Rafiki: Task-Level Capacity Planning in Distributed Stream Processing Systems

Benjamin J. J. Pfister[1](✉), Wolf S. Lickefett[1], Jan Nitschke[1], Sumit Paul[1], Morgan K. Geldenhuys[1], Dominik Scheinert[1], Kordian Gontarska[2], and Lauritz Thamsen[1]

[1] Technische Universität Berlin, Berlin, Germany
{benjamin.pfister,wolf.lickefett,jan.nitschke, sumit.paul}@campus.tu-berlin.de, {morgan.geldenhuys,dominik.scheinert,lauritz.thamsen}@tu-berlin.de
[2] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
kordian.gontarska@hpi.de

Abstract. Distributed Stream Processing is a valuable paradigm for reliably processing vast amounts of data at high throughput rates with low end-to-end latencies. Most systems of this type offer a fine-grained level of control to parallelize the computation of individual tasks within a streaming job. Adjusting the parallelism of tasks has a direct impact on the overall level of throughput a job can provide as well as the amount of resources required to provide an adequate level of service. However, finding optimal parallelism configurations that fall within the expected Quality of Service requirements is no small feat to accomplish.

In this paper we present Rafiki, an approach to automatically determine optimal parallelism configurations for Distributed Stream Processing jobs. Here we conduct a number of proactive profiling runs to gather information about the processing capacities of individual tasks, thereby making the selection of specific utilization targets possible. Understanding the capacity information enables users to adequately provision resources so that streaming jobs can deliver the desired level of service at a reduced operational cost with predictable recovery times. We implemented Rafiki prototypically together with Apache Flink where we demonstrate its usefulness experimentally.

Keywords: Distributed Stream Processing · Capacity planning · Resource optimization · Quality of Service · Parallelization · Profiling · Performance modeling

1 Introduction

Distributed Stream Processing (DSP) enables the processing of large volumes of unbounded streams of data with high throughput rates and low end-to-end latencies. Streams of data are generated in a growing number of contexts including IoT sensor networks, social media, and online transactions [5,9]. In order to meet Quality of Service (QoS) requirements regarding performance and availability, DSP systems must be configured and allocated a sufficient amount of resources to provide an adequate level of service. Determining configurations, how many resources to allocate to a DSP system, and what levels of throughput those resources and configurations can provide is challenging. Finding optimal configurations is typically time-consuming and requires expert-level knowledge of the DSP system and streaming job [1,10]. However, uncovering this information is important for all users of these systems, and providing an approach that automates and speeds up this process is necessary.

In the stream processing model, a series of tasks are performed on a stream of data, and each item in the data stream is processed by a task as soon as it becomes available [5]. Given an infinitely large stream of data, tasks will process as many items as possible using the available resources. To increase overall throughput and reduce end-to-end latency, the computation of tasks in DSP jobs can be run in parallel. Because each task performs a different function and can therefore process a different maximum number of messages per second, one of the most important configurations to adjust to match available resources to stream processing workloads is the number of parallel computations, or parallelism, of tasks.

By gaining insights into the capacity of a DSP job on a fine-grained level, resource allocation can be better optimized and QoS requirements can be more easily met. Finding optimal parallelism configurations and the capacity of a DSP job is often used for dynamic autoscaling. These reactive approaches typically profile running jobs and automatically rescale tasks when certain thresholds are reached [6]. Though useful for running applications, there is value in proactively profiling a DSP job in order to understand task-level capacity and set utilization targets.

In this paper we present Rafiki, an approach which automatically determines the processing capacities of individual tasks for any selected streaming job. Rafiki takes advantage of cloud computing, OS-level virtualization, and container orchestration technologies to deploy duplicate DSP pipelines and test their maximum capacity under realistic conditions. By running a series of proactive profiling runs, Rafiki finds optimal parallelism configurations at a task level and reports the maximum throughput possible for those configurations. With the insights gained from the profiling runs, a user can allocate sufficient resources to a DSP job in order to reach utilization targets, which allows for a more accurate estimation of recovery times as well as the identification and removal of performance bottlenecks. Additionally, our method provides an interface for monitoring the capacity utilization for any targeted job after profiling runs have been completed. We provide a prototype and evaluate its effectiveness experimentally with two DSP jobs.

