Bitflow: An In Situ Stream Processing Framework

Anton Gulenko, Alexander Acker, Florian Schmidt, Sören Becker, Odej Kao
TU Berlin, Germany
{*firstname.lastname*}@tu-berlin.de

***Abstract*—The timely processing of continuous data streams gains increasing importance in a variety of fields. Self-healing systems depend on efficient data analysis to detect problems and apply appropriate counter measures. This paper introduces Bitflow, a stream processing framework optimized for the data analysis tasks in self-healing IT systems. Numerous algorithmic contributions allow to mine monitoring data obtained from critical system components, in order to detect and classify anomalies, and to localize their root cause. These data analysis tasks are traditionally executed on big data processing platforms, which run on dedicated hosts and assume complete ownership over the occupied resources.**

Bitflow takes a different approach by analyzing the monitoring data directly at its source – i.e., *in situ*. We exploit the fact that IT systems are usually over-provisioned and a fraction of the computational resources can be allocated for self-healing functionality. Bitflow implements a dynamic modeling approach for dataflow graphs, which adapts to varying environments, such as changing data sources, or system components. Further, we describe Bitflow's scheduling approach, which determines when it is beneficial to migrate a data analysis process to a remote host. Experimental data from practical data analysis tasks shows the applicability of our scheduling solution.

***Index Terms*—data streams, stream processing, data analysis, in situ, self-healing, autonomic computing, cloud computing**

I. Introduction

Data stream processing is gaining attention in many domains. One area that depends on timely extraction of knowledge from raw data streams is autonomic computing. Especially the sub-field of self-healing systems requires timely analysis results to detect problems and apply appropriate counter measures.

Many recent contributions focus on applying machine learning and artificial intelligence methods to the field of autonomic computing, and especially on self-healing. This trend gave rise to the AIOps paradigm and self-healing system visions, such as an autonomic self-healing cloud platform [1]. The basis for any self-healing functionality is to continuously collect monitoring data from relevant system components. These data streams are analyzed in real-time to obtain an assessment of the system status. Amongst others, data analysis tasks include anomaly detection, anomaly classification, and root cause analysis.

However, the implementation, scheduling and execution of such data stream processing tasks is challenging, and a variety of frameworks have been developed to facilitate this process. After the introduction of MapReduce [2], several frameworks emerged for managing the execution of batch [3] and stream [4] data processing jobs on large server clusters.

Although these traditional big data frameworks have a wide range of use cases, their applicability in the field of autonomic computing is limited. Firstly, all commonly used stream processing frameworks require a static definition of a directed acyclic graph (DAG), which describes the dataflow (edges) through the data processing steps (graph nodes). After definition and deployment, the DAG structure remains static and does not adjust to changing external factors. Modern IT systems, however, are of highly dynamic nature. Having alternating monitoring data sources, changing data types and formats, or varying sampling frequencies, a stream processing framework must be able to rapidly adjust to such changes. Secondly, to ensure a timely remediation of anomaly situations, the data analysis latency must be minimized. Executing data processing steps on separate dedicated hardware, as it is commonly done in existing stream processing frameworks, necessitates data transmission over the network, which introduces an unpredictable latency. This poses a limitation when repair or remediation decisions rely on real-time analysis results.

To overcome these limitations we introduce the stream processing framework Bitflow, which is based on the concept of *in situ* data analysis to provide predictability and reduce the processing result latency. The main use case of the Bitflow framework is to support the realization of self-healing IT systems. To address the described difficulties, it supports the novel concept of a dynamic dataflow graph model and is designed to colocate stream analysis tasks with regular system workload. By scheduling the stream processing alongside the regular workload of IT systems, the latency overhead of the network is avoided. Furthermore, the colocation allows the data processing resources to naturally scale with the system.

Hence, this paper presents four distinct contributions. First, we introduce a general method for modeling adaptive dataflow DAGs that automatically adjust to the dynamics of IT systems. Second, we present the concept of in situ data stream processing, where data processing steps are colocated with regular system workload. Third, we define a formal scheduling criterion that determines when it is beneficial to move a data processing step to a remote host. Fourth, we provide a complete and functional prototype implementation of the proposed methods in a public Github repository: `https://github.com/bitflow-stream`.

