cdot e[w]}{N \cdot (t_e - t_s)} \right) \right\}$$
8 **return** S

Algorithm 2: Critical participation (CP metric)

For each graph snapshot, the CP-metric is computed using Algorithm 2. Snailtrail collects 'start' and 'end' nodes (lines 1–2) as seeds to traverse $G_{[t_s,t_e]}$. V_s (resp. V_e) includes the node(s) with the minimum (resp. maximum) timestamp $v[t]$ in $G_{[t_s,t_e]}$. Typically, $|V_s| = |V_e| = \ell$, where ℓ is the number of timelines, and so all nodes in V_s have timestamp t_s whereas all nodes in V_e have timestamp t_e.

Algorithm 2 computes the transient path centrality $c(e)$ of Equation (3.3) for all edges in $G_{[t_s,t_e]}$. Observe that $c(e) = c_1 \cdot c_2$, where c_1 is the number of paths from the source of e to any node in V_s, and c_2 is the number of paths from the destination of e to any node in V_e. The algorithm thus performs two simple traversals of $G_{[t_s,t_e]}$ in parallel, computing c_1 and c_2 for each edge (lines 3–4). Each traversal outputs pairs (e, c_i) and these are finally grouped by e to give CP values (lines 6–7).

Note that, while traversing $G_{[t_s,t_e]}$, we visit each edge in $G_{[t_s,t_e]}$ *only once* by propagating the *final* value c_1 (resp. c_2) from each edge to all its adjacent edges. This reduces the intermediate results of the computation significantly. We compute the CP according to Equation (3.3), which does not require path materialization.

Algorithm 2 requires two partitions of $G_{[t_s,t_e]}$: one on source, and one on destination identifiers. Worst-case time complexity is $O(d)$, where d is the diameter of $G_{[t_s,t_e]}$ in number of edges, i.e. the maximum number of edges in any transient critical path.

Performance summaries are constructed by user-defined groupings on the edge attributes and summing CP values over each group.

Snailtrail's accuracy depends on the quality of the instrumentation. A more complete set of dependencies increases the accuracy of the CP metric. It is especially important to correctly instrument messages and—if possible—already provide information about waiting workers as part of the event stream.

3.6 CP-based performance summaries

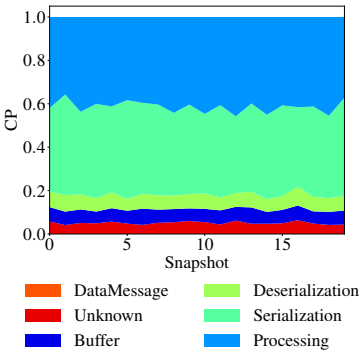
The CP metric provides an indication of an activity's contribution to the evolving critical path. Snailtrail can be configured to generate different types of performance summaries using the CP metric. Each summary type targets a specific aspect of an application's performance and is designed to reveal a certain type of bottleneck. In particular, Snailtrail provides four performance summaries which can answer four types of questions:

1. *Which activity type is on the critical path?* (Activity summary)
2. *Is there data skew?* (Straggler summary)
3. *Is there computation skew?* (Operator summary)
4. *Is there communication skew?* (Communication summary)

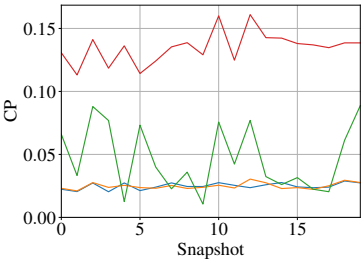
The performance summaries not only indicate potential bottlenecks, but also provide immediate actionable feedback on which activities to optimize, which workers are overloaded, which dataflow operator to re-scale, and how to minimize network communication.

Figure 3.5 shows examples of the four summary types for the Dhalion word-count benchmark [Flo+17] on Flink with 1 second snapshots. In the rest of this section, we describe each summary type in detail and we discuss how to use them in practical scenarios to improve an application's performance.

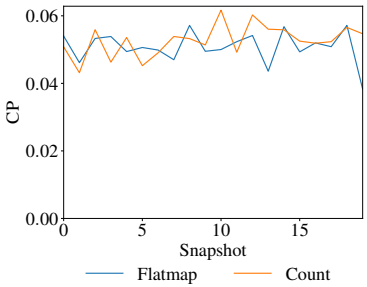
Activity summary *Is the fault-tolerance mechanism in the critical path when taking frequent checkpoints? Is coordination among parallel workers an overhead when increasing the application's parallelism?* An activity summary can answer this sort of questions about an application's performance. This summary plots the proportional CP value of selected *activity types* with respect



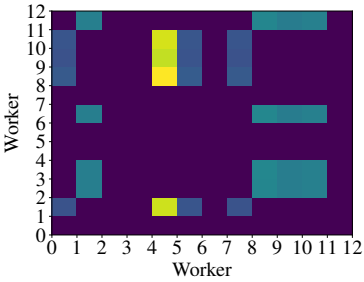
(a) Activity



(b) Straggler, lines indicate distinct workers



(c) Operator



(d) Communication, lighter colors indicate higher CP

Figure 3.5: Examples of Snailtrail summary types for the *Dhalion* [Flo+17] benchmark on Flink with 1 s snapshots. Figure (a) shows a performance summary aggregated by activity type. In Figure (b) we show a performance summary aggregated by worker to highlight stragglers. In Figure (c) we show a CP-based comparison between two operators, aggregated over all instances. Figure (d) shows a heatmap of the communication criticality, i.e. which worker communication appears to be on the critical path.

to the other activity types in a given snapshot. Activity reveal bottlenecks inherent to the system or its configuration. Having a ranking of activity types based on their critical participation essentially gives us an indication on which activities have the higher potential for optimization benefit. For example, if we find that serialization is on the critical path, we might want to try a different serialization library. The activity summary ranking can also help us choose good configurations for our application, like how to adjust the checkpoint interval or the parallelism. The activity summary of Figure 3.5 shows that serialization and processing have the higher potential for optimization. Activity summaries can be configured to plot selected activities only, as in Figure 3.1 where we only show the Spark driver's scheduling.

Straggler summary *Is there data skew? If so, which worker is the straggler?* Snailtrail can answer these questions with a straggler summary, which plots the critical participation of a worker's *timeline* in a certain snapshot. The straggler summary relies on the observation that if a worker is a straggler then many transient critical paths pass through its timeline. Hence, we can compare how how critical a worker's activities are as compared to the other workers in the computation and reveal computation imbalance. This ranking can serve as input to a work-stealing algorithm or guide a data re-distribution technique. The straggler summary of Figure 3.5 clearly shows one straggler worker in the Flink job. In Section 3.7.5, we look closer into detecting skew with Snailtrail.

Operator summary *Will re-scaling my dataflow improve performance? And if yes, which operator in the dataflow to re-scale?* An operator summary plots the critical participation of each operator's processing activity in a snapshot, normalized by the number of parallel workers executing the operator. This summary reveals bottlenecks in the dataflow caused by resource underprovisioning and serves as a good indicator for scaling decisions. Traditional profiling methods fail to detect that an operator might be limiting the end-to-end throughput of a dataflow even if its parallel tasks are perfectly balanced. Such bottlenecks are hard to detect by looking at traditional metrics such as queue sizes, throughput, and backpressure. The operator summary of Figure 3.5 shows that both operators have similar critical participation, thus the parallelism of the job is properly configured. In Section 3.7.5, we present a detailed use-case where operator summaries guide scaling decisions for streaming applications.

Communication summary *Is there communication skew? And if yes, which communication channels to optimize?* A communication summary plots the critical participation of communication activities between each pair of workers within a given snapshot. Contrary to traditional communication summaries, this CP-based summary does not rely on communication frequency or absolute message sizes. Instead, it ranks communication edges by their critical importance: the more often a communication edge belongs to a transient critical path, the higher it will be ranked by the summary. Communication summaries can be used to minimize network delays and optimize distributed task placement. If we find that a pair of workers' communication is commonly on the critical path, it is probably a good idea to physically deploy these two workers on the same machine. For example, the communication summary of Figure 3.5 indicates that co-locating worker 5 with workers 11–13 could benefit performance.

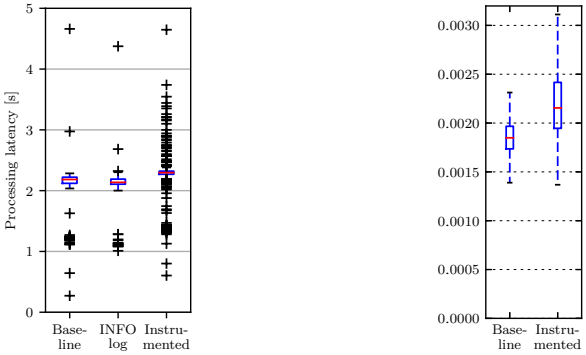
In our experience, the CP requires a sampling interval that is large enough to capture recurring patterns in the computational structure of the analyzed system. We observe stable results with a sufficiently large window size.

3.7 Evaluation

To show generality, we evaluate Snailtrail on four reference systems: Timely Dataflow 0.3.0, Apache Flink 1.2.0, Apache Spark 2.1.0, and TensorFlow 1.0.1. Our evaluation is divided into four categories. First, in Section 3.7.2 we show the instrumentation Snailtrail needs does not cause significant impact on the performance of reference systems. Second, in Section 3.7.3 we investigate Snailtrail's performance and show it can deliver results in real time with high throughput and low latency. Third, we compare the quality of Snailtrail's analysis and the utility of the CP metric with both conventional profiling and traditional critical path analysis (Section 3.7.4). Finally, we present use cases for Snailtrail with analysis results from using Snailtrail in practice in Section 3.7.5.

3.7.1 Experimental setting

Snailtrail uses *Timely Dataflow* [McS] 0.3.0 compiled with Rust 1.17.0. In all experiments, Snailtrail ran on an Intel Xeon E5-4640 2.40 GHz machine with 32 cores (64 threads) and 512 GiB RAM running Debian 7.8 ("wheezy"), and was configured to produce results by ingesting execution traces from a reference system on a different cluster.



(a) Overhead for Flink 1.2

(b) Overhead for Timely Dataflow.

Figure 3.6: Latency with and without instrumentation in seconds.

We compare Snailtrail to existing approaches with several traces generated by Flink, Spark, and TensorFlow using the following benchmarks. For Flink, we use the Yahoo streaming benchmark (YSB) [Chi+16] and the word-count benchmark of *Dhalion* [Flo+17]. For Spark, we use YSB and, for TensorFlow, we use the AlexNet [KSH12] program on ImageNet [Rus+15]. To evaluate Snailtrail performance we use Flink (configured with 48 parallel tasks) running a real-world *sessionization* program on a 10 minute window of operational logs from a large industrial datacenter. This generates a trace with a median number of 30 thousand events per second (around 7.5 million events for a 256 s snapshot, the largest we used). We also show the instrumentation overhead in Flink, with the same sessionization experiment, and Timely Dataflow, using a page-rank computation with 16 parallel workers on a random graph.

3.7.2 Instrumentation overhead

Snailtrail relies on tracing functionality in the reference system, and this incurs performance overhead. To evaluate the overhead of the instrumentation we added, we implemented a streaming analytic job, *sessionization*, in Flink and an iterative graph computation, page rank, in Timely Dataflow, and measured performance with tracing enabled and disabled. For TensorFlow and Spark we use their existing, and somewhat incomplete, tracing facilities.

Table 3.3: Snailtrail's median latency per snapshot in seconds for the online analysis of different snapshot intervals in seconds. The latency is given in seconds. The last row shows the median number of million events per snapshot.

Interval	1	2	4	8	16	32	64	128	256
Latency	0.06	0.14	0.29	0.62	1.40	2.93	5.91	13.16	24.84
Events	0.03	0.06	0.12	0.24	0.48	0.94	1.91	3.76	7.50

Figure 3.6 shows box-and-whisker plots of processing latency for Flink and Timely Dataflow implementations. Individual bars correspond to the cases where logging is completely turned off (*baseline*), the default logging level (*info*), and our detailed tracing (*instrumented*).

Flink shows a statistically significant difference of 9.7 % (±1.43 %) additional mean latency, or 203 ms (±29.9 µs) in absolute terms, at 95% confidence. This overhead is negligible, given that Flink typically runs with logging enabled in production deployments.

Timely includes a minimal logger which encodes log records in a compact binary format and writes them to stable storage in small batches. For Timely Dataflow, there is a statistically significant difference of 13.9 % (±5.5 %) increase in the mean latency, or 319 µs (±126.2 µs) in absolute terms, at 95% confidence.

Experiments with Spark and TensorFlow showed no discernible overhead for collecting the traces required by Snailtrail. Overall, we argue that performance penalties around 10% are an acceptable trade-off for greater insight, and could be additionally amortized in some cases.

3.7.3 Snailtrail performance

We evaluate Snailtrail's performance to demonstrate that (i) it always operates online and thus provides feedback to the running reference applications in real-time and (ii) its analysis scales to large deployments of reference applications without violating this online requirement.

Latency We require Snailtrail to be capable of constructing the PAG and computing the CP-metric for a snapshot of size x seconds in less than x seconds. The number of events in a snapshot depends on (i) the snapshot duration and (ii)

Table 3.4: Snailtrail's maximum achieved throughput (millions of processed events per second) and corresponding latency per snapshot in seconds for the online analysis of different snapshot intervals (length in seconds).

Interval	1	2	4	8	16	32	64	128	256
Throughput	1.2	1.2	1.2	1.1	1.1	1.0	0.8	0.5	0.4
Latency	0.7	1.4	3.2	7.1	10.1	10.2	16.8	24.9	30.8

the instrumentation granularity of the reference system. For this experiment, we vary the number of events in the snapshot by increasing its duration from 1 second to 256 seconds (in powers of 2) and we run Snailtrail on the Flink sessionization job trace, which is the densest one we have. Note that the public Spark traces from real-world cloud deployments [Ous] are not as dense as the ones generated by the Flink streaming computations we run.

We show median latency and number of events per snapshot in Table 3.3; Snailtrail is always capable of operating online and its latency increases almost linearly with the snapshot duration. Specifically, it can process 1 second of input logs in 6 ms and 256 s of input logs in under 25 s.

Throughput To evaluate Snailtrail's throughput, we interleave the processing of multiple snapshots to increase the number of events sent to the system. Table 3.4 shows the maximum achieved throughput (number of processed events per second) while respecting the online requirement and the corresponding latency for processing an input snapshot, including PAG construction and CP computation. For 1 second snapshots, Snailtrail can process 1.2×10^6 events per second; a throughput *two orders* of magnitude larger than the event rate we observed in all log files we have, including the Spark traces from *Spark Performance Analysis* [Ous]. Snailtrail keeps up with all tested workloads: the time to process a snapshot is always smaller than the snapshot's duration. Throughput decreases with larger snapshot sizes since the PAG gets bigger.

3.7.4 Comparison with existing methods

We examine how useful the CP-based summaries produced by Snailtrail are in practice, as compared to the weight-based summaries produced by conventional profiling, where activities are simply ranked by their total duration, and the

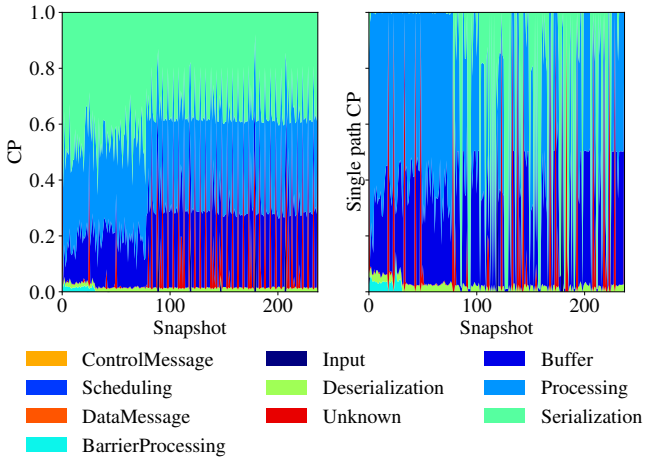


Figure 3.7: CP-based (left) and single-path (right) summaries for Flink on YSB (1 second snapshots).

single-path summaries, where CP is computed on a single transient critical path (in this experiment selected at random). We show examples of such summaries in Figures 3.7 to 3.9 for Flink, Spark, and TensorFlow, along with the configuration of each system.

First note that single-path summaries correspond to a straightforward application of traditional CPA on trace snapshots where only a single path is chosen at random. The plot on the right of Figure 3.7 exhibits high variation because different transient critical paths may consist of completely different activities, even within the same graph snapshot. In contrast, CP is a *fairer* metric that avoids this misleading critical activity "switching" by aggregating information from all transient critical paths in a snapshot.

Conventional profiling summaries do not account for overlapping activities, in contrast to CP-based summaries. They overestimate the participation of activities in the critical path (e.g., the processing activity in the right plot of Figure 3.8), resulting in activity durations that may even exceed the total duration of the snapshot. The CP-based summary of Figure 3.8 overcomes this problem and highlights the overhead of global coordination in micro-batch systems, a known result also pointed out in *Drizzle* [Ven+17].

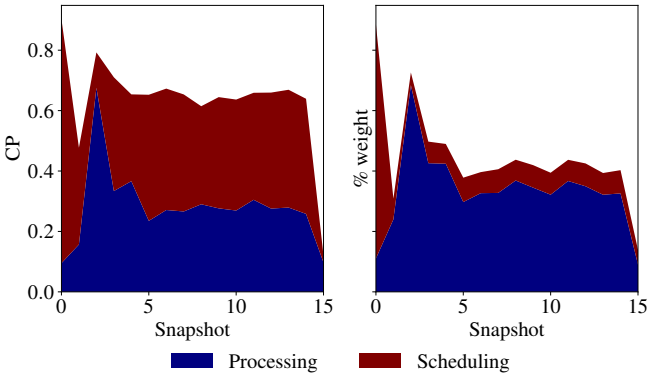


Figure 3.8: CP-based (left) and conventional profiling (right) summaries for Spark on YSB [Chi+16] (8 second snapshots). The CP-based summary clearly highlights the barrier role of Spark's driver.

Snailtrail is also different to traditional profiling in its ability to focus on different parts of a long-running computation. This feature is particularly useful in machine learning, where program phases have diverse performance characteristics. As an example, Figure 3.9 shows CP-based and conventional summaries for the accuracy phase of the AlexNet image processing application on TensorFlow with 16 workers. We plot processing and communication as separate bars for convenience and we further break down processing into the different operators appearing in this computation phase. The conventional summary overestimates the participation of communication and underestimates the importance of the Conv2D operator, which is the most critical one according to the CP-based summary. Processing in the conventional summary is dominated by the unknown activity type due to limited instrumentation in TensorFlow.

3.7.5 Snailtrail in practice

We select Apache Flink as the representative streaming system and demonstrate Snailtrail in action. We describe two use-cases and give examples of how the CP-based summaries can be used to understand and improve application performance of long-running computations.

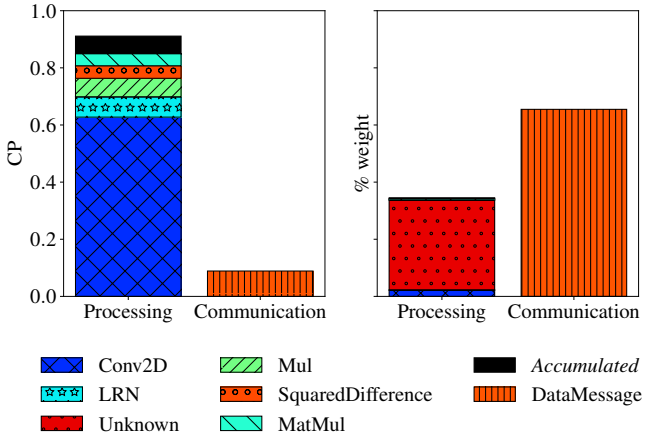
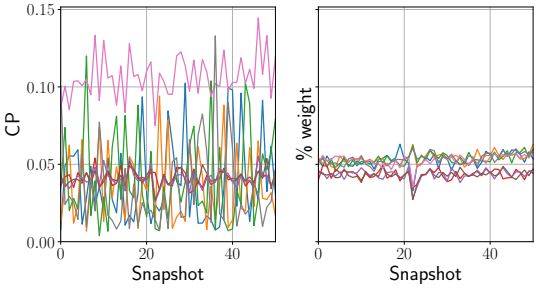


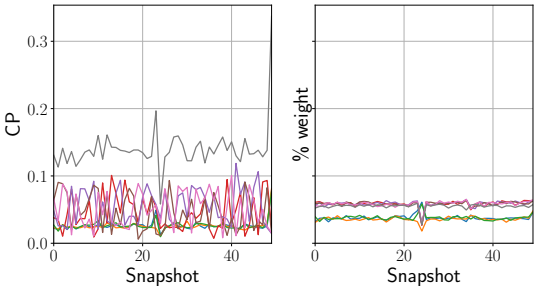
Figure 3.9: CP-based (left) and conventional profiling (right) summaries for the accuracy phase of AlexNet on TensorFlow (16 threads).

Detecting skew To demonstrate straggler summaries in action, we use the benchmark of *Dhalion* [Flo+17]. The benchmark contains a word-count application and a data generator. The data generator can be configured with a skewness percentage. We experiment with 30%, 50%, and 80% skewness. We configure the parallelism to be equal to four for all operators and we generate straggler summaries and conventional summaries shown in Figure 3.10. For small skew percentage, the conventional summaries fail to detect any imbalance and essentially indicate uniform load across workers. For higher skew percentages (50–80%) they indeed reveal a skew problem, yet they are unable to indicate a single worker as the straggler. Instead, they attribute the imbalance problem to several workers. On the other hand, the CP-based straggler summaries consistently and accurately detect the straggler worker, even for low skew percentage.

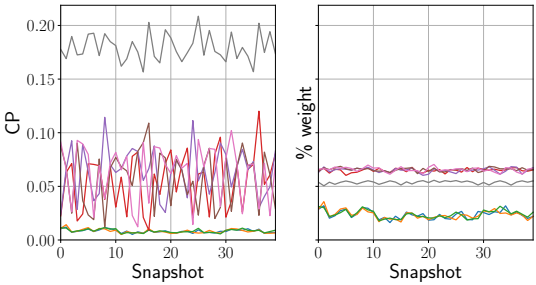
Optimizing operator parallelism We now demonstrate how Snailtrail can guide scaling decisions for streaming applications. We use Dhalion's [Flo+17] word-count benchmark again and initially under-provision the flatmap stage. We configure four parallel workers for the source, two parallel workers for the flatmap, and four parallel workers for the count operator.



