Towards Performance Prediction for Stream Processing Applications

Johannes Rank
Technical University of Munich
Boltzmannstr. 3
Garching, Germany
johannes.rank@in.tum.de

Dominik Paluch
Technical University of Munich
Boltzmannstr. 3
Garching, Germany
dominik.paluch@in.tum.de

Harald Kienegger
Technical University of Munich
Boltzmannstr. 3
Garching, Germany
harald.kienegger@in.tum.de

Helmut Krcmar
Technical University of Munich
Boltzmannstr. 3
Garching, Germany
helmut.krcmar@in.tum.de

ABSTRACT

Stream processing systems have become the major engine whenever data processing with low latency or real-time capability is required. Meeting the performance demands of these systems in terms of latency, throughput and resource utilization is hence crucial to ensure stable operation and correct functioning. Current performance modeling approaches target this issue by predicting performance characteristics on the architecture level, e.g by measuring existing deployments as a whole and estimating the impact of related changes. However, these approaches neglect the actual application logic of the streaming system and are hence neither able to identify bottlenecks in the processing behavior nor to predict performance during development. In this paper we propose a more comprehensive performance modeling approach that considers both, the deployment architecture and the actual implementation logic of stream processing systems on the example of the SAP HANA Streaming Analytics Server. For this means, we derive a performance model based on the actual source code of the stream processing solution and compare the simulation of our model with the actual measurements. Our results show that the approach provides much better findings allowing to identify bottlenecks in the application logic and to predict the point in time when an over-queuing situation occurs to provide valuable insights for both developers and performance analysts.

Keywords

Stream Processing, Performance Prediction, Over Queuing

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]

31st GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

1. INTRODUCTION

Current trends in the area of Big Data have increased the necessity to process data as it arrives instead of collecting the information for later analysis. In many scenarios, the data is most valuable at the time of its generation but loses its importance within few seconds or minutes [4]. A common example for such domains is predictive maintenance, which aims to estimate an equipment failure before the problem actually occurs and hence allows to prevent a complete breakdown by maintaining the system in time [13]. Such setups require monitoring and analyzing sensor data as it arrives. This is why so-called stream processing systems (SPS) become more and more important in areas such as IoT or Industry 4.0 [12].

SPS are designed to continuously handle incoming events by providing real-time computation to allow an immediate reaction on detected trends or patterns. Due to the sequential processing model of stream applications, an overflow of an internal queue affects all preceding operations and could spread until all prior queues are filled up. These so-called over-queuing situations heavily increase the latency of a system and, if real-time processing is required, potentially bring the whole business scenario to fail. The development of stream processing solutions thus requires not only to ensure a high throughput, but also to prevent over-queuing situations despite varying workload situations [7]. For these reasons, performance is not only a quality of service aspect, but vital for the whole streaming application scenario to succeed [14]. It is hence a crucial task to ensure performance of SPS during both, the development of the streaming application and the operation of the system.

However, despite the necessity to build SPS that meet the performance demands, there is currently a lack of expertise to develop and operate these applications in an efficient manner [3]. According to Requeno et al. (2017), "[...] there is now an urgent need for novel, performance oriented software engineering methodologies and tools capable of dealing with the complexity of such a new environment"[11]. Performance simulation and prediction approaches provide promising solutions for this issue by allowing to determine bottlenecks and answer sizing questions during both system operation and development [5].

We target this issue by proposing a performance prediction approach that simulates the performance aspects of the SPS for each processing task of the application logic, instead of the component as a whole. For this reason, we apply a transformation of the source code in to a performance model to provide simulation based predictions. In this paper we focus on the internal queues in order to predict over-queuing situations and determine the bottleneck task in order to provide better insights than traditional prediction approaches. We evaluated our approach by predicting the performance of an HANA Streaming Analytics Server and compared the results with actual measurements. This paper is organized as follows. Section 2 describes existing research in the area of performance prediction for SPS. Section 3 introduces the concept of stream processing as well as some related considerations regarding performance. Section 4 presents our modeling approach including our experiment. Finally, section 5 concludes this paper and states future research directions.

