Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

RaspaCN: Adaptive Scheduling Mechanism for High Performance Distributed Stream Processing in Kubernetes

Madushi Sarathchandra[2] | Chulani Karandana[2] | Winma Heenatigala[2] | Miyuru Dayarathna*[1] | Sanath Jayasena[2]

[1]WSO2, Inc., 787, Castro Street, Mountain View CA, USA, 94041
[2]Department of Computer Science & Engineering, University of Moratuwa, Moratuwa Sri Lanka

Correspondence
*Miyuru Dayarathna, WSO2, Inc., 787, Castro Street, Mountain View, CA, USA, 94041 Email: miyurud@wso2.com

Abstract

Recently distributed stream processors are increasingly being deployed in cloud computing infrastructures. In this paper we study performance characteristics of distributed stream processing applications in Google Compute Engine (GCE) which is based on Kubernetes. We identify performance gaps in terms of throughput which appear in such environments when using a round robin scheduling algorithm. As a solution, we propose resource aware stream processing scheduler called RaspaCN. We implement RaspaCN's job scheduler using two-step process. First, we use machine learning to identify the optimal number of worker nodes. Second, we use round robin and multiple knapsack algorithms to produce performance optimal stream processing job schedules. With three application benchmarks called HTTP Log Processor, Nexmark, and Email Processor representing real world stream processing scenarios we evaluate the performance benefits obtained via RaspaCN's scheduling algorithm. RaspaCN could produce percentage increase of average throughput values by at least 37%, 38%, and 10% respectively for HTTP Log Processor, Nexmark, and Email Processor benchmarks for fixed input data rates. Furthermore, we conduct experiments with varying input data rates as well and show 7% improved average throughput for HTTP Log Processor. These experiments show the effectiveness of our proposed stream processor job scheduler for producing improved performance.

KEYWORDS:
Stream processing, Software performance engineering, Scalability, Knapsack, Cloud computing, Kubernetes, Machine Learning, Event-based systems, IaaS, Auto-scaling

1 | INTRODUCTION

Stream Processing is a technology which conducts online real time processing of streams of data which appear in massive rates[1]. Multiple applications of stream processing can be found in areas such as telecommunications[2], crowd management, health informatics[3 4 5], cyber security[6], finance[7], marine sciences[8], energy[9 10], transportation[11], etc. Stream Processors are software platforms which allow their users to process and respond to incoming data streams faster. Recently, multiple distributed stream processors have appeared which can run on both shared-memory parallel and distributed memory systems[12].

Distributed stream processors have to use a job scheduling algorithm to deploy their queries across different data processing sites (i.e., workers). Job scheduling in computer systems has been widely studied area and there are a number of algorithms being used. There are two main approaches for stream processor job scheduling called *top-down* and *bottom-up*[13]. In practice, most often stream processing engines use only a simple round robin or hash-based scheduling. In this work we address the problem of performance optimization of distributed stream processing jobs in resource constrained environments[14].

As a solution in this paper we describe an improved adaptive job scheduling algorithm considering a distributed stream processor's performance behavior. We base our approach on bottom-up optimization technique where we initially distribute computation across all the workers but later on systematically aggregate these computations into few workers which reduces the requirement of keeping additional workers[15]. Furthermore, bottom-up approach enables us to avoid the requirement for maintaining highly provisioned worker nodes. In this work we use WSO2 Stream Processor as the example system to be optimized. We develop a mechanism called *Resource Aware Scheduler for Stream Processing Applications in Cloud Native Environments (RaspaCN)* which conducts performance optimal deployment of stream processing applications in the cloud native environment. We apply this mechanism on WSO2 Stream Processor which is based on Siddhi open source complex event processing library[16].

When deployed in a distributed stream processing cluster, stream processing applications are divided into multiple segments (i.e., Partial Siddhi Applications) by WSO2 Stream Processor's Manager component, based on their corresponding partitions. We schedule these partitions in the cluster using a scheduling algorithm. In this paper we use two different scheduling algorithms which are suitable for doing bottom-up scheduling. The first one is called *Round Robin (RR)* scheduler, which is based on popular round robin algorithm. The second one is based on *Branch-and-bound Knapsack (Knapsack)* algorithm[17]. The purpose of our scheduler is to generate throughput optimal stream processing query in a distributed stream processing cluster. From multiple experiments we have observed RR does not produce throughput optimal query plans compared to Knapsack. Furthermore, using a fixed Knapsack based scheduling plan does not always produce better performance because the system status changes based on time. Hence we propose adaptive scheduler based on Knapsack algorithm which produces optimal deployment of partial Siddhi applications to worker nodes. We refer mapping between partial Siddhi applications and their corresponding workers as *layout*. Our proposed approach intelligently switches between the different layouts to maintain high throughput.

We use three data stream benchmarks which we developed to evaluate the performance of the two schedulers. The first one is based on an HTTP Log file data set[18]. The second benchmark is a query from Google's Nexmark query[19]. The third benchmark is based on an email processing scenario. It was developed using the famous Enron email data set[20]. The three benchmarks resemble real world application scenarios, hence they are application benchmarks.

Based on these empirical observations we have developed a machine learning model using Random Forest algorithm[21] to select the optimal deployment configuration for worker components of the stream processor. Once the worker components number has been decided, the partial Siddhi applications are divided among the workers following RR. We use RR here initially because it evenly distributes components across different workers in the cluster. The application gets run in the cluster during the first epoch (first 15 minutes) and the performance numbers are collected at the performance measurement infrastructure. Next, the collected performance metrics are used along with the Knapsack algorithm to obtain high throughput performance. At the end of each scheduling epoch the necessity for re-arrangement of operators using the Knapsack algorithm is evaluated and the decision to continue using the same layout or use new layout proposed by the scheduler is taken. Note that here layout means the pattern how partial Siddhi applications get assigned to workers. With experiments done on Google Compute Engine (GCE) using the three benchmark applications HTTP Log Processor, Nexmark, and Email Processor we demonstrate RaspaCN could produce percentage increase of average throughput values by at least 37%, 38%, and 10% respectively for fixed input data rates compared to naive use of RR. With varying data rates for HTTP Log Processor we observed 7% increased average throughput for 30 minutes time period out of the last 45 minutes of the experiment. The contributions of our work can be listed as follows,

- *Resource Aware Scheduler for Stream Processing Applications in Cloud Native Environments* - We design and develop a scheduler for stream processors considering the dynamic nature of the cloud native environments. We name this mechanism as *RaspaCN*. We specifically focus on Kubernetes as the example cloud native environment.

- *Performance Optimal Branch and Bound Knapsack Scheduler* - The new scheduler we propose is an extension of multiple knapsack algorithm using branch and bound technique.

- *Unified Performance Measurement Infrastructure* - We measure the performance of system operation at three layers and we do unification of these measurements to construct a detailed picture of system operation.

- *Use Machine Learning to Predict Optimal Number of Workers* - We construct a machine learning model using Random Forest algorithm to predict the optimal number of Kubernetes Stream Processor Worker pods to be used.

The rest of the paper is organized as follows. In Section 2 we provide related work. In Section 3 we present an overview for stream processing software. Section 4 explains the methodology followed. Section 5 describes the evaluation conducted. We discuss the results in Section 6. We provide the conclusions in Section 7.

2 | RELATED WORK

Performance of distributed stream processing applications has been a widely studied area[22] [23] [24] [25]. Distributed stream processing has to handle the challenges of resource provisioning[26], scalability, information security, etc.

Distributed cluster's resource availability information are used for creating intelligent resource aware schedulers for distributed stream processors. R-Storm is one such example system where it tries to increase overall throughput by maximizing resource utilization while minimizing the network latency of Storm[27]. They mapped the problem of resource-aware scheduling of Storm applications to the problem of solving a Quadratic Multiple 3-Dimensional Knapsack problem. R-Storm has been designed with the assumption that the underlying software/hardware infrastructure is a dedicated one. However, in our distributed scheduler we handle the situation where the operation of stream processor is affected by the resource aware scheduling of the cloud native environment.