(a) 30% skew

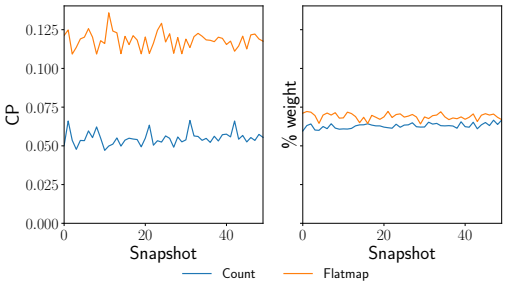


(b) 50% skew

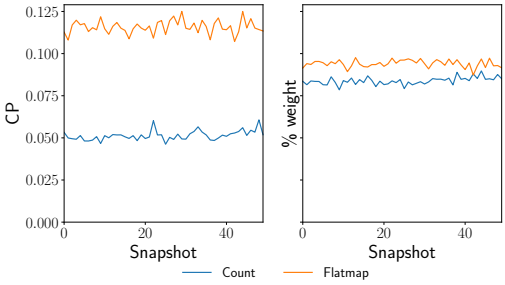


(c) 80% skew

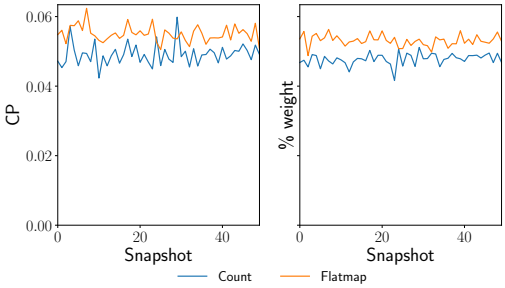
Figure 3.10: Straggler and conventional profiling summaries for the benchmark of *Dhalion* [Flo+17] on Flink and different skewness percentage. The data generator has been configured with 30% (a), 50% (b), and 80% (c) skewness.



(a) 4/2/4



(b) 1/2/4



(c) 4/4/4

Figure 3.11: Operator and conventional profiling summaries for the benchmark of *Dhalion* [Flo+17] on Flink and different configurations of operator parallelism. The source, flatmap, and count operators are configured with parallelism 4/2/4 (a), 1/2/4 (b), and 4/4/4 (c).

Figure 3.11a shows the operator and conventional profiling summaries for this configuration. We see that the operator summary detects that the flatmap workers are bottlenecks. On the other hand, the conventional summary shows a negligible difference between the parallel workers' processing. In addition, we gather metrics from Flink's web interface. Using those, we can observe backpressure, yet we have no indication of the cause. We next decrease the source's input rate, by changing its parallelism to one worker. Note that slowing down the source is a common system reaction to backpressure. Figure 3.11b shows the operator and conventional profiling summaries after this change. Notice how slowing down the source does not solve the problem and how the operator summary still provides more accurate information than the conventional one. The operator summary essentially indicates that the flatmap operator has a high CP value and needs to be re-scaled. Figure 3.11c shows the summaries after applying a parallelism of four to all operators. Checking Flink's web interface again we see that the backpressure disappears.

3.8 Critical participation: conclusion

Online critical path analysis represents a new level of sophistication for performance analysis of distributed systems, and Snailtrail shows its applicability to a range of different engines and applications. Looking ahead, Snailtrail's online operation suggests uses beyond providing real-time information to system administrators: Snailtrail's performance summaries could serve as immediate feedback for applications to perform automatic reconfiguration, dynamic scaling, or adaptive scheduling. The source code of Snailtrail has been released as open source.[1]

[1] `https://github.com/strymon-system/snailtrail`

4
DS2: Controlling distributed streaming dataflows

This chapter is based on the paper Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows *[Kal+18].*

We present DS2, a low-latency, robust controller for *dynamic scaling* of streaming analytics applications. DS2 can vary the resources available to a computation so as to handle variable workloads quickly and efficiently.

Static provisioning is a poor fit for continuous, long-running streaming applications. It forces users to choose a single point on the spectrum between allocating resources for worst-case, peak load (which is inefficient) and suffering degraded performance during load spikes. Fixing resources a priori almost inevitably leads to a system which is either over- or under-provisioned for much of its execution.

The solution is to dynamically scale the system in response to load, an idea used extensively in cloud environments [LML14; MS14]. This requires both a *mechanism* for scaling the computation, and a *scaling controller* which decides when and how to scale. This work focuses on the latter; DS2 is designed to be mechanism-agnostic. We will discuss an efficient mechanism to scale distributed stream processors in Chapter 5. DS2 can serve as a policy-based controller for the mechanism we present.

Figure 4.1 illustrates the problems discussed in Section 2.6 of the state-of-the-art Dhalion controller [Flo+17] of Heron, using the same word count dataflow as in the original paper. The dashed line shows the target throughput (source output rate), while the solid line tracks the achieved throughput, which varies due to backpressure and reconfiguration steps as Dhalion changes the computation scale. Dhalion performs six scaling decisions, taking more than 30 minutes to converge.

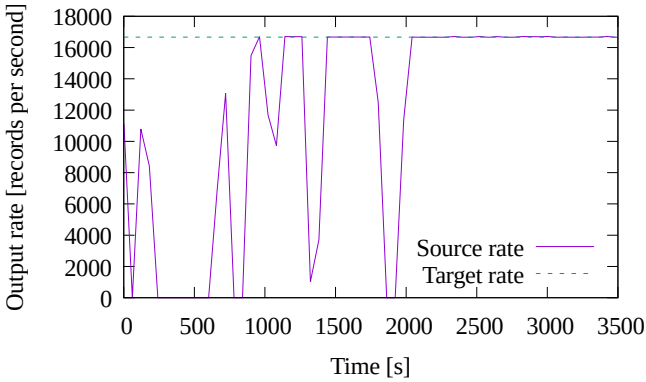


Figure 4.1: Effect of Dhalion's scaling decisions on the source rate when trying to match the target throughput of an under-provisioned word count dataflow. The dashed line shows target throughput, the solid line shows actual throughput over time.

We make the following contributions in this chapter. We review how existing dynamic scaling techniques can lead to inaccurate, unstable, or slow provisioning decisions. Specifically, we identify the causes of these effects in Section 4.1, which we attribute to the lack of a comprehensive performance model, dependence on heuristics, and use of coarse-grained, externally-observed execution metrics. We propose DS2, a general model and controller for automatic scaling of distributed streaming dataflows in Section 4.2. DS2 can accurately estimate parallelism for *all* dataflow operators within a *single* scaling decision, and operates reactively online. As a result, DS2 eliminates oscillation and over-provisioning when making scaling decisions. DS2 bases scaling decisions on real-time performance traces, and is general. It relies neither on specific signals like backpressure, such as *Dhalion* [Flo+17], nor simplistic assumptions like one-to-one operator selectivity, as in the work by Tu et al. [Tu+06].

DS2 gives leverage on existing state-of-the-art approaches: when used in Heron, it identifies the optimal backpressure-free configuration in a few seconds and one step, while Dhalion performs six steps to reach an over-provisioned configuration in the same scenario (Section 4.4.2). We apply DS2 on Apache Flink (Section 4.4.3) and demonstrate fully-automatic scaling of streaming dataflows under dynamic workload.

Finally, we show that DS2 is accurate and converges quickly for both Apache Flink and Timely Dataflow (Section 4.4.4 and Section 4.4.5).

4.1 Background and motivation

We identified that current systems fall short in two controller properties in Section 2.6. Firstly, they provide limited metrics which are not enough to make fast and accurate scaling decisions. Secondly, their policies are often simplistic and rule-based, leading to violations of the SASO properties.

A better approach A more promising approach for making scaling decisions would take into account both each operator's *true* processing and output capabilities, regardless of backpressure or other effects, and the dataflow topology and how scaling each operator will affect downstream operators.

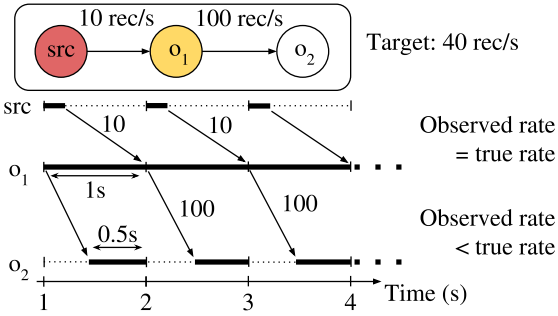


Figure 4.2: An under-provisioned dataflow and the execution timelines of its operators. Target throughput is 40 records per second, but o_1 is a bottleneck creating backpressure and limiting the observed source rate to 10 records per second.

Figure 4.2 gives an intuition of how this works. It shows the execution timelines of operator instances in a simple dataflow. Solid lines show *useful* work performed by an instance (e.g., record processing) while dotted lines show it waiting for input or output. Edges across timelines represent data transfer.

In this example, o_1 is a bottleneck slowing down both the source and o_2 by pausing their execution. Backpressure means that an external observer sees o_1

processing 10 records per second and o_2 processing 100 records per second. Based on this, a policy might provision three additional instances for o_1 to reach a target of 40 records per second, but it could not accurately estimate how much to scale o_2 and would need to make a speculative decision or apply an extra reconfiguration step.

A better approach would measure the *useful time* of an operator's timeline and would determine the true rate of o_1 as 10 records per second and that of o_2 as 200 records per second, inferring that when increasing the parallelism of o_1 to 4, it also needs to double the parallelism of o_2 to keep up with the output rate. Note this can be calculated globally, i.e. for all operators in the dataflow, in a single step.

DS2 does precisely this, obtaining rate measurements of each operator by lightweight instrumentation, which is already present in many streaming systems. In the rest of the chapter we define this notion, extend it to more complex dataflow graphs with multiple sources, and show how DS2 implements it to provide fast, accurate, and stable reconfiguration of streaming dataflows.

4.2 The DS2 model

DS2 identifies the optimal level of parallelism for each operator in the dataflow while the computation executes, based on real-time performance traces. It maintains a changing *provisioning plan*, which captures the number of resources allocated to each operator. It therefore works *online* and in a *reactive* setting.

Note that we do not target offline computation of an initial resource provisioning plan, as proposed by Bilal and Canini [BC17]. Such initial configurations quickly become sub-optimal in a live system where workloads and internal operator states change continuously. However, for static workloads known a priori, DS2 could use historical performance metrics and offline micro-benchmarks (as in [HDB11; Her+11; Gou+17]) to estimate the optimal levels of parallelism before deployment.

In this section, we define the scaling problem (Section 4.2.1), describe the DS2 model (Section 4.2.2), and discuss the model assumptions (Section 4.2.3) and properties (Section 4.2.4).

4.2.1 Problem definition

We target distributed streaming dataflow systems like *Apache Flink* [Car+15] and *Heron* [Kul+15] that execute data-parallel computations on shared-nothing clusters. Such a computation can be represented as a *logical* directed acyclic graph $G = (V, E)$, where vertices in V denote operators and edges in E are streams encoding data dependencies between the operators. A vertex with no incoming edges (no upstream operators) is a *source* and a vertex with no outgoing edges (no downstream operators) is a *sink*.

A dataflow computation runs as a *physical* execution plan which maps dataflow operators to provisioned compute resources (or workers). Let the graph $G' = (V', E')$ represent the execution plan. Vertices in V' are *operator* instances of a corresponding vertex in V and edges are streams. The assignment of operators to workers is system-specific. We show in Section 4.4 that DS2's scaling policy is independent of this assignment.

Problem 1 (Scaling problem)**.** *For a given logical dataflow with input streams $S_1, \ldots, S_n$ and rates $\lambda_1, \ldots, \lambda_n$, identify the minimum parallelism π_i per operator such that the physical dataflow can sustain all current source rates.*

Source operators generate records at a *rate* λ_s, defined by application data sources, for example sensors and stock market feeds. To maximize system throughput and not buffering data on the inputs while limiting processing latency, the execution plan has to have the required scale to sustain the full source rate at any point in time. This means that each operator must be able to process data without stalling its upstream operators from producing output.

Like any controller, DS2 targets workload changes on a timescale greater than its convergence time, and reacting to spikes or other changes on a shorter timescale than the convergence time would cause inefficient fluctuations. In these latter cases, the use of backpressure, buffering, or load shedding leads to more stable results than dynamic scaling at the cost of increased latency or lost data. This behavior can also be explained by the *Nyquist–Shannon sampling theorem* [Sha49], which states that to observe a signal at a frequency f, it must be sampled at at least the double frequency $2f$.

4.2.2 Performance model

We consider operator instances as repeatedly performing three activities in sequence: *deserialization*, *processing*, and *serialization*. This fits all types of

operators in most modern streaming dataflow systems, including Heron, Flink, and Timely Dataflow. When an operator instance is scheduled for execution, it pulls records from its input, deserializes them, applies its processing logic, and serializes the results (if any), which are pushed to the output. Serialization and deserialization are optional and happen only when data is moved between operator instances executed within different OS processes, otherwise data is usually exchanged via shared memory.

The model is based on the concept of *useful time*, which we define for an operator instance as follows:

Definition 14 (Useful time)**.** *The time spent by an operator instance in deserialization, processing, and serialization activities.*

The useful time excludes time spent waiting on input or output. Such waiting does occur in practice, for different reasons depending on the design of the reference system. In Flink, an operator instance may block on input when the input buffers are empty, or on output when there is no free space in the (bounded) output buffers. In Heron, instances may be forced to wait due to a backpressure signal from a slow downstream operator.

In all cases, the useful time amounts to the time an operator instance runs for if executed in an *ideal* setting where it never has to wait to obtain input or push output. In general, useful time differs from the total observed time the instance needs to process and output records, and plays a key role in solving Problem 1.

Based on this distinction, we define the *true* processing and output rate of an operator instance as follows:

Definition 15 (True rates)**.** *The true processing (or output) rate corresponds to how many records an operator instance can process (or output) per unit of useful time.*

The true rates denote the capacity of the operator instance, i.e. the *maximum* processing and output rate the instance could sustain for the current workload. In contrast to the true rates, the *observed rates* are those measured by simply counting the number of records processed and output by the instance over a unit of elapsed time, which might include waiting time.

Definition 16 (Observed rates)**.** *The observed processing (or output) rate corresponds to how many records an operator instance processes (or outputs) per unit of observed time.*

The observed rates are more sensitive to changing workloads, because they depend on the total amount of time an operator was active during the observation window. As more work is available, the active time goes up and the waiting time decreases. The true rates typically have lower variance, especially within short time periods (e.g., a few seconds of execution) as they are based the average "cost" to process and output a single record. This cost naturally can depend on factors like the size of the record, its content, and the state maintained by the operator instance. The average cost can be estimated using appropriate instrumentation of the operator without needing to saturate it. Also, the advance of time can trigger larger computations, for example on window boundaries. Here we assume that these spikes are short-lived and only slightly disturb the average computation time per record.

We define all rates in our model relative to windows of size W seconds of observed time. We denote the useful time for an operator instance W_u, where $0 \leq W_u \leq W$. More precisely:

$$\lambda_p = \frac{R_{\text{prc}}}{W_u} \qquad \text{(true processing rate)} \tag{4.1}$$

$$\lambda_o = \frac{R_{\text{psd}}}{W_u} \qquad \text{(true output rate)} \tag{4.2}$$

$$\widehat{\lambda}_p = \frac{R_{\text{prc}}}{W} \qquad \text{(observed processing rate)} \tag{4.3}$$

$$\widehat{\lambda}_o = \frac{R_{\text{psd}}}{W} \qquad \text{(observed output rate)} \tag{4.4}$$

where λ_p and λ_o are the true processing and output rate. It is undefined when $W_u = 0$. $\widehat{\lambda}_p$ and $\widehat{\lambda}_o$ are the observed processing and output rates (also undefined when $W = 0$), and R_{prc} (resp. R_{psd}) is the total number of records the instance processed (resp. pushed) in W. For a specific operator instance and a window W, the following inequalities hold: $0 \leq \widehat{\lambda}_p \leq \lambda_p$ and $0 \leq \widehat{\lambda}_o \leq \lambda_o$, since $0 \leq W_u \leq W$. In general, the less an operator instance waits on its input and output the smaller the difference between the observed and true rates. Table 4.1 summarizes the notation.

We instantiate the model with the logical dataflow graph G, the output rate of each data source, and the true processing and output rates (λ_p and λ_o) of each operator instance. G is static and known at compile time and does not change during execution, since the logical dataflow is unaffected by the scaling

decisions. The output rates of the data sources are continuously monitored outside the reference system, and the true rates of the operator instances are computed based on system-generated traces, as we explain in Section 4.3.1. The output of DS2 is an estimation of the optimal parallelism, i.e. the number of instances, for each logical operator in the graph G, subject to the constraints of the problem in Section 4.2.1.

Applicable operators and limitations We assume that the output rate of an operator corresponds with the input rate, specifically that changing the parallelism does not change the aggregate output rate over all instances of a specific operator. There are many operators that fall into this category, for example simple map and filter operators or keyed aggregation operators. However, there are notable exceptions. Many computations use a pre-aggregation step to limit the amount of data exchanged between remote operators. In this step, each operator instance computes a partial aggregation of the data it has available locally. The operator then sends a partially-aggregated result to a downstream operator, whose input is partitioned based on the data. This reduces the amount of data from the total amount of records to the total amount of groups in the data. In the worst case, every operator instance outputs the same number of records as each operator could see all groups. Then, the aggregate output rate scales linearly with the number of operator instances. Another example is a flat-map operator that divides input records into smaller records, for example splitting a sentence into words. Its input-to-output rate highly depends on the structure of the data and the model does not capture this part of the computation.

While the model does not cover these cases, it can still converge to a good solution in several iterations as it discovers new aggregate rates each time and does not remember old measurements. Still, it is not guaranteed to converge to a solution in this case. In our evaluation we demonstrate that DS2 converges to stable configurations for dataflows with operators that theoretically violate the discussed requirements.

The calculation of the level of parallelism proceeds as follows. Let A be the adjacency matrix of G. An entry $A_{ij} = 1$ iff the i-th operator outputs to the j-th operator, otherwise $A_{ij} = 0$. We consider operators numbered in topological order from $i = 0$ to $i = m - 1$, where m is the total number of operators in G. This means that if o_i outputs to o_j and, hence, $A_{ij} = 1$, then $0 \leq i < j < m$. There is a topological ordering of the nodes in G and it can be computed in linear time since G is an acyclic graph as defined in Section 4.2.1.

Table 4.1: Notation used throughout Chapter 4

Symbol	Description
G	A logical dataflow graph
m	Number of operators in G, $m > 1$
n	Number of source operators in G, $0 < n < m$
W	Size of a window in time units (observed time)
W_u	Useful time for an operator instance in W
R_{prc}	Number of records pulled from the input in W
R_{psd}	Number of records pushed to the output in W
$\widehat{\lambda_p}$	Observed processing rate of an operator instance
$\widehat{\lambda_o}$	Observed output rate of an operator instance
λ_p	True processing rate of an operator instance
λ_o	True output rate of an operator instance
o_i	i-th operator in G in topological order
p_i	Number of instances of the i-th operator
$o_i[\lambda_p]$	Aggregated true processing rate of the i-th operator
$o_i[\lambda_o]$	Aggregated true output rate of the i-th operator
π_i	Optimal number of instances for the i-th operator

For a time window W and operator o_i with p_i instances, $p_i \geq 1$, we define the *aggregated* true processing and output rates $o_i[\lambda_p]$ and $o_i[\lambda_o]$ as:

$$o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \qquad \text{(aggregated true processing rate)}$$

$$o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k \qquad \text{(aggregated true output rate)}$$

where λ_p^k and λ_o^k are the true processing and output rates of the k-th instance of o_i, as given by Equation (4.1) and Equation (4.2).

The optimal level of parallelism π_i for an operator o_i is now computed using the ratio of the aggregated true output rate of its upstream operators (when they keep up with their inputs) to the average true processing rate per instance of o_i. More formally:

$$\pi_i = \left\lceil \sum_{\forall j:\, j<i} A_{ji} \cdot o_j\left[\lambda_o\right]^* \cdot \left(\frac{o_i\left[\lambda_p\right]}{p_i} \right)^{-1} \right\rceil, \quad n \leq i < m \tag{4.5}$$

where m is the total number of operators in G, and n is the number of source operators in G, and $0 < n < m$.

$o_j[\lambda_o]^*$ denotes the aggregated true output rate of an operator o_j, when o_j itself and all operators before it (in topological order) are deployed with their optimal parallelism to keep up with their inputs. It is recursively computed using the following formula.

$$o_j\left[\lambda_o\right]^* = \begin{cases} o_j\left[\lambda_o\right] = \lambda_{\text{src}}^j, & 0 \leq j < n \\ \dfrac{o_j\left[\lambda_o\right]}{o_j\left[\lambda_p\right]} \cdot \displaystyle\sum_{\forall u:u<j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases} \tag{4.6}$$

where λ_{src}^j is the output rate of the j-th source operator, $0 \leq j < n$.

Note that $o_j[\lambda_o]^*$ depends on the ratio $\frac{o_j[\lambda_o]}{o_j[\lambda_p]}$, which denotes the selectivity of o_j, and the estimated true output rate of the upstream operators ($\forall u\colon u < j$ in the summation). The latter implies that $o_j[\lambda_o]^*$ and, hence, π_i can be efficiently

computed for *all* operators in the dataflow with a *single traversal* of G, starting from the sources. This property is important in practice, as it allows us to estimate the required number of instances for all operators in the dataflow in the same scaling decision.

4.2.3 Assumptions

DS2 makes the following assumptions about the dataflow system it is controlling.

Data-parallel operators An operator's output can be produced by partitioning its input on a key and applying the operator logic separately to each partition. Other than this, the operator's internal logic can be any user-defined function. Data-parallelism is essential for effective scaling decisions: executing multiple operator instances entails partitioning its state into chunks of data processed in parallel. In contrast, non-data-parallel operators do not benefit from scaling. System users could tag such operators for DS2 to ignore, or their lack of parallelism could be identified online by comparing input and output rates before and after scaling. As with existing systems, we leave the integration of such operators for future work.