2. RELATED WORK

Despite the fact that the first stream processing engines were already proposed several years ago [1] the number of performance prediction approaches related to these systems are sparse. One of the first approaches that focus on Apache Spark streaming was proposed by Kroß/Krcmar (2016). They applied the Palladio Component Model (PCM) [2] to simulate the Spark processing framework and parametrized the PCM instance based on measurements obtained from running the HiBench [1] benchmark suite. The approach focuses on scalability prediction by measuring an existing stream processing system and predicting its throughput under varying workload scenarios. The simulation yielded accurate results with prediction errors between 0,67% and 3,41% [7]. However, the approach treats the application logic as a black box and hence does not provide any insights into the application's behavior.

Requeno et al. (2017) propose a framework based on UML profiles to model Apache Storm applications. This includes the modeling of the Storm topology (layout of spouts and bolts) as well as the deployment of the respective nodes. Afterwards an automatic model-to-model transformation derives a generalized stochastic petri net for simulation purpose. The prediction covers the thread utilization (CPU) and execution time for each bolt [11]. However, the approach does not give further insights into the operations contained in each bolt as well as the internal queuing behavior. In addition, it requires to design and parametrize the whole model with UML which can be considered as time consuming and exhaustive. Lin et al. (2018) propose an ABS based model to simulate Spark streaming applications [9] similar to the approach in [7]. ABS is a language to describe the behavior of distributed object-oriented systems. Their approach allows to configure the stream application, as well as the Spark processing framework itself and therefore provides the capability to evaluate different deployment settings. The focus of the approach is to predict the throughput. RAM and CPU can be parametrized but their utilization is not part of the prediction. Again the approach does not provide detailed insights into the actual processing tasks contained in each stage and also internal queues are not considered.

[1] https://github.com/intel-hadoop/HiBench

3. STREAM PROCESSING

SPS are designed to continuously process large volumes of data. In contrast to traditional database systems, a streaming system does not necessarily persist any information, but rather treats the data on the fly by maintaining a global state through continuous analyzing or by modifying the data before forwarding it another system. The data sources for SPS are typically sensors or edged computing devices that monitor a certain object and report their observations continuously in the form of events. Current SPS distinguish between two processing models micro-batching and event-processing. Systems like Apache Spark Streaming that apply a micro-batch processing model do not immediately process events but rather collect them in a job. As soon as a predefined threshold is reached, the microbatch job is released and processed by the engine, which typically results in better throughput rates at the cost of a higher latency. In contrast to that, systems like Apache Storm or Flink apply event-processing and handle the arriving events immediately as they arrive. Other engines as for example the SAP HANA streaming server support both processing models and therefore leave the decision to the developer. The stream application as part of the SPS determines how an event is treated by the system by defining a sequence of connected operations (e.g. filters or functions) that form a directed acyclic graph (DAC). Depending on the actually used SPS framework, the terminology for an operation varies. Apache Spark Streaming uses the term *stage* to describe a set of operations that is executed by one node, whereas Storm uses the term *bolt* to describe a similar structure. In the context of this paper we use the HANA streaming terminology and simply refer to a stream operation as a *task*.

3.1 HANA Stream Processing

In contrast to the open source representatives Apache Spark Streaming and Apache Storm, the SAP HANA streaming server is a commercial product that is integrated into the HANA database platform and focuses on analytical workloads. Therefore it provides several libraries e.g. to support event-based machine learning based on data streams. The HANA streaming server has a monolithic structure and focuses on single deployments rather than distributed landscape with multiple worker-nodes. Streaming application are implemented in projects by using the Continuous Computation Language (CCL) and pushed onto the streaming server. CCL has an SQL like syntax but the server does not execute the statement only once but repeatedly for every incoming event. The supported IDE for CCL development is Eclipse in combination with the streaming development plugin. Aside from the source code view of the streaming project there is also a graphical representation provided. Figure 1 shows a visual example of a HANA streaming project, inspired by the RIOT ETL streaming application benchmark [12]. Events arrive in the *.csv* format and are firstly parsed into an internal structure. Afterwards a *range filter* is applied to omit outliers. The succeeding *bloom filter* checks if the tuple belongs to a predefined set of events and discards the information otherwise. Afterwards an interpolation is performed to enrich the data with additional information, followed by a *join operation* that merges the event with historical data provided by a HANA in-memory database. Finally, the streaming server *annotates* the tuple with a timestamp and forwards the information to a database for persistency.