The remainder of this paper is organized as follows. Section II gives an overview over alternative approaches to data stream processing and other related work. Section III introduces the in situ data stream processing framework Bitflow, its modelling

978-1-7281-8414-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ACSOS-C51401.2020.00053

approach for dynamic dataflow DAGs, and its approach to limiting the resources of managed processing steps. Section IV describes our scheduling algorithm and the criterion to decide when a data processing step is moved to a remote host. Finally, section V concludes the paper and outlines future research directions.

II. Related Work

The research direction of *big data* focuses on analyzing data volumes that are not possible to process on a single machine [5, 6]. Platforms for big data processing implement different programming models with varying features and levels of flexibility. Googles MapReduce platform [2] implements massively parallel execution of the well established map-reduce model which allows data transformation through user-defined map and reduce operations. Other widely used platforms support more general data processing workflows on batches of data (e.g. Spark [3]), or both batch and stream-wise processing (e.g. Stratosphere [7] or Asterix [4]). Such platforms usually offer abstraction levels that on the one hand allow concise representation of the operations for the user, and on the other hand allow the platform to optimize the underlying transportation and computation of the data. What all big data platforms have in common is that they assume complete ownership of the underlying compute resources.

Despite their wide range of use cases, big data frameworks are not suited for the domain of autonomic computing. The unpredictable latency induced by the network transmission to dedicated analysis hardware together with the static definition of DAGs are the two major limitations.

Astrolabe [8] is a hierarchical system for storing and aggregating highly distributed data records. Its hierarchical and scalable approach is similar to the data analysis architecture proposed in this paper, but limited to aggregations, and not directly applicable to continuously updated streams of data.

Intanagonwiwat et al. have proposed *directed diffusion*, a communication paradigm for scalable data aggregation by hierarchically distributing the data and the processing workload [9]. The main difference between directed diffusion and Astrolabe is that the former is specifically designed to handle data streams. The TAG (Tiny AGgregation Service) [10] system builds on similar concepts as directed diffusion, but offers a simpler, declarative interface for describing user queries.

Of the published platforms for explicitly monitoring or managing computer systems, only few include a specifically designed data processing strategy. Moore et al. propose Splice, a knowledge plane for data centers that couples static and dynamic metrics of various infrastructural elements with physical meta information such as location and energy consumption [11]. Dean et al. propose to use *residual* resources for analyzing data that is collected on the hypervisors of a cloud data center. The analysis is performed on a dedicated Virtual Machines (VMs) with special scheduling constraints to avoid over-utilization of the hypervisor resources by the analysis processes.

III. The Bitflow Framework

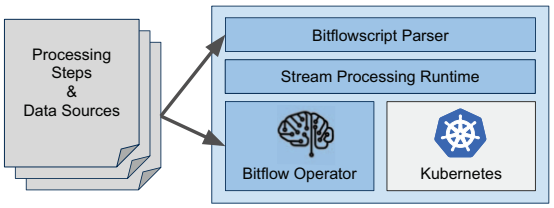

Figure 1: High-level components of Bitflow

This section gives an overview of the Bitflow data stream processing framework and describes how it defines and executes dynamic in situ dataflow graphs. The Bitflow source code is available online[1], as well as its documentation[2].

Bitflow schedules processing steps as containers and uses Kubernetes as the underlying execution platform. Kubernetes is a distributed orchestration platform for containers and supports many different abstractions for expressing how containers are executed and connected. The built-in capabilities of Kubernetes include running individual containers, automatic replication and scaling of identical container instances, precise control over the target execution node, and hard resource usage limitations for each container. Kubernetes itself is lightweight and can be installed on a wide range of devices and infrastructures, ranging from cloud servers to Internet of Things (IoT) and edge computing devices. More importantly, Kubernetes can be installed on the hypervisors of a cloud platform, alongside any cloud operating system. For the remainder of this paper we focus on the use case of a self-healing cloud. However, Bitflow is applicable to any IT system where Kubernetes can be used to orchestrate containers among available nodes.