No data or computation imbalance Our scaling model addresses neither data skew across operator instances nor computational stragglers. Both these types of imbalance can trigger backpressure which cannot be tackled by changing the degree of parallelism of one or more operators. Several robust solutions to the skew and straggler problems exist and have been incorporated into real systems. Techniques such as partial key grouping [Nas+15] introduced in Storm [Nas+16] and further evaluated by Katsipoulakis, Labrinidis, and Chrysanthis [KLC17], and work-stealing for straggler mitigation in MapReduce [Kwo+12] and Google Dataflow [KD16] are complementary to DS2. In Section 4.3.2 we describe how DS2 could be integrated in a general controller for streaming applications which would not only handle dynamic scaling but also include skew and straggler handling components.

Stable workloads during scaling Like existing scaling mechanisms, DS2 operates with the understanding that workload characteristics remain stable between a scaling decision being made and the new parallelism configuration

being deployed. This window is the time taken for DS2 to make a decision (which we evaluate in Section 4.4) plus the time to deploy the new configuration, which depends on the dataflow system in use. In practice, we find this timescale is dominated by the latter in current systems.

4.2.4 Properties

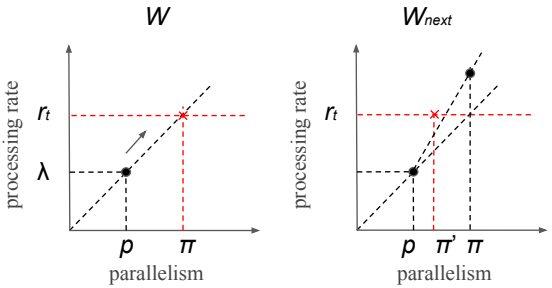
DS2 estimates the optimal parallelism for each operator assuming perfect scaling, that is, the true processing and output rates change linearly with the number of instances. In general, however, true rates are described by non-linear, most commonly sub-linear functions. Super-linear speedups are possible [Gou+17] (e.g., when state fits in cache after a scale-up) but are rare in practice. When this "perfect scaling" assumption holds, DS2 estimations based on Equation (4.5) correspond to bounds, and the model enjoys the following two properties:

Property 3 (No overshoot)**.** *A scale-up decision will not result in over-provisioning. The estimated optimal number of instances π_i for an under-provisioned operator is always less than or equal to the minimum required to keep up with the target rate $r_t = \sum_{\forall j:j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Equation* (4.5)*.*

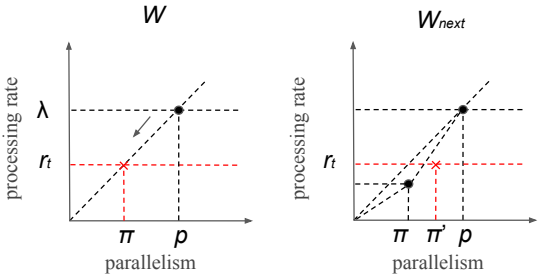
Property 4 (No undershoot)**.** *A scale-down decision will not result in under-provisioning (and, hence, backpressure). The estimated optimal number of instances π_i for an over-provisioned operator is always greater than or equal to the minimum needed to keep up with the target rate $r_t = \sum_{\forall j:j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Equation* (4.5)*.*

Figure 4.3 shows hypothetical scale-up and scale-down scenarios, each during two consecutive time windows, W and W_{next}. Consider an operator initially configured with parallelism p and aggregated processing rate $\lambda < r_t$, where r_t is the target rate, as shown in Figure 4.3a. Assuming linear scaling, our model assigns π instances to reach the target rate r_t. Property 3 states that there exists no $\pi' < \pi$ such that π' matches r_t. Indeed such a π' can only exist in W_{next} if the aggregated processing rate scales super-linearly, as shown in Figure 4.3a.

Similarly, if an operator is initially configured with parallelism p and aggregated processing rate $\lambda > r_t$, as in Figure 4.3b, our model assigns $\pi < p$ instances to scale down to r_t. Property 4 states that there exists no $\pi' > \pi$ such



(a) No overshoot when scaling up



(b) No undershoot when scaling down

Figure 4.3: Given a target rate r_t and aggregated true processing rate λ which does not scale super-linearly, our model guarantees no over-provisioning when scaling up and no under-provisioning when scaling down.

that π' matches the target r_t. As shown in Figure 4.3b, such a π' would violate the assumption of non-superlinear aggregated true processing rate.

Together, these properties imply that repetitive applications of DS2 do not oscillate: they will monotonically converge to the target rate from below or above. This ensures stability without the need to blacklist previous decisions, and simplifies the scaling mechanism significantly.

When true rates are linear and the target rate r_t is accurately estimated for each operator, DS2 converges in at most one step. When one of these two conditions does not hold, for example, true rates do not scale well due to other overheads (e.g., worker coordination) or dataflow operators have data-dependent output rates, DS2 needs more steps to converge to a stable configuration. In each of these steps, DS2 tries to minimize the error of its previous decision to get closer to the target, as any typical controller does.

4.3 Implementation and deployment

The DS2 controller consists of about 1500 lines of Rust running as a standalone process. Here we describe the instrumentation requirements it imposes and discuss the issues encountered integrating it with three different stream processing engines: Flink, Timely Dataflow, and Heron.

4.3.1 Instrumentation requirements

DS2 requires a subset of the instrumentation required by bottleneck detection tools for stream processors like Snailtrail (Chapter 3). The stream processor must periodically collect and report records processed, records produced, and useful time (serialization, deserialization, processing) or waiting time per operator instance. Some stream processors already provide enough information to derive the metrics required by DS2. For example, *Apache Heron* [Heron] provides sufficient metrics. Other systems, such as Flink and Timely Dataflow require additional instrumentation, which we discuss next.

Flink gathers some of the metrics required by DS2 (e.g., records read and produced) by default but we extended its runtime so that each operator instance maintains local counters for (de-)serialization and processing duration as well as for buffer wait time, reporting them to DS2 in configurable intervals. For record-at-a-time systems like Flink, tracking and emitting metrics for every record might incur significant overhead. Instead, we aggregate measurements

per input buffer for all operators, except for sources where we aggregate per output buffer. Specifically, we have implemented a `MetricsManager` module which is responsible for gathering, aggregating, and reporting policy metrics. We assign one `MetricsManager` instance per parallel thread executing operator logic. Each thread maintains local counters for records read, records produced, (de-)serialization duration, processing duration, and waiting for input and output buffers. Source operator instances send their current local counters to the `MetricsManager` every time an output buffer gets full and regular operator instances send their local counters every time they receive a new input buffer for processing. The `MetricsManager` maintains a data structure with the current aggregate metrics of its operator instance and reports them to the outside world in configurable intervals.

Timely Dataflow [McS] outputs raw tracing information, which we aggregate in configurable intervals to produce metrics for DS2. We use a similar `MetricsManager` as in Flink, which receives streams of logged events coming from Timely Dataflow workers and aggregates them on the fly. Each Timely Dataflow worker logs individual events of different types, such as scheduling an operator or sending a message over a data channel, along with their timestamp in nanoseconds. Recall that operator instances in Timely Dataflow are not blocked on their input or output queues; instead, they are continuously spinning, i.e. they are scheduled for execution based on progress-tracking, potentially even if there are no data records to process. Spinning results in a large amount of scheduling event logs, which quickly saturate the `MetricsManager`, although most of these log records are not needed for computing the true rates. To tackle this problem, we modified Timely Dataflow's logger to trace and send to the `MetricsManager` only the "useful" scheduling events, i.e. those that correspond to an operator instance doing some "useful work" for the actual computation.

Apache Heron [Heron] also by default outputs detailed, aggregated metrics, which are periodically collected and fed into DS2. The aggregation window depends on how frequently Heron samples its metrics and can be configured.

4.3.2 Integration with stream processors

DS2 is mechanism-agnostic and can be integrated with any stream processor capable of dynamically varying resources and migrating state. Figure 4.4 shows the high-level architecture of such an integration. Instrumented streaming jobs periodically report metrics to a repository. DS2 consists of a *Scaling Policy*

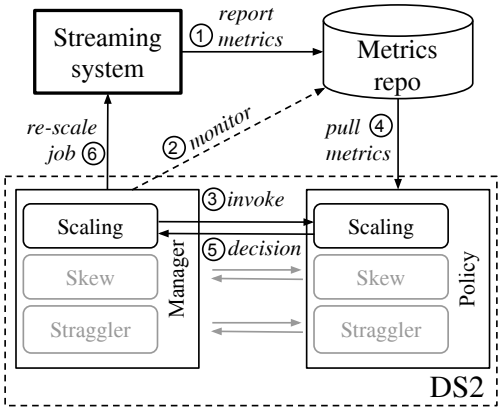


Figure 4.4: DS2 integration with streaming systems

component implementing the model of Section 4.2.2, and a *Scaling Manager* monitoring the repository, invoking the policy when new metrics are available, and sending scaling commands to the stream processor.

While DS2 currently only offers scaling functionality, it could be extended with skew and straggler mitigation techniques as shown in Figure 4.4. In this case, the system would consist of multi-purpose Manager and Policy components, where the first detects the problem type (e.g., presence of skew) and the latter invokes the appropriate policy.

We have integrated DS2 with Apache Flink, which employs a simple scaling mechanism: when instructed, Flink takes a *savepoint*, a consistent snapshot of the job state, halts the computation, and redeploys it with the updated parallelism [Hue16]. We demonstrate this integration in action and evaluate it under a dynamic source rate in Section 4.4.3.

Scaling manager Operational issues in real deployments that are not captured by the model must be handled by the implementation instead. To deal with factors that might affect scaling decisions in practice, the Scaling Manager provides the following configuration parameters:

The *policy interval* defines the frequency with which metrics are gathered and the policy invoked. Tuning the policy interval allows the scaling manager to aggregate metrics meaningfully, e.g., to ensure enough data is available to

compute averages for processing and output rates. Long intervals give stable metrics but also increase reaction time. The interval must also be tuned based on the reconfiguration mechanism of the reference system. In our experiments, we found 5–30 second intervals reasonable for Flink and Timely Dataflow. For Heron, we found the default 60 s suitable.

The *warm-up time* is the number of consecutive policy intervals ignored after a scaling action, since rate measurements can be unstable at the start of a computation or before backpressure builds up.

The *activation time* specifies when DS2 applies a scaling decision, as the number of consecutive policy decisions considered by the scaling manager before issuing a scaling command. Activation time plus an appropriate policy interval mitigates the effects of irregularities in some streaming computations, such as non-incremental tumbling windows or data-dependent operators. For instance, consider naïvely-implemented window operators that buffer records and only apply the computation logic after the window fires. As long as input is simply assigned to a window, the operator's processing rate will appear high but once the window fires and the actual computation is performed the processing rate will suddenly drop. DS2 can consider several consecutive policy decisions and, for example, compute the maximum or median parallelism across intervals before applying a scaling action.

The *target rate ratio* defines a maximum allowed difference between the observed source rate achieved by the policy and the target rate, addressing the practical issue that processing and output rates might be affected by overheads not captured by instrumentation. For instance, adding workers to a distributed computation might incur higher coordination, channel selection cost, or resource contention, and so a computation might need more resources to achieve the target rate than the policy indicates. DS2 estimates the additional resources required by computing the ratio between the currently achieved rate and the target rate.

Practical considerations DS2 also ignores minor changes (e.g., changing an operator's parallelism by one or two), which can be triggered by noisy metrics. External disruptions, such as garbage-collection in Java-based systems or disk I/O, can also influence rates measurements. For example, when integrating DS2 with Flink, we took care to properly configure task managers, heap memory, and network buffers. We are also aware that system performance might degrade after a scaling action (though we have not observed this in practice). If this were

to happen, DS2 rolls back to the previous configuration. Similarly, consecutive decisions resulting in very small improvements indicate a performance issue (e.g., data skew, stragglers) that cannot be improved by scaling. DS2 can limit the number of decisions to prevent further reconfiguration.

DS2 in the presence of skew Even though the scaling model assumes no data imbalance and the current implementation of DS2 does not offer skew mitigation functionality, it is worth discussing how the system behaves if skew actually appears in a streaming application it is controlling. In such a case, the system makes a scaling decision assuming data balance (Section 4.2.3) by averaging true processing and output rates. Thus, DS2 proposes a configuration which might not meet the target throughput but at the same time will not over-provision the system. Further, due to DS2's ability to limit the number of decisions (Section 4.3.2), the policy is guaranteed to converge. We have verified the above behavior experimentally on Flink varying the skew parameter in the Dhalion benchmark from 20% to 50% and 70%. In all cases, DS2 converged after two steps to the configuration which would be optimal if there was no skew, but which in this experiment did not meet the target throughput.

4.3.3 DS2 and execution models

DS2's policy can be applied on many streaming systems. In Flink and Heron each dataflow operator is assigned a number of worker threads that define its level of parallelism, i.e. the number of parallel instances executing the operator's logic. In this case, Equation (4.5) can be directly used to configure operator parallelism independently. In Timely Dataflow, on the other hand, parallelism is configured globally for the whole dataflow. Each worker runs every operator in the dataflow graph according to a scheduling strategy based on progress tracking.

Many computations for Timely Dataflow are implemented in an open-loop setting. This means that the computation can self-adapt to the available work by varying the batching. Less work leads to smaller batches while more available work increases the batch size. Additionally, a computation can control the amount of outstanding data by only feeding new data at specific points in time. For this reason, determining an adequate level of parallelism depends on a target processing latency, which is directly influenced by the amount of unprocessed state in the computation. DS2 optimizes for a throughput target

and to apply it on Timely Dataflow we fix the batch size to prevent this form of self-adaptation.

For Timely Dataflow, DS2 estimates the optimal number of total workers by summing up the optimal level of parallelism, as given by Equation (4.5), for all operators in the dataflow. The intuition here is simple: an operator that needs π_i instances to keep up with its input actually needs $\pi_i \cdot 100\%$ computing power per unit of time. In an execution model like Timely Dataflow's where operators share computing resources (worker threads), the total computing power needed so that the system can keep up with its input is $\sum_{\forall i} \pi_i \cdot 100\%$. We experimentally validate the accuracy of DS2 decisions on Timely Dataflow in Section 4.4.5.

4.4 Experimental evaluation

Our evaluation covers DS2 in use with three different streaming systems: Heron, Flink, and Timely Dataflow. We start our evaluation by comparing DS2 with the state-of-the-art Dhalion scaling controller used in Heron, with the benchmark in the original Dhalion publication [Flo+17]. We then demonstrate DS2 in action through end-to-end, dynamic scaling experiments with Flink, followed by measurements of DS2 convergence and accuracy in using both Flink and Timely Dataflow. Finally, we evaluate the overhead of the instrumentation used by DS2.

4.4.1 Setup

We run all Flink and Timely Dataflow experiments on up to four machines, each with 16 Intel Xeon E5-2650 @2.00 GHz cores and 64 GiB of RAM, running Debian GNU/Linux 9.4. We use Apache Flink 1.4.1 configured with 12 task managers, each with 3 slots (maximum parallelism per operator is 36), and Timely Dataflow 0.5.0 compiled with Rust 1.24.0. For the comparison experiment, we run Heron 0.17.8 on a four socket-machine equipped with AMD Opteron 6276, with 64 threads in total and 256 GiB of memory.

To demonstrate generality across diverse computations and streaming operators, we selected six queries from the Nexmark benchmarking suite of Apache Beam [Tuc+02; NexB; NEX]. Specifically, we test the policy with queries 1–3, 5, 8, and 11, which contain various representative streaming operators: stateless streaming transformations, i.e. map and filter in **Q1** and **Q2**, and a

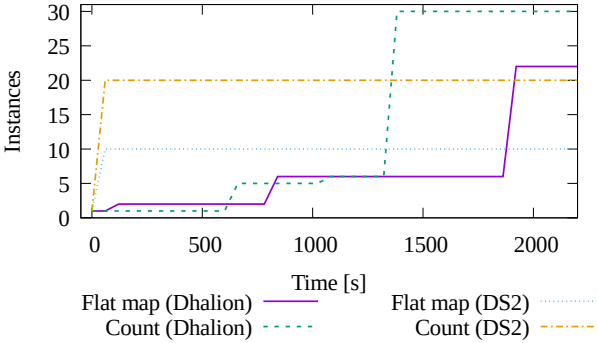


Figure 4.5: Comparison of DS2 vs. Dhalion on Heron using the word count dataflow of Dhalion [Flo+17].

stateful record-at-a-time two-input operator (incremental join) in **Q3**. It also contains various window operators: sliding window in **Q5**, tumbling window join in **Q8**, and session window in **Q11**. These queries specify computations both in processing and event time domains [Aki+15].

We use the wordcount dataflow as specified by Floratou et al. [Flo+17] for the comparison with Dhalion (Section 4.4.2) and the end-to-end experiment on Flink (Section 4.4.3).

4.4.2 DS2 compared to Dhalion on Heron

We compare the accuracy and convergence steps of DS2 with Dhalion, recreating the benchmark in *Dhalion* [Flo+17]. We run Heron with Dhalion and its dynamic resource allocation policy enabled. The source operator of the three-stage wordcount topology (Source, FlatMap, Count) produces sentences at a fixed rate of 1×10^6 per minute. The FlatMap and Count operators are rate-limited to simulate bottlenecks: each FlatMap instance splits at most 100×10^3 sentences per minute, and each Count instance counts up to 1×10^6 words per minute, which are the ratios presented in Dhalion paper. We start under-provisioned with one instance per operator and let Heron stabilize until backpressure is absent.

We have already seen how the source rate evolves to match the target throughput in this experiment in Figure 4.1. Figure 4.5 shows the parallelism

of the FlatMap and Count operators over time, from the start until convergence. Dhalion makes six scale-up decisions (each involving a single operator) and reaches a stable configuration with 22 FlatMap instances and 30 Count instances after 2000 seconds.

We then apply DS2 on the same initial under-provisioned configuration using a 60 s decision interval, no warm-up, one interval activation time, and 1.0 target ratio (cf. Section 4.3). DS2 indicates a required parallelism of 10 for FlatMap and 20 for Count, which indeed is the minimum configuration that handles one million sentences per minute. Note that DS2 correctly estimates the optimal parallelism in a single step, after only one minute of collecting the default Heron performance metrics.

Dhalion requires several re-configuration steps, each affecting a single operator, and reaches a final configuration that is significantly over-provisioned, even in this simple wordcount dataflow. In contrast, DS2 correctly identifies the optimal configuration in a single step and two orders of magnitude less time than Dhalion.

Besides those discussed in Section 4.1, another reason Dhalion takes so long to reach a backpressure-free configuration is that its reaction time depends on the size of the operator queues. By default, Heron has a 100 MiB buffer per operator queue, which may take some time to fill (depending on the workload) before backpressure kicks in and Dhalion can react. In contrast, DS2 only depends on the decision interval where metrics are aggregated, arbitrarily specified by the user and typically much smaller.

4.4.3 DS2 on Flink

We now show DS2 driving Apache Flink, in order to demonstrate the benefits of DS2 when combined with a fast re-configuration mechanism such as that in Flink. Here, DS2 uses a 10 s decision interval, 30 s warm-up time, one interval activation time, and 1.0 target ratio. DS2 hence ignores the first three decisions after re-configuration, applying a decision immediately after.

We use the same wordcount dataflow as before, this time with two phases corresponding to scale-up and scale-down scenarios respectively. In the first phase, the source rate is 2 million sentences per second and Flink starts under-provisioned with 10 FlatMap instances and 5 Count instances. In this state, the FlatMap operators can not keep up with the source rate, neither can the Count operators handle FlatMap's output rate. Once Flink has reached a backpressure-free configuration, we keep the source rate stable for 10 minutes. During the

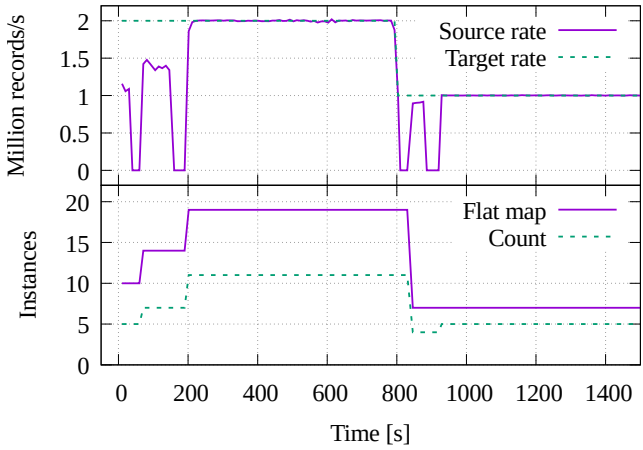


Figure 4.6: Dynamic scaling experiment with Flink using DS2 on the word count dataflow of [Flo+17].

second phase, we decrease the source rate to one million sentences per second and keep it stable for another 10 minutes.

Figure 4.6 shows observed source rate and operator parallelism over time. DS2 applies two scale-up actions. First, at 40 s it re-deploys the dataflow with 14 FlatMap instances and 7 Count instances. This happens right after the warm-up and activation time, and Flink takes around 30 s to snapshot state and restart from the savepoint [Hue16]. At 150 s DS2 acts again to increase FlatMap to 19 and Count to 11 instances. This time Flink takes about 50 s to redeploy the backpressure-free configuration at 200 s. At 803 s (3 s into the second phase) DS2 reacts to the reduced source rate by reducing the configuration to 7 FlatMap and 4 Count instances at 845 s. At 900 s it makes a final decision to increase Count parallelism by one, and Flink successfully applies the change at 930 s, reaching the new optimal configuration.

This shows that DS2 plus an efficient re-configuration mechanism can offer robust dynamic scaling for streaming dataflows, allowing the reference system to react to changes in its workload in just a few seconds, which is significantly faster than any other systems we are aware of.

Table 4.2: Target source rate (records per second) configuration for the NEXMark queries on Apache Flink and Timely Dataflow.

	Bids		**Auctions**		**Persons**	
	Flink	Timely	Flink	Timely	Flink	Timely
Q1	4M	5M	—		—	
Q2	4M	5M	—		—	
Q3	—		500K	3M	100K	800K
Q5	500K	2M	—		—	
Q8	—		420K	4M	120K	4M
Q11	1M	9M	—		—	

Table 4.3: DS2 convergence steps for Nexmark queries on Flink. Values are the level of parallelism of the main operator of each query. Leftmost column shows initial parallelism (from 8 to 28 instances); subsequent columns show optimal level of parallelism as estimated by DS2 in each step. Final decisions converged to by DS2 are highlighted.

