Adaptive Scheduling of Parallel Jobs in Spark Streaming

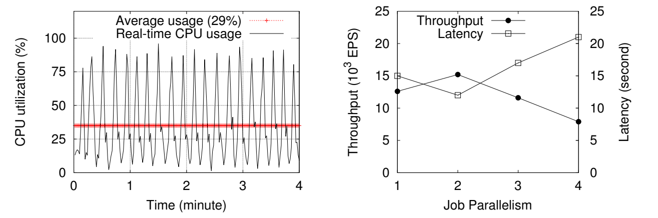
Dazhao Cheng[*], Yuan Chen[†], Xiaobo Zhou[‡], Daniel Gmach[†] and Dejan Milojicic[†]

[*]Department of Computer Science, University of North Carolina at Charlotte, USA
[†]System Software Group, Hewlett Packard Labs, Palo Alto, USA
[‡]Department of Computer Science, University of Colorado, Colorado Springs, USA
Email addresses: dazhao.cheng@uncc.edu, yuan.chen@hpe.com, xzhou@uccs.edu, daniel.gmach@hpe.com, dejan.milojicic@hpe.com

***Abstract*—Streaming data analytics has become increasingly vital in many applications such as dynamic content delivery (e.g., advertisements), Twitter sentiment analysis, and security event processing (e.g., intrusion detection systems, and spam filters). Emerging stream processing systems, such as Spark Streaming, treat the continuous stream as a series of micro-batches of data and continuously process these micro-batch jobs. Such micro-batch based stream processing provides several advantages over traditional stream processing systems, which process streaming data one record at a time, including fast recovery from failures, better load balancing and scalability. However, efficient scheduling of micro-batch jobs to achieve high throughput and low latency is very challenging due to the complex data dependency and dynamism inherent in streaming workloads. In this paper, we propose A-scheduler, an adaptive scheduling approach that dynamically schedules parallel micro-batch jobs in Spark Streaming and automatically adjusts scheduling parameters to improve performance and resource efficiency. Specifically, A-scheduler dynamically schedules multiple jobs concurrently using different policies based on their data dependencies and automatically adjusts the level of job parallelism and resource shares among jobs based on workload properties. We implemented A-scheduler and evaluated it with a real-time security event processing workload. Our experimental results show that A-scheduler can reduce end-to-end latency by 42% and improve workload throughput and energy efficiency by 21% and 13%, respectively, compared to the default Spark Streaming scheduler.**

I. Introduction

Real time processing of live data has become increasingly vital, such as web site statistics/analytics, intrusion detection systems, and spam filters. Spark Streaming [1] is a stream processing framework built on top of Spark [2] to support near real-time distributed stream processing. Instead of processing streaming data one record at a time like traditional stream processing systems (e.g., Storm [3] and TelegraphCQ [4]), Spark Streaming processes many records together. These small sets of records are called micro-batches and Spark Streaming treats them as continuously arriving RDDs. These are then passed to the Spark Engine as traditional batch workloads. It takes advantage of RDDs and their lineage properties to divide this infinite stream of data into finite chunks. The use of micro-batches and RDDs enables Spark Streaming to provide fault tolerance at scale. However, it is very challenging to schedule these micro-batch jobs in Spark Streaming to maximize the performance and resource efficiency due to the complexity inherent in Sparking Streaming workloads.



(a) CPU utilization under the default sequential scheduling.

(b) Performance impacts under different job parallelism.

Fig. 1. CPU utilization and performance impact by different schedulings

As often used for in-memory batch computations, Spark is capable of handling static dependency between jobs within an application [5]. By contrast, in more recent workloads such as Spark Streaming workloads, a batch often involves multiple jobs and there are complex and dynamic data dependencies between these jobs. The data flows generated between consecutive jobs, which we call micro jobs, are inter-dependent. A job often relies on the output of other jobs (e.g., `union` and `updateStateByKey`), and it has to wait until all the parent jobs are finished. For example, aggregating multiple objects of a job has to collect outputs from all its parent jobs.

The default Spark Streaming simply executes streaming jobs sequentially. This results in low resource utilization and poor performance. Figure 1(a) shows the CPU utilization obtained when running a real time security analytic workload with Spark Streaming using the default sequential scheduling. As shown in the Figure, the system is significantly under-utilized and the average CPU utilization is about 29%. Also, there are many spikes of CPU usage because different stages of a job have different resource consumption characteristics. For instance, map stages typically consume more CPU resources while reduce stages consume more memory resources.

A solution is simply to increase the parallelism of job execution and run the job in parallel. This can be achieved in Spark Streaming by setting `spark.streaming.concurrentJobs` to a value greater than 1. By default, Sparks scheduler runs jobs in FIFO fashion. Although FIFO policy is able to capture job dependencies well, it can increase latency because a long running job can delay jobs behind it and increase latency. Alternatively, we can use fair scheduler. Fair scheduler enables multiple jobs to share resources (equal share or weight based), but determining the ideal parallelism level and resource shares between multiple jobs is a difficult task due to uncertainties and complex data dependencies in streaming applications. Moreover, the level of parallelism and weights in Spark are manually set and remain unchanged during the entire application execution. Figure 1(b) shows the throughput and the end-to-end latency under different job parallelism levels in our experiment. The result demonstrates that a parallelism level of two achieves the best performance for this workload. Low parallelism causes inefficient resource utilization leading to poor performance while high parallelism can potentially schedule more inter-dependent jobs and result in poor performance too. As shown in Figure 1(b), increasing the parallelism to 4 results in worse performance. The latency increases by 27% and the throughput drops by 36%. The results show that native parallel execution does not necessarily improve the performance and a good scheduler must consider the dynamic resource demand between jobs, which is missing in the current Spark Streaming job scheduler. Without the fine-grained resource management, running multiple jobs by applying simple fair sharing policy can hurt the performance.