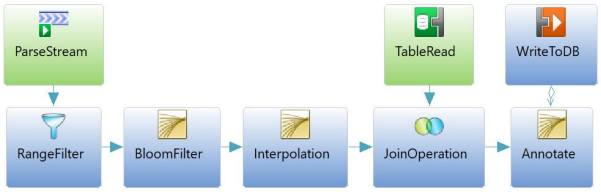

Figure 1: Visualized Stream Application

3.2 Performance Considerations

Based on the example introduced in Figure 1, several aspects are worth considering in terms of performance that apply to most streaming solutions. Firstly, due to the time criticality of most stream processing business cases, the latency is of special importance for any SPS due to the reasons explained in section 1. Since the information value of many events expire within short time periods it is crucial to ensure that the data items are processed in a timely fashion. Secondly, the throughput is a valuable characteristic to decide which volumes of data the streaming engine is able to handle. It is defined by the slowest *task* (bottleneck) of the streaming application and hence should be the focus of any performance improvement activities. For SPS also the memory is of particular interest, since most SPS are designed to keep all the events and processing logic in memory. If data has to be stored on disk or paging is required due to limited RAM resources, the latency heavily increases [14]. Some streaming frameworks such as Apache Flink even crash if memory over allocation occurs [10].

Another characteristic of SPS are the internal queues. The HANA streaming engine automatically provides one queue prior to each task to temporarily buffer events if the corresponding task is still occupied by another data tuple. Since most SPS are applied in the context of IoT, the corresponding workload is mostly data-driven and often harder to predict in terms of volume, velocity and variety [8]. Internal queues hence allow to handle temporary bursts of data that exceed the usual throughput of the system without crashing the whole application. The situation when one queue is filled up is called over-queuing and is one of the severest vulnerabilities of a SPS. Since every event has to be processed according to the DAG, preceding tasks will start to queue-up as soon as the bottleneck queue is full, because they cannot continue with their operation as long as there is no space left to store the output. Figure 2 depicts an example with three queues. Since *task3* is the bottleneck, due to its processing rate of 1/s, its preceding *queue2* will start to fill up as soon as *queue3* is full. Afterwards, *task2* is not longer able to sustain its high processing rate because no space is left for putting the processed events. Therefore, *queue2* and eventually *queue1* will also queue-up until no more new events can be accepted leading to a forced load-shedding. Even if not the whole system is over-queued, each additional event increases the whole latency of the application. In the worst-case, this could lead to a situation where all the events that are currently in progress by the system are already outdated. For developers and performance analysts it is hence most important to identify the bottleneck task and its throughput. This allows to determine at which rate the queue starts to fill up and how long it would take until over-queuing arises in situation of unexpected data bursts.

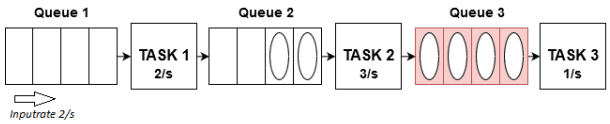

Figure 2: Component Diagram

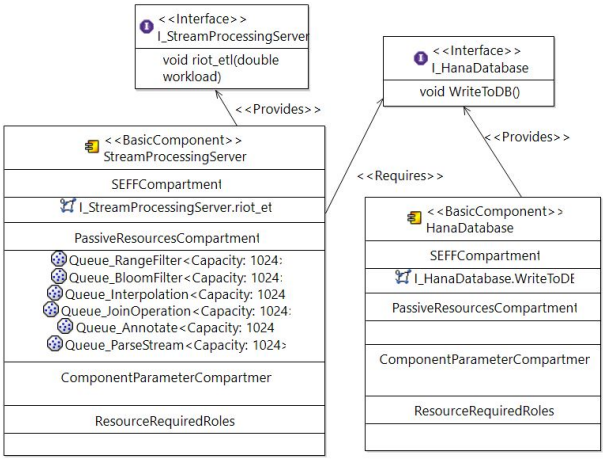

Figure 3: Component Diagram

4. PERFORMANCE PREDICTION

Due to the characteristics stated in section 3.2 we emphasize the importance to build performance models that simulate not only the SPS as a whole but considers all the individual tasks and queues of the application logic as well. For this reason, we perform a manual transformation of a streaming application based on its CCL source code into an instance of the Palladio Component Model. PCM is a meta-model that consists of several model parts to separately describe the different performance aspects of the modeled system such as its components, resource environment, usage profile and behavior.