The remainder of this paper is organized as follows. In Sect. 2 we explain our approach. In Sect. 3 we outline Rafiki's design and evaluate our approach with two DSP jobs. We conclude with related work in Sect. 4 and a brief conclusion in Sect. 5.

2 Approach

In order to measure task-level capacity and apply the gained insights to a given DSP job, we propose Rafiki, a three-step solution. First, a set of profiling runs are iteratively executed with increasing and optimal parallelism configurations to obtain maximum capacities across the different tasks. Second, the capacity information for all tasks is deduced based on the metrics gathered from the profiling runs of the tested parallelism configurations. Third, the gathered capacity information is applied to a running job to target a specific utilization. Additionally, real-time insights are provided into the potential effects of changing task parallelisms in an interactive dashboard. An overview can be seen in Fig. 1. We have implemented this approach to validate it and promote its usability.[1]

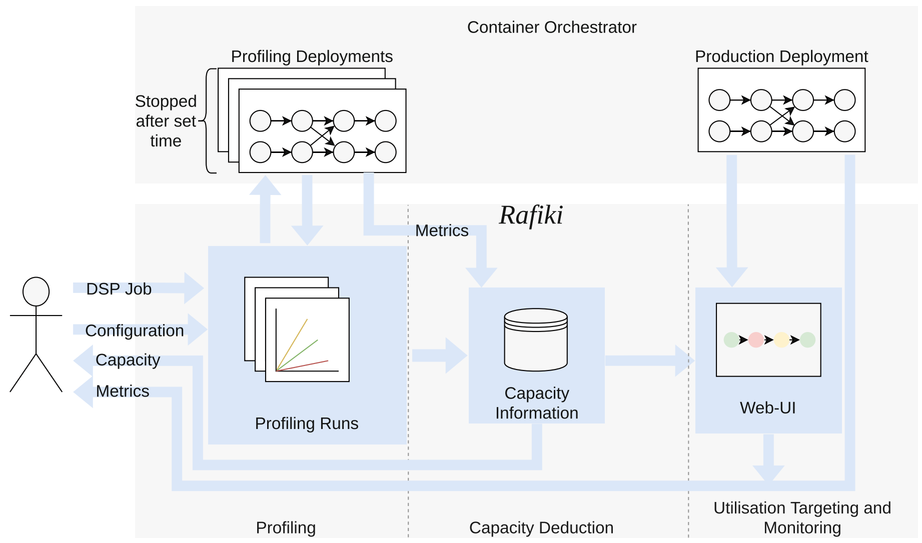


Fig. 1. Overview of the architecture and interactions

2.1 Profiling Runs

The main goal of this step is to calculate the maximum capacity for each task at a given level of parallelism. We achieve this by executing a set of brief profiling runs while stressing the system's processing capacity in every iteration. By ensuring a sufficient amount of messages are available to be consumed, we can flood the entire pipeline to detect tasks that are unable to process messages at the rate they are received. These tasks cause backpressure and indicate the maximum processing capacity of the task at the current parallelism configuration. In each

[1] https://github.com/ciklista/rafiki.

run, the essential idea is to detect tasks that are the bottleneck given the current configuration and increase their parallelism in the next iteration.

A set of profiling runs is always started with all tasks having a parallelism of one. We then subsequently increase the parallelism of certain tasks, one at a time, until no more bottlenecks can be enforced, or a user-defined maximum parallelism has been reached. With each profiling run, we take advantage of cloud computing and container orchestration technologies to deploy duplicate DSP jobs with increasing parallelism configurations that read from the same Apache Kafka source. Messages are replayed from the same offset to ensure that each experiment iteration receives the same message sequence.