Operator fusion has been used as a method for optimizing SPADE applications in System S. Khandekar *et al.* described an optimization scheme that is applied at the compile time that groups operators in a SPADE program into appropriate run-time software units called Processing Elements (PEs)[28]. Our methodology is different from them in two aspects. First, they follow a top-down fusion approach where initially all the operators are fused together and then the largest PE is split into two separate smaller PEs. The optimization is continued until a suitable operator to PE partitioning is met. In our approach we follow a bottom-up method where we initially place all partial Siddhi applications (i.e., operators) into distinct Workers (i.e., PEs) and fuse them together till the Knapsack's optimization criteria is met. Gedik *et al.* follows a similar bottom-up approach with FINT fusion strategy[29]. Soulé *et al.* described a hybrid scheduler for stream processing languages[30]. They are focused on the problem of scheduling for arriving at the optimized solution. Kalyvianaki *et al.* conducted a similar work on a stream query planner that consider allocation of resources to queries[31].

Cervino *et al.* try to solve the problem of providing a resource provisioning mechanism to overcome inherent deficiencies of cloud infrastructure[32]. They have conducted some experiments on Amazon EC2 to investigate the problems that might affect badly on a stream processing system. Performance aspects of Kubernetes has been an issue recently being investigated. Medel *et al.* investigated deploying, terminating and maintaining performance of Docker containers with Kubernetes, detecting operational states that occur with the related pod-container[33]. However, none of the previous work describe a stream processor scheduler implementation for Kubernetes. Developing a stream processor for Kubernetes environment has to consider the fact that processes (i.e., pods) running inside the Kubernetes are resource constrained entities. Hence the scheduling algorithm cannot exceed the allocated amount of resources which makes the pods to restart.

3 | OVERVIEW OF STREAM PROCESSING SOFTWARE

We provide short introductions to WSO2 Stream Processor and Siddhi which are the data stream processor and the underlying event processing engine used for implementing RaspaCN. Moreover, we provide brief introductions to HTTP Log Processor, Nexmark, and Email Processor benchmarks used during the experiments.

3.1 | Overview of WSO2 Stream Processor

We have developed RaspaCN on top of the WSO2 Stream Processor which is an Open Source, Cloud Native and Scalable Stream Processing platform[34]. It can collect, analyze, and process events instantaneously in real time.

Siddhi is the SQL streaming language used by the WSO2 Stream processor. A Siddhi Application is a collection of Siddhi execution elements. Siddhi Query Language is designed to process event streams, identify complex event occurrences, and notify them in real time. Siddhi queries explain how to process existing event streams and create new output event streams if they do not exist.

The distributed deployment of the WSO2 Stream Processor facilitates scalability where the inflated workloads can be distributed among multiple stream processor instances. In distributed stream processing, an execution group (i.e., partial Siddhi application) is the smallest unit of execution which is considered as a collection of Siddhi execution elements. It facilitates to execute multiple instances of each execution group parallely in multiple stream processor instances.

Next, we describe the three benchmark applications used in the experiments conducted in this paper. The partitioner component (P) which appears at the beginning of these applications conducts partitioning of input stream's events based on the field *groupID*. We set the *groupID* field's value in the range 1,2,3,...,9 (9 different integer value) in a round robin manner. This allows for equal distribution of workload across all the nine partial Siddhi applications which got created while deploying the benchmarks.

3.2 | HTTP Log Processor Benchmark

HTTP Log Processor is a stream processing benchmark developed[1] by us based on the HTTP log data provided by the Division of Economic and Risk Analysis (DERA) of Securities and Exchange Commission of USA[18 35 36]. This data set contains the details about the web browsers and related crawlers that visited the site. Events in the HTTP log stream has sixteen fields as *iij_timestamp, groupId, ip, date, time, zone, cik, accession, code, size, idx, norefer, noagent, find, crawler, and browser*. The streaming application detects the malicious attacks to the site based on the HTTP log details.

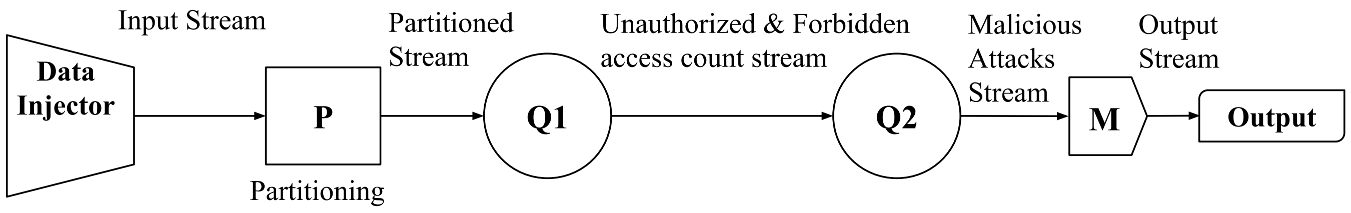

FIGURE 1 Architecture of HTTP Log Processor Benchmark

Here Q1 runs an external time batch window which detects unauthorized (HTTP 401) and forbidden (HTTP 403) access counts. Q2 runs a malicious attacks detection function based on the access counts calculated by the Q1. The partitioner component denoted by P partitions input event stream based on the value of groupID. Merging of these partitioned streams happen through the merger component (M). Note that the same notation is used with the other two benchmarks as well.

3.3 | Nexmark Benchmark

Nexmark Benchmark was developed by us based on the Google Nexmark queries. It is based on an auctioning system that comprises items, bidders, and auctions. In a real world auction system, numerous auctions are opened. Based on the continuous actions, three basic business objects can be identified as users, sellers, and buyers.[19]

In the Auction system the users are registered with the Person Business object to act the roles of buyers and sellers and added to the person stream. The sellers submit the details about the items as *description, start day, end day, time for auction, and quantity*. The description and the quantity of the item is streamed to the main system as the item entity and the reserver (person who reserves), Start and End time of the auction. When the auction expires, it is signalled via the closed auction stream. The streaming application we developed notifies the hot selling item based on the Bid Count Stream which has the six fields *iij_timestamp, auction_id, partition_id, time, person, and bid.*

[1]https://github.com/miyurud/log-processor

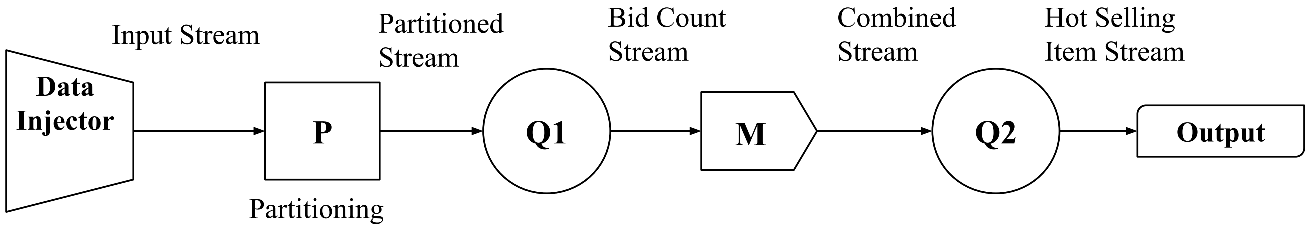

FIGURE 2 Architecture of Nexmark Benchmark

The first query of the Nexmark hot-selling item Siddhi application calculates the bid count per each auction item within a selected time window. The second query selects the hot-selling item which has the maximum bid count within the selected time batch window. The sources of our Nexmark application is accessible from[2].

3.4 | Email Processor Benchmark

Email Processor Benchmark was developed based on the Enron Email data set[20]. This data set contains 517,417 emails generated by the employees of Enron Corporation.[37] The events in the Email Stream has the nine fields, *iij_timestamp, groupID, fromAddress, toAddresses, ccAddresses, bccAddresses, subject, body,* and *regexstr*. Several operations were applied on the email stream including, *Email Address Filtering, Email Body Modification, Most Frequent Word Calculation, String Replacements of the Email Body*, and *Metrics Calculation*.

In our benchmark implementation[3] the input email data stream is injected into the distributed stream processor by the data injector. Then the Input Stream is partitioned using the Siddhi Query Engine and the query Q1 filters all the email addresses end with `@enron.com` from the fields ToAddress, CCAddress and BCCAddress. Q2 changes each and every email and Q3 query gathers metrics of each email. These metrics include the number of characters, words, paragraphs in the email body. Then the processed email stream is sent via query Q4 and generates the output log stream. The architecture of the Email Processor benchmark is shown below.