Init	**Q1**	**Q2**	**Q3**	**Q5**	**Q8**	**Q11**
8	12→**16**	11→13→**14**	16→**20**	14→15→**16**	**10**	12→22→**28**
12	**16**	**14**	18→**20**	**16**	**10**	22→**28**
16	**16**	12→**14**	**20**	**16**	8→**10**	26→**28**
20	**16**	13→**14**	**20**	14→**16**	8→**10**	**28**
24	**16**	**14**	**20**	14→**16**	8→**10**	**28**
28	**16**	**14**	**20**	13→**16**	8→**10**	**28**

4.4.4 Convergence

We now show DS2 convergence from both over- and under-provisioned states on more complex dataflows. We use the same Flink configuration as before, and execute each query with fixed source rates and initial configurations of varying parallelism. The source rates for each query are specified in Table 4.2. We run each query-configuration combination for 5 minutes and evaluate DS2 with 30 s decision interval, 30 s warm-up time, 1.0 target ratio, and five intervals activation. We consider the policy to have converged if the decision is unchanged over 5 consecutive intervals.

Table 4.3 shows the indicated parallelism per decision step for the main operator of each query on Flink. Note that queries Q3, Q5, Q8, and Q11 include many operators, but we show results for the main operator of each for simplicity. DS2 converges in one step for simple queries and initial configurations close to optimal (e.g., Q1 with parallelism 12), and in at most three steps for complex queries and initial configurations far from optimal (e.g., Q5 with initial parallelism 8).

In all cases, DS2 takes at most three steps to converge. It needed three steps in 3 experiments (with Q2, Q5, and Q11), two steps in 14 experiments, and a single step in 19 out of 36 total experiments. We also ran the same queries using Timely Dataflow and the results were similar.

This shows that DS2 provides two important SASO properties: *stability* and *short settling time*.

Intuitively, one DS2 step moves close to optimal by estimating ideal linear scaling (Section 4.2.4). For far-from-optimal initial configurations, the second step "refines" this decision with a more accurate measurement, and the third step compensates for uncaptured overheads.

4.4.5 Accuracy

We next show accuracy: DS2 converges to configurations that exhibit no backpressure (and thus keep up with the source rates) while minimizing resource usage. In particular, we show that for a given dataflow, fixed input rate, and initial configuration, DS2 identifies the optimal parallelism regardless of whether the job is initially under- or over-provisioned. We further show that there exists no other backpressure-free configuration with lower parallelism than the one DS2 computes. Finally, we show that this configuration gives low latency by minimizing waiting time per operator instance.

We set source rates as in Table 4.2 and parallelism given by the convergence experiment. Figures 4.7 and 4.8 plot observed source rates (left) and per-record latency (right) for the main operator of each Nexmark query on Flink with different configurations. For queries with two sources, Q3 in Figure 4.7c and Q8 in Figure 4.8b, we show results for the higher-rate source (results for the low-rate sources are similar). In all cases, DS2 successfully identifies the lowest parallelism that can keep up with the source rate. Further increasing the parallelism does not significantly improve latency and would waste resources, while lower parallelism would cause backpressure.

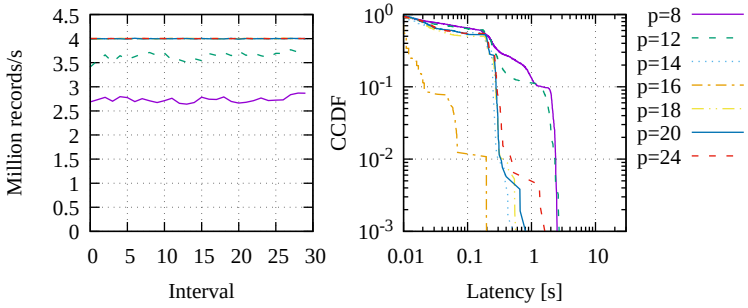
Timely Dataflow does not have a backpressure mechanism so data sources are never delayed and the observed source rates are always equal to the initial fixed rate (instead, queues grow when the system cannot keep up). We therefore simply show complementary cumulative distribution functions (CCDFs) of per-epoch latencies with different configurations for Timely. Figure 4.9 shows these for Q3, Q5, and Q11; results are similar for other queries. Each epoch in the CCDFs corresponds to 1 second of data, which must be processed in less than 1 second. The optimal parallelism indicated by DS2 is four in all queries, regardless of the starting configuration. For Q3 (left) and Q11 (right), a parallelism of four is clearly the configuration that can keep up with the 1-second target (vertical line in the plots) using minimum required resources. For Q5, 18% of the epochs are above the target by up to 0.5 seconds. Here, the larger percentage of epochs that cannot keep up is because of the window operator, which stashes data and then forwards it at certain time points. This manifests as load spikes, which require additional resources for the system to keep up. Longer decision intervals smooth out the spikes but tend to affect policy decisions towards higher optimal configurations, which is why DS2 indicated a parallelism of four (cf. Section 4.3.2).

In summary, DS2 identified optimal configurations in all experiments and never overshot (provisioned more resources than needed), thereby exhibiting the remaining two SASO properties: *accuracy* and *no overshoot*.

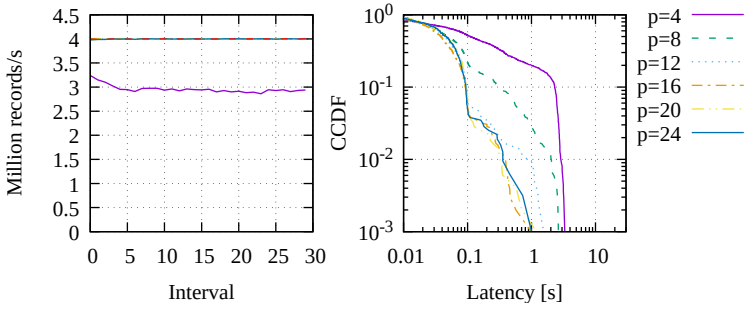
4.4.6 Instrumentation overhead

Finally, we evaluate instrumentation overhead. We run the Nexmark queries for 5 minutes with source rates from Table 4.2 and a 10 s decision interval—the smallest we use in this work, which results in the most frequently aggregated logs and has the highest potential overhead on the system performance.

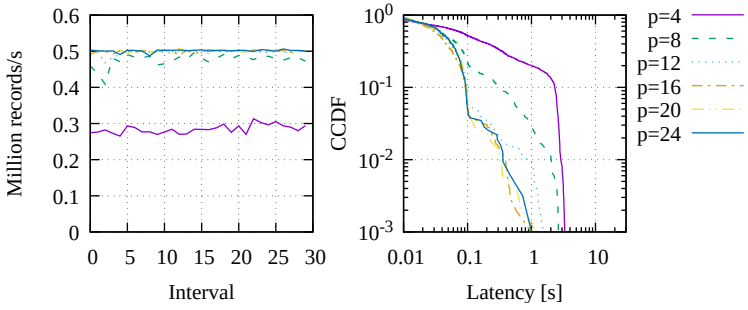
We measure per-record latency in Flink using its built-in metric and per-epoch latency in Timely Dataflow using 1 s event-time epochs. Figure 4.10 shows boxplots for both systems. Individual columns show latency with logging completely off (vanilla) and instrumentation activated (instr). Overheads are small: at most 13% on Flink (40 ms absolute difference) and at most 20% on Timely Dataflow (5 ms absolute difference) across all queries. Performance penalties are an acceptable trade-off for a good scaling policy, and could be further reduced with a larger decision interval and pre-aggregation of metrics. Note that Heron incurs no overhead since it gathers the required metrics by default.



(a) [Q1] Indicated parallelism: 16

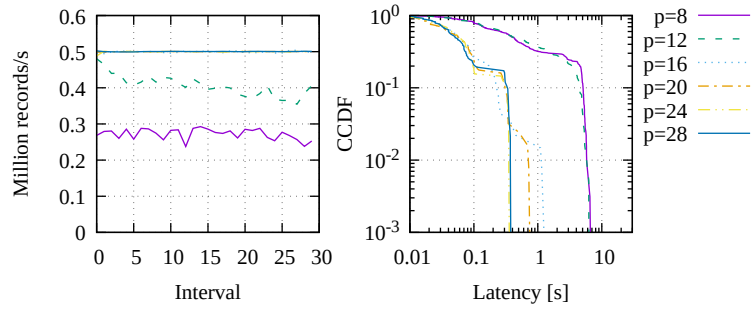


(b) [Q2] Indicated parallelism: 14

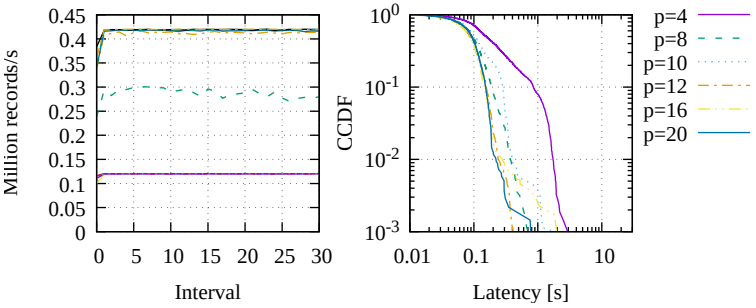


(c) [Q3] Indicated parallelism: 20

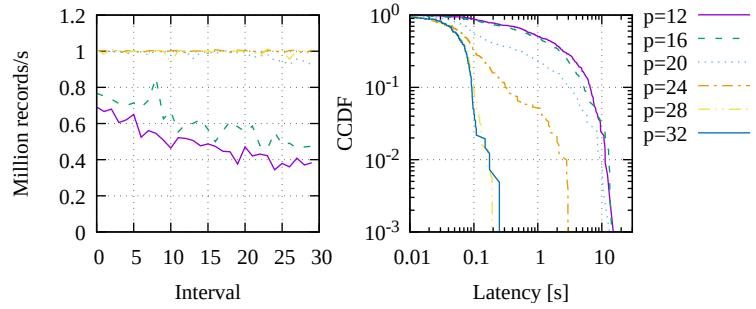
Figure 4.7: Observed source output rates and per-record latency CCDFs for different configurations of the Nexmark operators Q1–Q3 on Apache Flink.



(a) [Q5] Indicated parallelism: 16



(b) [Q8] Indicated parallelism: 10



(c) [Q11] Indicated parallelism: 28

Figure 4.8: Observed source output rates and per-record latency CCDFs for different configurations of the Nexmark operators Q1, Q8 and Q11 on Apache Flink.

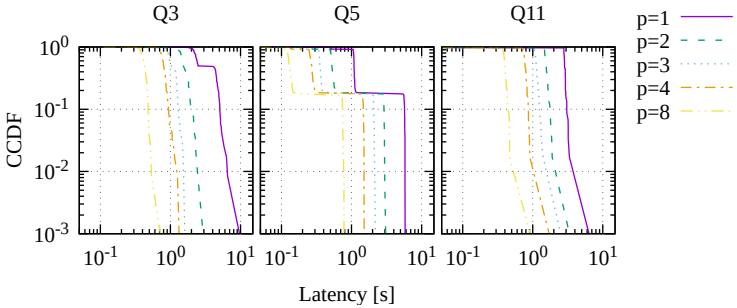
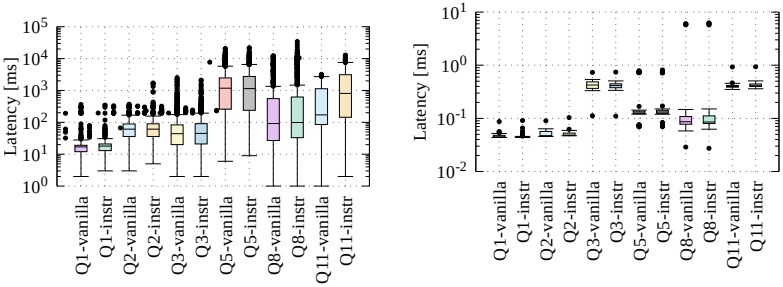


Figure 4.9: CCDFs of per-epoch latencies for different configurations of the Nexmark operators on Timely Dataflow. All queries have an indicated parallelism of 4.



(a) Flink instrumentation overhead

(b) Timely Dataflow instrumentation overhead

Figure 4.10: Policy instrumentation overhead for the Nexmark queries of Table 4.2 with instrumentation disabled (vanilla) and enabled (instr) for both Flink (a) and Timely Dataflow (b). Note the different scale of the y-axis.

4.5 DS2: conclusion

In this chapter we have described and evaluated DS2, a novel automatic scaling controller for distributed streaming dataflows. Unlike existing scaling approaches, which rely on coarse-grained metrics and simplistic models, DS2 leverages knowledge of the dataflow graph, the computational dependencies among operators, and estimates the operators' true processing and output rates.

DS2 uses a general performance model that is mechanism-agnostic and broadly applicable to a range of streaming systems. We have implemented DS2 on different stream processing engines: Apache Flink, Timely Dataflow, and Apache Heron, and showed that it is capable of accurate scaling decisions with fast convergence, while incurring negligible instrumentation overheads.

An interesting question for future work is what kind of scaling and adaptation mechanisms are a good match for a controller like DS2. The efficiency of DS2's model means that responsiveness is often limited by the latency of the scaling mechanism of the stream processor (when it is not determined by the granularity of measurement). All the stream processors we test against implement scaling actions by checkpointing the dataflow, redeploying, and restoring from the checkpoint. A faster, more dynamic reconfiguration mechanism might allow DS2 to operate on shorter timescales than the tens of seconds it allows in current systems. We present a reconfiguration mechanism with low latency impact in Chapter 5. DS2 is open source, together with all code and data used to produce the results in this chapter.[1]

[1] `https://github.com/strymon-system/ds2`

5

Megaphone: Latency-conscious state migration for distributed streaming dataflows

This chapter is based on the paper Megaphone: Latency-conscious state migration for distributed streaming dataflows *[Hof+19].*

In this chapter we present Megaphone, a technique for fine-grained migration in a stream processor which delivers maximum latencies orders of magnitude lower than existing techniques, based on the observation that a stream processor's structured computation and logical timestamps allow the system to *plan* fine-grained migrations. Megaphone can specify migrations on a key-by-key basis, and then optimizes this by batching at varying granularities; as Figure 5.1 shows, the improvement over all-at-once migration can be dramatic.

Applying existing fine-grained live migration techniques to a streaming engine is non-trivial. While systems like Squall target online transaction processing (OLTP) workloads with short-lived transactions, streaming jobs are long-running. In such a setting, Squall's approach to acquire a global lock during initialization is not a viable solution. Furthermore, many of Squall's remedies are reactive rather than proactive (because it has to support general transactions whose data needs are hard to anticipate), which can introduce significant latency on the critical path.

The core idea behind Megaphone's migration mechanism is to multiplex *fine-grained* state migration with actual data processing, coordinated using logical timestamps common in stream processors. This is a proactive approach to migration that relies on the prescribed structure of streaming computations, and the ability of stream processors to coordinate with high frequency using

logical timestamps. Such systems, including Megaphone, avoid the need for system-wide locks by pre-planning the rendezvous of data at specific workers.

Our main contribution is *fluid migration* for stateful streaming dataflows: a state migration technique that enables consistent online reconfiguration of streaming dataflows and smoothens latency spikes without using additional resources (Section 5.1) by employing fine-grained planning and coordination through logical timestamps. Additionally, we design and implement an API for reconfigurable stateful *timely dataflow* dataflow operators that enables fluid migration to be controlled simply through additional dataflow streams rather than through changes to the dataflow runtime itself (Section 5.2). Finally, we show that Megaphone has negligible steady-state overhead and enables fast direct state movement using the NEXMark benchmarks suite and microbenchmarks (Section 5.3).

Megaphone is built on *Timely Dataflow* [McS], and is implemented purely in library code, requiring no modifications to the underlying system. We first review existing state migration techniques in streaming systems, which either cause performance degradation or require resource overprovisioning. We also review live migration in DBMSs and identify the technical challenges to implement similar methods in distributed stream processors (Section 2.7).

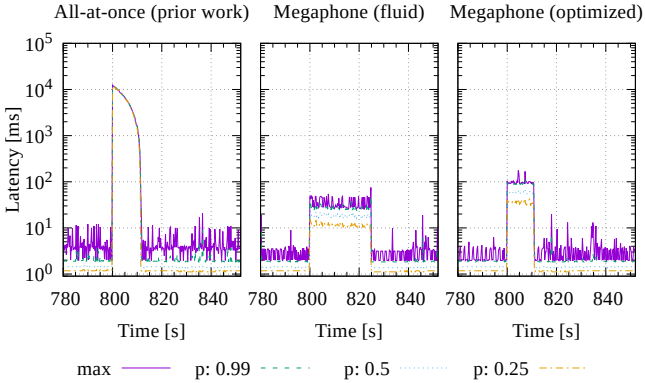


Figure 5.1: A comparison of service latencies in prior coarse-grained migration strategies (all-at-once) with two of Megaphone's fine-grained migration strategies (fluid and optimized), for a workload that migrates one billion keys consisting of 8GB of data.

Table 5.1: Notation used throughout Chapter 5

Symbol	Description
$\text{operator}_{\text{key}}$	User-supplied operator function definition
configuration	Function assigning keys to workers based on time
update	Changes to the configuration function
F	Operator managing routing and state locality
S	Operator hosting a user-supplied operator
$P(t)$	The state associated with a specific key at time t
S_a	An operator instance named a
C	A configuration at a specific point in time

5.1 State migration design

Megaphone's features rely on core streaming dataflow concepts such as logical time, progress tracking, data-parallel operators, and state management. Basic implementations of these concepts are present in all modern stream processors, such as Apache Flink [Car+15], Millwheel [Aki+13], and Google Dataflow [Aki+15]. In the following, we rely on the Naiad [Mur+13] *timely dataflow* model as the basis to describe the Megaphone migration mechanism. Timely dataflow natively supports a superset of dataflow features found in other systems in their most general form.

5.1.1 Migration formalism and guarantees

We will use *timely dataflow* frontiers to separate migrations into independent arbitrarily fine-grained timestamps and logically coordinate data movement without using coarse-grained pause-and-resume for parts of the dataflow.

To frame the mechanism we introduce for live migration in streaming dataflows, we first lay out some formal properties that define correct and live migration. In the interest of clarity we keep the descriptions casual, but each can be formalized.

We consider stateful dataflow operators that are *data-parallel* and *functional*. Specifically, an operator acts on input data that are structured as (key, val) pairs, each bearing a logical timestamp. The input is partitioned by its *key* and the operator acts independently on each input partition by sequentially applying each *val* to its state in timestamp order. For each *key*, for each *val* in

timestamp order, the operator may change its per-key state arbitrarily, produce arbitrary outputs as a result, and it may schedule further per-key changes at future timestamps (in effect sending itself a new, post-dated *val* for this *key*).

$$\text{operator}_{key}\colon (state, val) \rightarrow (state', [outputs], [(vals, times)]) \tag{5.1}$$

The output triples are the new state, the outputs to produce, and future changes that should be presented to the operator.

For a specific *operator*, we can describe the correctness of an implementation. We introduce the notation of *in advance of* as follows.

Definition 17 (in advance of)**.** *A timestamp t is **in advance of***

1. *a timestamp t' if t is greater than or equal to t';*
2. *a frontier F if t is greater than or equal to an element of F.*

In-advance-of corresponds to the less-or-equal relation for partially ordered sets. For example, a time 6 is in advance of 5.

Property 5 (Correctness)**.** *The **correct outputs through time** are the time-stamped outputs that result from each key from the timestamp-ordered application of input and post-dated records bearing timestamps not in advance of time.*

For each migrateable operator, we also consider a *configuration* function, which for each timestamp assigns each key to a specific worker.

$$\text{configuration}\colon (time, key) \rightarrow worker \tag{5.2}$$

With a specific *configuration*, we can describe the correctness of a migrating implementation.

Property 6 (Migration)**.** *A computation is **migrated according to*** **configuration** *if all updates to* key *with timestamp* time *are performed at worker* configuration(time, key).

A configuration function can be represented in many ways, which we will discuss further. In our context we will communicate any changes using a *timely dataflow* stream, in which configuration changes bear the logical timestamp of their migration. This choice allows us to use *timely dataflow*'s frontier mechanisms to coordinate migrations, and to characterize liveness.

Property 7 (Completion (liveness))**.** *A migrating computation is* ***completing*** *if, once the frontiers of both the data input stream and configuration update stream reach F, then (with no further requirements of the input) the output frontier of the computation will eventually reach F.*

Our goal is to produce a mechanism that satisfies each of these three properties: Correctness, Migration, and Completion.

5.1.2 Configuration updates

State migration is driven by updates to the *configuration* function (Equation (5.2)) introduced in Section 5.1.1. In Megaphone these updates are supplied as data along a *timely dataflow* stream, each bearing the logical timestamp at which they should take effect. Informally, configuration updates have the form

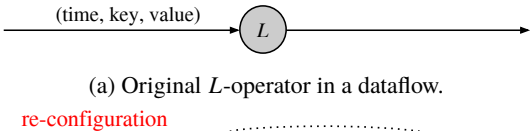
$$\text{update}\colon (\textit{time}, \textit{key}, \textit{worker}) \tag{5.3}$$

indicating that as of *time* the state and values associated with *key* will be located at *worker*, and that this will hold until a new update to *key* is observed with a greater timestamp. For example, an update could have the form of (*time:* 16, *key:* a, *worker:* 0), which would define the configuration function for times of 16 and beyond.

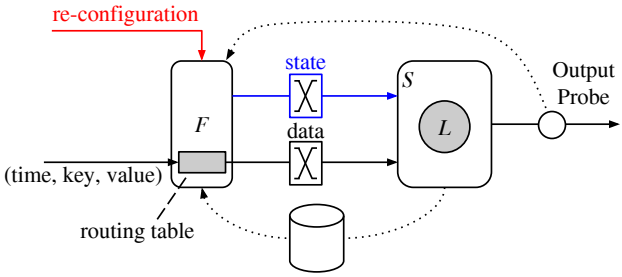
As configuration updates are simply data, the user has the ability to drive a migration process by introducing updates as they see fit. In particular, they have the flexibility to break down a large migration into a sequence of smaller migrations, each of which have lower duration and between which the system can process data records. For example, to migrate from one configuration C_1 to another C_2, a user can use different migration strategies to reveal the changes from C_1 to C_2:

All-at-once migration To simultaneously migrate all keys from C_1 to C_2, a user could supply all changed (*time*, *key*, *worker*) triples with one common *time*. This is essentially an implementation of the *partial pause-and-restart* migration strategy of existing streaming systems as described in Section 2.7.1.