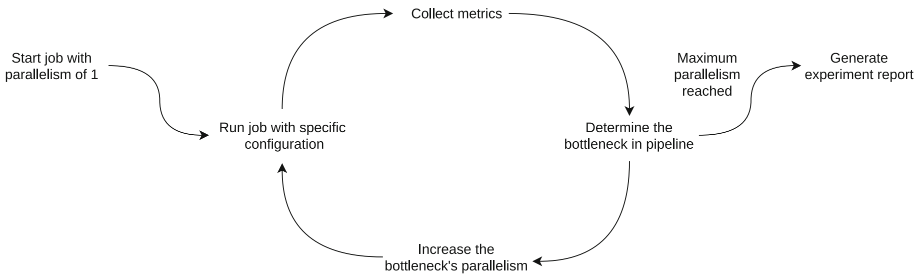

Start job with parallelism of 1
Collect metrics
Maximum parallelism reached
Generate experiment report
Run job with specific configuration
Determine the bottleneck in pipeline
Increase the bottleneck's parallelism

Fig. 2. Profiling run loop

Figure 2 illustrates the experiment loop as well as the decision process of which task parallelism to increase for the next iteration. After a run has successfully completed and metrics have been collected, we check if backpressure was detected on any task throughout the run. If this was not the case, we can assume none of the tasks reached its maximum capacity and no bottleneck was found within the system. In this case, we increase the parallelism on the source task, allowing messages to be consumed at a larger rate. This process is repeated until backpressure is found on any task. Once that happens, we can then deduce which task parallelism needs to be increased in order to resolve this bottleneck. If backpressure is reported for a single task, the subsequent task is not able to consume messages at the same rate at which they are produced. Following this reasoning, we therefore increase the parallelism of the task following the task that experienced backpressure. If we see multiple tasks experiencing backpressure, we will increase parallelism on the task subsequent to the last task that experienced backpressure, in the order of the data flow. We repeat this process until increasing the parallelism of a certain task would exceed the predefined maximum allowed parallelism. This upper bound is set by the user and is derived from financial or host system constraints.

2.2 Deducing Capacity

After each profiling run, throughput and backpressure metrics are collected and used to define the maximum throughput of individual tasks. Based on observed backpressure, we can deduce different types of information for any task x with any task parallelism configuration y as shown in Table 1. Rafiki assumes, that all tasks are isolated and that parallel instances of an operator are similar, i.e. receive a similar share of messages, and that the underlying system operates without failures.

Table 1. Capacity assumptions for task x at parallelism y

Case	Task $x-1$ under backpressure	Task x under backpressure	Assumption
1	✓	✗	Throughput is maximum capacity
2	✗	✓	Throughput is lower bound for capacity
3	✗	✗	Throughput is lower bound for capacity
4	✓	✓	Throughput is lower bound for capacity

One additional special case is a source task, as it does not have a preceding task to monitor in order to determine its capacity. There are two approaches to solving this issue. First, one could identify a metric that follows the concept of backpressure in the system generating the input stream of the DSP job. Alternatively, one could simply aggregate capacity information across all runs for a given parallelism configuration of the source task. By design of the profiling runs, this provides a lower bound for the given configuration. Further, under the assumption of infinite messages at the time of a profiling run, maximum capacity for source tasks is defined as the capacity observed when none of the tasks experienced backpressure. It can then be assumed, that the source task operated at its maximum capacity.

2.3 Utilization Targeting

After successfully completing the profiling runs, the maximum, or at least a lower bound for the maximum, number of messages that a task can process at a specific parallelism has been recorded in the database. With this throughput table we can monitor a running job and deduce the current capacity to achieve a target utilization. A DSP job should typically be run at a percentage of the maximum capacity in order to be able to recover from failures that will likely occur over the lifespan of a long-running job. Effective DSP systems use fault tolerance mechanisms such as checkpointing to periodically create consistent states to recover from in case a failure occurs. Upon failure, messages since the last saved state must be reprocessed in addition to the messages that continue to arrive. Targeting a specific utilization allows a DSP system enough processing capacity to be able to recover from failure. For example, a job processing incoming messages at 100% has no additional capacity for recovery, but a job running at 70% of the maximum capacity has 30% processing capacity for recovery. Having insights into the processing capacity and target utilization also make it possible to estimate recovery times. It is crucial that the running job we monitor is the same as the job we run our experiments with since the capacity depends on the implementation.