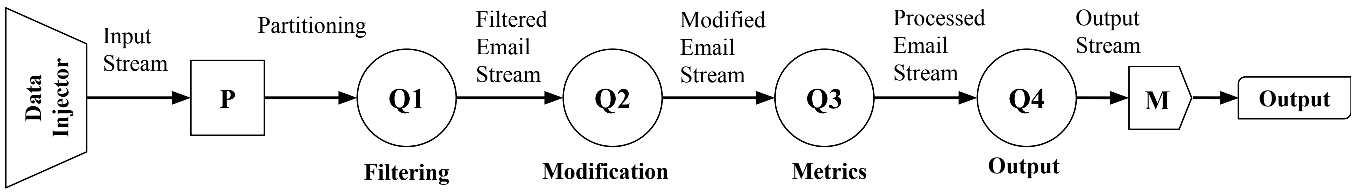

FIGURE 3 Architecture of Email Processor Benchmark

4 | METHODOLOGY

RaspaCN schedules stream processing applications in a given cloud native environment with optimal performance (throughput). The architecture of RaspaCN and how it operates during run time is shown in Figure 4. The box named RaspaCN, Knapsack Scheduler, and Performance DB are the main components which we implement.

Performance DB collects performance information from Siddhi applications and Google Compute Engine (GCE). We collect throughput and latency information from Siddhi applications online (simultaneously while the applications execute). From GCE we collect performance statistics of Kubernetes nodes and containers. We collect the node and the container level data by running

[2]https://github.com/WinmaH/NexmarkAuctionStreamProcessor
[3]https://github.com/miyurud/EmailProcessor_Siddhi

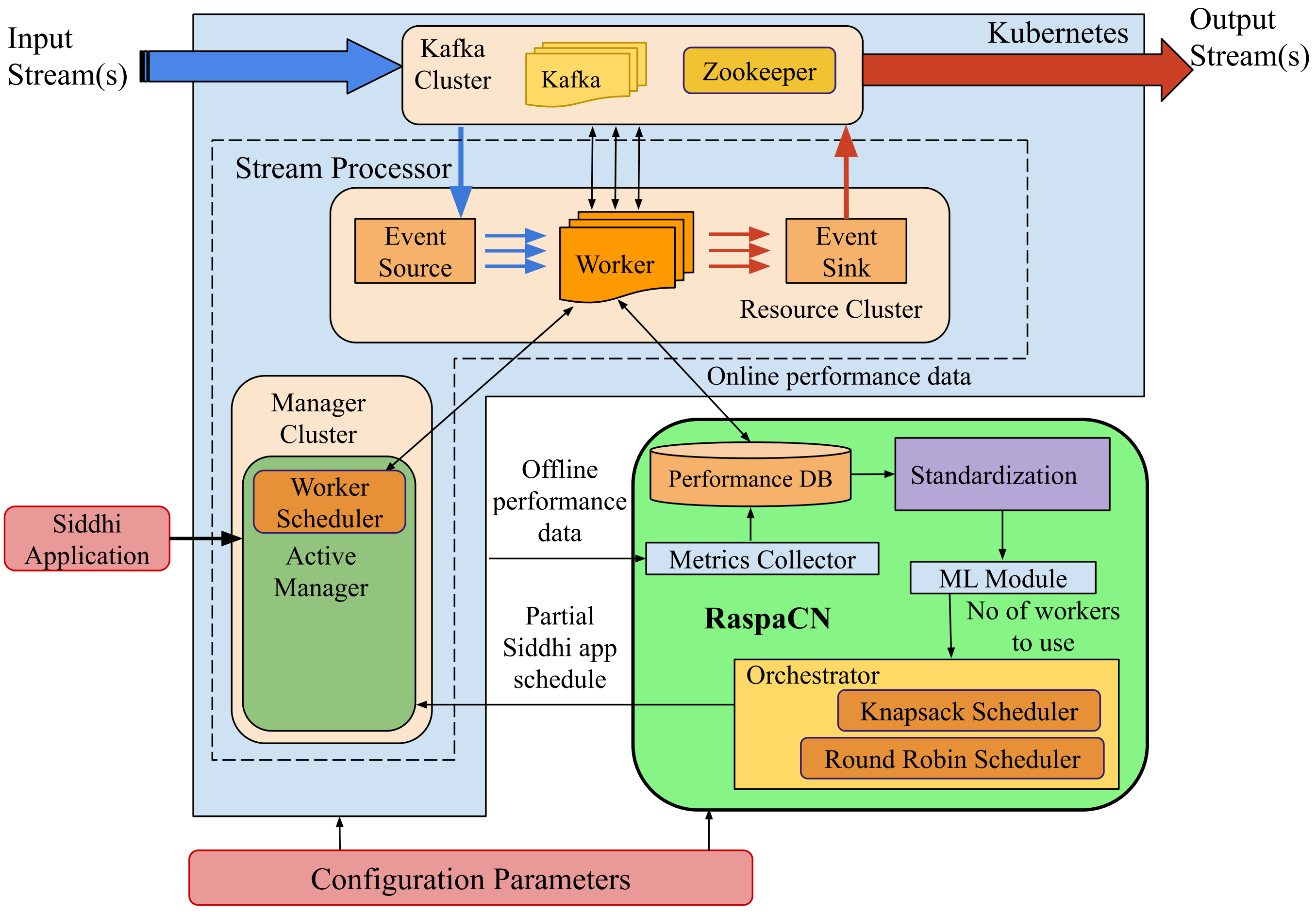

FIGURE 4 Architecture of RaspaCN

an offline metrics collector application developed by us. RaspaCN uses these information for training an ML module which is used during the run time before deploying the stream processor cluster to determine the optimal number of workers to be used. The Orchestrator is the module which starts/stops the worker instances in Kubernetes.

When run in distributed mode, WSO2 Stream Processor creates three different types of clusters in Kubernetes. Manager Cluster is the one which holds the manager components of the distributed stream processor. We deploy two managers in Active/-Passive mode. This means one manager is running as the Active Manager handling all the cluster management requests while the other one (not shown in Figure 4 for simplicity) runs as the passive one without handling any requests, synchronizing its state with the Active Manager.

Streams of events which are received via input streams are received by Kafka cluster (i.e., a cluster of Kafka message brokers). Note that in our work we use Kafka only as a message broker and we do not use any of its stream processing capabilities. Event sources fetches these data and pushes them to the workers (as indicated via blue color arrows). Communication between the workers happen via the same Kafka cluster. This means one worker writes the processed data to a Kafka topic, and another worker fetches these data and processes them further. Finally, the outcomes of the processing are returned by the workers via Event Sink component back to the Kafka cluster (as indicated by the red color arrows). Third party event listeners (not shown in Figure 4) fetches these processed data from Kafka cluster.

Note that although RaspaCN's architecture is drawn considering WSO2 Stream processor, it can be generalized and applied for other industrial stream processing systems since it is lightly coupled with the stream processor. This can be observed in Figure 4 as well because RaspaCN's components are located outside the stream processor. If one needs to extend RaspaCN into some other stream processor, he needs to do two things. First, he needs to implement a component which collects throughput/latency performance metrics from the partial stream processing applications. Here partial stream processing applications are similar to partial Siddhi apps where stream processing applications are divided into multiple partial stream processing apps. Second, he needs to change Orchestrator to generate a schedule of stream processor application components.

4.1 | Partial Siddhi Application Deployment

Once a Siddhi application has been submitted to an Active Manager, it gets split into partial Siddhi applications. WSO2 Stream Processor allows for having multiple partial Siddhi applications in a single worker pod. Having one partial Siddhi application per pod is also a possible approach but in the context of resource-limited environments this may lead to additional CPU and memory usage for background tasks. Hence we allow multiple partial Siddhi apps to be scheduled per worker pod. These partial Siddhi applications are scheduled in the cluster using the Round Robin (RR) Scheduler located within the manager instance. With RR we first arrange the list of workers based on their alphabetical naming and then we allocate partial Siddhi applications one by one in the workers. When we reach the last item of the list, we start allocating second partial Siddhi application per worker from the beginning of the list. We continue this pattern of allocation until all the partial Siddhi applications have been assigned to workers. Once the partial Siddhi applications have completed running for a period of t, it runs the Knapsack scheduler to determine the scheduling layout for the next time slot $t + D$. Here D is a constant which represents the time period (i.e., scheduling interval) between running of two consecutive scheduling processes. t is the start time of the current scheduling period. Here t is the time at which each D time slot starts. Next, we describe how the Knapsack scheduling algorithm is executed.