4.1 Modeling Approach

Preliminary to the model transformation, we perform a load test with a simple training workload in order to obtain the maximum processing rate of each task as well as the average drop rates of the different filter operations. For this reason we use the built in *streamingmonitor* utility to measure each second the number of processed events for every task. We use this information later to parameterize our model. Furthermore, we configure our streaming system to apply event-processing instead of micro-batching. As a first step of our model creation, we define the general architecture of our system by using the PCM repository diagram. As depicted in Figure 3 this diagram provides information about our system's components as well as the provided interfaces and services. The performance simulation in this paper focuses on the stream processing server. However, since our stream application persists the processed events eventually to the database, we need an additional component to model these external calls. This allows us in future work to extend the model with a prediction of the database utilization, depending on the streaming servers output. The stream processing server contains six internal queues, one for each of the diffe-

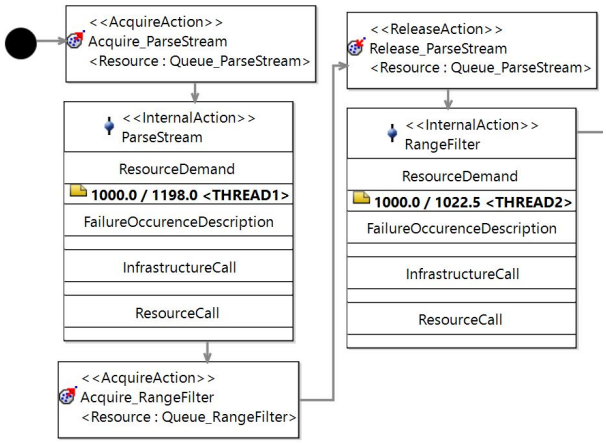

Figure 4: SEFF Queuing Process

rent processing tasks. The *TableRead* and *WriteToDB* operations are part of the related processing tasks and hence do not own a dedicated queue. On the architecture level, we model a queue as a passive resource with a capacity of 1024 events, which is the default size for the HANA streaming server. For each service provided by a component via its interface, a so-called *Service Effect Specification* (SEFF) is created. The SEFF itself is another model that is linked to the repository diagram and describes the actual behavior of the respective services depcited as a finite state machine. In order to obtain the SEFF for the streaming application, we perform a manual transformation of the CCL source code into its respective PCM model elements. Each streaming task is mapped to an *internal action* and parametrized with the average processing rates obtained by the initial measurements. An *internal action* describes a local resource demand invoked by processing a single event. In this example we only define a throughput rate in order to limit the number of events that can be processed per second. In future work an assignment of hardware resources such as CPU or RAM would also be possible. Before entering the task, each event has to acquire a token from the related *passive resource* (as defined in the repository diagram) and is only allowed to enter the task in case at least one token is available, which is similar to concept of semaphores. After leaving the task the event has to acquire the token from the next queue before releasing its hold. Figure 4 depicts this behavior on the example of the *parse stream* and *range filter* action.