3 Evaluation

In this section, we show through experiments that using Rafiki is both practical and beneficial for obtaining the task-level capacity information of DSP jobs.

3.1 Prototype Implementation

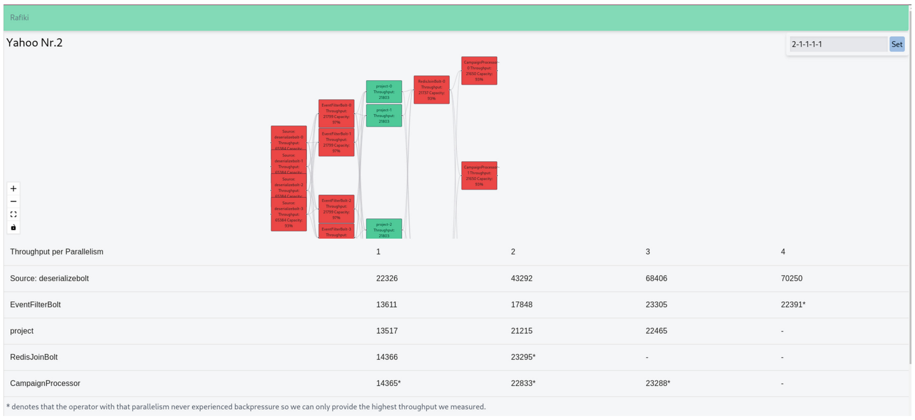
To evaluate our approach, we implemented Rafiki prototypically to work with Apache Flink. The prototype consists of three main components, a Java application, a database, and a web UI. All components ship as docker containers. The core is a Java Spring Boot application that triggers the profiling runs. It publishes an API that can start the execution of jobs on a remote DSP system and supervises the profiling loop. After completing a profiling run, it records and stores the metrics in a PostgreSQL database.

The web UI, depicted in Fig. 3, is a React application that enables a user to upload a custom Java executable to Flink and to set parameters relevant for the profiling runs such as the highest level of parallelism. The web UI calls the APIs exposed by the Java application. Once results are available, the web UI allows for real-time monitoring of a running job and applies the capacity information to single tasks of that job, indicating current capacity via color codes. The web UI also features a sandbox mode that enables users to simulate a different level of parallelism on their job and observe changes in capacity.

3.2 Experiment Setup

Profiling runs were conducted on the Google Cloud Platform[2] in a three node Kubernetes [14] cluster using the Google Kubernetes Engine.[3] Hardware and software specifications are shown in Table 2. Flink was deployed natively in Kubernetes with HDFS [12] being used for the storage of Flink checkpoints. All streaming jobs were configured to consume messages from the Kafka [7] streaming platform. Based on the cluster setup, a maximum parallelism of six

[2] https://cloud.google.com/.
[3] https://cloud.google.com/kubernetes-engine.



Throughput per Parallelism	1	2	3	4
Source: deserializebolt	22326	43292	68406	70250
EventFilterBolt	13611	17848	23305	22391*
project	13517	21215	22465	-
RedisJoinBolt	14366	23295*	-	-
CampaignProcessor	14365*	22833*	23288*	-

* denotes that the operator with that parallelism never experienced backpressure so we can only provide the highest throughput we measured.

Fig. 3. Rafiki UI. Task-level capacity is indicated via different colors. (Color figure online)

was used. The duration of a profiling run iteration was set to two minutes to allow enough time for messages to accumulate and be processed in windowing periods. Each job was profiled with four successive runs with the mean being used for the evaluation.