4.2 | Knapsack Scheduling Algorithm

In this work we introduce an advanced job scheduler for distributed stream processor to allocate partial Siddhi applications among the workers to get the optimal throughput performance under the resource constraints we considered. We implemented the scheduler using multiple Knapsack algorithm[38] because partial Siddhi applications are allocated among the multiple workers. We found that the general multiple Knapsack algorithm cannot be used as it is in our scheduler because we need to reduce the inter-node communication cost among the workers. Therefore, we define knapsack capacities for each worker and allocate partial Siddhi applications based on the branch and bound method in such a way to fill workers one after another.

In our current implementation, we define a knapsack capacity for each worker based on the average CPU usage value of partial Siddhi applications. The partial Siddhi applications are considered as the items to be scheduled. Each item has a weight and a profit value and we define them as CPU usage and entire throughput for the run of each item respectively. The algorithm returns a subset which gives the maximum throughput such that the sum of the CPU usage values of this subset is smaller than or equal to the Knapsack capacity that we defined.

The algorithm consists of the following core parts for finding the subset out of n partial Siddhi applications with maximum throughput performance. After selecting the subset, the remaining partial Siddhi applications are scheduled in such a way to pack the maximum number of items to a worker without exceeding the knapsack capacity. Here we used greedy like heuristic method which schedules partial Siddhi applications based on the throughput value per CPU usage of each partial Siddhi application. The steps of this algorithm include calculating the upper bound, calculating the profit value, finding all possible solutions, and selecting the optimum solution.

Algorithm 1 depicts the pseudo-code for calculating the Upper Bound. When calculating the upper bound, $KnapsackCapacity(k)$, *Throughputs[0, ..., (n-1)]*, *CPU Usages[0, ..., (n-1)]* and the *sorted set of partial Siddhi applications based on CPU Usages (pa')* to be scheduled are obtaining as the inputs. Then we calculate the Upper Bound using the entire throughput value for the run of each partial Siddhi application such that the sum of the CPU usage values of added partial Siddhi applications is smaller than or equal to the Knapsack capacity (Algorithm 1 line 5-8).

Algorithm 2 depicts the pseudo-code for calculating the profit value for a tree node in decision tree. For this calculation, *Upper Bound (ub)*, $max\_weight(mw)$, $remaining\_weights(wr)$, and $remaining\_profit(pr)$ are taken after calculating the upper bound in Algorithm 1. We calculate the profit value by adding the remaining fraction of throughput value to the calculated upper bound.

Algorithm 3 depicts the pseudo-code for traversing the branch and bound decision tree to find all possible solutions. In this section, we discuss in detail how we determine the maximum throughput subset of *Throughputs[0, ..., (n-1)]* out of n partial Siddhi applications. The upper bound and the profit value are calculated for each tree node in the branch and bound decision tree based on the Algorithm 1 and 2. Then it compares the bound with the current best solution before exploring the tree. If the best in the sub-tree is worse than the current best, it simply ignores the node and its sub-tree. The algorithm recursively does this processing for the entire decision tree (Algorithm 3 line 6-18).

Algorithm 1: Calculate Upper Bound

```
Input : KnapsackCapacity(k), CPU_Usages[0, …, (n-1)] ,
Throughputs[0, …, (n-1)],
Sorted Set of Partial Siddhi apps based on CPU_Usages(pa')
Output : Upper Bound
Description : calculateUpperBound(PartialSiddhiApps pa')
1: max_weight ← 0
2: upper_bound ← 0
3: last_profit_value ← 0
4: for all $p_i$ in pa' do
5:     if (k.getCapacity() > max_weight) &&
         (k.getCapacity() > $p_i$.getWeight() ) then
6:         max_weight += $p_i$.getWeight()
7:         upper_bound += $p_i$.getProfit()
8:         last_profit_value = $p_i$.getProfit()
9:     end if
10:    if k.getCapacity() > max_weight then
11:        upper_bound -= last_profit_value
12:        break
13:    end if
14: end for
15: return upper_bound
```

Algorithm 2: Calculate Profit Value

```
Input : KnapsackCapacity(k), CPU_Usages[0, …, (n-1)] ,
Throughputs[0, …, (n-1)], max_weight(mw),
remaining_weights(wr), remaining_profit(pr), Sorted Set of
Partial Siddhi apps based on CPU_Usages(pa')
Output : Profit Value for a Tree Node
Description : calculateProfitValue(PartialSiddhiApps pa')
1: ub ← calculateUpperBound(pa')
2: if wr is not equal 0 then
3:     profit_value ← ub + (pr/wr) * (k.getCapacity() - mw)
4: else
5:     profit_value ← ub
6: end if
7: return profit_value
```

Algorithm 4 depicts the pseudo-code for selecting the final solution after traversing the decision tree. Here, the algorithm finds all the possible solutions and then selects the best possible optimal solution out of these possible solutions by tree traversing.

Algorithm 3: Find all possible solutions

```
Input : KnapsackCapacity(k), CPU_Usages[0, …, (n-1)] ,
Throughputs[0, …, (n-1)], Sorted Set of Partial Siddhi apps
based on CPU_Usages(pa’)
Output : Solution nodes from Branch and Bound Decision
Tree
Description : getPossibleSolutions(PartialSiddhiApps pa’)
1: iub ← getInitialUpperBound()
2: level ← 0
3: pa’ ← pa.clone()
4: ub_j ← calculateUpperBound(pa’) * -1
5: P_j ← calculateProfitValue(pa’)*-1
6: if iub > ub_j then
7:     iub ← ub_j
8: end if
9: if p_j <= iub then
10:    branch&boundTreeNodes.add(node_j)
11:    temp.add(node_j)
12: end if
13: if (pa.get(level) in pa’) && (level < pa.size()) then
14:    pa’.remove(pa.get(level))
15: else
16:    go to step 19
17: end if
18: go to step 4
19: level++
20: while temp is not empty do
21:    pa’ ← temp.get(0).getPartialSiddhiApps()
22:    temp.remove(temp.get(0))
23:    go to step 4
24: end while
25: return ← branch&boundTreeNodes
```

4.3 | Optimal Worker Count Prediction

From our experience with cloud native stream processor performance studies, we understood that the performance does not always grow with the increase of the amount of worker nodes in the cloud native environment[39]. Therefore, using an optimal number of workers leads to cost optimization and reduces the inter node communication overhead. In our implementation we used a machine learning based prediction model to predict the optimal number of workers to instantiate. Machine Learning model was built using a data set[4] obtained from experiment results that has the features such as the number of workers, the number of partial Siddhi applications, the input data rate, the number of (String/float/int/long) attributes in the input stream, the types of windows present (external time batch/length/time batch), the number of integer operators, the number of aggregation functions, the number of (select/insert/from) operators, the number of filter conditions, and the number of extensions used. Correlation analysis (a statistical method for evaluating the strength of relationship between two quantitative variables) resulted in selecting the features such as the number of workers, the number of partial Siddhi applications, the data rate, the no of string attributes

[4]https://github.com/Lakshini/Machine-Learning-For-Predicting-Workers-Dataset

Algorithm 4: Select the best possible solution

```
Input : KnapsackCapacity(k), CPU_Usages[0, ..., (n-1)] ,
        Throughput[0, …, (n-1)], Sorted Set of Partial Siddhi
        apps based on CPU_Usages(pa')
Output : Maximum Profit sub set out of n Partial Siddhi Apps
Description : getOptimumSolution(PartialSiddhiApps pa')
1: nodes ← getPossibleSolutions(pa')
2: max_profit ← 0
3: for all n_i in nodes do
4:     weight ← 0
5:     for all p_i in n_i.getPartialSiddhiApps() do
6:         weight += p_i.getWeight()
7:     end for
8:     if ((weight < k.getCapacity()) &&
          (max_profit <= n_i.getProfit())) then
9:         max_profit ← n_i.getProfit()
10:        final_node ← n_i
11:    end if
12: end for
13: return final_node.getPartialSiddhiApps()
```

in the input stream, and the no of float attributes in the input Stream. We used the supervised classification algorithm, Random Forest, to train the model. The random forest classifier uses Bagging and Randomness in predicting using several decision trees. Since it enhances the accuracy of the results, we used that algorithm to train the model. In order to optimally solve the machine learning problem we did a hyper parameter tuning using the grid method. K fold cross validation was used to train the model in order to prevent the model getting over-fitted. We used the number of folds as 10 with 80 % data on training and the rest on testing the model. With these techniques being used our machine learning model could obtain Mean Absolute Percentage Error (MAPE) of 15.61% and accuracy (based on MAPE) of 84.39%. From the predicted throughput values using the Machine Learning model when the resource availability is given, the Orchestrator identifies the number of workers associated with the best throughput. Then the corresponding amount of workers are automatically deployed in the cloud initially.