In summary, the default sequential job scheduling in Spark Streaming is inefficient. Simply increasing job parallelism without considering dynamic resource allocation can not guarantee good performance for workloads with complex dependencies such as stream processing. Further, it is challenging to determine the level of parallelism and resource shares to achieve optimal performance in stream processing. These observations motivate us to develop a more efficient job scheduler for Spark Streaming by considering the complexity and data dependency in streaming workloads. We propose A-Scheduler, a novel scheduler that adaptively schedules parallel jobs in Spark Streaming with optimal policies and parameters to achieve better performance and resource efficiency. Specifically, we make the following contributions:

- We propose A-scheduler that adaptively schedules parallel micro-batch jobs in Spark Streaming. More specifically, A-scheduler classifies and groups micro-batch jobs into two different categories and job pools based on their data dependencies: (a) independent jobs that have their input data sets available, and (b) dependent jobs that rely on the output of other jobs which is not yet calculated, and then uses FIFO and fair policies to schedule independent jobs and dependent job pool respectively.
- We propose a reinforcement learning based online tuning algorithm that automatically and adaptively adjusts the scheduling parameters, including job parallelism level and resource shares between concurrently running jobs based on changes in performance, workload characteristics and resource availability.
- We implemented A-scheduler in open-source Spark and conducted a comprehensive evaluation with a real-time security analytics workload. The experimental results show that A-scheduler can reduce the end-to-end latency by 42%, increase the system throughput by 21%, and improve energy efficiency by 13% compared to the default Spark Streaming scheduling.

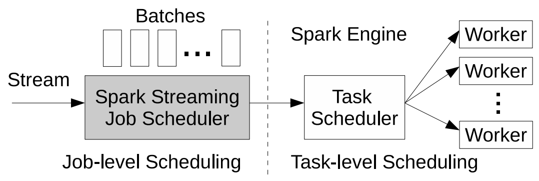


Fig. 2. Job and task level scheduling in Spark Streaming.

The rest of this paper is organized as follows. Section II introduces Spark Streaming and presents the system design. Section III and IV describe the resource allocation approach and online tuning algorithm. The implementation and experimental results are discussed in Section V. Section VI reviews related work and Section VII concludes the paper.

II. Design

A. Background

Spark is a general cluster compute engine for scalable data processing. Spark Streaming is an extension of the core Spark Engine that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. It treats streaming as a series of deterministic batch jobs. As shown in Figure 2, it internally buffers the incoming stream and converts it into small batches of a fixed duration, e.g., 1 second. These small batches, called micro batches, are represented as an RDD then handed over to the Spark framework as traditional batch jobs. We note that each of these batch jobs can consist of multiple consecutive jobs as some operations such as, e.g., joins, need previous stages to be fully completed and thus spawn new jobs. Further, each job typically consists of many tasks, e.g., a map operation executed on many workers. As shown in Figure 2, there are two scheduling levels for Spark Streaming workload, i.e., job level and task level. The job level scheduler receives the periodically generated jobs and decides when and how to schedule them in the Spark cluster for execution. The task level scheduler is then responsible for assigning individual tasks to the worker nodes for processing. In this work, we focus on the job level scheduling in Spark Streaming.

In Spark Streaming, there exists a two-dimensional dependency chain as shown in Figure 3. `In-batch dependency` operation means that some jobs rely on output data of another job of a previous stage. This is typically the case if an operation can only start once all input data is calculated. For example, `join` and `groupBy` operations require that previous

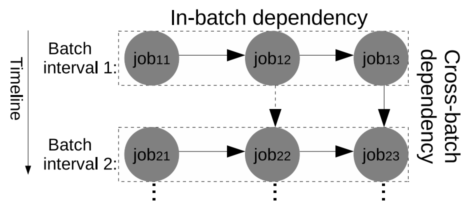


Fig. 3. Two-dimensional dependency in Spark Streaming.

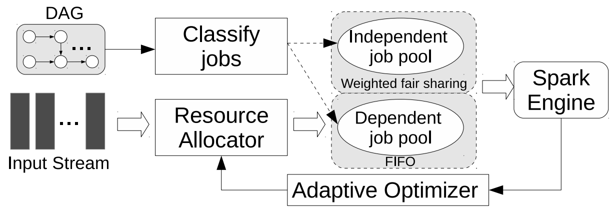


Fig. 4. Overview of A-scheduler.

stages are fully calculated. Such operations are spawned as new jobs. Further, in Spark Streaming for each micro batch a new job set is submitted to Spark.

`Cross-batch dependency` refers to dependencies between consecutive jobs as shown in Figure 3. Some operations such as `updateStateByKey` generate intermediate data that is cached locally at the individual nodes. This cached data is then required as input for the consecutive job execution of this operation in the next interval. For example, job_{23} may rely on the output of job_{13} as shown in Figure 3. It is a cross-batch dependency between job_{23} and job_{13}. In this work, we define an `independent job` as a job that relies only on already existing RDDs in the cluster. We define a `dependent job` as a job that relies on the output RDD of another job that is not yet calculated and cached.

To maintain consistent stateful operations regarding those complex data dependencies, Spark Streaming runs jobs in FIFO fashion and guarantees ordered processing of RDDs under the default implementation. As shown in Figure 3, jobs are submitted to the Spark Engine sequentially following the order of their indexes. This policy ensures that each job begins execution only after all its parent jobs are completed. However, it also leads to low parallelism of job execution for a single Spark Streaming application.