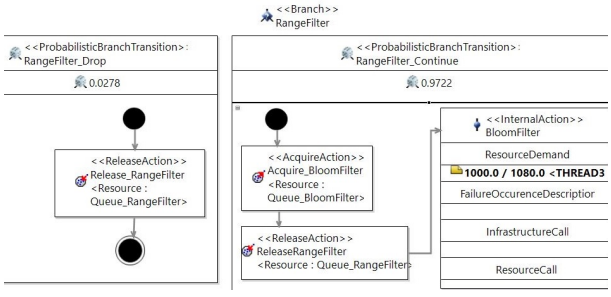

Figure 5: SEFF Filtering Process

In case the task is a filter operation, it is followed by a

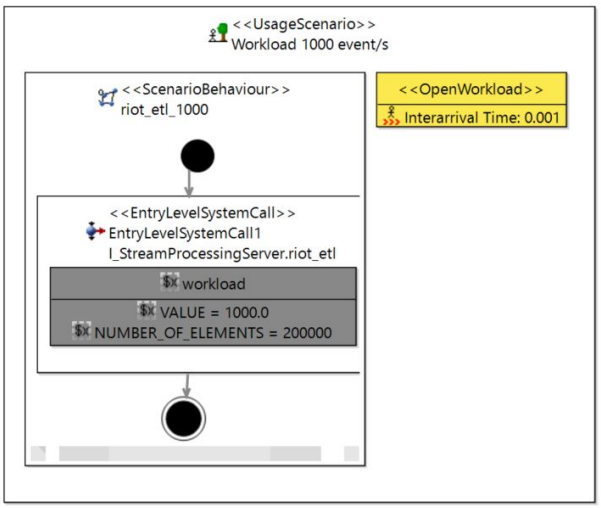

Figure 6: SEFF Queuing Process

branch action. PCM supports two types of branch transitions, the *guarded transition* requires a boolean expression to decide which branch to choose, whereas the *probabilistic transition* only needs a probability assigned to it. For our model we choose a *probabilistic transition* in order to represent the average drop rates of the respective filters. As depicted in Figure 5 a filter processes an incoming task and hence invokes a processing resource demand. If the event is dropped it still has to release its claim on the current queue, otherwise the event is forwarded to the next internal action. The workload is defined in the *usage model*. For our predictions we define two different *usage scenarios* for the data rates 1000 and 1600 events per second. This allows us to simulate both load situations without changing the model. We choose an *open workload* since requests arrive at a certain rate which implies that the total number of events in the system can be variable in contrast to a *closed workload* model. This is required in order to predict the over-queuing situations. The *scenario behavior* describes how each event interacts with the system. In our case it just enters the SEFF of our application and passes the variable *workload* that contains some characteristics about the current usage scenarios. Figure 6 displays the scenario for 1000 events per second.

4.2 Experiment

For our experiment we used an electricity-usage dataset obtained from the Boston Central Library [2] as our workload. The dataset consists of about 250000 power measurements that were recorded every 5 minutes since the year 2016. We used the first 200000 tuples of the dataset and split them into a training, and test set with 100000 records each. For our streaming system we setup a virtual machine with 1.0 processing units and 32 GB RAM based on an IBM Power E870 Server with 4.19 GHz and SLES12 SP02. We used the training set to perform a simple load test, while simultaneously monitoring each task via the *streaming monitor* utility to obtain the parametrization for our model. Afterwards we ran the simulation of our PCM instance multiple times