Fluid migration To smoothly migrate keys from C_1 to C_2, a user could repeatedly choose one key changed from C_1 to C_2, introduce the new (*time*, *key*, *worker*) triple with the current *time*, and await the migration's completion before choosing the next key.



(a) Original L-operator in a dataflow.



(b) Megaphone's operator structure in a dataflow.

Figure 5.2: Overview of Megaphone's migration mechanism. The solid lines indicate regular streams. The dotted lines indicate information sharing for the state, and progress information obtained from the probe.

Batched migration To trade off low latency against high throughput, a user can produce batches of changed (*time*, *key*, *worker*) triples with a common *time*, awaiting the completion of the batch before introducing the next batch of changes.

We believe that this approach to reconfiguration, as user-supplied data, opens a substantial design space. Not only can users perform fine-grained migration, they can prepare future migrations at specific times, and drive migrations based on *timely dataflow* computations applied to system measurements. Most users will certainly need assistance in performing effective migration, and we will evaluate several specific instances of the above strategies.

5.1.3 Megaphone's mechanism

We now describe how to create a migrateable version of an operator L by implementing a deterministic, data-parallel *operator* as described in Section 5.1.1. A non-migrateable implementation would have a single dataflow operator with a single input dataflow stream of (*key*, *val*) pairs, exchanged by *key* before they arrive at the operator.

Instead, we create two operators F and S. F takes the data stream as input and as an additional input the stream of configuration updates and produces data pairs and migrating state as outputs. S takes as inputs exchanged data pairs and exchanged migrating state, and applies them to a hosted instance of L, which implements *operator* and maintains both state and pending records for each key. Figure 5.2b presents a schematic overview of the construction. Recall that in timely dataflow instances of all operators in the dataflow are multiplexed on each worker (thread). The F and S on the same worker share access to L's state.

This construction can be repeated for all the operators in the dataflow that need support for migration. Separate operators can be migrated independently (via separate configuration update streams), or in a coordinated manner by re-using the same configuration update stream. Operators with multiple data inputs can be treated like single-input operators where the migration mechanism acts on both data inputs at the same time.

Operator F Operator F routes (*key*, *val*) pairs according to the configuration at their associated *time*, buffering pairs if *time* is in advance of the frontier of the configuration input. For times in advance of this frontier, the configuration is not yet certain as further configuration updates could still arrive. The configuration at times not in advance of this frontier can no longer be updated. As the data frontier advances, configurations can be retired.

Operator F is also responsible for initiating state migrations. For a configuration update (*time*, *key*, *worker*), F must not initiate a migration for *key* until its state has absorbed all updates at times strictly less than *time*. F initiates a migration once *time* is present in the output frontier of S, as this implies that there exist no records at timestamps less than *time*, as otherwise they would be present in the frontier in place of *time*.

Operator F initiates a migration by uninstalling the current state for *key* from its current location in operator S, and transmitting it bearing timestamp *time* to the instance of operator S on *worker*. The state includes both the state for *operator*, as well as the list of pending (*val*, *time*) records produced by *operator* for future times.

Operator S Operator S receives exchanged (*key*, *val*) pairs and exchanged state as the result of migrations initiated by F. S immediately installs any received state. S applies received and pending (*key*, *val*) pairs in timestamp

order using *operator* once their timestamp is not in advance of either the data or state inputs.

We provide details of Megaphone's implementation of this mechanism in Section 5.2.

Proof sketch For each key, the function $operator_{key}$ from Equation (5.1) defines a timeline corresponding to a single-threaded execution, which assigns to each time a pair $(state, [(val, time)])$ of state and pending records *just before* the application of input records at that time. Let $P(t)$ denote the function from times to these pairs for *key*.

For each key, the *configuration* function from Equation (5.2) partitions this timeline into disjoint intervals, $[t_a, t_b)$, each of which is assigned to one operator instance S_a.

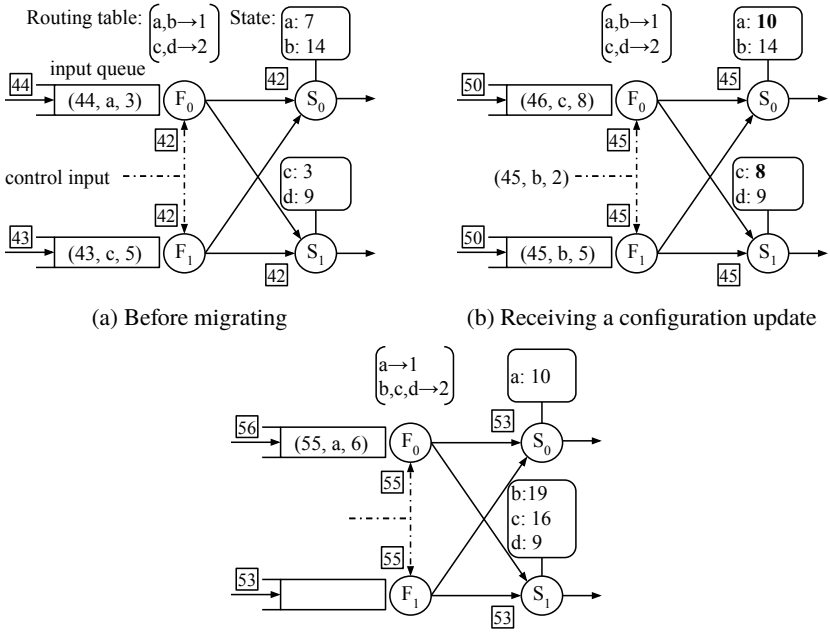
Claim: F migrates exactly $P(t_a)$ to S_a.

First, F always routes input records at $time$ to S_a, and so routes all input records in $[t_a, t_b)$ to S_a. If F also presents S_a with $P(t_a)$, it has sufficient input to produce $P(t_b)$. More precisely,

1. because F maintains its output frontier at t_b, in anticipation of the need to migrate $P(t_b)$, S_a will apply no input records in advance of t_b. And so, it applies exactly the records in $[t_a, t_b)$.
2. Until S_a transitions to $P(t_b)$, its output frontier will be strictly less than t_b, and so F will not migrate anything other than $P(t_b)$.
3. Because F maintains its output frontier at t_b, and S_a is able to advance its output frontier to t_b, the time t_b will eventually be *in* the output frontier of S.

5.1.4 Example

Figure 5.3 presents three snapshots of a migrating streaming word-count dataflow. The figure depicts operator instances F_0 and F_1 of the upstream routing operator, and operator instances S_0 and S_1 of the operator instances hosting the word-count state and update logic. The F operators maintain input queues of received but not yet routable input data, and an input stream of logically timestamped configuration updates. Although each F maintains its own routing table, which may temporarily differ from others, we present one for clarity. Input frontiers are represented by boxed numbers, and indicate timestamps that may still arrive on that input.



(a) Before migrating

(b) Receiving a configuration update

(c) After migration

Figure 5.3: A migrating word-count dataflow executed by two workers. The example is explained in more detail in Section 5.1.4

In Figure 5.3a, F_0 has enqueued the record (44, a, 3) and F_1 has enqueued the record (43, c, 5), both because their control input frontier has only reached 42 and so the destination workers at their associated timestamps have not yet been determined. Generally, F instances will only enqueue records with timestamps in advance of the control input frontier, and the output frontiers of the S instances can reach the minimum of the data and control input frontiers.

In Figure 5.3b, both control inputs have progressed to 45. The buffered records (44, a, 3) and (43, c, 5) have been forwarded to S_1 and S_2, and the count operator instances apply the state updates accordingly, shown in bold. Additionally, both operators have received a configuration update for the key b at time 45. Should the configuration input frontier advance beyond 45, both F_0 and F_1 can integrate the configuration change, and then react. Operator F_0 would observe that the output frontier of S_0 reaches 45, and initiate a state migration. Operator F_1 would route its buffered input at time 45, to S_1 rather than S_0.

In Figure 5.3c the migration has completed. Although the configuration frontier has advanced to 55, the output frontiers are held back by the data input frontier of F_1 at 53. According to Definition 4, the frontier guarantees that no record with a time earlier than 53 will appear at the input. If the configuration frontier advances past 55 then operator F_0 could route its queued record, but neither S operator could apply it until they are certain that there are no other data records that could come before the record at 55.

5.2 Implementation

Megaphone is an implementation of the migration mechanism described in Section 5.1. In this section, we detail specific choices made in Megaphone's implementation, including the interfaces used by the application programmer, Megaphone's specific choices for the grouping and organization of per-key state, and how we implemented Megaphone's operators in Timely Dataflow. We conclude with some discussion of how one might implement Megaphone in other stream processing systems, as well as alternate implementation choices one could consider.

5.2.1 Megaphone's operator interface

Megaphone presents users with an operator interface that closely resembles the operator interfaces Timely Dataflow presents. In several cases, users can use the same operator interface extended only with an additional input stream for configuration updates. More generally, we introduce a new structure to help users isolate and surface all information that must be migrated (state, but also pending future records). These additions are implemented strictly above Timely Dataflow, but their structure is helpful and they may have value in Timely Dataflow proper.

The simplest stateful operator interface in Megaphone and Timely Dataflow is the `state_machine` operator, which takes one input structured as pairs (*key*, *val*) and a state update function which can produce arbitrary output as it changes per-key state in response to keys and values. In Megaphone, there is an additional input for configuration updates, but the operator signature is otherwise identical.

More generally, Timely Dataflow supports operators of arbitrary numbers and types of inputs, containing arbitrary user logic, and maintaining arbitrary state. In each case a user must specify a function from input records to integer keys, and the only guarantee Timely Dataflow provides is that records with the same key are routed to the same worker. Operator execution and state are partitioned by worker, but not necessarily by key.

For Megaphone to isolate and migrate state and pending work we must encourage users to yield some of the generality Timely Dataflow provides. However, Timely Dataflow has already required the user to program partitioned operators, each capable of hosting multiple keys, and we can lean on these idioms to instantiate more fine-grained operators, partitioned not only by worker but further into finer-grained *bins* of keys. Routing functions for each input are already required by Timely Dataflow, and Megaphone interposes to allow the function to change according to reconfiguration. Timely Dataflow per-worker state is defined implicitly by the state captured by the operator closure, and Megaphone only makes it more explicit. The use of a helper to enqueue pending work is borrowed from an existing Timely Dataflow idiom (the `Notificator`). While Megaphone's general API is not identical to that of Timely Dataflow, it is a similar framing of the same idioms. From our experience, a programmer familiar with Timely Dataflow can adapt to Megaphone's API in little time.

Listing 5.1 shows how Megaphone's operator interface is structured. The interface declares unary and binary stateful operators for single input or dual

input operators as well as a state-machine operator. The logic for the state-machine operator has to be encoded in the `fold`-function. Megaphone presents data in timestamp order with a corresponding state and notificator object. Here, migration is transparent and performed without special handling by the operator implementation.

Example Listing 5.2 shows an example of a stateful word-count dataflow with a single data input and an additional `control` input. The `stateful_unary` operator receives the `control` input, the state type, and a key extraction function as parameters. The control input carries information about where data is to be routed as discussed in the previous section. During migration, the state object is converted into a stream of serialized tuples, which are used to reconstruct the object on the receiving worker. State is managed in groups of keys, i.e. many keys of input data will be mapped to the same state object. The key extraction function defines how this key can be extracted from the input records.

5.2.2 State organization

State migration as defined in Section 5.1.1 is defined on a per-key granularity. In a typical streaming dataflow, the number of keys can be large in the order of million or billions of keys. Managing each key individually can be costly and thus we selected to group keys into *bins* and adapt the *configuration* function from Equation (5.2) as follows:

$$\mathit{configuration} : (\mathit{time}, \mathit{bin}) \rightarrow \mathit{worker}.$$

Additionally, each key is statically assigned to one equivalence class that identifies the bin it belongs to.

In Megaphone, the number of bins is configurable in powers of two at startup but cannot be changed during run-time. A stateful operator gets to see a bin that holds data for the equivalence class of keys for the current input. Bins are simply identified by a number, wich corresponds to the most significant bits of the exchange function specified on the operator.[1]

[1] Otherwise, keys with similar least-significant bits are mapped to the same bin; Rust's `HashMap`-implementation suffers from collisions for keys with similar least-significant bits.

```
fn state_machine(
  control: Stream<ControlInstr>,
  input: Stream<(K, V)>,
  exchange: K -> Integer
  fold: |Key, Val, State| -> List<Output>,
) -> Stream<Output>;

fn unary(
  control: Stream<ControlInstr>,
  input: Stream<Data>,
  exchange: Data -> Integer,
  fold: |Time, Data, State, Notificator| -> List<Output>,
) -> Stream<Output>;

fn binary(
  control: Stream<ControlInstr>,
  input1: Stream<Data1>,
  input2: Stream<Data2>,
  exchange1: Data1 -> Integer,
  exchange2: Data2 -> Integer,
  fold: |Time, Data1, Data2, State, Notificator1, Notificator2|
      -> List<Output>,
) -> Stream<Output>;
```

Listing 5.1: Abstract definition of the Megaphone operator interfaces. Arguments `State` and `Notificator` are provided as mutable references which can be operated upon. Parameters in angle brackets indicate generic parameters, such as the data type of a stream. Function types are declared with the `->` operator, with parameters on the left and return type on the right.

```
worker.dataflow(|scope| {

  // Introduce configuration and input streams.
  let conf = conf_input.to_stream(scope);
  let text = text_input.to_stream(scope);

  // Update per-word accumulate counts.
  let count_stream = megaphone::unary(
    conf,
    text,
    |(word, diff)| hash(word),
    |time, data, state, notificator| {
      // map each (word, diff) pair to the accumulation.
      data.map(|(word, diff)| {
        let mut count = state.entry(word).or_insert(0);
        *count += diff;
        (word, count)
      })
    }
  );
});
```

Listing 5.2: A stateful word-count operator. The operator reads (*word*, *diff*)-pairs and outputs the accumulated count of each encountered word. For clarity, the example suppresses details related to Rust's data ownership model. The complete example is available in Megaphone's source release.

Megaphone's mechanism requires two distinct operators, F and S. The operator S maintains the bins local to a worker and passes references to the user logic L. Nevertheless, the S-operator does not have a direct channel to its peers. For this reason, F can obtain a reference to bins by means of a shared pointer. During a migration, F serializes the state obtained via the shared pointer and sends it to the new owning S-operator via a regular Timely Dataflow channel. Note that sharing a pointer between two operators requires the operators to be executed by the same process (or thread to avoid synchronization), which is the case for Timely Dataflow.

5.2.3 Timely Dataflow instantiation

In Timely Dataflow, data is exchanged according to an *exchange* function, which takes some data and computes an integer representative value:

$$exchange : data \rightarrow Integer.$$

Timely dataflow uses this value to decide where to send tuples. In Megaphone, instead of assigning data to a worker based on the exchange function, we introduce an indirection layer where bins are assigned to workers. That way, the exchange function for the channels from F to S is by a specific worker identifier.

5.2.3.1 Monitoring output frontiers

Megaphone ties migrations to logical time and a computation's progress. A reconfiguration at a specific time is only to be applied to the system once all data up to that time has been processed. The F operators access this information by monitoring the output frontier of the S operators. Specifically, Timely Dataflow supports *probes* as a mechanism to observe progress on arbitrary dataflow edges. Each worker attaches a probe to the output stream of the S operators, and provides the probe to its F operator instance.

5.2.3.2 Capturing Timely Dataflow idioms

For Megaphone to migrate state, it requires clear isolation of per-key state and pending records. Although Timely Dataflow operators require users to write operators that can be partitioned across workers, they do not require the state and pending records to be explicitly identified. To simplify programming

migrateable operators, we encapsulate several Timely Dataflow idioms in a helper structure that both manages state and pending records for the user, and surfaces them for migration.

Timely Dataflow has a `Notificator` type that allows an operator to indicate future times at which the operator may produce output, but without encapsulating the keys, states, or records it might use. We implemented an extended notificator that buffers future triples (*time*, *key*, *val*) and can replay subsets for times not in advance of an input frontier. Internally the triples are managed in a priority queue, unlike in Timely Dataflow, which allows Megaphone to efficiently maintain large numbers of future triples. By associating data (keys, values) with the times, we relieve the user from maintaining this information on the side. As we will see, Megaphone's notificator can result in a net reduction in implementation complexity, despite eliciting more information from the user.

5.2.4 Discussion

Up to now, we explained how to map the abstract model of Megaphone to an implementation. The model leaves many details to the implementation, several of which have a large effect on an implementation's run-time performance. Here, we want to point out how they interact with other features of the underlying system, what possible alternatives are and how to integrate Megaphone into a larger, controller-based system.

Other systems We implemented Megaphone in Timely Dataflow, but the mechanisms could be implemented in any sufficiently expressive stream processor with support for event time, progress tracking, and state management. Specifically, Megaphone relies on the ability of F operators to 1. observe timestamp progress at other locations in the dataflow, and 2. to extract state from downstream S operators for migration. With regard to the first requirement, systems with out-of-band progress tracking like *MillWheel* [Aki+13] and Google Dataflow [Aki+15] also provide the capability to observe dataflow progress externally, while systems with in-band watermarks like Flink would need to provide an additional mechanism. Extracting state from downstream operators is straightforward in Timely Dataflow where workers manage multiple operators. In systems where each thread of control manages a single

operator, external coordination mechanisms could be used to effect the same behavior.

Fault tolerance Megaphone is a library built on Timely Dataflow abstractions, and inherits fault-tolerance guarantees from the system. For example, the Naiad implementation of *timely dataflow* provides system-wide consistent snapshots, and a Megaphone implementation on Naiad would inherit fault tolerance. At the same time, Megaphone's migration mechanisms effectively provide programmable snapshots on finer granularities, which could feed back into finer-grained fault-tolerance mechanisms.

Reconfiguration with stateful operators and fault-tolerance both require operators to expose their internal state. The system hosting the operators can decide when to persist an operator's state or migrate an operator instance to another physical worker. The main difference between a mechanism for fault-tolerance and state-migration is that the former should be done asynchronously to limit the impact on steady-state latency while the latter is likely on the critical path and thus needs to be engineered to have a low latency impact in the computation's performance.

Alternatives to binning Megaphone's implementation uses binning to reduce the complexity of the *configuration* function. An alternative to a static mapping of keys to bins could be achieved by the means of a prefix tree (e.g., a longest-prefix match as in Internet routing tables). Extending the functionality of bins to split bins into smaller sets or merge smaller sets into larger bins would allow run-time reconfiguration of the actual binning strategy rather than setting it up during initialization without the option to change it later on.

Migration controller We implemented Megaphone as a system that provides an input for configuration updates to be supplied by an external controller. The only requirement Megaphone places on the controller is to adhere to the control command format as described in Section 5.1.2. A controller could observe the performance characteristics of a computation on a per-key level and correlate this with the input workload. For example, the DS2 system presented in Chapter 4 automatically measures and re-scales streaming systems to meet throughput targets.

Independently, we have observed and implemented several details for effective migration. Specifically, we can use bipartite matching to group migrations

that do not interfere with each other, reducing the number of migration steps without much increasing the maximum latency. We can also insert a gap between migrations to allow the system to immediately drain enqueued records, rather than during the next migration, which reduces the maximum latency from two migration durations to just one.

5.3 Evaluation

We present a tripartite evaluation of Megaphone. We are interested in particular in the latency of streaming queries, and how they are affected by Megaphone both in a steady state (where no migration is occuring) and during a migration operation.

First, in Section 5.3.1 we use the NEXMark benchmarking suite [NEX; Tuc+02] to compare Megaphone with prior techniques under a realistic workload. NEXMark consists of queries covering a variety of operators and windowing behaviors. Next, in Section 5.3.2 we look at the overhead of Megaphone when no migration occurs: this is the cost of providing migration functionality in stateful dataflow operators, versus using optimized operators which cannot migrate state. Finally, in Section 5.3.3 we use a microbenchmark to investigate how parameters like the number of bins and size of the state affect migration performance.

We run all experiments on a cluster of four machines, each with four Intel Xeon E5-4650 v2 @2.40 GHz CPUs (each 10 cores with hyperthreading) and 512 GiB of RAM, running Ubuntu 18.04. For each experiment, we pin a Timely Dataflow process with four[2] workers to a single CPU socket. Our open-loop testing harness supplies the input at a specified rate, even if the system itself becomes less responsive (e.g., during a migration). We record the observed latency every 250 ms, in units of nanoseconds, which are recorded in a histogram of logarithmically-sized bins.

Unless otherwise specified, we migrate the state of the main operator of each dataflow. We initially migrate half of the keys on half of the workers to the other half of the workers (25% of the total state), which results in an imbalanced assignment. We then perform and report the details of a second migration back to the balanced configuration.

[2]Timely Dataflow's progress tracking protocol did not scale well to configurations with large number of workers at the time of writing this chapter. Thus, we limited our evaluation to four machines and four Timely Dataflow workers.

Table 5.2: Lines of code in NEXMark query implementations. The native implementation is based on unmodified Timely Dataflow and includes manual optimizations.

Implementation	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Native	12	14	58	128	73	130	55	58
Megaphone	16	18	41	74	46	74	54	29

5.3.1 NEXMark benchmark

The NEXMark suite models an auction site in which a high-volume stream of users, auctions, and bids arrive, and eight standing queries are maintained reflecting a variety of relational queries including stateless streaming transformations (e.g., map and filter in Q1 and Q2 respectively), a stateful record-at-a-time two-input operator (incremental join in Q3), and various window operators (e.g., sliding window in Q5, tumbling window join in Q8), and complex multi-operator dataflows with shared components (Q4 and Q6).

We have implemented all eight of the NEXMark queries in both native Timely Dataflow and using Megaphone. Table 5.2 lists the lines of code for queries 1–8. *Native* is a hand-tuned implementation, *Megaphone* is implemented using the stateful operator interface. Note that the implementation complexity for the native implementation is higher in most cases as we include optimizations from Section 5.2 which are not offered by the system but need to be implemented for each operator by hand.