Table 2. Cluster specifications

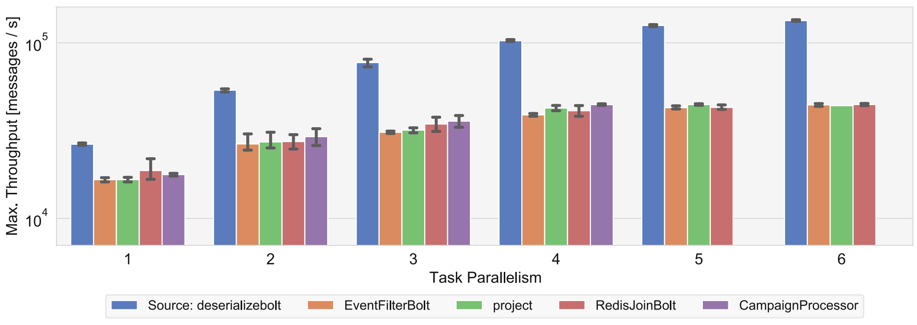
Resource	Details
OS	Ubuntu 18.04
CPU	4 vCPU
Memory	16 GB RAM
Software	Java v1.11, Flink v1.12, Kafka v2.6, ZooKeeper v3.6, Redis v5.0, Prometheus v2.25, Docker v19.3, Kubernetes v1.18, HDFS v2.8

3.3 DSP Jobs

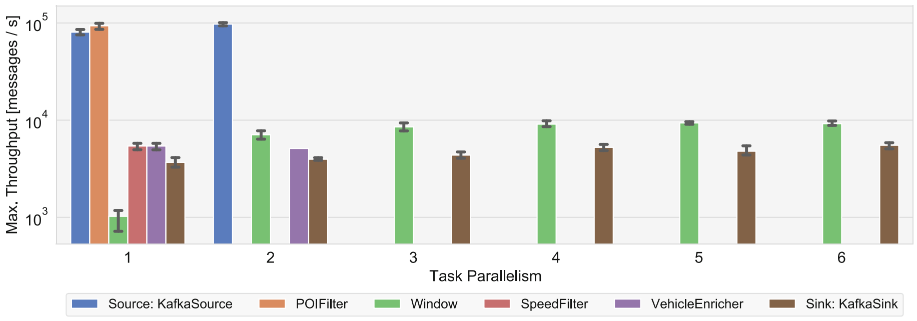
Rafiki was tested with two DSP jobs, the Yahoo Streaming Benchmark [2] and an IoT Vehicles Experiment [3]. Source code for both jobs can be found in the Rafiki repository.[4] The Yahoo Streaming Benchmark is an advertising analytics use case that counts how many times ads from an ad campaign were viewed in a given time window. In the benchmark, ad campaigns with corresponding ad events are synthetically generated at a constant rate. The stream processing job then deserializes events from Kafka, filters them based on an ad type, matches the ad id to a campaign id, and counts how many times ads from a campaign

[4] https://github.com/ciklista/rafiki.

were viewed in a 10-s window. The IoT Vehicles Experiment is an IoT traffic sensor use case that detects speeding vehicles. The experiment uses pregenerated vehicle data that includes the positional data of the vehicles. The data stream is filtered based on certain points of interest and then collected in a window. The vehicle speed is calculated based on the window time and the update interval of the vehicles. Vehicles that exceed a predefined speed are then reported to a different Kafka sink.



(a) Yahoo Streaming Benchmark



(b) IoT Vehicles Experiment

Fig. 4. Profiling results of experiments conducted with our two DSP jobs. For each job, task, and task parallelism, we report the average maximum throughput and the corresponding standard deviation across all conducted runs. For most tasks, it can be observed that the influence of the task parallelism on the maximum throughput is fairly linear.

3.4 Results

Rafiki tested on average 25 configurations for the Yahoo Streaming Benchmark and 17 configurations for the IoT Vehicles Experiment on each run. The results depicted in Fig. 4 show the highest measured throughput for each level of parallelism. Cases where throughput was not recorded indicate that the task could adequately handle the overall throughput at a lower parallelism, and that overall throughput was limited by another task. With a maximum parallelism of 6, each task shows a linear increase in processing capacity with additional task parallelism. Despite slight variance across runs, repeated profiling experiments found the same bottlenecks and generally tested configurations in the same order. We therefore conclude that Rafiki proved its ability to measure task-level capacity and identify bottlenecks that would benefit most from increased parallelism.