4.4 | Redundant Worker Removal

When the Knapsack based scheduling algorithm is used in partial Siddhi application allocation, certain workers may be idling based on the scheduling layout. Such redundant (i.e., empty) workers are removed by the RaspaCN's Orchestrator which also eventually helps in optimization of system resources. The Orchestrator is capable of observing the cluster, determining the redundant workers, and removing them. First, it observes the cluster and records the details of the active workers. Once the Branch and Bound Knapsack scheduler determines the scheduling layout and deploys the partial Siddhi applications, the Orchestrator checks for the redundant workers and removes them.

5 | EVALUATION

We did evaluation of the performance behavior of the distributed stream processing applications run by RaspaCN on a Google Compute Engine (GCE) cloud. We used four Kubernetes nodes of machine type `n1-standard-2` which were running in the zone `us-east1-b`. Each node had 2 vCPU cores (Intel Haswell), 7.5 GB RAM, and a boot disk of 100GB which was a standard persistent disk.

The deployment architecture of the Kubernetes cluster used in the experiments is shown in Figure 5. We used WSO2 Stream Processor 4.2.20, Apache Kafka 2.11 and mysqlserver 5.7. Out of the four nodes two of the nodes ran stream processor worker pods which ran the stream processing applications. The remaining two were used for background processes such as running manager components of the distributed stream processor, running RaspaCN, running Kafka, data publishing, etc. SP-RDBMS corresponds to the MySQL database server which collected the performance information. Each deployed pod had memory allocated as shown in Figure 5. Each worker and each Zookeeper had an equal amount of memory (0.85 Gi and 0.8 Gi respectively). Here 1 Gi is equal to 1.07374 giga bytes of RAM. Note that we release RaspaCN's source code under open source license[5].

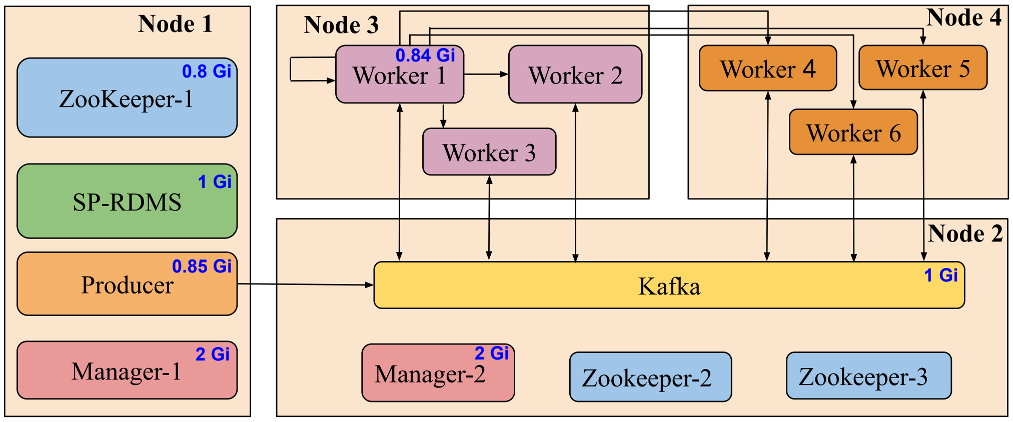

FIGURE 5 Deployment Architecture of Distributed Stream Processor in Google Cloud Platform

In the first round we conducted three experiments. In each experiment we ran one of the three benchmarks described in Section 3. Each experiment was conducted for one hour time period with a fixed scheduling interval of 15 minutes for Knapsack algorithm. Use of 15 minutes as the scheduling interval allows the JVMs to stabilize doing Just-In-Time (JIT) compilation of the Java byte code while allowing us to complete the experiments in the minimum time duration. The results presented in this paper are single round experiments. We remove the ash color highlighted time periods from average throughout calculation which are the periods where scheduling algorithm runs. Since the Knapsack algorithm gets deployed after running RR 15 minutes, we use only throughput data collected after the first 15 minutes in this paper for all the experiments when we calculate the average throughput.

ML model we trained for these experiments predicted 6, 5, and 5 worker counts as the optimal workers to be used with Log Processor, Nexmark, and Email Processor respectively. In the case of RR we do not do recalculation and the same layout was kept for running for 1 hour. However, in two of the experiments we repeatedly rescheduled the same layout for RR and Knapsack algorithms in order to observe the impact of the periodic rescheduling process for throughput variation. All the three benchmarks had a steady flow of input data stream (HTTP Log Processor input data pattern is plotted in Figure 6).

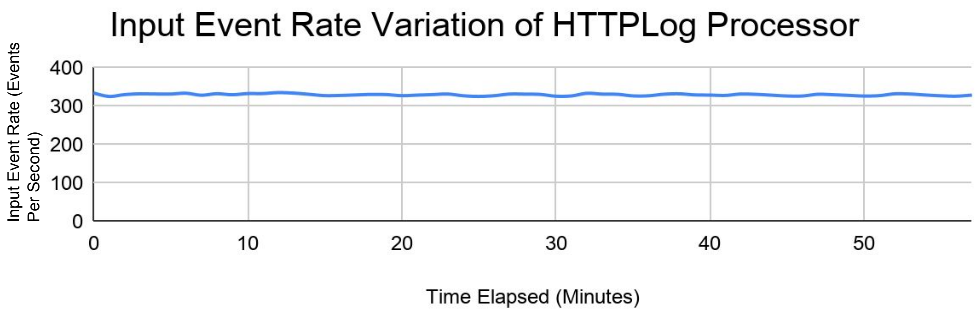

FIGURE 6 Input Event Rate variation of HTTP Log Processor

[5]https://github.com/miyurud/RaspaCN

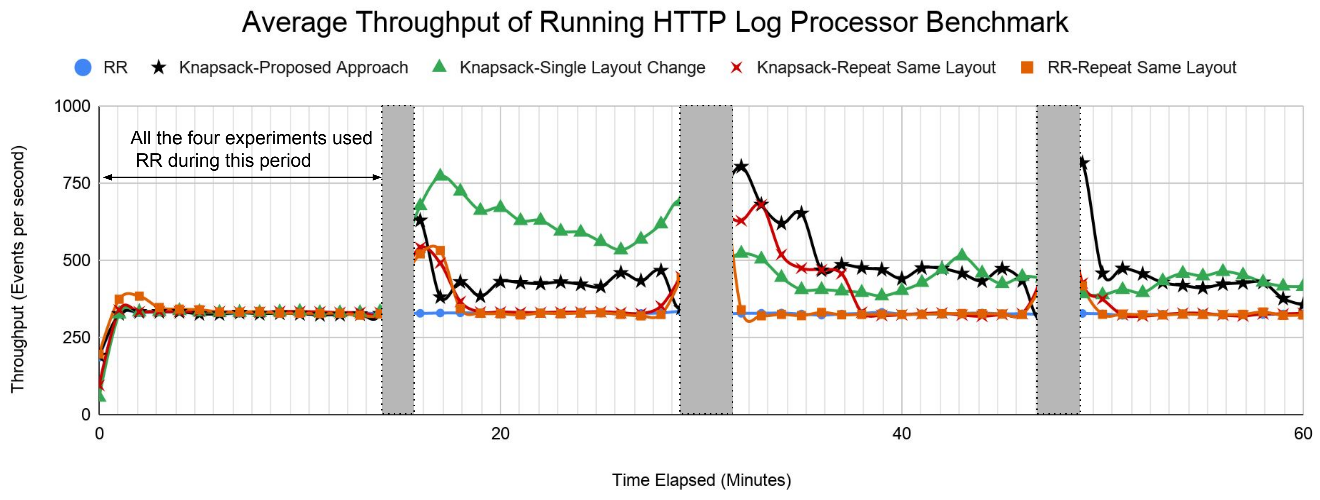

FIGURE 7 Average Throughput of Running HTTP Log Processor Benchmark

The input data rates were almost fixed for Email Processor and Nexmark benchmarks. Hence we do not include those data rate curves in this paper. The average input data rates for all the three benchmarks are 330, 242, 51 (all in events/second) for HTTP Log Processor, Nexmark, and Email Processor respectively. The throughput performance results of running the benchmarks are shown in Figures 7, 8, and 9. The ash color columns appearing in each chart corresponds to time periods in which scheduling process has been run. During this period the data producer of the benchmark continues publishing data to Kafka. Yet the data injector component stops collecting data from Kafka. Since the producer component has been running even though the stream processor stops for a short period, this resulted in a workload spike just after the new layout has been deployed. Note that, in all the three charts, the first 15 minutes corresponds to running RR even though they are marked as Knapsack.