Figure 1 shows the main components of Bitflow. The core component is the *Bitflow Operator*, which uses the Kubernetes operator pattern[3] to schedule processing step containers across the cluster. The spawned containers execute instances of Bitflow's *Stream Processing Runtime*, which implements data stream operator graphs in the Go programming language. Bitflow's default processing step containers are openly available[4]. The Bitflow Processing Runtime supports a number of common stream operators, such as filtering, normalization, and aggregation. More advanced user-defined operators, such as clustering or machine learning algorithms, are added through a plugin mechanism. The actual operator graphs that run in a processing step container, as well as their configurations, are expressed in a lightweight Domain Specific Language (DSL) *Bitflowscript*. Bitflowscript allows to define a directed acyclic graph of operators in a function-like syntax. An important benefit of the Bitflowscript DSL is that it allows to modify the operator graph in a processing step template without

[1] https://github.com/bitflow-stream
[2] https://bitflow.readthedocs.io/en/latest
[3] https://kubernetes.io/docs/concepts/extend-kubernetes/operator
[4] https://hub.docker.com/orgs/bitflowstream

compiling, building and deploying a new processing step container.

Figure 2 shows how Bitflow distributes processing step containers in the Kubernetes cluster. The main input for the orchestrator is a model of the dataflow graph, which is described below (Section III-A). The orchestrator uses this model to schedule processing step containers on the appropriate hosts.

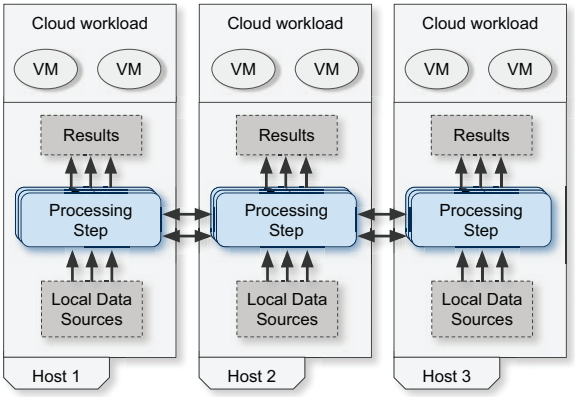

Figure 2: In situ stream processing with Bitflow: processing of local data sources

Each processing step container reads its input data in form of high-frequency data streams. Bitflow transmits data in a custom, dense binary format. A single processing step can also receive multiple input streams. For example, it is often necessary to combine the data of multiple monitoring systems to obtain a richer representation of a single system component. Another important use case for multiple input streams are higher-level processing steps, which further analyze the intermediate results of lower-level steps.

A. Modeling the Dataflow Graph

An in situ dataflow in Bitflow is a nested directed acyclic graph (DAG) that contains *data sources* and *processing steps* as nodes, as well as data streams as edges. Bitflow dataflows are not defined as static, fixed graphs, but rather as a loosely coupled collection of *processing step templates*, which the system uses to dynamically construct the actual dataflow graph. This model supports highly dynamic, self-healing IT systems, where monitoring data sources continuously appear and disappear. The main task of the dataflow model is to express the processing step templates and data sources that form a dataflow graph. The model is visualized in Figure 3. Both data sources and processing step templates are formatted as YAML or JSON objects and stored directly in the underlying Kubernetes cluster, where the Bitflow operator evaluates them to construct the dynamic dataflow graph.

Every data source contains a list of descriptive attributes (*labels*). Examples of potentially useful data source labels include the monitoring system that collects the data, the system layer of the monitored component (e.g. `physical`, `virtual`, `service`), the component that was monitored,