Table 3. Validation results

Job	# Messages	Estimated processing time	Average real processing time	Average deviation
Yahoo Streaming Benchmark	51.8 M	398.7 s	445.3 s	−10.31%
IoT Vehicles Experiment	56 M	625.8 s	594 s	5.38%

To validate the measured capacity results, we compared the predicted processing time against the actual processing time, as seen in Table 3. To do this, we accumulated messages in Kafka and measured the time needed to process these messages. We then compared our estimated processing time with the actual processing time. Rafiki underestimated the maximum processing capacity in the Yahoo Streaming Benchmark by 5–14% with an average of 10% and overestimated the maximum processing capacity in the IoT Vehicles Experiment by 3–6% with an average of 5%.

3.5 Discussion

Rafiki was implemented to deduce information about task-level capacity in order to make it possible to monitor and reach utilization targets. While the results for the IoT Vehicles Experiment also show the same relation, we cannot observe as high of a capacity gain per parallelism as in the Yahoo Streaming Benchmark experiment. Although we can still use the capacity information to set a utilization target, the results indicate that the limiting factor for this job is likely not the parallelism. While Rafiki is able to reveal such bottlenecks, it cannot show its source.

The current implementation of Rafiki is limited by a few aspects. External factors such as failing nodes or a bottleneck in the underlying network are not detectable by Rafiki and would alter the results. This issue could be solved in future iterations by extending the range of collected metrics to include these factors and for Rafiki to react to the events. Another limitation is jobs that have multiple sources. If Rafiki does not detect backpressure in the system, it would increase the parallelism for all the sources, likely resulting in a higher parallelism than needed for a subset of the sources. This issue could also be solved by extending our approach.

4 Related Work

Our approach is inspired by previous work in DSP capacity planning. Related work that assesses the maximum capacity of a DSP system typically uses analytical models to predict capacity or profiling techniques to monitor actual capacity in a running system. Profiling-based approaches have been shown to be more accurate. Roy et al. use both analytical modeling and profiling to find the capacity of distributed systems [11]. They find that the accuracy of their models decrease as system activity increases. Bilal and Canini [1] argue that using analytical modeling to predict throughput and latency reduces the accuracy of results as assumptions must be simplified to create models and these models must be regularly updated to reflect changing environmental conditions.

Kalavri et al. [6] use task processing rates to create a model to automatically scale a DSP job according to the current workload. Theirs is a reactive profiling-based approach that works on running systems. Rafiki, by contrast, uses parallel profiling runs to determine the capacity information of a chosen DSP job so that utilization targets can be easily set by a user for different throughput rates thus ensuring any QoS requirements are met.

In [8], the authors propose a prototype called Flink-ER which represents the DSP execution graph as a flow network. Here each task has a capacity and a flow representing the maximum message processing rate and current processing rate respectively. Flink-ER uses graph theory and partitions to identify performance bottlenecks, using network bandwidth and latency as the basic capacity measurement.

To automate finding optimal parameters in DSP systems, Bilal and Canini [1] propose a framework that compares the results of various optimization algorithms. Their profiling-based approach focuses on minimizing latency while providing a minimum level of throughput to obtain realistic results.

Tang and Gedik [13] use an estimation of each task's CPU utilization to generally measure capacity at a task level. The tasks with the highest CPU utilization are identified as bottlenecks and are allocated more resources. After increasing the parallelism of the bottleneck task, it is then tested to see if the throughput increased. In contrast, Rafiki directly identifies bottleneck tasks using backpressure metrics.

Stela [15] identifies bottleneck tasks based on their input and output rate to dynamically scale individual tasks of a DSP system up or down. Congestion is found when the input rate of the data stream is higher than the number of messages that can be processed. This measure of capacity is used by Rafiki, however, it is obtained as a metric directly from backpressure metrics. Stela is an online, reactive application, while Rafiki duplicates configurations pipelines in parallel and intentionally overloads the system to find bottlenecks.

Our overall approach borrows from Chiron [4]. Chiron uses a profiling-based approach to measure the capacity of DSP jobs with QoS requirements to find optimal checkpoint intervals. OS-virtualization, container orchestration, and IaC methods are used to deploy isolated and duplicated pipelines with varying checkpoint interval configurations. In order to test the maximum capacity and increase the number of events processed by the DSP job, events are read from an earlier timestamp. All duplicate pipelines read from the same Kafka topic to increase accuracy [4]. Chiron builds on Timon [3], which tests alternate DSP configurations by deploying parallel pipelines that read from production data streams.