Out of the five data series, the RR had lower average throughput value while all the three Knapsack experiments had better throughput compared to RR. In Figures 7, 8, and 9 there are two performance curves having RR experiments. The first one ('RR') did not run any scheduling process at all. In the second curve ('RR-Repeat Same Layout') at the end of each 15 minutes interval we reschedule the same RR layout. This resulted in throughput peaks because of the same reason mentioned above. Repeated rescheduling of RR layout resulted in 9% improved performance compared to not running any scheduling for RR.

In the third round we changed the scheduling algorithm from RR to Knapsack after 15 minutes elapsed (shown as 'Knapsack-Single Layout Change'). This resulted in improved throughput yet it gets depleted as the time passes. In the case of Log Processor this can be observed after passing 30 minutes. We observe switching results in better throughput during 30-60 minutes time period. During that period 'Knapsack-Single Layout Change' resulted in only 33% performance improvement.

Next, we rescheduled the same initial Knapsack layout again and again at the end of each 15 minutes interval and observed the performance behavior. The resulting performance curve is shown in Figure 7 as 'Knapsack-Repeatedly Schedule Same Layout'. Repeating the same Knapsack layout again and again resulted only 23% improved performance compared to RR. It can be observed that this performance curve is the lowest out of the three Knapsack curves. This indicates that repeated rescheduling of the same layout does not bring performance improvement, rather dynamically switching the Knapsack layout provides improved performance. I.e., the improved throughput results because of reconfiguration of the stream processing application components and it is not mainly because of buffered events are piped through the system. Finally, we run our proposed approach (shown as 'Knapsack-Proposed Approach').

Here we reschedule the Log Processor application with a new layout which is calculated by the Knapsack scheduling algorithm. With this experiment we observed the best throughput values out of the three Knapsack experiments which was 37% average throughput improvement in the overall experiment. Especially 'Knapsack-Proposed Approach' resulted in 54% performance improvement during the 30-60 minutes time period of the experiment. We saw similar behavior of the performance curves for Nexmark and Email Processor benchmarks as well. This indicates the effectiveness of our scheduling algorithm in maintaining better throughput performance.

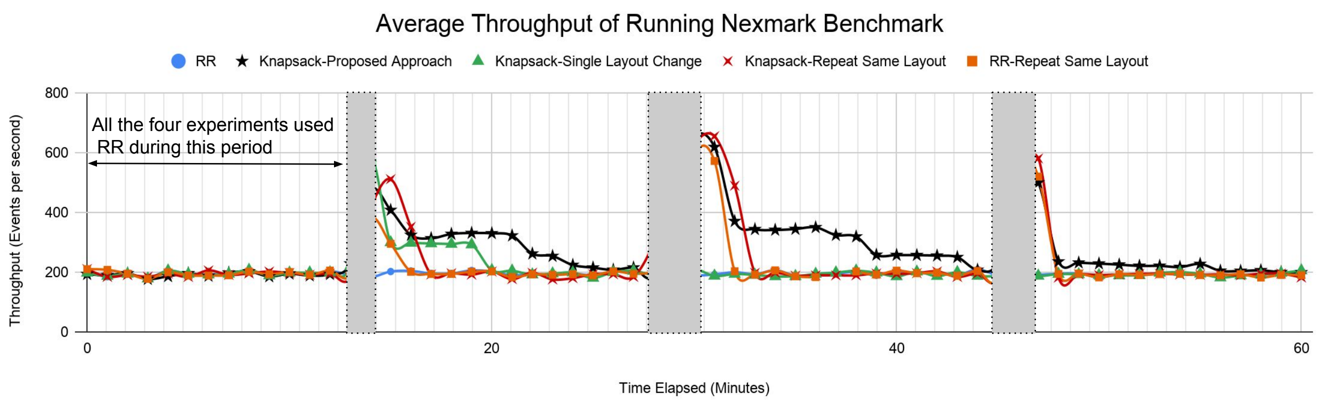

FIGURE 8 Average Throughput of Running Nexmark Benchmark

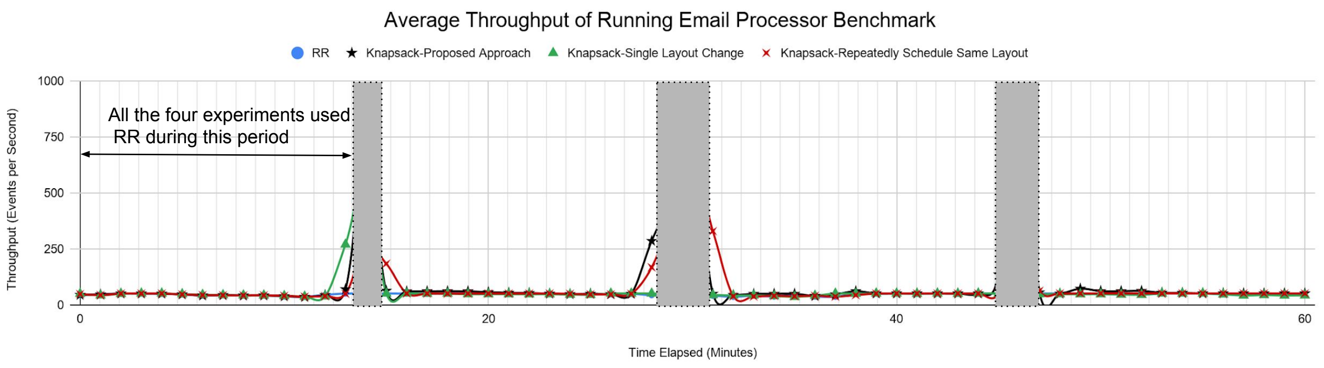

FIGURE 9 Average Throughput of Running Email Processor Benchmark

We calculate the average throughput increase produced by the use of Knapsack and our scheduling algorithm (Knapsack-Proposed Approach in Figures 7, 8, and 9) compared to naive use of RR. For HTTP Log Processor, Nexmark, and Email Processor the average throughput was increased by use of our technique by 37%, 38%, and 10% respectively.

How partial Siddhi applications of HTTP Log Processor benchmark get deployed in each Kubernetes nodes during the complete 60 minutes time period is shown in Figures 10, 11, 12, and 13. It can be observed that in the case of Knapsack based scheduling layout the partial Siddhi applications get deployed in one worker less compared to the scheduling layout shown in Figure 10 which is based on RR. Hence the resources allocated for that worker gets released which increases the efficiency of the application execution. We saw similar scheduling behavior with the other two benchmarks as well. Furthermore, even with lesser number of workers our mechanism provides significant performance improvements compared to RR.