[2]https://data.boston.gov/dataset/central-library-electricity-usage

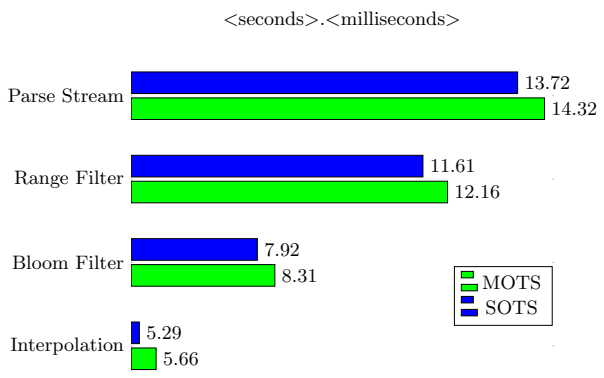

Figure 7: Over-queuing Prediction 1000/s

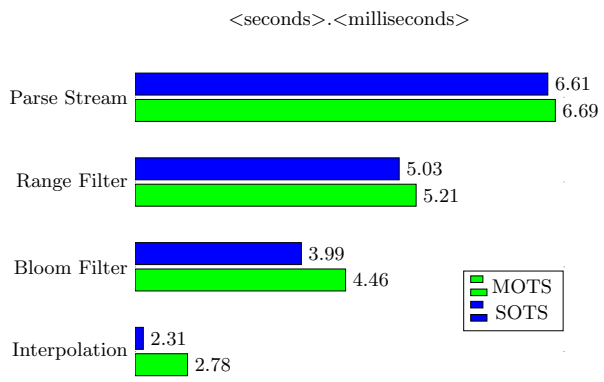

Figure 8: Over-queuing Prediction 1600/s

by using the SimuCom[3] plugin to predict the performance for the two different workload situations. We choose the event rates 1000/sec and 1600/sec to include a small scalability prediction and compared the results with corresponding measurements obtained by running the test workload. Our simulation focused on the utilization of the internal queues or more precisely at which point the different queues in the system will be full. Figure 7 compares the measured over-queuing timestamp (MOTS) and the simulated over-queuing timestamp (SOTS) for an event rate of 1000 records per second. After 5.66 seconds, the queue of the *Interpolation* task is filled up. Through this effect, also the preceding *Bloom Filter* queues eventually up after 8.31 seconds, even if its capable processing rate is much faster than 1000/s. Finally, the whole streaming server over-queues after 14.32 seconds. This leads to an omitting of new events due to dynamic load shedding. For a time-critical system, this would imply that not only the real-time requirement could fail due to the increased latency (queuing delay), but the also results of the streaming servers calculations would be distorted due to the loss of the latest events. Our simulation predicts that the over-queuing already occurs after 13.72 seconds, whereas the measured time is 0.6 seconds later. A developer or analyst would get several valuable insights from such a result. Firstly, the bottleneck can be identified as the *Interpolation* task, since it is the first task that over-queues. Through such an insight any performance improvement activities can be focused on the application level, which would be much more efficient than just providing more hardware resources. Hirzel, et al. (2014), for example proposes a catalog consisting of 11 stream processing optimizations that are mostly applied on task level [6]. Secondly, the prediction yields that system is not capable of handling a workload of 1000/s. However, due to the data-driven characteristic of most workloads related to SPS, unexpected data bursts can always occur. The most important insight is hence that according to the simulation, the system would, in case of such a burst, still remain stable for 13.72 seconds. This prediction is hence a good example why task based performance considerations are important for stream processing systems.

Figure 8 depicts the simulation and measurement results with a workload of 1600/s. Even though the data rate has only increased by 60%, the whole application already over-queued after 2.78 seconds. Both simulations provided reasonable results with an average accuracy of 95.01% (1000/s) and 91.97% (1600/s). The model can hence also be used for predicting different workload scenarios. The differences between simulation and measurement can be mostly attributed to two factors. Firstly, our probabilistic filter operations do not perfectly reflect the actual drop rates of the measurement. In summary, the difference between the average droprate of the training and the test dataset is less than 2%. The other factor is the dynamic load-balancing. If a task over-queues, the run-time environment of the streaming server reallocates the CPU shares among the threads. For this reason our defined processing rates need to be modified as well in order to reflect these dynamic changes. Since, the PCM framework does not allow any kind of dynamic *ResourceDemand* redistribution, we have to perform multiple simulation runs. Each time a queue runs full, a new SEFF has to be generated containing the adapted average processing rates. In addition the queue-sizes in the component diagram need to be adapted according to the state before the load-balancing took effect. The whole simulation of the usage scenario is hence performed in four stages, with each stage depending on the results of the previous one. Inaccuracies of previous stage are hence carried to the following simulation stages.

4.3 Limitations

As described in the previous section, we needed four simulation runs for each of our usage scenarios due to the integrated load-balancing of the HANA streaming platform. This can be considered as one of the strongest limitations of the current approach, since the manual creation of multiple SEFF stages is time-consuming. The best solution for this issue would be to extend the PCM framework in order to support dynamic *ResourceDemands*. For this reason a representation as depicted in Figure 9 would probably be sufficient. In this case, depending on the current state of the queue the processing rate would be increased or reduced. Another option would be to automate the transformation of CCL source code to PCM instances. An automatic recreation of models based on previous simulation runs would also allow more flexibilty. Another limitation is, that our current setup only encompasses stateless *tasks* such as filters or annotations. Especially for analytical workload scenarios, stateful operations like *sliding-windows* are very common and required when working with aggregation operators such