B. Architecture of A-scheduler

A-scheduler is a self-adaptive resource scheduling approach for parallel job execution in Spark Streaming. Unlike the default Spark Streaming scheduler, the new scheduler centers on two key designs:

- Multiple jobs of one Spark Streaming application are concurrently running in Spark Engine to achieve high parallelism and efficient resource utilization;
- Streaming jobs are classified into two categories, i.e., independent job and dependent job and scheduled using different scheduling policies based on the dependency.

Figure 4 shows the architecture of A-scheduler. A-scheduler first classifies micro-batch jobs into two different categories based on their data dependencies and accordingly submits jobs into two different job pools. A-scheduler uses a FIFO scheduling policy for jobs in the dependent job pool and a weighted fair sharing scheduling policy for jobs in the independent job pool. The scheduler adaptively adjusts the job parallelism level and the resource sharing ratio between concurrently running jobs. A-scheduler further collects the statistics such as the end-to-end latency for each job and system throughput. It gradually improves system performance by refining good performing configurations and discarding poor performing configurations. After a few rounds of adjusting the job parallelism settings and resource sharing policies, A-scheduler automatically achieves good performance. Since A-scheduler starts with default settings and improves them automatically, it does not require any prior knowledge of submitted jobs. The two key components in A-scheduler are its Resource Allocator and Adaptive Tuning Optimizer.

- **Resource Allocator** applies the priority based fair sharing scheduling and FIFO policy for independent jobs and dependent jobs, respectively. It further optimizes the weighted based fair sharing policy to improve the resource allocation between the two job pools and thus better satisfy overall demands.
- **Adaptive Tuning Optimizer** automatically fine-tunes the job parallelism level and resource sharing ratio between the two different job pools. It uses feedback control to adapt these parameters dynamically based on performance metrics from completed jobs.

III. Resource Allocation

A. Classifying jobs

A-scheduler classifies micro-batch jobs into two different categories by identifying data dependencies of streaming jobs based on the Directed Acyclic Graph (DAG) analysis. It uses this information to apply different resource allocation policies for the two different job categories.

Identifying dependencies between jobs: Dependencies can appear between jobs executing one micro-batch if, for example, data shuffling is required between two specific stages in the RDD lineage. In such a case the next job relies on the output of one or multiple previous jobs and it has to wait until all the previous jobs are finished. Spark defines a novel data structure named RDD which expresses DAG with RDD lineage. It provides transformations and actions to build RDD lineage and explicitly express the algorithm logic of applications. Based on the above dependency analysis, we find that the information on edges in DAG graph can be naturally used to classify the job level dependency. So we identify the data dependency between jobs by profiling the DAG graph of applications while accepting job submission.

Identifying dependency between batches: The dependency between batches happens across multiple batch intervals. It means `stateful` transformations for computations

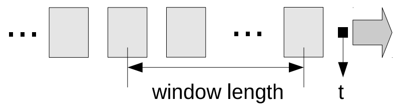


Fig. 5. Dependency between batches in window operation.

span multiple intervals, such as `sliding windows` transformation. As shown in Figure 5, the `window` operations gather together data over a sliding window. It computes function at time t by applying `union` operation over all RDDs between time `t` and time `(t- window length)`. Due to the consistency semantics of RDD definition, the output RDD of each batch interval reflects all of the input data received in that and previous intervals. If some of the previous RDD computations have not finished, Spark cannot automatically use the cached values of these RDDs for the following computations at time `t`. Accordingly, it causes resource inefficiency and poor performance. In Figure 5, the `window` operation at time `t` relies on the results of previous batches. We identify such relationship as the dependency between batches and these jobs as the dependent jobs at batch level. Similarly to the detection of job level dependency, we identify the the data dependency between batches by profiling the DAG graph of applications.

B. Resource Allocator

Compared with the default sequential scheduling, it is necessary to control the resource allocation when multiple jobs are concurrently running on the shared Spark cluster. As shown in Figure 6, the resource allocator aims to control the resource allocation at two levels, i.e., pool level and job level. At the same time, we optimize the job parallelism setting to control the total number of concurrently running jobs. Specifically, the resource allocator includes three components.

Resource allocation between job pools: We use weighted sharing policy to control the resource allocation between two different types of job pools. The proportion of independent jobs and dependent jobs decides how to set resource sharing ratio between two job pools. We define the resource sharing ratio as the parameter $r_{pool} = \frac{r_{independent}}{r_{dependent}}$, which should be dynamically adjusted in the real system to find the optimal value for the specific workload. As FIFO scheduling policy is applied to the dependent job pool, all of the resources allocated to this pool will be allocated to a single job. It prevents multiple dependent jobs to run concurrently to avoid resource waste.

Resource allocation among independent jobs: A-scheduler applies a priority based resource sharing policy to control the resource allocation among multiple parallel running independent jobs. We find that independent jobs from different batch intervals actually have different priority demands. Specifically, jobs from the former batch interval should have the higher priority than those from the latter batch interval. This is due to the fact that the independent jobs from the former batch can finish as soon as possible to achieve better latency when they are given higher priorities. Thus, A-scheduler defines the resource sharing ratio between jobs from different batch intervals as the parameter $r_{job} = \frac{r_{former}}{r_{latter}}(r_{job} \geq 1)$. The independent jobs from the same batch interval are given the same priority for fairness.