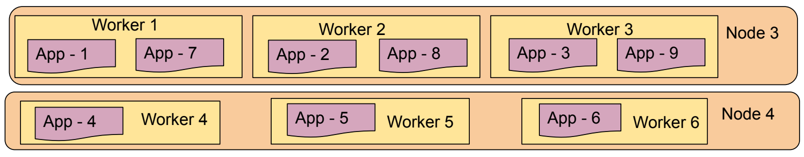

FIGURE 10 Layout for first 15 minutes HTTP Log Processor Benchmark Execution based on RR

In the second round we selected HTTP Log Processor benchmark and we used varying input event rates. Here we choose only HTTP Log Processor because it produced the highest throughput improvement out of the three benchmarks. This was performed

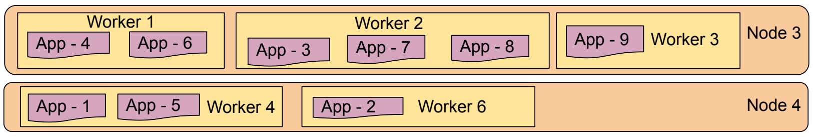

FIGURE 11 Layout for 15-30 minutes HTTP Log Processor Benchmark Execution based on Knapsack

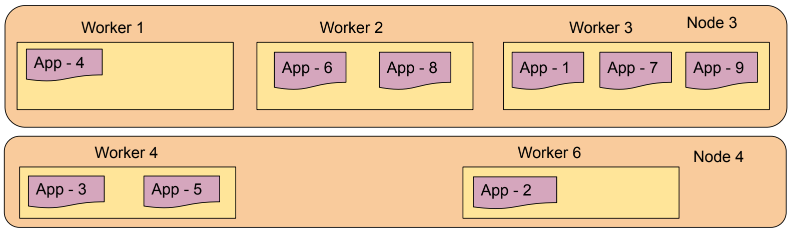

FIGURE 12 Layout for 30-45 minutes HTTP Log Processor Benchmark Execution based on Knapsack

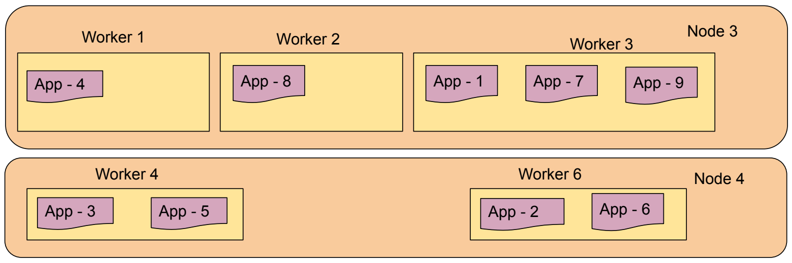

FIGURE 13 Layout for 45-60 minutes HTTP Log Processor Benchmark Execution based on Knapsack

to identify whether our scheduling mechanism can with stand such varying event rate scenarios since the experiments described in the paper so far were conducted with fixed input event rates. The corresponding varying input event rate pattern of HTTP Log Processor is shown in Figure 14.

Throughput results of running the HTTP Log Processor Benchmark is shown in Figure 15. In the case of such significantly varying input event rate as well our Knapsack based scheduler was able to produce high throughput values compared to RR during the 15-45 minute period. We observed a 7% increase in average throughput with use of Knapsack algorithm compared to RR.

6 | DISCUSSION

The results show that our approach produces a considerable improvement of average throughput compared to the naive use of round robin algorithm. Use of periodic re-evaluation of the scheduling layout enables us to maintain high throughput performance throughout the execution of the stream processing application. Furthermore, compared to the naive scheduler, RaspaCN produces improved performance with the use of a lesser number of workers (i.e., Kubernetes pods). This enables those resources to be utilized for other processing tasks increasing the efficiency of the use of resources in the Kubernetes cluster. The purpose of this paper was to describe RaspaCN's ability to produce increased throughput performance with a lesser number of workers rather than the use of naive scheduling algorithms. Even though the experiments described in this paper has been performed in four node Kubernetes cluster, we plan to conduct large scale experiments with RaspaCN in future which includes much larger number of Kubernetes nodes. The current experiments have been conducted with single application settings. Since each Siddhi application in a distributed stream processor has been handled as a collection of partial Siddhi applications, it is straight forward to use the same scheduling algorithm with different stream processing applications because ultimately

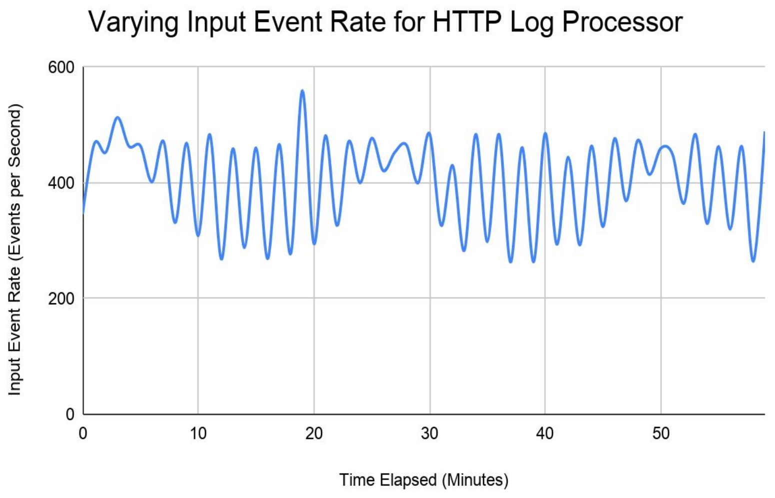

FIGURE 14 Input event rate pattern of HTTP Log Processor with significant variations in input event rate

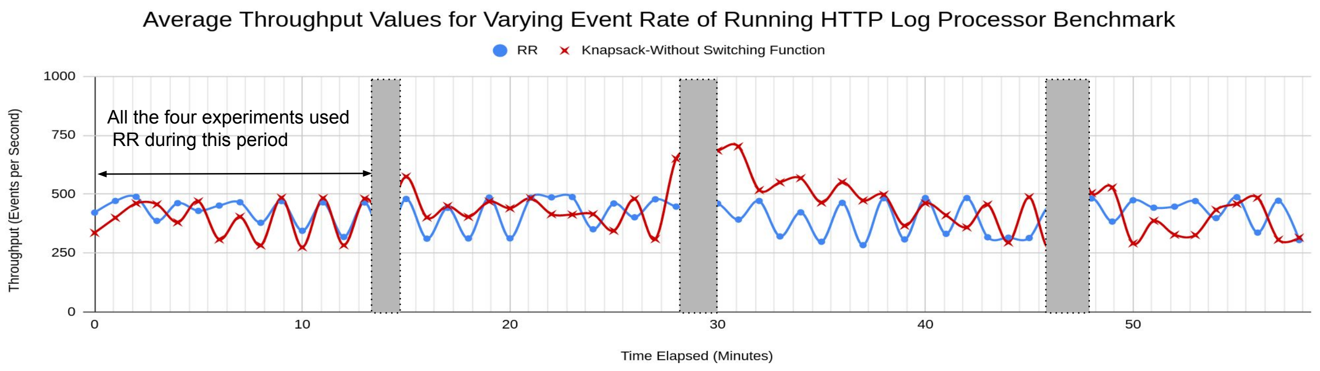

FIGURE 15 Average throughput of running HTTP Log Processor benchmark with significantly varying input event rates

ihttps://www.overleaf.com/project/5de2a8c9dd267500011a74cct becomes a problem of scheduling a collection of partial Siddhi applications in an efficient manner to produce high throughput. Furthermore, in this paper we are concerned of producing high throughput stream processing applications. Since we use throughput as the evaluation metric, we do not consider percentile values to evaluate the benefits of the proposed approach. The same mechanism can be extended for producing low latency. Current RaspaCN scheduler has been implemented without considering the statefulness[40] aspects of the stream processing queries. This means by default we assume the state gets persisted on a database server in such a way that scheduling process and state persistence does not interfere with each other.

7 | CONCLUSIONS

In this paper we present a resource aware stream processor scheduling mechanism for distributed stream processors running in Kubernetes environment which produces and maintains high throughput compared to naive use of round robin scheduling. The scheduling mechanism uses machine learning (Random Forest algorithm) to decide on the optimal number of workers to be used in a distributed stream processor deployment considering both properties of cloud infrastructure, past application execution, as well as current application's characteristics. Once the optimal number of workers are determined the partial Siddhi applications are deployed in the cluster using a scheduling algorithm based on branch and bound knapsack algorithm. RaspaCN ensures it produces increased throughput with efficient use of cluster resources. We conducted evaluation of RaspaCN by using three real world benchmark applications running on distributed stream processor, with RaspaCN as the scheduling mechanism on Google Compute Engine. We observed RaspaCN was able to produce percentage increase of throughput values by at least 37%, 38%, and 10% respectively for HTTP Log Processor, Nexmark, and Email Processor benchmarks with fixed input data rate patterns. Moreover, 7% increased throughput was observed for HTTP Log Processor with significantly varying input data rate. Furthermore, such throughput improvements were produced with a lesser number of workers compared to the naive approach. In future we hope to investigate performance optimization of stream processing applications with checkpointing enabled. There we hope to use latency behavior of the checkpointed applications as the main metric for scheduling. We also hope to expand the scheduling mechanism to leverage cluster auto scaler of Kubernetes in the future.