To test our hypothesis that Megaphone supports efficient migration on realistic workloads, we run each NEXMark query under high load and migrate the state of each query without interrupting the query processing itself. Our test harness uses a reference input data generator and increases its rate. The data generator can be played at a higher rate but this does not change certain intrinsic properties. For example, the number of active auctions is static, and so increasing the event rate decreases auction duration. For this reason, we present time-dilated variants of queries Q5 and Q8 containing large time-based windows (up to 12 hours). We run all queries with 4×10^6 updates per second. For stateful queries, we perform a first migration at 400 s and perform and report a second re-balancing migration at 800 s. We compare *all-at-once*, which is essentially equivalent to the partial pause-and-restart strategy adopted by existing systems, and *batched*, Megaphone's optimized migration strategy, which

strikes a balance between migration latency and duration. We use 2^{12} bins for Megaphone's migration; in Section 5.3.2 we study Megaphone's sensitivity to the bin count.

Figure 5.6 through 5.11 show timelines for the second migration of stateful queries Q3 through Q8. Generally, the all-at-once migrations experience maximum latencies proportional to the amount of state maintained, whereas the latencies of Megaphone's batched migration are substantially lower when the amount of state is large. The maximum latencies directly correspond to the latency observed by clients when applying a configuration update.

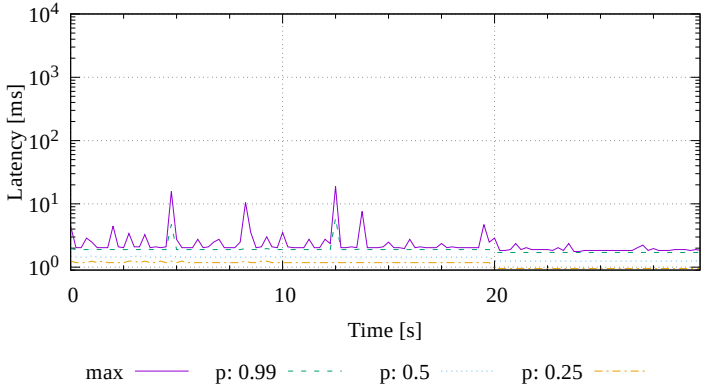


Figure 5.4: NEXMark query latency for Q1, 4×10^6 requests per second, reconfiguration at 10 s and 20 s. No latency spike occurs during migration as the query does not accumulate state.

Query 1 and Query 2 maintain no state. Q1 transforms the stream of bids to use a different currency, while Q2 filters bids by their auction identifiers. Despite the fact that both queries do not accumulate state to migrate, we demonstrate their behavior to establish a baseline for Megaphone and our test harness. Figures 5.4 and 5.5 show query latency during two migrations where no state is thus transferred; any impact is dominated by system noise.

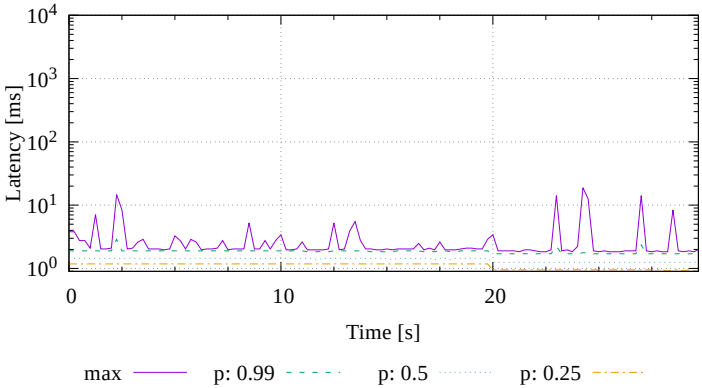
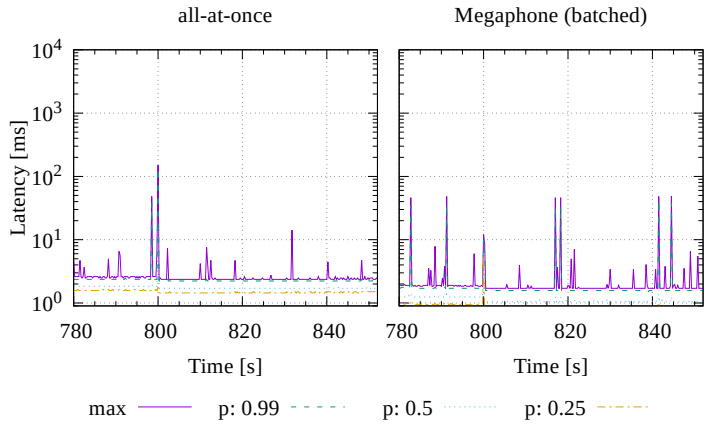


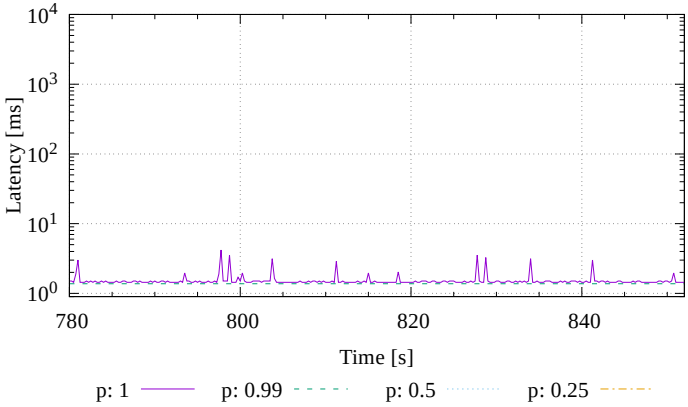
Figure 5.5: NEXMark query latency for Q2, 4×10^6 requests per second, reconfiguration at 10 s and 20 s. No latency spike occurs during migration as the query does not accumulate state.

Query 3 joins auctions and people to recommend local auctions to individuals. The join operator maintains the auctions and people relations, using the seller and person as the keys, respectively. This state grows without bound as the computation runs. Figure 5.6 shows the query latency for both Megaphone, and the native Timely Dataflow implementation. We note that while the native Timely Dataflow implementation has some spikes, they are more pronounced in Megaphone, whose tail latency we investigate further in Section 5.3.2.

Query 4 reports the average closing prices of auctions in a category relying on a stream of closed auctions, derived from the streams of bids and auctions, which we compute and maintain, and contains one operator keyed by auction id which accumulates relevant bids until the auction closes, at which point the auction is reported and removed. The NEXMark generator is designed to have a fixed number of auctions at a time, and so the state remains bounded. Figure 5.7 shows the latency timeline during the second migration. The all-at-once migration strategy causes a latency spike of more than two seconds whereas the batched migration strategy only shows an increase in latency of up to 100 ms.



(a) Query 3 implemented with Megaphone.



(b) Query 3 native implementation.

Figure 5.6: NEXMark query latency for Q3. A small latency spike can be observed at 800 s for both all-at-once and batched migration strategies, reaching more than 100 ms for all-at-once and 10 ms for batched migration. Although the state for query 3 grows without bounds, this did not bear significance after 800 s.

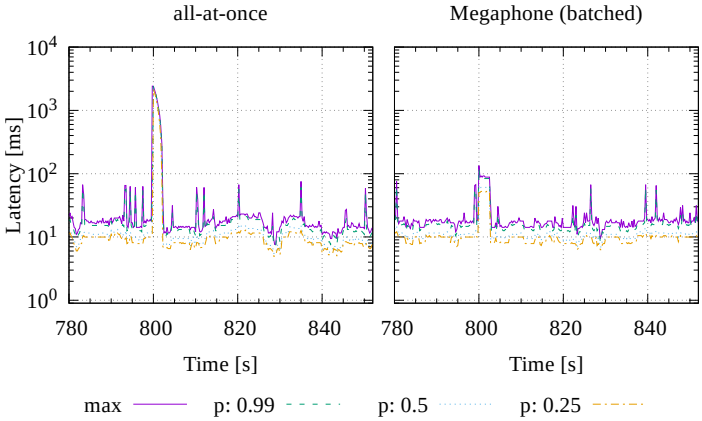


Figure 5.7: NEXMark query latency for Q4, 4×10^6 requests per second, reconfiguration at 800 s.

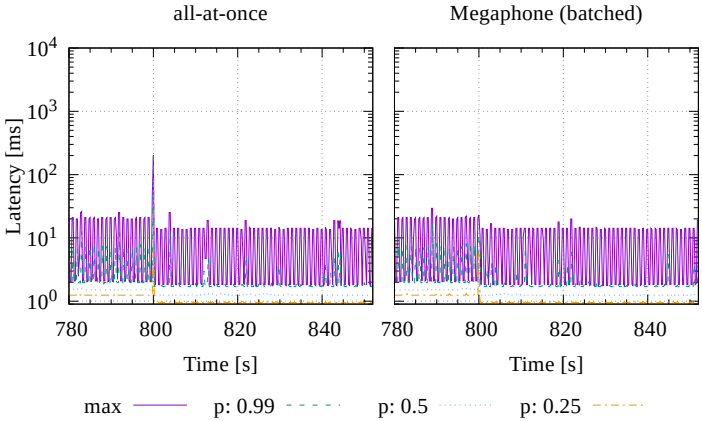


Figure 5.8: NEXMark query latency for Q5, 4×10^6 requests per second, reconfiguration at 800 s with time dilation.

Query 5 reports, each minute, the auctions with the highest number of bids taken over the previous sixty minutes. It maintains up to sixty counts for each auction, so that it can both report and retract counts as time advances. To elicit more regular behavior, our implementation reports every second over the previous minute, effectively dilating time by a factor of 60. Figure 5.8 shows the latency timeline for the second migration; the all-at-once migration is an order of magnitude larger than the per-second events, whereas Megaphone's batched migration is not distinguishable.

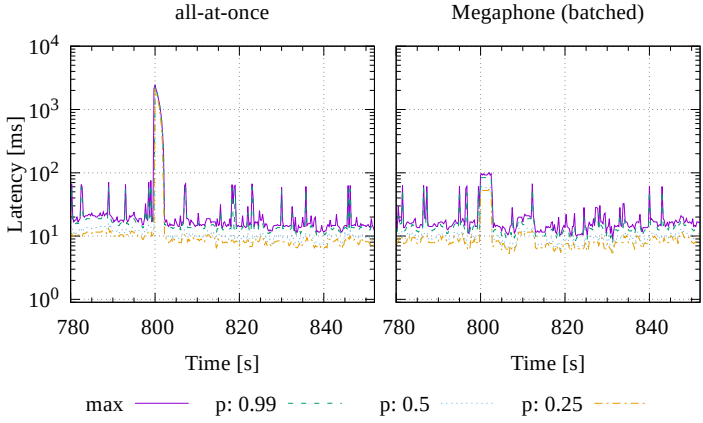


Figure 5.9: NEXMark query latency for Q6, 4×10^6 requests per second, reconfiguration at 800 s.

Query 6 reports the average closing price for the last ten auctions of each seller. This operator is keyed by auction seller, and maintains a list of up to ten prices. As the computation proceeds, the set of sellers, and so the associated state, grows without bound. Figure 5.9 shows the timeline at the second migration. The result is similar to query 4 because both have a large fraction of the query plan in common.

Query 7 reports the highest bid each minute, and the results are shown in Figure 5.10. To elicit more regular behavior, our implementation reports every ten seconds, effectively dilating time by a factor of 6. The query shows regular

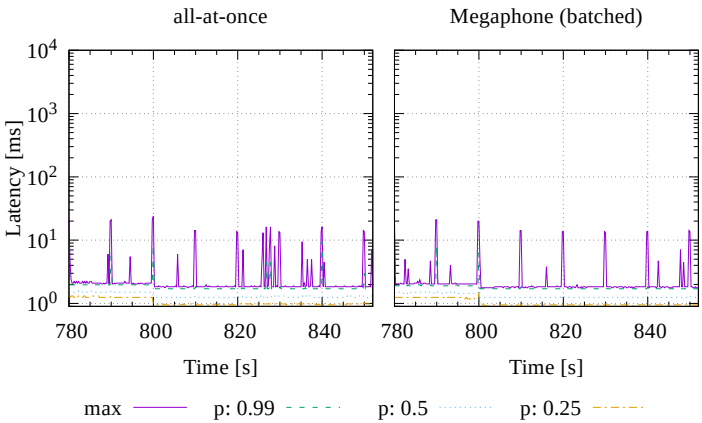


Figure 5.10: NEXMark query latency for Q7, 4×10^6 requests per second, reconfiguration at 800 s.

spikes every ten seconds, which happen when the output for completed windows is computed. This query has minimal state (one value) but does require a data exchange to collect worker-local aggregations to produce a computation-wide aggregate. Because the state is so small, there is no distinction between all-at-once and Megaphone's batched migration.

Query 8 reports a twelve-hour windowed join between new people and new auction sellers. This query has the potential to maintain a massive amount of state, as twelve hours of auction and people data is substantial. Once reached, the peak size of state is maintained. To show the effect of twelve-hour windows, we dilate the internal time by a factor of 79. The reconfiguration time of 800 s corresponds to approximately 17.5 h of event time. Despite this, as Figure 5.11 shows, very little latency is introduced into the result stream as a result of the migration at 800 seconds.

These results show that for NEXMark queries maintaining large amounts of state, all-at-once migration can introduce significant disruption, which Megaphone's batched migration can mitigate. In principle, the latency could be reduced still further with the fluid migration strategy, which we evaluate in Section 5.3.3. Some queries collect state over time, which they reduce at

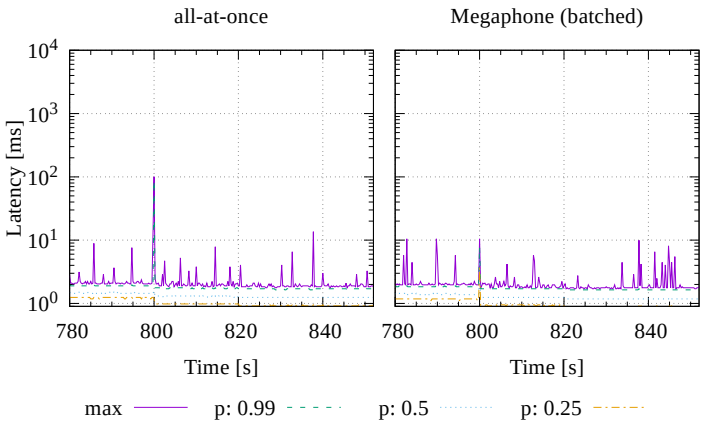


Figure 5.11: NEXMark query latency for Q8, 4×10^6 requests per second, reconfiguration at 800 s with time dilation.

regular intervals. A migration can benefit from migrating such queries after they reduce their temporary state to limit the mount of data transferred between operators.

5.3.2 Overhead of the interface

We now use a counting microbenchmark to measure the overhead of Megaphone, from which one can determine an appropriate trade-off between migration granularity and this overhead. We compare Megaphone to native Timely Dataflow implementations, as we vary the number of bins that Megaphone uses for state. We anticipate that this overhead will increase with the number of bins, as Megaphone must consult a larger routing table. For all experiments, we define the throughput and measure the latency at which results are produced.

The workload uses a stream of randomly selected 64-bit integer identifiers, drawn uniformly from a domain defined per experiment. The query reports the cumulative counts of the number of times each identifier has occurred. In these workloads, the state is the per-identifier count, intentionally small and simple so that we can see the effect of migration rather than associated computation. We consider two variants, an implementation that uses hashmaps for bins (“hash

count"), and an optimized implementation that uses dense arrays to remove hashmap computation ("key count").

Both query variants use a similar dataflow. It consists of the driver logic to produce data, and the counting operator. The input to the counting operator is partitioned so that equal keys will be routed to the same operator instance. The *native* implementation uses one operator, the query implemented on Megaphone has the additional F operator in its dataflow.

Each experiment is parameterized by a domain size (the number of distinct keys) and an input rate (in records per second), for which we then vary the number of bins used by Megaphone. Each experiment pre-loads one instance of each key to avoid measuring latency due to state re-allocation at runtime. We report the latency results using CCDFs, which show latency on the x-axis and the density on the y-axis.

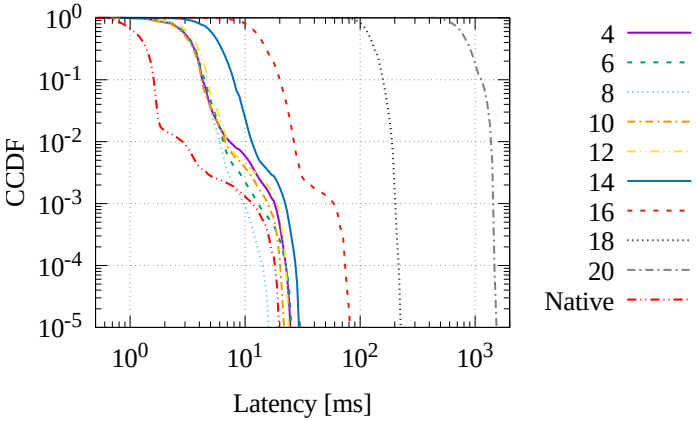
Figure 5.12 shows the CCDF of per-record latency for the hash-count experiment with 256×10^6 distinct keys and a rate of 4×10^6 updates per second. Figure 5.13 shows the CCDF of per-record latency for the key-count experiment with 256×10^6 distinct keys and a rate of 4×10^6 updates per second. Figure 5.14 shows the CCDF of per-record latency for the key-count experiment with 8192×10^6 distinct keys and a rate of 4×10^6 updates per second. Each figure reports measurements for a native Timely Dataflow implementation, and for Megaphone with geometrically increasing numbers of bins.

For small bin counts, the latencies remain a small constant factor larger than the native implementation, but this increases noticeably once we reach 2^{16} bins. We conclude that while a performance penalty exists, it can be an acceptable trade-off for flexible stateful dataflow reconfiguration. A bin-count parameter of up to 2^{12} leads to largely indistinguishable results, and we will use this number when we need to hold the bin count constant in the rest of the evaluation.

5.3.3 Migration micro-benchmarks

We now use the counting benchmark from the previous section to analyze how various parameters influence the maximum latency and duration of Megaphone during a migration. Specifically:

1. In Section 5.3.3.1 we evaluate the maximum latency and duration of migration strategies **as the number of bins increases**. We expect Megaphone's maximum latencies to decrease with more bins, without affecting duration.

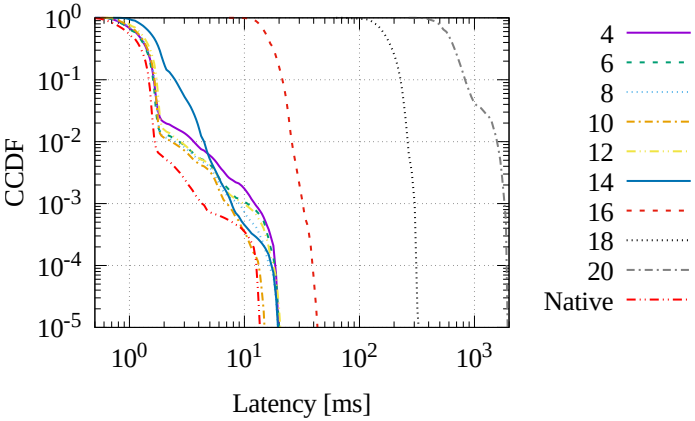


(a) CCDF of per-record latencies

Experiment	90%	99%	99.99%	max
4	4.46	7.60	18.87	25.17
6	4.46	6.55	13.11	26.21
8	4.46	6.03	9.96	16.78
10	4.19	6.82	16.25	23.07
12	4.98	7.08	19.92	24.12
14	8.13	11.53	23.07	30.41
16	20.97	27.26	60.82	83.89
18	159.38	192.94	209.72	226.49
20	1140.85	1409.29	1476.40	1543.50
Native	1.64	2.88	12.06	19.92

(b) Selected percentiles and their latency in ms

Figure 5.12: Hash-count overhead experiment with 256×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.

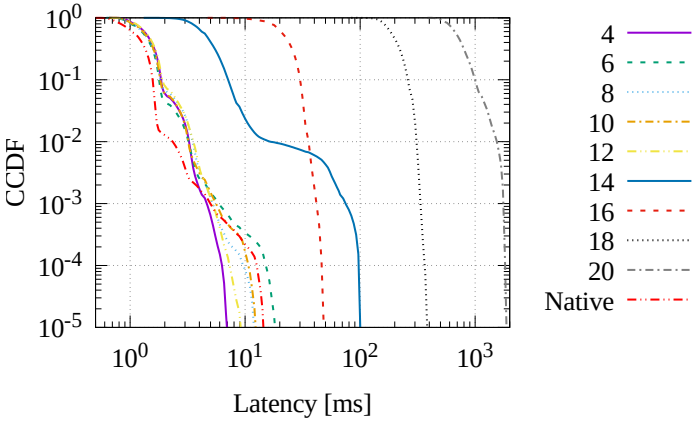


(a) CCDF of per-record latencies

Experiment	90%	99%	99.99%	max
4	1.64	3.67	12.58	19.92
6	1.64	2.75	11.01	20.97
8	1.70	2.49	9.44	19.92
10	1.70	2.36	7.08	15.20
12	1.77	2.88	9.96	20.97
14	2.49	4.46	7.86	19.92
16	22.02	26.21	32.51	46.14
18	234.88	268.44	301.99	335.54
20	838.86	1610.61	1879.05	1946.16
Native	1.51	1.70	4.46	14.16

(b) Selected percentiles and their latency in ms

Figure 5.13: Key-count overhead experiment with 256×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.



(a) CCDF of per-record latencies

Experiment	90%	99%	99.99%	max
4	1.90	3.28	4.72	7.34
6	1.84	3.28	6.03	18.87
8	1.84	3.54	5.24	12.58
10	1.84	3.28	5.77	13.11
12	1.90	3.67	5.24	9.44
14	7.34	16.78	75.50	100.66
16	30.41	35.65	41.94	50.33
18	268.44	318.77	335.54	385.88
20	1006.63	1610.61	1811.94	1879.05
Native	1.57	2.36	4.98	14.68

(b) Selected percentiles and their latency in ms

Figure 5.14: Key-count overhead experiment with 8192×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.

2. In Section 5.3.3.2 we evaluate the maximum latency and duration of migration strategies **as the number of distinct keys increases**. We expect all maximum latencies and durations to increase linearly with the amount of maintained state.
3. In Section 5.3.3.3 we evaluate the maximum latency and duration of migration strategies **as the number of distinct keys and bins increase proportionally**. We expect that with a constant per-bin state size Megaphone will maintain a fixed maximum latency while the duration increases.
4. In Section 5.3.3.4 we evaluate **the latency under load** during migration and steady-state to estimate the potential throughput. We expect a smaller maximum latency for Megaphone migrations.
5. In Section 5.3.3.5 we evaluate **the memory consumption** during migration. We expect a smaller memory footprint for Megaphone migrations.