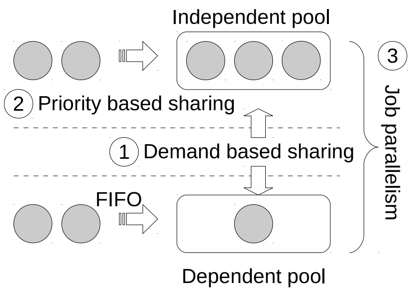


Fig. 6. Resource allocation policies and job job parallelism control.

Job parallelism control: The job parallelism parameter `spark.streaming.concurrentJobs` controls the total number of active running jobs (i.e., independent jobs and dependent jobs) in a Spark cluster. As FIFO scheduling is applied in the dependent job pool, the job parallelism control mainly focuses on the job concurrency management of the independent job pool. As the analysis in Section I, both low parallelism setting and high parallelism setting of job execution can be inefficient in Spark Streaming. An ideal job parallelism should be selected based on the real workload and the cluster hardware configurations.

Observations above show that resource allocated to individual jobs is jointly determined by three parameters, i.e., pool level ratio r_{pool}, job level ratio r_{job} and `spark.streaming.concurrentJobs`. In the following, we adaptively configure these three parameters in an online manner so as to ensure good performance and satisfy various resource allocation demands.

IV. Adaptive Online Tuning

In this section, we design an online Reinforcement Learning (RL) based algorithm to adaptively tune the level of job parallelism and resource share weights between job pools and job to achieve desired performance. It identifies efficient configurations by comparing the measured performance (i.e., latency and throughput) of completed streaming jobs with different configurations in a self-adaptive manner. Due to the periodically running property and characteristic of Spark Streaming workload, it is able to continuously improve resource allocation solution by adaptively adjusting configuration for that specific workload type. We formulate the online parameter tuning problem for parallel Spark Streaming execution as a reinforcement learning process. We define the state space S and action set A when applying the reinforcement learning approach. We use reinforcement learning optimization approach to obtain an online and adaptive parameter tuning scheme.

A. Problem formulation and parameter searching space

The parameter tuning problem of resource allocation can be formulated as a finite Markov Decision Process (MDP). Formally, for a set of states S and a set of actions A, the MDP is defined by the transition probability $P_a(s, s')$ and a reward function R. At each step t, the adapter perceives its current state $s_t \in S$ and the available action set $A(s_t)$. By taking action $a_t \in A(s_t)$, the adapter transits to the next state s_{t+1} and receives a reward R_{t+1} from the environment. The value function of taking action a in state s can be defined as: $Q(s, a) = E\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a\}$, where $0 \le \gamma < 1$ is a discount factor.

We define the state space S as the set of possible parameter values. States defined on the configurations are deterministic in that $P_a(s, s) = 1$, which simplifies the RL problem. We represent the state space as a collection of state vectors: $s = [r_{pool}, r_{job}, parallelism]$. The elements in the state vector are scheduling parameters to control the resource allocations among multiple jobs. r_{pool} and r_{job} are weight ratio at pool level and job level respectively. We shrink the searching space of these parameters to a reasonable range (i.e., $r_{pool} \in [0.1, 10]$ and $r_{job} \in [1, 10]$) in order to accelerate the search speed. The tuning granularity of r_{pool}, r_{job} and $parallelism$ are set to 0.1, 1 and 1, respectively. The action set (A) is represented as a collection of action vectors (a) : $A = [a_{pool}, a_{job}, a_{parallelism}]$.

B. Performance-aware reward function

In the resource sharing configuration problem, the desired configurations are the ones which improve system-wide performance (i.e., throughput and latency). The rewards are the summarized performance of streaming processing feedbacks on the resulted new configuration. The performance is measured by a reward which is the ratio of current throughput (thrpt) to a reference throughput plus possible penalties when latency based SLAs are violated:

$$\begin{aligned} reward &= \frac{thrpt}{thrpt_{ref}} - penalty, \\ penalty &= \begin{cases} 0 & \text{if latency} \le \text{SLA}, \\ \frac{latency}{SLA} & \text{Otherwise.} \end{cases} \end{aligned} \tag{1}$$

The reference throughput ($thrpt_{ref}$) value is the maximum achievable streaming processing rate by applying the current default sequential scheduling policy. We obtained the reference for one streaming application by tuning the system to an ideal batch interval under the SLA constraint. SLA equals the batch interval size so that the latency is maintained comparable to the batch size. This constraint guarantees that the latency is not continuously increasing and the system that can remain stable [6]. A low reward indicates either waste of resource or interference between parallel running jobs, both of which should be avoided when tuning parameters.

C. Solution of search algorithm

The solution of the RL problem aims to maximize the cumulative rewards at each state. It is equivalent to finding an estimation of $Q(s, a)$ which approximates its actual value. The basic RL algorithms in experience based solution are called temporal-difference methods, which update $Q(s, a)$ at each time a sample is collected:

$$\begin{aligned} Q(s_t, a_t) =& Q(s_t, a_t) + \\ & \alpha * [R_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)], \end{aligned} \tag{2}$$

where α is the learning rate and γ is the discount factor. The Q values are usually stored in a look-up table and updated by writing new values to the corresponding entries in the table. Starting from any initial policy, the adapter gradually refines the policy based on the feedback perceived at each step.

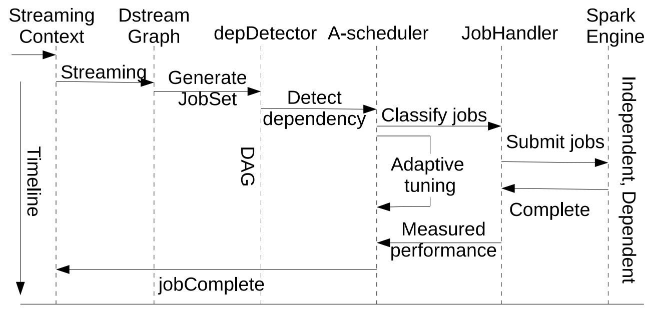


Fig. 7. A-scheduler workflow.