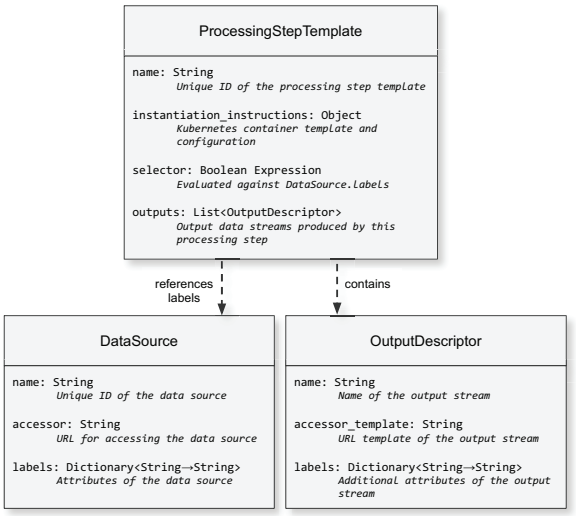

Figure 3: Dataflow model with decoupled data sources and processing steps

possible parent components, and the data sampling frequency. A processing step template consists of an operator subgraph, as well as a boolean *selector* expression, which is matched against the labels of all known data sources and determines whether the processing step is compatible with the respective data source. Compatible pairs of data sources and processing step templates result in an instantiated processing step in the actual dataflow graph. When a processing step is instantiated, every contained leaf operator node implicitly declares an artificial data source, which is recursively matched against all known processing step templates. This recursive algorithm terminates when no new processing steps are instantiated, or when a processing step would consume its own output (to avoid loops). Aside from labels, each data source definition contains a data stream *accessor*: the URL for connecting to the data stream.

On the other hand, a processing step template contains all instructions necessary to deploy an instance of the respective processing step container. This consists of a Kubernetes container template, including the container name and all necessary configurations. Besides the container template, the processing step template defines an operator graph expressed in Bitflowscript. Finally, the *data source selector* is a list of rules that Bitflow evaluates against the labels of each known data source. The result determines whether a data source is compatible with the respective processing step. Compatible pairs of data sources and processing step templates result in the deployment of a processing step container.

B. Enforcing Resource Limitations

Bitflow enforces resource limitations on the managed processing step containers. These limitations are based on user-defined bounds of resources that are reserved in the cluster for data analysis. These bounds cover all relevant system

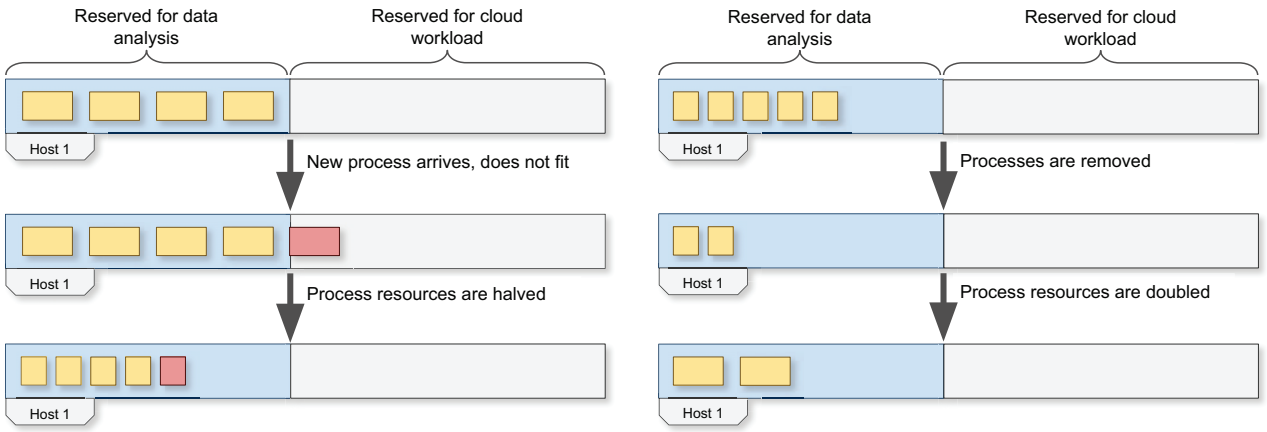

Figure 4: Exponential growth policy when starting (left) and stopping (right) data processing steps

resources, including CPU utilization, allocated memory, hard disk access, and network bandwidth. In order to make the configuration task easier for the user, the simplest way to configure these bounds is by providing a single figure that determines the global share of all resources reserved for data analysis (e.g. `2%`). Alternatively, the user can express the resource bounds individually, and either in relative or absolute terms. Bitflow computes the resources available for every spawned container, and uses Kubernetes' native mechanisms for enforcing the computed limits.