Each of our migration experiments largely resembles the shapes seen in Figure 5.1, where each migration strategy has a well defined *duration* and *maximum latency*. For example, the all-at-once migration strategy has a relatively short duration with a large maximum latency, whereas the bin-at-a-time (*fluid*) migration strategy has a longer duration and lower maximum latency, and the batched migration strategy lies between the two. In these experiments we summarize each migration by the duration of the migration, and the maximum latency observed during the migration.

5.3.3.1 Number of bins vary

We now evaluate the behavior of different migration strategies for varying numbers of bins. As we increase the number of bins we expect to see fluid and batched migration achieve lower maximum latencies, though ideally with relatively unchanged durations. We do not expect to see all-at-once migration behave differently as a function of the number of bins, as it conducts all of its migrations simultaneously.

Holding the rates and bin counts fixed, we will vary the number of bins from 2^4 up to 2^{14} by factors of four. For each configuration, we run for one minute to establish a steady state, and then initiate a migration and continue for one another minute. During this whole time the rate of input records continues uninterrupted.

Figure 5.15 reports the latency-vs-duration trade-off of the three migration strategies as we vary the number of bins. The connected lines each describe

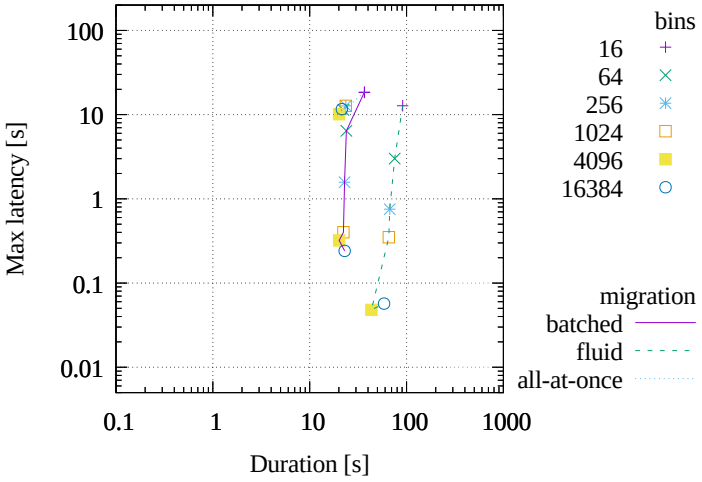


Figure 5.15: Key-count migration latency vs. duration, varying bin count for a fixed domain of 4096×10^6 keys. The vertical lines indicate that increasing the granularity of migration can reduce maximum latency for fluid and batched migrations without increasing the duration. The all-at-once migration datapoints remain in a cluster independent of the migration granularity.

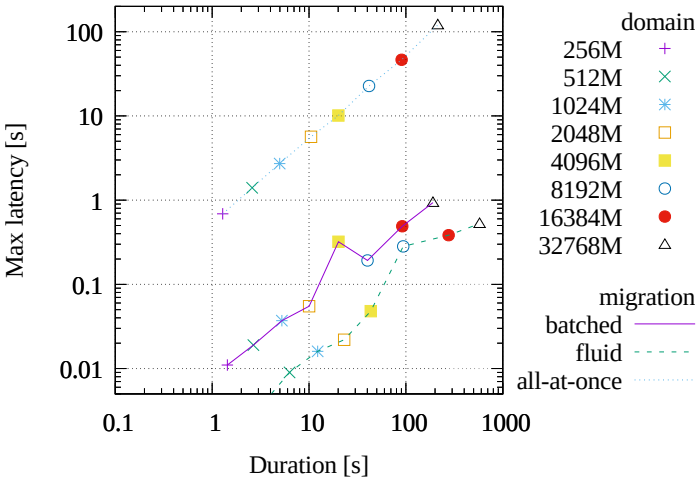


Figure 5.16: Key-count migration latency vs. duration, varying domain for a fixed rate of 4×10^6. As the domain size increases the migration granularity increases, and the duration and maximum latencies increase proportionally.

one strategy, and the common shapes describe a common number of bins. We see that all all-at-once migration experiments are in a low duration high latency cluster. Both fluid and batched migration achieve lower maximum latency as we increase the number of bins, without negatively impacting the duration.

5.3.3.2 Number of keys vary

We now evaluate the behavior of different migration strategies for varying domain sizes. Holding the rates and bin counts fixed, we will vary the number of keys from 256×10^6 up to 8192×10^6 by factors of two. For each configuration, we run for one minute to establish a steady state, and then initiate a migration and continue for one another minute. During this whole time the rate of input records continues uninterrupted.

Figure 5.16 reports the latency-vs-duration trade-off of the three migration strategies as we vary the number of distinct keys. The connected lines each

describe one strategy, and the common shapes describe a common number of distinct keys. We see that for any experiment, all-at-once migration has the highest latency and lowest duration, fluid migration has a lower latency and higher duration, and batched migration often has the best qualities of both.

5.3.3.3 Number of keys and bins vary proportionally

In the previous experiments, we either fixed the number of bins or the number of keys while varying the other parameter. In this experiment, we vary both bins and keys together such that the total amount of data per bin stays constant. This maintains a fixed migration granularity, which should have a fixed maximum latency even as the number of keys (and total state) increases. We run the key count experiment and fix the number of keys per bin to 4×10^6. We then increase the domain in steps of powers of two starting at 256×10^6 and increase the number of bins such that the keys per bin stays constant. The maximum domain is 32×10^9 keys.

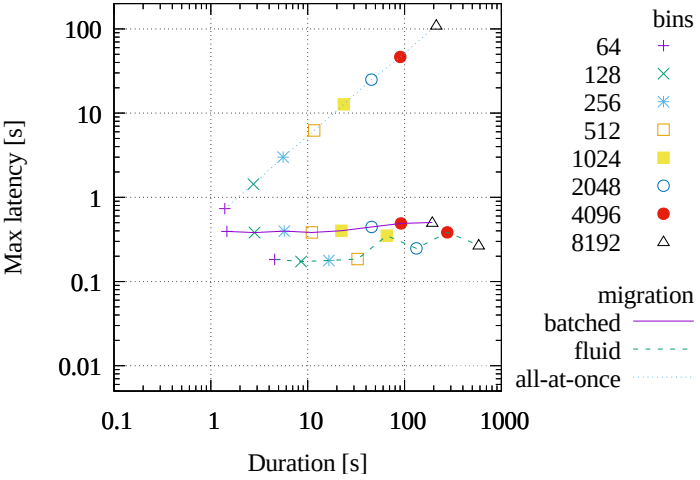


Figure 5.17: Latency and duration of key-count migrations for fixed state per bin. By holding the granularity of migration fixed, the maximum latencies of fluid and batched migration remain fixed even as the durations of all strategies increase.

Figure 5.17 reports the latency-versus-duration trade-off for the three migration strategies as we increase domain and number of bins while keeping the state per bin constant. The lines describe one migration strategy and the points describe a different configuration. We can observe that for fluid and batched migration the latency stays constant while only the duration increases as we increase the domain. For all-at-once migration, both latency and duration increase.

We conclude that fluid and batched migration offer a way to bound the latency impact on a computation during a migration while increasing the migration duration, whereas all-at-once migration does not.

5.3.3.4 Throughput versus processing latency

In this experiment, we evaluate what throughput Megaphone can sustain for specific latency targets. As we increase the offered load, we expect the steady-state and migration latency to increase. For a specific throughput target, we expect the all-at-once migration strategy to show a higher latency than batched, which itself is expected to be higher than fluid.

To analyze the latency, we keep the number of keys and bins constant, at $16'384 \times 10^6$ and 4096, and vary the offered load from 250×10^3 to 32×10^6 in powers of two. We measure the maximum latency observed during both steady-state and migration for each of the three migration strategies described earlier.

Figure 5.18 shows maximum latency observed when the system is sustaining a certain throughput. All three migration strategies and non-migrating show a similar pattern: Up to 16×10^6 records per second they do not show a significant increase in latency. At 32×10^6, the latency increases significantly, indicating that the system is now overloaded.

We conclude that the system's latency is mostly throughput-invariant until the system saturates and eventually fails to keep up with its input. Both fluid and batched migration sustain a throughput of up to 4×10^6 per second for a latency target of 1 s: Megaphone's migration strategies can satisfy latency targets 10-100x lower than all-at-once migration with similar throughput.

5.3.3.5 Memory consumption during migration

In Section 5.3.3.3 we analyzed the behavior of different migration strategies when increasing the total amount of state in the system while leaving the state

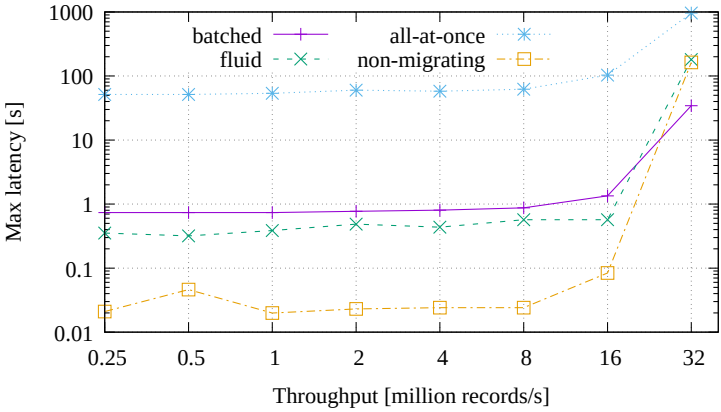


Figure 5.18: Offered load versus max latency for different migration strategies for key-count. The migration is invariant of the rate up to 16 million records per second.

per bin constant. Our expectation was that the all-at-once migration strategy would always offer the lowest duration when compared to batched and fluid migrations. Nevertheless, we observe for large amounts of data being migrated the duration for a all-at-once migration is longer than for batched migration.

To analyze the cause for this behavior we compared the memory consumption for the three migration strategies over time. We run the key count dataflow with 16×10^9 keys and 4096 bins. We record the resident set size (RSS) as reported by Linux over time per process.

Figure 5.19 shows the RSS reported by the first Timely Dataflow process for each migration strategy. Batched and fluid migration show a similar memory consumption of 35 GiB in steady state and do not expose a large variance during migration at times 400 s and 800 s. In contrast to that, all-at-once migration shows significant allocations of approximately additional 30 GiB during the migrations.

The experiment gives us evidence that a all-at-once migration causes significant memory spikes in addition to latency spikes. The reason for this is that during a all-at-once migration, each worker extracts and serializes the data to be migrated and enqueues it for the network threads to send. The network thread's send capacity is limited by the network throughput, limiting the throughput at which data can be transferred to the remote host. Batched and fluid migration

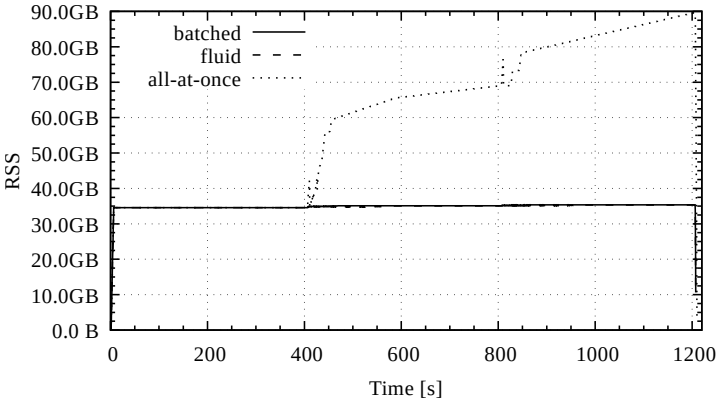


Figure 5.19: Memory consumption of key-count per process over time for different migration strategies. The fluid and batched strategies require less additional memory in each migration step than the all-at-once migration, which migrates all state at once.

patterns only perform another migration once the previous is complete and thus provide a simple form of flow-control effectively limiting the amount of temporary state.

5.4 Megaphone: conclusion

We presented the design and implementation of Megaphone, which provides efficient, minimally disruptive migration for stream processing systems. Megaphone plans fine-grained migrations using the logical timestamps of the stream processor, and interleaves the migrations with regular streaming dataflow processing. Our evaluation under realistic workloads shows that migration disruption was significantly lower than with prior all-at-once migration strategies.

We implemented Megaphone in Timely Dataflow, without any changes to the host dataflow system. Megaphone demonstrates that dataflow coordination mechanisms (timestamp frontiers) and dataflow channels themselves are sufficient to implement minimally disruptive migration. Megaphone is available as open source.[3]

[3] `https://github.com/strymon-system/megaphone`

I don't believe in infinity,
I think there's an end

(NOFX: I Believe in Goddess)

6 Conclusions

Stream processors have evolved significantly during the last decades. First generation stream processors were tailored to special use cases. Most were limited to single machines without support for scaling to multiple computers. A significant technology jump occurred with the first scale-out architectures, which were designed to cope with an ever-increasing amount of data that needed to be processed. A scale-out architecture increases the processing and storage resources available to a stream processor and can enable higher throughput with more complex queries. Still, the systems lacked properties which we consider crucial: they had limited elasticity, meaning that in overload situations data would be dropped, and they did not support generalized stream processing, allowing only specific join operations and lacking a well-defined notion of time.

Independent of stream processing, new technologies were needed to process large amounts of data distributed across many machines. Google's MapReduce filled this gap and demonstrated how computations can be scaled to hundreds of machines. The stream processing community applied similar approaches to stream processing, leading to a new generation of stream processors that enabled better scalability. Nevertheless, many newer systems, although offering a high throughput, did not have adequate query latency.

Modern stream processors aim at optimizing both latency and throughput. They exploit different levels of parallelism, having a notion of time independent of system time, and reducing data batch sizes for lower latency. Such optimization comes with the price of increased system complexity. Therefore, to realize the full potential, additional features, tools and mechanisms are required to help both programmers and operators.

At the beginning of this thesis we identified a gap between the capabilities of current stream processors and the tools available to programmers and

operators to fully utilize them. Specifically, we observed that existing tracing infrastructure for distributed stream processors did not enable debugging latency problems, which we think is a common task in the scope of time-critical stream processing. We noted that stream processors lacked manageability tools to automate configuration parameter tuning, such as setting the level of parallelism for dataflows. Also, the trace data they supplied did not necessarily help to pinpoint problems as the data was pre-aggregated, or simply measured the wrong indicators.

In this thesis, we presented techniques to enable programmers, users and operators to close the gap between the theoretical capabilities of stream processors and deployments in practice.

In Chapter 3, we discussed the problem of how to derive and present a performance breakdown of factors effecting processing latency. We presented Snailtrail, a system to generate latency traces on-line, using the *critical participation* metric. This metric, based on critical path analysis, but adapted to long-running computations, allows to generate a detailed real-time breakdown of factors that contribute to processing latency. Our evaluation showed that critical participation greatly improves analyzing latency problems in distributed stream processors. We were able to identify problems previously discussed by other authors fully automatically. Snailtrail solves the problem of debugging stream processors and gives precise and helpful diagnostics about their computational structure.

In Chapter 4, we analyzed the problem of automatically controlling a distributed stream processor. We presented DS2, a controller that is able to configure a stream processor's parallelism fast and accurately, using a fine-grained performance model. The performance model captures dependencies between operators in a dataflow and allows for global reconfigurations in a single step. It is based on the knowledge of the dataflow structure combined with detailed internal measurements, and external requirements in the form of SLOs. Our evaluation showed that DS2 needs very few steps to reach a stable configuration and has a fast reaction time. The evaluation also revealed that many stream processors suffer from substantially increased query latency while applying a reconfiguration.

Lastly, in Chapter 5, we presented a reconfiguration mechanism for stream processors that is designed to minimize the latency impact on a running computation. Megaphone is based on the idea of interleaving configuration updates with query processing by slicing these updates into small units, thus reducing the latency impact on query processing. Our evaluation showed that

Megaphone reduces the maximum latency by orders of magnitude compared to a state-of-the-art stop-and-restart approach. This enables frequent and fine-grained configuration updates to adjust resource utilization to match SLOs and fast reactions to workload fluctuations.

All our contributions are purely generic, either implemented as stand-alone systems with well-defined interfaces or as libraries on top of unmodified existing stream processing systems. Also, while each one works independently of the other, they show clear benefits when combined.

We presented systems and techniques to address the two most urgent problems identified in the introduction: Stream processors need to provide debugging mechanisms as well as manageability and understandability features to enable low-latency query processing. In this thesis, we presented Snailtrail to address debuggability and understandability, as well as DS2 and Megaphone to simplify managing a stream processor.

6.1 Requirements for efficient stream processing

Throughout this dissertation, the discussion of existing stream processors revealed various architectural problems and recurring patterns, affecting both performance and expressiveness. In our studies, we encountered some crucial factors governing performance and efficiency of stream processing. We conclude this thesis with a set of requirements that we think are important for efficient stream processing.

Stream processing systems constitute a category of distributed systems, with special properties imposed by their use-cases. Stream processing systems can be expected to be long-running with queries aggregating over long periods of time. Queries can be deployed for weeks, months or even years and the system should provide the necessary infrastructure for such queries to execute correctly. This includes graceful handling of workload and processing fluctuations, proper maintenance of intermediate state, and some form of fault tolerance. Additionally, the definitions for queries might be updated while they are executing, either manually or when a query optimizer generates a new plan, which requires the system to adapt to new or changed queries. For these reasons, a stream processor should honor the following requirements for efficient stream processing.

Understandability Stream processors should give detailed information about the execution performance of the queries they are maintaining. This information should be interpretable by users in order to understand and optimize their queries.

Manageability Additionally, stream processors should support reconfiguration of all important system properties, such as updating queries or adding and removing resources. While a new configuration is applied, it should still honor the other properties outlined here.

Liveness Stream processors should provide answers to queries over time based on incoming data. Eventually, a stream processor has to produce a result for a query.

Correctness A stream processor should produce correct results based on the installed queries and the data it received. Problems originating from inside the system should not have an effect on results.

Query latency A stream processor should try to provide answers to queries with low latency with respect to maintaining system stability, i.e. the delay between observing incoming data and providing an updated query result should be minimized.

Elasticity The rate at which input data arrives can vary over time. A stream processor has to gracefully adapt to changes in the arrival rate.

Generality The query language or interface should be general, allowing to express arbitrary queries with their own constraints, e.g., processing of joins and time-based window semantics.

Abstract time Due to the nature of how data is processed, stream processing systems need an abstract notion of time that allows to relate results with input data independent of when a computation was performed. Data from external sources can suffer from delay, which can result in data arriving out of order. To tolerate time fluctuations on the input and during processing, query evaluation must be independent of the system time.

Expressiveness The stream processor should provide a set of general operators to be recombined for specific queries and it should permit the user to add other operators.

Continuity Queries on streams can be defined on arbitrarily-sized time- and data-based windows. The stream processor should offer the necessary infrastructure to host such queries over a long time period while ensuring the other properties outlined here.

Scalability Stream processors should support both scale-up by better utilizing additional resources within the same machine as well as scale-out to add additional resources connected from other machines.

Debuggability Many stream processors are inherently distributed. To aid the user, they should provide means to debug operators, even when executing in a distributed cluster. A stream processor should be able to debug user code that it executes and report sufficient metrics about its internal state. Both help to track performance anomalies back to their roots.

Stonebraker, Çetintemel, and Zdonik [SÇZ05] in 2005 presented a list of eight requirements for stream processing, some of which are similar to ours. Most importantly, they identify the need for low latency as the primary objective for stream processing.

Most of the items on this list are actually covered by many current stream processing systems. Hence, it is more interesting to determine which requirements they do not support. None of the stream processors mentioned in this thesis did have an adequate debugging infrastructure, or met our goal of being understandable and manageable, which is why this thesis aims at solving these problems. We note that many current stream processors support liveness and correctness, but fail to optimize explicitly for low-latency query processing, with Timely Dataflow being a notable exception. Elasticity and scalability are supported, at least as a basic mechanism. Most stream processors allow for some form of continuity, but there is no agreed notion on how it should be achieved and what the desired granularity is. We observe that stream processors have gained many features and currently there exist many maintained solutions, both academic and commercial, to choose from. However, we are not aware of any stream processor fulfilling all the requirements outlined in this section.

6.2 Directions for future work

This dissertation opens up several interesting perspectives for future research topics on stream processors and distributed systems. In this part, we want to highlight some of them.

With Megaphone we demonstrate that fine-grained configuration updates can limit a reconfiguration's latency impact on query performance. This enables moving the configuration between workers in the stream processor. However, it does not solve the problem of attaching or detaching physical workers to a running system. We believe that this is an interesting and challenging coordination problem by itself.

The controller presented in Chapter 4 estimates the parallelism for a computation to meet its throughput SLO. It solves the problem of backpressure in a stream processor. In reality, other problems can cause a stream processor to miss its targets. DS2 cannot address data skew where one worker has significantly more work than others and cannot keep up with it. To address data skew, a controller needs to have an understanding how the processing characteristics vary between shards of data, for example by tracking "hot" keys in the stream. In addition to that, a controller has to make correct decisions to move shards of data and processing to specific workers. Megaphone has a configuration interface that is broad enough for a controller to supply reconfigurations for mitigation of both stragglers and skew.

The controller currently optimizes a stream processor to keep up with its throughput SLO. An interesting research question is how to build a similar controller that optimizes a stream processor's configuration to meet latency SLOs. This involves different metrics to observe where processing time is spent in addition to the dependency information from the dataflow and processing as provided by Snailtrail. A result could be a controller that trades off latency versus throughput to create a well-balanced configuration.

Megaphone's current implementation assumes a partitioning of the data fixed at compile time. This limits the reconfiguration granularity at runtime. The evaluation demonstrates that a very fine-grained partitioning of data has negative effects on processing latency. To enable adaptive partitioning at runtime, a different, non-prepartitioned approach would need to be implemented. This would allow a controller to specify configurations down to the key-level where currently only groups of keys can be configured.

Many stream processors offer continuity in the form of fault tolerance. For example, *Apache Flink* [Car+15] uses periodic snapshots based on barriers

interleaved with the stream of records. Its implementation is limited to acyclic computations and incurs latency spikes whenever a snapshot is generated because data needs to be written to an external system. We think a better fault-tolerance mechanism can both avoid latency spikes as well as support cyclic computations. An interesting direction would be improving on the fault-tolerance support mentioned in *Naiad* [Mur+13] and implementing it on *Timely Dataflow* [McS].