5 Conclusion

Finding optimal parallelism configurations for DSP jobs and determining the maximum throughput those configurations can provide is no easy feat. In this paper we proposed Rafiki, an automated approach for finding optimal configurations and gaining insights into the task-level capacity of a DSP job. A number of proactive profiling runs are conducted where Rafiki uncovers capacity information for individual tasks. This capacity information is collected and makes the process of allocating sufficient resources to meet QoS requirements easier for users. It can be used to estimate recovery times through selecting specific utilization targets, thus helping to create more efficient and reliable DSP jobs. Rafiki was tested experimentally using two DSP jobs and found to accurately measure capacity within an average of 5–10%. Rafiki offers increased usability by providing a web UI that allows for real-time capacity monitoring and experiment evaluation. Future work could enhance and build upon the proposed solution in a number of ways. Though tested prototypically with Apache Flink, the concepts of task parallelism and bottlenecks in stream processing pipelines are common across most DPS systems. Mapping these abstractions to different systems would increase Rafiki's versatility. Bottlenecks could also be defined by metrics other than backpressure, such as processing rates, latency, or CPU utilization.

Acknowledgment. This work has been supported through grants by the German Ministry for Education and Research (BMBF) as BIFOLD (funding mark 01IS18025A) and WaterGridSense 4.0 (funding mark 02WIK1475D).

References

1. Bilal, M., Canini, M.: Towards automatic parameter tuning of stream processing systems. In: SoCC 2017, pp. 189–200. Association for Computing Machinery, New York, NY, USA (2017)
2. Chintapalli, S., et al.: Benchmarking streaming computation engines: storm, flink and spark streaming. In: IPDPSW. IEEE (2016)
3. Geldenhuys, M.K., Thamsen, L., Gontarska, K.K., Lorenz, F., Kao, O.: Effectively testing system configurations of critical IoT analytics pipelines. In: Baru, C., et al. (eds.) Big Data, pp. 4157–4162. IEEE (2019)
4. Geldenhuys, M.K., Thamsen, L., Kao, O.: Chiron: optimizing fault tolerance in QoS-aware distributed stream processing jobs. In: Wu, X., et al. (eds.) Big Data, pp. 434–440. IEEE (2020)
5. Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F.H., Khan, S.: A survey of distributed data stream processing frameworks. IEEE Access **7**, 154300–154316 (2019)

6. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M., Roscoe, T.: Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: OSDI, pp. 783–798. USENIX Association, Carlsbad, CA (2018)
7. Kreps, J.: Kafka: a distributed messaging system for log processing (2011)
8. Li, Z., et al.: Flink-ER: an elastic resource-scheduling strategy for processing fluctuating mobile stream data on flink. Mob. Inf. Syst. **2020**, 5351824:1–5351824:17 (2020)
9. Nasiri, H., Nasehi, S., Goudarzi, M.: Evaluation of distributed stream processing frameworks for IoT applications in smart cities. J. Big Data **6**, 52 (2019)
10. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Comput. Surv. **52**(2), 36:1–36:37 (2019)
11. Roy, N., Dubey, A., Gokhale, A., Dowdy, L.: A capacity planning process for performance assurance of component-based distributed systems. In: ICPE 2011, pp. 259–270. Association for Computing Machinery, New York, NY, USA (2011)
12. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: Khatib, M.G., He, X., Factor, M. (eds.) MSST, pp. 1–10. IEEE Computer Society (2010)
13. Tang, Y., Gedik, B.: Autopipelining for data stream processing. IEEE Trans. Parallel Distrib. Syst. **24**(12), 2344–2354 (2013)
14. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Réveillère, L., Harris, T., Herlihy, M. (eds.) EuroSys, pp. 18:1–18:17. ACM (2015)
15. Xu, L., Peng, B., Gupta, I.: Stela: enabling stream processing systems to scale-in and scale-out on-demand. In: IC2E, pp. 22–31. IEEE Computer Society (2016)