References

1. Dayarathna M, Perera S. Recent Advancements in Event Processing. *ACM Comput. Surv.* 2018; 51(2): 33:1–33:36. doi: 10.1145/3170432

2. Theeten B, Bedini I, Cogan P, Sala A, Cucinotta T. Towards the Optimization of a Parallel Streaming Engine for Telco Applications. *Bell Labs Technical Journal* 2014; 18(4): 181-197. doi: 10.1002/bltj.21652

3. Blount M, Ebling MR, Eklund JM, et al. Real-Time Analysis for Intensive Care: Development and Deployment of the Artemis Analytic System. *IEEE Engineering in Medicine and Biology Magazine* 2010; 29(2): 110-118. doi: 10.1109/MEMB.2010.936454

4. Dayarathna M, Suzumura T. Hirundo: A Mechanism for Automated Production of Optimized Data Stream Graphs. In: ICPE ’12. ACM; 2012; New York, NY, USA: 335–346

5. Dayarathna M, Suzumura T. Automatic optimization of stream programs via source program operator graph transformations. *Distributed and Parallel Databases* 2013; 31(4): 543–599. doi: 10.1007/s10619-013-7130-x

6. Miller Z, Dickinson B, Deitrick W, Hu W, Wang AH. Twitter Spammer Detection Using Data Stream Clustering. *Inf. Sci.* 2014; 260: 64–73. doi: 10.1016/j.ins.2013.11.016

7. Wu H, Salzberg B, Zhang D. Online Event-driven Subsequence Matching over Financial Data Streams. In: SIGMOD ’04. ACM; 2004; New York, NY, USA: 23–34

8. Hayes JP, Kolar HR, Akhriev A, Barry MG, Purcell ME, McKeown EP. A real-time stream storage and analysis platform for underwater acoustic monitoring. *IBM Journal of Research and Development* 2013; 57(3/4): 15:1-15:10. doi: 10.1147/JRD.2013.2245973

9. Akram N, De Silva S, Foti M, et al. Real time data analytics platform for power grid smart applications. In: ; 2017: 1-6

10. Hazra J, Das K, Seetharam DP, Singhee A. Stream Computing Based Synchrophasor Application for Power Grids. In: HiPCNA-PG ’11. ACM; 2011; New York, NY, USA: 43–50

11. Jayasekara S, Perera S, Dayarathna M, Suhothayan S. Continuous Analytics on Geospatial Data Streams with WSO2 Complex Event Processor. In: DEBS ’15. ACM; 2015; New York, NY, USA: 277–284

12. Kamburugamuve S, Ramasamy K, Swany M, Fox G. Low Latency Stream Processing: Apache Heron with Infiniband Intel Omni-Path. In: UCC ’17. Association for Computing Machinery; 2017; New York, NY, USA: 101–110

13. Seshadri S, Kumar V, Cooper B, Liu L. A Distributed Stream Query Optimization Framework through Integrated Planning and Deployment. *IEEE Transactions on Parallel and Distributed Systems* 2009; 20(10): 1439-1453. doi: 10.1109/TPDS.2008.232

14. Dayarathna M, Suzumura T. *A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters*: 164–178; Berlin, Heidelberg: Springer Berlin Heidelberg . 2013

15. Dayarathna M, Suzumura T. Multiple stream job performance optimization with source operator graph transformations. *Concurrency and Computation: Practice and Experience*; n/a(n/a): e5658. e5658 cpe.5658doi: 10.1002/cpe.5658

16. Suhothayan S, Gajasinghe K, Loku Narangoda I, Chaturanga S, Perera S, Nanayakkara V. Siddhi: A Second Look at Complex Event Processing Architectures. In: GCE '11. ACM; 2011; New York, NY, USA: 43–50

17. GeeksforGeeks . 0/1 Knapsack using Branch and Bound. URL: https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/; 2019.

18. DERA . EDGAR Log File Data Set. URL: https://www.sec.gov/dera/data/edgar-log-file-data-set.html; 2019.

19. Google . Nexmark benchmark suite. URL: https://beam.apache.org/documentation/sdks/java/testing/nexmark/; 2019.

20. Ravindra S, Dayarathna M, Jayasena S. Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams. In: ICPE '17. ACM; 2017; New York, NY, USA: 91–102

21. Breiman L. Random Forests. *Machine Learning* 2001; 45(1): 5–32. doi: 10.1023/A:1010933404324

22. Dayarathna M, Suzumura T. A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking. In: Joshi K, Siegle M, Stoelinga M, D'Argenio PR., eds. *Quantitative Evaluation of Systems*Springer Berlin Heidelberg; 2013; Berlin, Heidelberg: 225–240.

23. Chintapalli S, Dagit D, Evans B, et al. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In: ; 2016: 1789-1792

24. Lu R, Wu G, Xie B, Hu J. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In: ; 2014: 69-78

25. Karimov J, Rabl T, Katsifodimos A, Samarev R, Heiskanen H, Markl V. Benchmarking Distributed Stream Data Processing Systems. In: ; 2018: 1507-1518

26. Dayarathna M, Suzumura T. A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters. In: Meng W, Feng L, Bressan S, Winiwarter W, Song W., eds. *Database Systems for Advanced Applications*Springer Berlin Heidelberg; 2013; Berlin, Heidelberg: 164–178.

27. Peng B, Hosseini M, Hong Z, Farivar R, Campbell R. R-Storm: Resource-Aware Scheduling in Storm. In: Middleware '15. ACM; 2015; New York, NY, USA: 149–161

28. Khandekar R, Hildrum K, Parekh S, et al. COLA: Optimizing Stream Processing Applications via Graph Partitioning. In: 2009 (pp. 308-327).

29. Gedik B, Andrade H, Wu KL. A code generation approach to optimizing high-performance distributed data stream processing. In: ; 2009: 847–856.

30. Soulé R, Gordon MI, Amarasinghe S, Grimm R, Hirzel M. Dynamic expressivity with static optimization for streaming languages. In: DEBS '13. ACM; 2013: 159–170

31. Kalyvianaki E, Wiesemann W, Vu QH, Kuhn D, Pietzuch P. SQPR: Stream query planning with reuse. In: ; 2011: 840-851.

32. Cervino J, Kalyvianaki E, Salvachua J, Pietzuch P. Adaptive Provisioning of Stream Processing Systems in the Cloud. In: ; 2012: 295-301

33. Medel V, Rana O, Bañares J Arronategui U. Modelling Performance Resource Management in Kubernetes. In: ; 2016: 257-262.

34. WSO2 . Fully Distributed Deployment. URL: https://docs.wso2.com/display/SP400/Fully+Distributed+Deployment; 2018.

35. Rodrigo A, Dayarathna M, Jayasena S. Latency-Aware Secure Elastic Stream Processing with Homomorphic Encryption. *Data Science and Engineering* 2019; 4(3): 223–239. doi: 10.1007/s41019-019-00100-5

36. Rodrigo A, Dayarathna M, Jayasena S. Privacy Preserving Elastic Stream Processing with Clouds Using Homomorphic Encryption. In: Li G, Yang J, Gama J, Natwichai J, Tong Y., eds. *Database Systems for Advanced Applications*Springer International Publishing; 2019; Cham: 264–280.

37. Ray, Brian . Enron Email Dataset. URL: https://data.world/brianray/enron-email-dataset; 2017.

38. Han B, Leblet J, Simon G. Hard multidimensional multiple choice knapsack problems, an empirical study. *Computers and Operations Research* 2010; 37(1): 172 - 181. doi: https://doi.org/10.1016/j.cor.2009.04.006

39. Janakan S, Dayarathna M. Scaling a Distributed Stream Processor in a Containerized Environment. URL: https://www.infoq.com/articles/distributed-stream-processor-container/; 2019.

40. Castro Fernandez R, Migliavacca M, Kalyvianaki E, Pietzuch P. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In: SIGMOD '13. ACM; 2013; New York, NY, USA: 725–736

How to cite this article: M. Sarathchandra, C. Karandana, W. Heenatigala, M. Dayarathna, and S. Jayasena (2020), Scheduler Mechanism for Auto Scaling Stateful Distributed Data Stream Processors in Cloud Native Environment, *Concurrency and Computation: Practice and Experience*, *2020;00:1–6*.