These techniques for limiting resource consumption do not allow for online updates of the enforced limits. Therefore, constrained containers must be restarted when these limits are changed. Updating the limits becomes necessary when too many processing step containers are spawned on one host, so that the previously running containers must be "shrunk" in order to fit within the available resource bounds. This can lead to a cascade of restarting containers when containers are frequently started and stopped. Bitflow mitigates this problem by using an *exponential growth policy* when computing the share of resources given to each container. Instead of always splitting all resources between the running containers, Bitflow leaves spare resources that allow to spawn a number of additional containers without any restarts. When the spare room is filled up, the share of resources given to each container is halved, resulting in twice as many slots for running containers. The same procedure is applied when containers are stopped: Once three quarters of the reserved resources are free, the number of reserved slots is halved, resulting in a new situation where half of the resources are allocated, while the other half is reserved for starting new containers. As a result, all containers receive twice the resources as before. Figure 4 demonstrates this procedure, both when increasing and decreasing the number of reserved slots for containers. To avoid an indefinitely growing queue of input data, Bitflow implements load shedding and a light form of backpressure propagation to signal that a dataflow is congested.

IV. Scheduling of Processing Step Containers

For Bitflow's scheduling approach, we assume a flat network topology where all hypervisor nodes have the same distance to one another. Further, each hypervisor can host an unlimited number of processes due to the adaptive resource limitations described in the previous section. These assumptions result in a straightforward scheduling algorithm: Processing steps are placed on the node that contains their data source.

To mitigate overload situations for individual processing steps, we add an adaptive extension to this static scheduling algorithm. When an processing step exhibits an overload situation, it can be beneficial to move it to another node, in order to distribute the load more evenly. After such a migration, the input data stream must be forwarded from the data source to the new execution host, which results in an increased data processing time. However, the evenly distributed load, and the better resource availability on the migration target, potentially outweigh this performance penalty. Therefore, we define the following dynamic scheduling criterion: A processing step is migrated, when the migration would result in a lower data processing time, despite the added time for transporting the data over the network.

In order to compute Bitflow's migration criterion, we have to compare the current data processing time of the processing step with the predicted processing time after the migration. We predict this processing time using a formal *processing time model*. Before defining the processing time model, we declare the following notation.

A sample is an atomic piece of data received and produced by processing steps. Let $T(H_1, H_2)$ denote the time to transmit a sample over the network from host H_1 to host H_2. For the sake of simplicity, we assume a constant network latency and speed over time, and also that each sample has the same size.

Let the function $send(\mathcal{H}, H)$ denote the time to send a sample from all hosts in the host set $\mathcal{H} = \{H_1, ..., H_n\}$ to the host H. This time is zero when forwarding the sample within the same host, and it is equal to the largest forwarding

time, when sending data from at least one remote host, since all samples are sent in parallel:

$$send(\mathcal{H}, H) = \begin{cases} 0 & if \;\; \mathcal{H} = \{H\} \\ \max_{H_i \in \mathcal{H}}(T(H_i, H)) & else \end{cases} \quad (1)$$

Let $compute_A(R)$ denote the time for the processing step A to compute the results for one sample, when it is executed on the resources R.

Now we can define the data processing time of a processing step A as the function $P_A(\mathcal{H}, H, R)$, which is the sum of sending all samples from the n data sources $\mathcal{H} = \{H_1, ..., H_n\}$ to host H, and the time to compute the result:

$$P_A(\mathcal{H}, H, R) = send(\mathcal{H}, H) + compute_A(R) \quad (2)$$

If we can calculate P_A for every combination of hosts, processing steps, and resources, we can use it to test our migration criterion the following way:

1) Compute $P_A(\mathcal{H}, H, R)$, where H is the current execution host and R the currently available resources.
2) Compute $P_A(\mathcal{H}, H^*, R^*)$ for every potential migration target H^*, setting R^* to the resources that would be available on that host.
3) If one of the migration targets leads to a lower result than the current execution host, migrate the processing step there.