V. Implementation and Evaluation

A. Implementation

We implemented A-scheduler in Spark 1.4.0 by modifying the `JobScheduler`, `JobGenerator` and `StreamingListener` classes. A new component `depDetector` is added to detect the data dependencies for each job, classify each job based on its data dependencies and submit it to the corresponding job pool. During job execution, our `jobAnalyzer` collects the status of each completed job by using `StatsReportListener` and `SparkJobInfo`. The end-to-end latency and system throughput are reported by `StreamingListener` periodically and tagged with the corresponding `jobId`. Another new component is `Adapter`, which implements the online tuning algorithm.

Figure 7 shows the workflow of our A-scheduler implementation. Two job pools named `Independent` and `Dependent` are created during the initialization phase of the Spark Streaming cluster. When a new streaming job arrives, the `depDetector` detects the data dependencies by profiling the application DAG graph. `ssc.sc.setLocalProperty` is used to select the proper job pool i.e., `Independent` and `Dependent` for each individual job based on its data dependency. `Adapter` adjusts the value of `spark.streaming.concurrentJobs` to specify the number of concurrent jobs (i.e. level of parallelism). `Adapter` gradually adjusts the value by refining good performing configurations and discarding poor performing configurations.

The online adapter has a table-based Q function, which is initialized to all zeros. In our experiment, we used the

TABLE I
COMPARISON AMONG DIFFERENT APPROACHES.

Approach	Job Parallelism	Scheduling Policy	Scheduling Parameters
Sequential	1	FIFO	Manual and Static
P-FIFO	2	FIFO	Manual and Static
P-Fair	2	Fair	Manual and Static
A-scheduler	Adaptive	FIFO+Fair	Automatic and Dynamic

Sarsa(0) algorithm [7] with $\alpha = 0.15$, $\gamma = 0.7$ to drive the configuration adapter. The configuration interval is set to 15 seconds. The selection of the tuning interval is a trade-off between the parameter searching speed and the batch processing time. If the interval is too long, it will take more time to find good configurations. If the interval is too short, the jobs with new configurations may not complete and no performance feedback can be collected. Thus, we choose a control interval of 15 seconds, which is approximately 2 times of the average processing time for individual batches.

B. Experiment Setup

Testbed. For the testbed, we deploy Spark version 1.4.0 onto a cluster consisting of 17 VMs. One VM is hosting the master node and the other 16 VMs each host a Spark worker node. Each VM runs Ubuntu Server 14.04 with Linux kernel 3.16 and has allocated 4 virtual CPUs and 8 GB memory.

Distributed Real Time Event Workload. To evaluate the performance of A-scheduler, we use "Distributed Real-time Event Analysis (DREA)", which mimics a real-world production workload from industry. DREA is a rule-based security event (e.g., DNS events) evaluation system that performs complex event processing over a large number of incoming events at high rate. For the evaluation, we replay an event trace that was recorded from the real-world security event analysis application. For the experiment in this work, we employ one workload generator for each node in the cluster. Arriving events are buffered into micro-batches and these micro-batches are then processed by Spark Streaming. The application takes advantage of multiple jobs to process events in parallel.

We compare the performance of the proposed A-scheduler with three other scheduling policies: the default sequential scheduling policy of Spark Streaming (**Sequential**), parallel job scheduling with FIFO policy (**P-FIFO**) and parallel job scheduling with Fair sharing policy (**P-Fair**). The detailed characteristics of four different scheduling approaches used in the experiment are summarized in Table I. For P-FIFO and P-Fair, we manually set the job parallelism to two in the experiment, which obtain the best performance.

We compare the different scheduling policies using the following key metrics: latency, throughput and energy efficiency. We first demonstrate the effectiveness of the adaptive parallel scheduling approach. We then show the improvement from adaptive parameter tuning. And finally, we discuss the overhead and scalability of A-scheduler and the impact of batch interval and number of receivers in Spark Streaming.

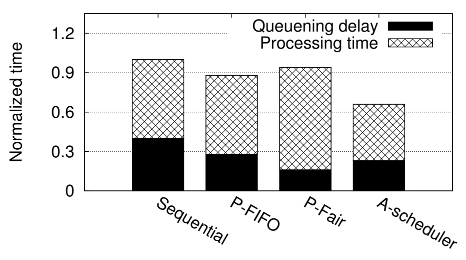
C. Effectiveness of Adaptive Parallel Scheduling

Reducing end-to-end latency. Figure 8(a) compares the performance of Sequential, P-FIFO, P-Fair and A-scheduler in terms of overall end-to-end latency, queueing delay and processing time of DREA workload. The end-to-end latency includes queueing delay and processing time for each job. Queueing delay further includes the delay caused by batch interval and the waiting time for job submission. The end-to-end latency is normalized by the sequential approach's latency.

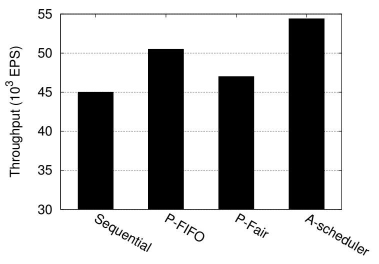
Both P-Fair and A-scheduler significantly reduce the queueing delay compared with Sequential. This is due the fact that P-Fair and A-scheduler have higher job parallelism than Sequential. It allows more jobs to concurrently run in the cluster instead of waiting in the queue. P-FIFO slightly reduces the queueing delay compared with Sequential because it applies a FIFO scheduling policy to schedule parallel jobs. Though P-Fair has the minimum queueing delay among all policies, it has the worst processing time (23% worse than Sequential). The reason is that P-Fair is dependency-oblivious and can lead to resource waste with multiple concurrent jobs. P-Fair may schedule more jobs quickly, but for jobs that depend on other jobs' result Spark will execute these other jobs a second time. Spending time on executing jobs multiple times clearly wastes resources and the results reveal that simply increasing job parallelism without considering data dependency may not improve processing time.