List of Tables

List of Figures

Glossary

CCDF complementary cumulative distribution function. 93–96, 125

CP critical participation. 34, 35, 41–45, 50, 53, 55–57, 59, 61–65, 68

CPA critical path analysis. 22–24, 33, 34, 36–38, 41, 44, 45, 63

CQL continuous query language. 14

DBMS database management system. 13

MPI message passing interface. 22, 36

OLTP online transaction processing. 99

PAG program activity graph. 22, 23, 37–40, 42, 47, 48, 50, 51, 53, 54, 61, 62

RDD resilient distributed dataset. 47, 52

RSS resident set size. 134

SLO service level objective. 25, 31, 138, 139, 142

SQL structured query language. 13, 14, 16

YSB Yahoo streaming benchmark. 60

Bibliography

[Aba+03] Daniel J. Abadi et al. “Aurora: a new model and architecture for data stream management”. In: *VLDB J.* 12.2 (2003), pp. 120–139. DOI: 10.1007/s00778-003-0095-z (cit. on pp. 12, 13).

[Aba+05] Daniel J. Abadi et al. “The Design of the Borealis Stream Processing Engine”. In: *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 277–289. URL: http://cidrdb.org/cidr2005/papers/P23.pdf (cit. on pp. 14, 26).

[Aba+13] Martín Abadi et al. “Formal Analysis of a Distributed Algorithm for Tracking Progress”. In: *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*. Ed. by Dirk Beyer and Michele Boreale. Vol. 7892. Lecture Notes in Computer Science. Springer, 2013, pp. 5–19. ISBN: 978-3-642-38591-9. DOI: 10.1007/978-3-642-38592-6_2 (cit. on p. 21).

[Aba+16] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi (cit. on p. 47).

[AD16] Eric Anderson and Marian Dvorsky. *Comparing Cloud Dataflow autoscaling to Spark and Hadoop*. 2016. URL: https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop (cit. on p. 26).

[Aki+13] Tyler Akidau et al. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044 (cit. on pp. 18, 25, 101, 114).

[Aki+15] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803. ISSN: 2150-8097. DOI: 10.14778/2824032.2824076 (cit. on pp. 24, 88, 101, 114).

[Ale+98] Cedell A. Alexander et al. "Near-Critical Path Analysis: A Tool for Parallel Program Optimization". In: *Southern Symposium on Computing*. 1998 (cit. on p. 23).

[Ara+03] Arvind Arasu et al. *STREAM: The Stanford Stream Data Manager*. Tech. rep. 1. 2003, pp. 19–26. URL: http://sites.computer.org/debull/A03mar/paper.ps (cit. on p. 14).

[ARH94] Cedell Alexander, Donna Reese, and James C. Harden. "Near-Critical Path Analysis of Program Activity Graphs". In: *International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*. 1994 (cit. on p. 34).

[Arm+18] Michael Armbrust et al. "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA, 2018, pp. 601–613. ISBN: 978-1-4503-4703-7 (cit. on pp. 28, 30).

[AVB17] Marcos Dias de Assunção, Alexandre Da Silva Veith, and Rajkumar Buyya. "Resource Elasticity for Distributed Data Stream Processing: A Survey and Future Directions". In: *CoRR* abs/1709.01363 (2017). arXiv: 1709.01363 (cit. on p. 25).

[Bar+12] Sean Barker et al. “"Cut Me Some Slack": Latency-aware Live Migration for Databases”. In: *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT '12. Berlin, Germany: ACM, 2012, pp. 432–443. ISBN: 978-1-4503-0790-1 (cit. on pp. 29, 31).

[BC17] Muhammad Bilal and Marco Canini. “Towards automatic parameter tuning of stream processing systems”. In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24–27, 2017*. 2017, pp. 189–200. DOI: 10.1145/3127479.3127492 (cit. on p. 72).

[bdb] *BigData Benchmark*. URL: https://amplab.cs.berkeley.edu/benchmark/ (visited on 09/2017) (cit. on p. 35).

[Bed+13] Ivan Bedini et al. “Modeling performance of a parallel streaming engine: bridging theory and costs”. In: *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*. Ed. by Seetharami Seelam et al. ACM, 2013, pp. 173–184. ISBN: 978-1-4503-1636-1. DOI: 10.1145/2479871.2479895. URL: http://dl.acm.org/citation.cfm?id=2479871 (cit. on p. 24).

[Böh+12] David Böhme et al. “Scalable Critical-Path Based Performance Analysis”. In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 1330–1340. ISBN: 978-1-4673-0975-2. DOI: 10.1109/IPDPS.2012.120. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6266782 (cit. on p. 22).

[Bur75] W. H. Burge. “Stream Processing Functions”. In: *IBM Journal of Research and Development* 19.1 (Jan. 1975), pp. 12–25. ISSN: 0018-8646 (print), 2151-8556 (electronic) (cit. on p. 11).

[Car+15] Paris Carbone et al. “Apache Flink: Stream and batch processing in a single engine”. In: *Data Engineering* 38.4 (2015) (cit. on pp. 9, 18, 28, 47, 73, 101, 142).

[Car+17] Paris Carbone et al. “State Management in Apache Flink: Consistent Stateful Distributed Stream Processing”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097 (cit. on pp. 25, 28, 30).

[Car12] David Carasso. *Exploring Splunk*. Evolved Technologist Press, 2012 (cit. on p. 33).

[CB15] Charlie Curtsinger and Emery D. Berger. "Coz: Finding Code That Counts with Causal Profiling". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 184–197. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815409 (cit. on p. 24).

[CC15] Jian Chen and Russell M. Clapp. "Critical-path candidates: scalable performance modeling for MPI workloads". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2015 (cit. on p. 36).

[Cha+03] Sirish Chandrasekaran et al. "TelegraphCQ: Continuous Dataflow Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 668–668. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872857 (cit. on p. 14).

[Chi+16] Sanket Chintapalli et al. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. 2016, pp. 1789–1792. DOI: 10.1109/IPDPSW.2016.138 (cit. on pp. 60, 64).

[Cho+14] Michael Chow et al. "The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 217–231. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685066 (cit. on p. 22).

[CM12] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing". In: *ACM Comput. Surv.* 44.3 (June 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677 (cit. on p. 7).

[Dai+11] Jinquan Dai et al. "HiTune: Dataflow-based Performance Analysis for Big Data Cloud". In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'11. 2011, p. 24 (cit. on p. 24).

[Das+11] Sudipto Das et al. "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration". In: *Proc. VLDB Endow.* 4.8 (May 2011), pp. 494–505. ISSN: 2150-8097. DOI: 10.14778/2002974.2002977 (cit. on pp. 29, 31).

[Des17] Salvatore Desimone. *Storage reimagined for a streaming world*. 2017. URL: http://blog.pravega.io/2017/04/09/storage-reimagined-for-a-streaming-world/ (cit. on pp. 24, 26).

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150 (cit. on p. 15).

[DK10] Isaac Dooley and Laxmikant V. Kalé. "Detecting and using critical paths at runtime in message driven parallel programs". In: *IEEE International Symposium on Parallel and Distributed Processing*. 2010 (cit. on p. 23).

[Elm+11] Aaron J. Elmore et al. "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 301–312. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989356 (cit. on pp. 29, 31).

[Elm+15] Aaron J. Elmore et al. "Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 299–313. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2723726 (cit. on pp. 29, 31).

[Fan+17] Junhua Fang et al. "Parallel Stream Processing Against Workload Skewness and Variance". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '17. Washington, DC, USA, 2017, pp. 15–26. ISBN: 978-1-4503-4699-3 (cit. on p. 28).

[Fer+13] Raul Castro Fernandez et al. "Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management". In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 2013, pp. 725–736. DOI: 10.1145/2463676.2465282 (cit. on pp. 17, 24, 26–28, 30).

[Flo+17] Avrilia Floratou et al. "Dhalion: Self-regulating Stream Processing in Heron". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1825–1836. ISSN: 2150-8097. DOI: 10.14778/3137765.3137786 (cit. on pp. 24, 26–28, 56, 57, 60, 65–67, 69, 70, 87, 88, 90).

[Flo83] Thomas H. Flowers. "The Design of Colossus (foreword by Howard Campaigne)". In: *IEEE Annals of the History of Computing* 5.3 (1983), pp. 239–252. DOI: 10.1109/MAHC.1983.10079 (cit. on p. 11).

[Fu+17] Tom Z. J. Fu et al. "DRS: Auto-Scaling for Real-Time Stream Analytics". In: *IEEE/ACM Trans. Netw.* 25.6 (2017), pp. 3338–3352. DOI: 10.1109/TNET.2017.2741969 (cit. on pp. 26, 27).

[Gar+12] Elmer Garduno et al. "Theia: Visual Signatures for Problem Diagnosis in Large Hadoop Clusters". In: *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques*. lisa'12. San Diego, CA: USENIX Association, 2012, pp. 33–42. URL: http://dl.acm.org/citation.cfm?id=2432523.2432526 (cit. on p. 24).

[Ged+14] Buğra Gedik et al. "Elastic Scaling for Data Stream Processing". In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (June 2014), pp. 1447–1463. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.295 (cit. on pp. 24, 26–28).

[Gou+17] Anastasios Gounaris et al. “Dynamic Configuration of Partitioning in Spark Applications”. In: *IEEE Trans. Parallel Distrib. Syst.* 28.7 (2017), pp. 1891–1904. DOI: 10.1109/TPDS.2017.2647939 (cit. on pp. 72, 80).

[Gul+12] Vincenzo Gulisano et al. “StreamCloud: An elastic and scalable data streaming system”. In: *IEEE Transactions on Parallel and Distributed Systems* (2012). ISSN: 10459219. DOI: 10.1109/TPDS.2012.24 (cit. on pp. 24, 26–28, 30).

[Guo+13] Qi Guo et al. “Correlation-based performance analysis for full-system MapReduce optimization”. In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. 2013, pp. 753–761. DOI: 10.1109/BigData.2013.6691648 (cit. on p. 23).

[Hadoop] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 01/2019) (cit. on p. 15).

[HDB11] Herodotos Herodotou, Fei Dong, and Shivnath Babu. “No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics”. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC ’11. Cascais, Portugal: ACM, 2011, 18:1–18:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038934 (cit. on p. 72).

[Hei+14a] Thomas Heinze et al. “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS ’14. Mumbai, India: ACM, 2014, pp. 13–22. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611294 (cit. on p. 26).

[Hei+14b] Thomas Heinze et al. “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS ’14. Mumbai, India: ACM, 2014, pp. 13–22. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611294 (cit. on p. 30).

[Hel+04] Joseph L. Hellerstein et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN: 047126637X (cit. on p. 25).

[Her+11] Herodotos Herodotou et al. “Starfish: A Self-tuning System for Big Data Analytics”. In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 2011, pp. 261–272. URL: http://cidrdb.org/cidr2011/Papers/CIDR11%5C_Paper36.pdf (cit. on p. 72).

[Heron] *Apache Heron*. URL: https://apache.github.io/incubator-heron (visited on 02/2019) (cit. on pp. 9, 82, 83).

[Hof+18] Moritz Hoffmann et al. “SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 95–110. ISBN: 978-1-931971-43-0. URL: https://www.usenix.org/conference/nsdi18/presentation/hoffmann (cit. on pp. 5, 33).

[Hof+19] Moritz Hoffmann et al. “Megaphone: Latency-conscious state migration for distributed streaming dataflows”. In: *PVLDB* 12.9 (2019). DOI: 10.14778/3329772.3329777 (cit. on pp. 5, 99).

[Hol96] Jeffrey K. Hollingsworth. “An Online Computation of Critical Path Profiling”. In: *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. SPDT ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 11–20. ISBN: 0-89791-846-0. DOI: 10.1145/238020.238024 (cit. on p. 23).

[Hue16] Fabian Hueske. *Savepoints: Turning Back Time*. 2016. URL: https://data-artisans.com/blog/turning-back-time-savepoints (cit. on pp. 84, 90).

[IBMSt] *IBM Streams*. URL: https://www.ibm.com/ch-en/marketplace/stream-computing (visited on 11/2017) (cit. on p. 30).

[JMC89] Douglas Stott Parker Jr., Richard R. Muntz, and H. Lewis Chau. “The Tangram Stream Query Processing System”. In: *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*. IEEE Computer Society, 1989, pp. 556–563. ISBN: 0-8186-1915-5. DOI: 10.1109/ICDE.1989.47262 (cit. on p. 12).

[Kal+18] Vasiliki Kalavri et al. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 783–798 (cit. on pp. 5, 69).

[KD16] Eugene Kirpichov and Malo Denielou. *No shard left behind: dynamic work rebalancing in Google Cloud Dataflow*. 2016. URL: https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow (visited on 01/2019) (cit. on pp. 24, 27, 79).

[KLC17] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. "A holistic view of stream partitioning costs". In: *PVLDB* 10.11 (2017), pp. 1286–1297. URL: http://www.vldb.org/pvldb/vol10/p1286-katsipoulakis.pdf (cit. on p. 79).

[KR16] Kimberly Keeton and Timothy Roscoe, eds. *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cit. on p. 60).

[Kul+15] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788 (cit. on pp. 17, 24, 25, 73).

[Kwo+12] YongChul Kwon et al. "SkewTune: Mitigating Skew in Mapreduce Applications". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD

'12. Scottsdale, Arizona, USA: ACM, 2012, pp. 25–36. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213840 (cit. on p. 79).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563 (cit. on p. 38).

[LJK15] Bjorn Lohrmann, Peter Janacik, and Odej Kao. "Elastic Stream Processing with Latency Guarantees". In: *Proceedings - International Conference on Distributed Computing Systems*. 2015. ISBN: 9781467372145. DOI: 10.1109/ICDCS.2015.48 (cit. on pp. 26, 27).

[LML14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments". In: *J. Grid Comput.* 12.4 (Dec. 2014), pp. 559–592. ISSN: 1570-7873. DOI: 10.1007/s10723-014-9314-7 (cit. on p. 69).

[LogI] *VMware vRealize Log Insight*. URL: http://www.vmware.com/products/vrealize-log-insight.html (visited on 01/2019) (cit. on p. 33).

[Mai+18] Luo Mai et al. "Chi: a scalable and programmable control plane for distributed stream processing systems". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1303–1316 (cit. on p. 30).

[McS] Frank McSherry. *A modular implementation of timely dataflow in Rust*. URL: https://github.com/frankmcsherry/timely-dataflow (visited on 01/2019) (cit. on pp. 9, 20, 27, 47, 59, 83, 100, 143).

[MS14] Sunilkumar S. Manvi and Gopal Krishna Shyam. "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey". In: *Journal of Network and Computer Applications* 41 (2014), pp. 424–440. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2013.10.004. URL: http://www.sciencedirect.com/science/article/pii/S1084804513002099 (cit. on p. 69).

[Mur+13] Derek Gordon Murray et al. “Naiad: a timely dataflow system”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738. URL: http://dl.acm.org/citation.cfm?id=2517349 (cit. on pp. 19, 21, 27, 47, 101, 143).

[Nag] *Nagios*. URL: https://www.nagios.org (visited on 09/2017) (cit. on p. 33).

[Nas+15] Muhammad Anis Uddin Nasir et al. “The power of both choices: Practical load balancing for distributed stream processing engines”. In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 2015, pp. 137–148. DOI: 10.1109/ICDE.2015.7113279 (cit. on p. 79).

[Nas+16] Muhammad Anis Uddin Nasir et al. “When two choices are not enough: Balancing at scale in Distributed Stream Processing”. In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 2016, pp. 589–600. DOI: 10.1109/ICDE.2016.7498273 (cit. on p. 79).

[NEX] *NEXMark benchmark*. URL: http://datalab.cs.pdx.edu/niagaraST/NEXMark (visited on 01/2019) (cit. on pp. 87, 116).

[NexB] *Apache Beam Nexmark benchmark suite*. URL: https://beam.apache.org/documentation/sdks/java/testing/nexmark (visited on 02/2019) (cit. on p. 87).

[Ous] Kay Ousterhout. *Spark Performance Analysis*. URL: https://kayousterhout.github.io/trace-analysis/ (visited on 04/2017) (cit. on pp. 35, 52, 62).

[Ous+15] Kay Ousterhout et al. “Making Sense of Performance in Data Analytics Frameworks”. In: *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 293–307. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout (cit. on p. 23).

[PokGO] *Bringing Pokemon GO to life on Google Cloud*. URL: https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html (visited on 06/2018) (cit. on p. 28).

[Raj+18] Sumanaruban Rajadurai et al. "Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA, 2018, pp. 98–112. ISBN: 978-1-4503-4911-6 (cit. on pp. 29, 30).

[Ram+16] Navaneeth Rameshan et al. "Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy". In: *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE. 2016, pp. 233–244. DOI: 10.1109/CCGrid.2016.71 (cit. on p. 27).

[Rus+15] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on p. 60).

[Sac+03] Federico D. Sacerdoti et al. "Wide Area Cluster Monitoring with Ganglia". In: *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China*. IEEE Computer Society, 2003, p. 289. ISBN: 0-7695-2066-9. DOI: 10.1109/CLUSTR.2003.1253327 (cit. on p. 33).

[Sai+08] Ali G Saidi et al. "Full-System Critical Path Analysis". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. 2008 (cit. on p. 23).

[Sch+09] Scott Schneider et al. "Elastic scaling of data parallel operators in stream processing". In: *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*. 2009. ISBN: 9781424437504. DOI: 10.1109/IPDPS.2009.5161036 (cit. on pp. 27, 28).

[Sch05] M. Schulz. "Extracting Critical Path Graphs from MPI Applications". In: *IEEE International Conference on Cluster Computing* (2005) (cit. on pp. 22, 36).

[SCM13] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. "ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation". In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT '13. Genoa, Italy: ACM, 2013, pp. 53–64. ISBN: 978-1-4503-1597-5. DOI: 10.1145/2452376.2452384 (cit. on p. 31).

[SÇZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 Requirements of Real-time Stream Processing". In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504 (cit. on p. 141).

[Sha+02] Mehul A. Shah et al. "Flux: An Adaptive Partitioning Operator for Continuous Query Systems". In: *In ICDE*. 2002, pp. 25–36 (cit. on p. 30).

[Sha49] C. E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (Jan. 1949), pp. 10–21. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1949.232969 (cit. on p. 73).

[SparkDy] *Dynamic Resource Allocation in Spark*. URL: https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation (visited on 01/2019) (cit. on pp. 26, 28).

[Ste97] Robert Stephens. "A Survey of Stream Processing". In: *Acta Informatica* 34.7 (July 1997), pp. 491–541. ISSN: 0001-5903 (print), 1432-0525 (electronic). URL: http://link.springer-ny.com/link/service/journals/00236/bibs/7034007/70340491.htm (cit. on pp. 7, 11).

[Sto86] Michael Stonebraker. "The case for shared nothing". In: *IEEE Database Eng. Bull.* 9.1 (1986), pp. 4–9 (cit. on p. 10).

[Sul96] Mark Sullivan. "Tribeca: A Stream Database Manager for Network Traffic Analysis". In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 594–. ISBN: 1-55860-382-4. URL: http://dl.acm.org/citation.cfm?id=645922.673489 (cit. on p. 12).

[Tos+14] Ankit Toshniwal et al. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641 (cit. on pp. 17, 24, 25).

[Tu+06] Yi-Cheng Tu et al. "Load Shedding in Stream Databases: A Control-based Approach". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 787–798. URL: http://dl.acm.org/citation.cfm?id=1182635.1164195 (cit. on p. 70).

[Tuc+02] Pete Tucker et al. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Tech. rep. OGI School of Science & Engineering at OHSU, 2002 (cit. on pp. 87, 116).

[Twi] *New Tweets per second record, and how!* URL: https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html (visited on 06/2018) (cit. on p. 28).

[Ven+17] S. Venkataraman et al. "Drizzle: Fast and Adaptable Stream Processing at Scale". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017 (cit. on pp. 17, 63).

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Vol. 303. Academic Press London, 1985 (cit. on p. 11).

[Wan+12] Chengwei Wang et al. "VScope: Middleware for Troubleshooting Time-sensitive Data Center Applications". In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag New York, Inc., 2012, pp. 121–141. ISBN: 978-3-642-35169-3. URL: http://dl.acm.org/citation.cfm?id=2442626.2442635 (cit. on p. 24).

[WK09] Daniel Warneke and Odej Kao. "Nephele: Efficient Parallel Data Processing in the Cloud". In: *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*. MTAGS '09. Portland, Oregon: ACM, 2009, 8:1–8:10. ISBN: 978-1-60558-714-1. DOI: 10.1145/1646468.1646476 (cit. on pp. 18, 27).

[WT15] Yingjun Wu and Kian-Lee Tan. "ChronoStream: Elastic stateful stream computation in the cloud". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by Johannes Gehrke et al. IEEE Computer Society, 2015, pp. 723–734. ISBN: 978-1-4799-7964-6. DOI: 10.1109/ICDE.2015.7113328. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7109453 (cit. on pp. 25, 29, 30).

[XPG16] Le Xu, Boyang Peng, and Indranil Gupta. "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand". In: *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*. 2016, pp. 22–31. DOI: 10.1109/IC2E.2016.38 (cit. on p. 26).

[YDG89] S. H. Yen, D. H. Du, and S. Ghanta. "Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis". In: *Proceedings of the 26th ACM/IEEE Design Automation Conference*. DAC '89. 1989, pp. 649–654 (cit. on p. 34).

[YM88] Cui-Qing Yang and Barton P Miller. "Critical Path Analysis for the Execution of Parallel and Distributed Programs". In: *IEEE International Conference on Distributed Computing Systems*. 1988 (cit. on pp. 22, 34).

[Yu+08] Yuan Yu et al. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 1–14. ISBN: 978-1-931971-65-2 (cit. on p. 16).

[Zah+12] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 2012, pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia (cit. on pp. 16, 47).

[Zah+13] Matei Zaharia et al. “Discretized streams: Fault-tolerant streaming computation at scale”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 423–438 (cit. on pp. 17, 24, 25, 30).

[Zha+16] Xu Zhao et al. “Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 603–618. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao (cit. on p. 24).

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. “Dynamic Plan Migration for Continuous Queries over Data Streams”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’04. Paris, France, 2004, pp. 431–442. ISBN: 1-58113-859-8 (cit. on p. 28).