In order to calculate $P_A(\mathcal{H}, H, R)$, we need to compute the sub terms $send(\mathcal{H}, H)$ and $compute_A(R)$. Computing $send(\mathcal{H}, H)$ depends on the values $T(H_i, H)$ for all $H_i \in \mathcal{H}$, which we obtain by observing the data stream characteristics during runtime. Due to the assumption that the network latency and size of samples remains constant, we do not model dynamic characteristics of the network. We model the compute time $compute_A(R)$ with the following parametric function:

$$compute_A(R) = a * (R + b)^{-c} + d \quad (3)$$

This polynomial function with four parameters models a declining compute time with rising resources. Figure 5 illustrates this parametric function, and how it affects Bitflow's migration criterion for a processing step that runs on the same host as its data source. Intuitively, the time to compute a result increases with decreasing resources. With zero resources, the processing time is infinite. However, the processing time also never falls below a threshold defined by the parameter d, which is bounded by the processing step's complexity and by the underlying hardware. The resources can be reduced without any implications, until $compute_A(R)$ starts rising. At this point, we can continue reducing the resources, until $compute_A(R)$ increases by more than the network transportation time to a remote host. Now, if we have to reduce the resources even further, it is more beneficial to migrate the processing step to a remote host.

In order to compute the parametric function $compute_A(R)$, we have to find fitting values for the parameters a, b, c, and d. These values are specific for a combination of a processing step and an underlying execution platform. The procedure to obtain the parameter values consists of a series of controlled experiments. First, the processing step is executed with the maximum available resources, while measuring the average time it takes for it to process one sample. Then the resources are reduced in regular steps, while continuing to take the processing time measurements. This procedure produces a set of sample points for the $compute_A(R)$ function. In the final step, we fit the parametric function $compute_A(R)$ to the collected points, using the Levenberg-Marquardt algorithm for non-linear least squares optimization. The result is a $compute_A(R)$ function with specific parameter values, which predicts the processing time of the step for arbitrary resources on the underlying hardware.

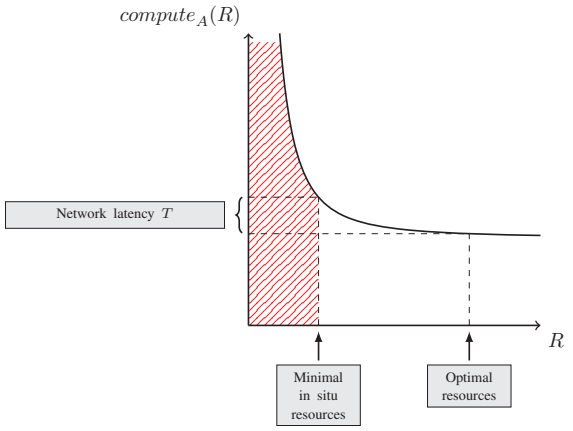

Figure 5: Predicting the processing latency after migrating the processing step

To demonstrate the applicability of our approach, we have performed the procedure to obtain the parameters for $compute_A(R)$ for three different algorithms. The evaluated algorithms include different models used for anomaly detection. The first evaluated model is based on a neural network with Long-short term memory (LSTM) neurons. The second model uses multivariate online ARIMA [13], an approach for modeling and predicting periodic time series. The third model is not based on neural networks, but instead uses a variation of the clustering algorithm BIRCH, optimized for evolving data streams. We performed the experiments to obtain the necessary data on a physical machine with a Quadcore Intel Xeon CPU (E3-1230 V2 3.30GHz), 8 virtual cores, and 16 GM of RAM. In each experiment run, a data set consisting of 10.000 samples, with 28 metrics per sample, was processed by the respective algorithm, while measuring the average processing time per sample. Starting with 8 virtual CPUs (which results in no resource limitation at all), the number of CPUs allocated for the process were reduced in steps of 0.3 after each experiment run, until reaching the minimum assignable resources at 0.1 CPUs.