The dependency awareness of A-scheduler can effectively reduce both queueing delay and processing time by opportunistically scheduling parallel jobs and avoiding resource waste. Compare to Sequential, A-scheduler reduces the delay and processing time by 13% and 29%, respectively. The overall latency is improved by 42%, 32% and 37% compared with Sequential, P-FIFO and P-Fair approaches, respectively. A-scheduler benefits from the dependency awareness and the high job parallelism of the streaming workload in Spark.

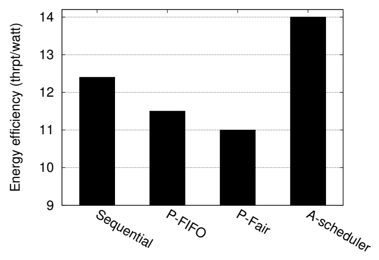
Workload performance. Figure 8(b) compares the event throughput in events per second (EPS) of the DREA workload achieved by Sequential, P-FIFO, P-Fair and A-scheduler, respectively. For this experiment, the workload generators overload the system and we monitor the throughput that the application can achieve. The result shows that A-scheduler, P-FIFO and P-Fair can increase throughput by 21%, 11% and 5% compared to Sequential, respectively. P-Fair achieves a slight performance gain by allowing multiple jobs to execute concurrently. Figure 8(b) also illustrates that A-scheduler increases the throughput by 8% and 16% compared to P-FIFO and P-Fair. A-scheduler achieves the highest throughput improvement due to its adaptive job scheduling capability that achieves a higher degree of parallelism and the fine-grained resource management strategy.



(a) Reducing end-to-end latency.

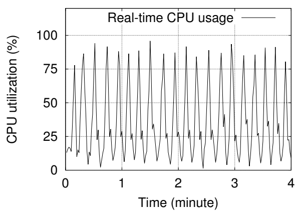


(b) Increasing system throughput.

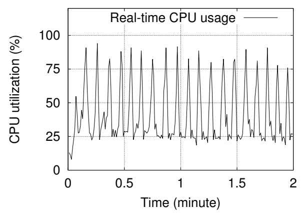


(c) Improving energy efficiency.

Fig. 8. Effectiveness of A-scheduler in terms of the end-to-end latency, system throughput and energy efficiency.



(a) Utilization under Sequential.



(b) Utilization under A-scheduler.

Fig. 9. Real-time CPU utilization under Sequential and A-scheduler.

Improving energy efficiency. A-scheduler does not just improve performance, but also increases energy efficiency by improving utilization and avoiding inefficient executions. In our experimental study, we use a performance-to-power ratio as a measure of energy efficiency. We measure the system's throughput (events/sec) and the total power consumption (watt) of the cluster to obtain throughput per watt (i.e., events/(sec$\times$watt) or events per joule). The power consumption is obtained by using VMware ESXi 5.5 Intelligent Power Management Interface.

Figure 8(c) shows the system energy efficiency achieved under different approaches. The result illustrates that A-scheduler improves energy efficiency by 13%, 22% and 27%, compared to Sequential, P-FIFO and P-Fair, respectively. As P-Fair consumes more CPU resources than A-scheduler and Sequential does, it achieves the worst energy efficiency. This is due to the fact that lots of computations in P-Fair are wasting compute resources. In particular, Figure 9 shows that A-scheduler achieves higher CPU utilization and more smooth CPU consumption compared to Sequential. We also find that memory resource utilizations under different approaches are similar to each other. Because the streaming workload in Spark is CPU-intensive rather than memory-intensive, the considered scheduling approaches have little impact on the memory utilization.

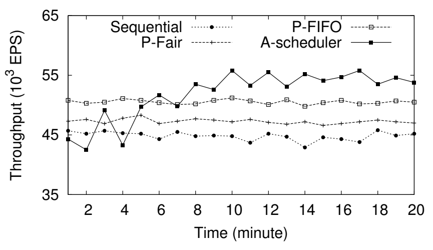
Summary. We analyzed the performance and energy efficiency impact of applying the adaptive parallel job scheduling strategy. Indeed, we find that P-FIFO and P-Fair treat all jobs in Spark Streaming as dependent jobs and independent jobs, respectively. Results above show that both, the P-FIFO and P-Fair approach, achieve sub-optimal performance and energy efficiency compared to A-scheduler. This is due to the fact that treating all jobs as dependent reduces job parallelism while treating all jobs as independent leads unnecessary resource waste. The adaptiveness of A-scheduler can effectively improve both performance and energy efficiency.

D. *Effectiveness of Online Adaptive Tuning*

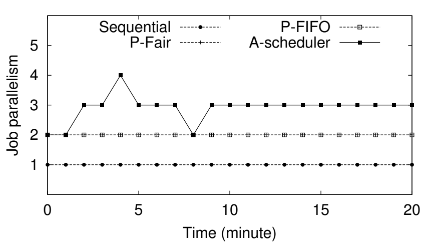
We study the benefits of the online adaptive tuning in A-scheduler. The algorithm does not assume any model of the system in consideration. It derives policies from interactions and continues to adjust the configuration parameters according to changes in workload and system behaviors. Figure 10 shows the adaptiveness under Sequential, P-FIFO, P-Fair and A-scheduler. Both P-FIFO and P-Fair are configured with a static initialization policy. They tune the parameters only based on initial performance measurement. We validate the effectiveness of the proposed online turning algorithm in terms of throughput improvement. We show the job parallelism and the resource sharing ratio tuning process under A-scheduler.