[3]A framework for simulating PCM instances

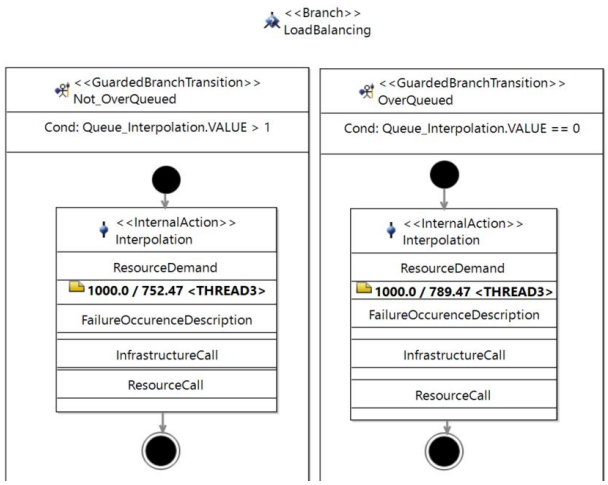

Figure 9: SEFF Queuing Process

as MAX or AVG. These tasks demand a modified consideration, due to the reason that the total number of events present within the system is not longer defined only based on the individual processing rates and queue-sizes, but also dependent on the retention policy of the window-operations. Finally assumptions are made regarding the expected filter drop rates.

5. CONCLUSION

This paper addresses the gap of performance predictions that consider the application logic of a SPS on a task level. Our experiment showed that accurate results can be achieved even for varying workload scenarios. The approach provides good insights into the inner workings of a stream application allowing to predict over-queuing situations and identify bottlenecks. It is hence a first step towards a tooling that allows developers and analysts to build and operate stable streaming systems. In addition, we intend to automate the CCL to PCM transformation in order to provide developers a useful tool to provide performance insights already during system development. Since CCL development is performed in the Eclipse IDE, such a feature could be provided and integrated as an additional plugin. We could already show that such transformation from different programming languages than Java are possible [15]. Finally, the prediction of hardware resources such as RAM or CPU would be a valuable extension. Especially the prediction of RAM resources is normally difficult due to non-deterministic factors such as the Garbage Collector in Java systems. However, since our current approach is already able to simulate the approximate number of events within our system, we can probably come up with a reasonable prediction regarding RAM utilization.

6. REFERENCES

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

[2] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[3] I. Bedini, S. Sakr, B. Theeten, A. Sala, and P. Cogan. Modeling performance of a parallel streaming engine: bridging theory and costs. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 173–184. ACM, 2013.

[4] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale. Dynamic datacenter resource provisioning for high-performance distributed stream processing with adaptive fault-tolerance. In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, page 13. ACM, 2013.

[5] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, et al. Performance-oriented devops: A research agenda. *arXiv preprint arXiv:1508.04752*, 2015.

[6] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[7] J. Kroß and H. Krcmar. Modeling and simulating apache spark streaming applications. *Softwaretechnik-Trends*, 36(4):1–3, 2016.

[8] J. Kroß, S. Voss, and H. Krcmar. Towards a model-driven performance prediction approach for internet of things architectures. *Open Journal of Internet Of Things (OJIOT)*, 3(1):136–141, 2017.

[9] J.-C. Lin, M.-C. Lee, I. C. Yu, and E. B. Johnsen. Modeling and simulation of spark streaming. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 407–413. IEEE, 2018.

[10] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández. Spark versus flink: Understanding performance in big data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442. IEEE, 2016.

[11] J.-I. Requeno, J. Merseguer, and S. Bernardi. Performance analysis of apache storm applications using stochastic petri nets. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 411–418. IEEE, 2017.

[12] A. Shukla, S. Chaturvedi, and Y. Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[13] R. Sipos, D. Fradkin, F. Moerchen, and Z. Wang. Log-based predictive maintenance. In *Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1867–1876. ACM, 2014.

[14] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[15] A. Streitz, M. Barnert, J. Rank, H. Kienegger, and H. Krcmar. Towards model-based performance predictions of sap enterprise applications. In *Proceedings of the 9th Symposium on Software Performance (SOSP)*, 2018.