Figure 6 shows the results of our experiments. The crosses show the experimentally collected samples, while the lines

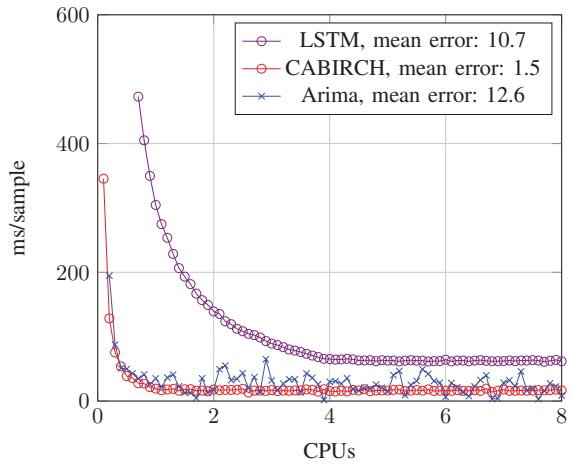

Figure 6: Modeling the processing time of different classes of algorithms with the parameterized $compute_A(R)$ function

show the resulting parameterized $compute_A(R)$ function. All algorithms show the expected behavior of rapidly increasing processing time with dropping resources. For every algorithm, we computed a *mean error* values, which is the mean absolute deviation of the fitted $compute_A(R)$ from the actual measurements. The low mean error values, which we obtained for the three algorithms, confirm that our parameterized model of $compute_A(R)$ is well-suited to represent their behavior.

V. Conclusion

In this paper, we presented the open source data stream processing engine Bitflow. Bitflow follows the concept of in situ data analysis to overcome the drawbacks of a dedicated data analysis infrastructure. Instead of sending data to a remote host for processing, Bitflow leverages the local resources of each node that collects or generates data. The main use case for such a data processing engine is a self-healing IT system that continuously monitors and analyzes all of its relevant components. Bitflow features a dynamic dataflow graph model that adapts to rapidly changing systems, yet is capable of expressing arbitrary DAGs of processing steps.

Further, this paper presented Bitflow's dynamic scheduling approach that optimizes the data processing time of resource-constrained processing steps, while considering both the network transportation of the input data and the computation time itself. Our approach is based on a data-driven model of the algorithm's computation time, depending on the allocated CPU resources. Using three example algorithms, we showed that our computation time model is applicable to a wide range of algorithm classes.

We believe that in situ data analysis is a viable alternative to traditional big data platforms in many use cases and future research will further uncover its potential in areas such as scheduling of data processing steps.

References

[1] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, "A System Architecture for Real-Time Anomaly Detection in Large-Scale NFV Systems," in *Procedia Computer Science*, vol. 94. Elsevier, 2016, pp. 491–496.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[3] M. Zaharia, M. Chowdhury, T. Das *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[4] A. Behm, V. R. Borkar, M. J. Carey *et al.*, "Asterix: towards a scalable, semistructured data platform for evolving-world models," *Distributed and Parallel Databases*, vol. 29, no. 3, pp. 185–216, 2011.

[5] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, pp. 137–144, 2015.

[6] P. Zikopoulos, C. Eaton *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[7] A. Alexandrov, R. Bergmann, S. Ewen *et al.*, "The stratosphere platform for big data analytics," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 23, no. 6, pp. 939–964, 2014.

[8] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM transactions on computer systems (TOCS)*, vol. 21, no. 2, pp. 164–206, 2003.

[9] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *international conference on Mobile computing and networking*. ACM, 2000, pp. 56–67.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[11] J. Moore, J. Chase, K. Farkas, and P. Ranganathan, "A sense of place: Toward a location-aware information plane for data centers," *Hewlett Packard Technical Report*, 2004.

[12] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *international conference on Autonomic computing*. ACM, 2012, pp. 191–200.

[13] F. Schmidt, F. Suri-Payer, A. Gulenko, M. Wallschläger, A. Acker, and O. Kao, "Unsupervised anomaly event detection for vnf service monitoring using multivariate online arima," in *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2018, pp. 278–283.