Figure 10(a) shows the real-time throughputs achieved by Sequential, P-FIFO, P-Fair and A-scheduler, respectively. The tuning process can be coarsely divided into three stages based on the performance improvement. Static policies (i.e., Sequential, P-FIFO and P-Fair) achieve higher throughput than A-scheduler does at the beginning stage since the adaptive policy needs time to search the efficient configurations. A-scheduler continuously monitors the system level performance metrics and identifies workload demand changes. During the following stage (3_{th} to 7_{th} minute), A-scheduler throughput is unstable due to dynamic tuning. The parameters (i.e. r_{pool}, r_{job} and $parallelism$) are tuned according to the output of the RL algorithm. At the final stage, A-scheduler consistently achieves better performance than the other policies do. It outperforms Sequential, P-FIFO and P-Fair by 21%, 11% and 5% in terms of throughput improvement, respectively. The result suggests that A-scheduler is able to automatically find an efficient configuration in a small number of steps. The throughput is increased and maintained at a high level due to the property of RL algorithm.

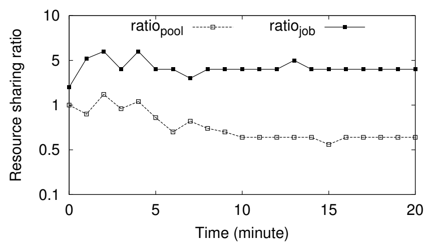
Figure 10(b) plots the adaptive tuning process of job parallelism setting during the experiment. It shows that the job parallelism of Sequential equals one statically guarantees the sequential scheduling policy while the job parallelism of P-FIFO and P-Fair set to two is the optimal under current setup.



(a) Throughput.



(b) Job parallelism.



(c) Resource sharing ratio.

Fig. 10. Adaptiveness of online tuning.

The job parallelism of A-scheduler starts from the initial value, i.e., two, and then keeps increasing the value until four. After that, job parallelism decreases to three because setting the value to four leads the performance degradation at the 4_{th} minute as shown in Figure 10(a). Job parallelism drops occasionally around the 8_{th} minute but comes back to three again immediately. The job parallelism of A-scheduler eventually is tuned to three as the ideal value in the experiment.

In addition to the level of parallelism, the online tuning adjusts the resource shares between different job pools and different job. Figure 10(c) plots the tuning process of resource sharing ratio at job pool level and job level by A-scheduler in the experiment. r_{pool} starts from the initial value, i.e., one, to search the ideal value. It firstly oscillates around the initial value and then drops below one after the 4_{th} minute. A-scheduler eventually finds $r_{pool} = 0.6$ as an ideal value for DREA workload. The initial value r_{job} is set to two because the jobs from former batches have higher priorities than those from the following batches. Then A-scheduler tries to increase the value to r_{job} for giving more priorities to jobs from former batches. However, it hurts the job parallelism when the r_{job} is set to six as shown in Figure 10(c). A high r_{job} value means decreasing the concurrency at batch level, which may deteriorate the desired benefit of A-scheduler. For DREA workload in the experiment, we find $r_{job} = 4$ is a good value for system performance improvement. Both r_{pool} and r_{job} could change occasionally during the stable stage but they can recapture the ideal values in a short while.

Summary. Figure 10 illustrates that A-scheduler achieves consistent throughput improvement by adaptively tuning the job parallelism and resource sharing ratio simultaneously. Compared with static tuning policies, A-scheduler benefits from increasing job parallelism while avoiding unnecessary resource waste. The adaptive resource sharing policy further ensures the resource allocation of individual jobs is efficient to satisfy its different demands. The result suggests that A-scheduler is able to continuously improve the system performance without prior knowledge of submitted jobs.

E. Overhead and Scalability

A-scheduler relies on Spark's daemon processes such as `StreamingListener` and `StatsReportListener` to monitor job execution and perform adaptive job scheduling. It collects the throughput and latency information of individual

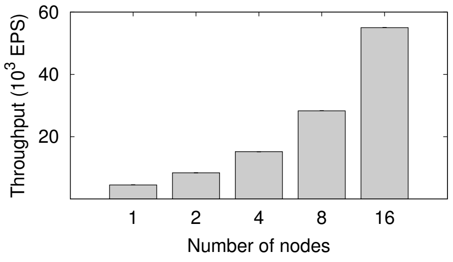


Fig. 11. Scalability.

jobs from `SparkJobInfo`. Hence, there is no additional monitoring overhead introduced by A-scheduler. The dependency analysis only profiles applications once at the initialization stage so that the overhead is minor. The parallel job scheduling does not launch any new process for job submission and handling. Furthermore, the self-adaptive RL algorithm itself takes approximately 170 ms to complete. This overhead is negligible compared to A-schedulers control interval of 15 seconds. We also evaluate the application's scalability with our A-scheduler. Figure 11 shows that the application's throughput scales well with the increasing number of worker nodes, throughput scaling from 4.5k EPS (1 node) to 56k EPS (16 nodes). For a very large cluster, multiple factors could become the bottlenecks and a further study is needed.

F. Discussion

Many other parameters, including the batch window size, the number of receivers, the size of input records, and whether the input data is serialized can have impact on the performance and resource efficiency in Spark Streaming. In particular, we discuss the impact of batch interval size and number of receivers in detail as follows.

Size of batch interval. While the micro-batch approach increases throughput at the expense of latency by coalescing input data before processing, the size of the batch interval does not form a linear relationship with the throughput. In fact, the ideal latency is obtained when the batch interval is set to slightly larger than the time it takes to process a batch. In practice, it would be better if the system can dynamically adjust the duration of the batch interval based on inputs. Dynamic batch sizes in addition will have the ability to adjust to sudden changes in data incoming rates [8].

Number of receivers. When evaluating throughput of streaming systems, the size of input data should be taken into consideration. To overcome the possible bottleneck in data intake, the solution can be increasing the number of receivers. It is also possible to obtain the performance gain by setting the number of receivers larger than the number of worker nodes.

VI. Related Work

Data stream processing can be achieved in two ways, i.e., record-by-record model and micro-batch model. Stream processing engines, such as TelegraphCQ [4], TimeStream [9] and Storm [3] are all based on the record-by-record model. These continuous operators process streaming data one record at a time. However, with today's trend towards larger scale and more complex real-time analytics, this traditional architecture has also met some challenges, such as fault tolerance and load balancing. Recently, Zaharia *et al.* [1] proposed D-Streams, a new model for distributed streaming computation that enables fast recovery from both faults and stragglers without the overhead of replication. Instead of processing the streaming data one record at a time, Spark Streaming discretizes the streaming data into micro-batches. Therefore, lost state can be simply recovered by recomputing the lost partition.

Many prior studies have shown that the performance of big data processing can be significantly improved by various scheduling techniques [10], [11], [12], [13]. The simple FIFO Scheduler may not work well since a long job can exclusively take the computing resource, and cause large delays for other jobs. Thus, many schedulers, e.g., Capacity Scheduler, Fair Scheduler, can share resources among multiple jobs. Ousterhout *et al.* [14] presented Sparrow, a stateless distributed scheduler that adapted a load balancing technique to the domain of parallel task scheduling. Hu *et al.* [15] proposed ELF, a novel decentralized model for stream processing that can simultaneously run hundreds of concurrent jobs. Our work differs from these efforts in that we investigate adaptive scheduling for parallel job execution in Spark Streaming to improve application performance.

There are growing interests in improving Spark Streaming performance with various techniques, e.g., adaptive batching size selection, online performance tuning and dynamic resource allocation. Das *et al.* [8] proposed an online adaptive algorithm that allowed the streaming system to automatically adapt the batch size. It was able to quickly adapt the batch size under changes in data rates, workload behaviors and available resources. Heinze *et al.* [16] presented an elastic scaling data stream processing prototype, which allowed to trade off monetary cost against the offered quality of service. It used an online parameter optimization, which minimized the monetary cost for the user. Although these approaches achieved performance improvement, they paid little attention to optimizing job scheduling in Spark Streaming.

VII. Conclusion

Spark is inefficient in scheduling workloads composed of multiple jobs with complex data dependencies between jobs, as is seen in many real Spark Streaming workloads and analytics applications. We propose A-scheduler, an adaptive job scheduler for parallel job execution in Spark Streaming. A-scheduler dynamically schedules multiple concurrent jobs based on the data dependency and adaptively tunes the level of job parallelism and resource shares between jobs. Our implementation and extensive evaluation with a real workload demonstrate that A-scheduler can significantly improve the application performance and energy efficiency over the existing Spark job scheduler. We plan to investigate its applicability and effectiveness to a broad range of Spark workloads. We are also interested in exploring the integration of A-scheduler with task-level scheduling in the future.

Acknowledgement

This work was partly done during Dazhao Cheng's internship at Hewlett Packard Labs. It is also partly supported by NSF CAREER award CNS-0844983, grants CNS-1217979 and CNS-1422119.

References

[1] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. of ACM SOSP*, 2013.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A faulttolerant abstraction for inmemory cluster computing," in *Proc. of USENIX NSDI*, 2012.

[3] "Apache storm," http://storm.apache.org/.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *Proc. of ACM SIGMOD*, 2003.

[5] H. Wang, L. Chen, K. Chen, Z. Li, Y. Zhang, H. Guan, Z. Qi, D. Li, and Y. Geng, "Flowprophet: Generic and accurate traffic prediction for data-parallel cluster computing," in *Proc. of IEEE ICDCS*, 2015.

[6] "Apache spark," http://spark.apache.org/docs.

[7] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in Neural Information Processing Systems*, 1996.

[8] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. of ACM SoCC*, 2014.

[9] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proc. of ACM EuroSys*, 2013.

[10] D. Cheng, C. Jiang, and X. Zhou, "Heterogeneity-aware workload placement and migration in distributed sustainable datacenters," in *Proc. of IEEE IPDPS*, 2014.

[11] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," in *Proc. of IEEE IPDPS*, 2015.

[12] D. Cheng, J. Rao, Y. Guo, and X. Zhou, "Improving mapreduce performance in heterogeneous environments with adaptive task tuning," in *Proc. of ACM/IFIP/USENIX Middleware*, 2014.

[13] D. Cheng, P. Lama, C. Jiang, and X. Zhou, "Towards energy efficiency in heterogeneous hadoop clusters by adaptive task assignment," in *Proc. IEEE ICDCS*, 2015.

[14] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. of ACM SOSP*, 2013.

[15] L. Hu, K. Schwan, H. Amur, and X. Chen, "Elf: Efficient lightweight fast stream processing at scale," in *Proc. of USENIX ATC*, 2014.

[16] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. of ACM SoCC*, 2015.