Dslash: Managing Data in Overloaded Batch Streaming Systems

Robert Birke
IBM Research - Zurich
bir@zurich.ibm.com

Mathias Björkqvist
IBM Research - Zurich
mbj@zurich.ibm.com

Evangelia Kalyvianaki
City University London
evangelia.kalyvianaki.1@city.ac.uk

Lydia Y. Chen
IBM Research - Zurich
yic@zurich.ibm.com

***Abstract*—Real-time processing of big data has become a core operation in various areas of business, such as extracting value from real-time social network data. Big data workloads in the wild show a strong temporal variability that not only poses the risk of slow responsiveness in data analysis, but also leads to a high risk of service outage. The recent development of batch streaming systems based on the MapReduce framework is shown effective on non-overloaded systems. However, little is known on how to enhance the performance of the batch streaming systems for bursty workloads. In this paper, we propose a latency-driven data controller, Dslash, which aims to process as much data as possible, while processing these as fast as the application target latency and system capacity allow. In particular, we implement Dslash on Spark Streaming – an emerging and complex batch streaming system. Dslash features include (i) placing data in an augmented distributed memory, (ii) shedding out-of-date data, (iii) improving the processing locality of Map tasks, and (iv) delaying data processing in transient overloads. Extensive evaluations on a large number of workloads show that Dslash can ensure stable and fast responsiveness compared to vanilla Spark Streaming systems.**

I. Introduction

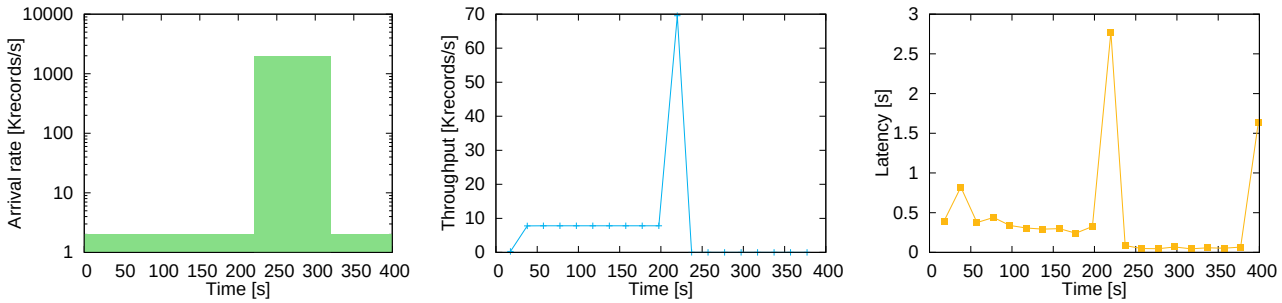
Recently, there has been a surging demand for real-time processing of large volumes of social media data and extracting information pertaining to business, politics, and society. For example, real-time information from Twitter has been used for analyzing stock market trends [1] and warning of natural hazards [2]. Social media data tend to show strong temporal variability leading to volatile and bursty workloads that pose challenges in resource management. According to [3], the peak load can be an order of magnitude higher than the average load. Consequently, the momentary peak data load can exceed the system capacity and lead to severe service outages, which can seriously undermine the user experience. Examples include the Slashdot effect [4] and the Fail Whale of Twitter – consequences of a sudden workload surge causing a service outage.

Due to the popularity of streaming applications several different systems exist. Systems such as IBM System Info Stream [5] and Storm [6], are conventionally based on a continuous operator model, in which long-running, operators receive data records, update their internal state in case of stateful operators, and output analysis results. Other systems [7] motivated by the efficacy of the MapReduce paradigm in processing big data leverage the MapReduce-based analysis platforms as processing engines, including the very recent development of Spark Streaming [8], where query analysis is executed by a set of map and reduce tasks. A long standing challenge across distributed streaming systems is to optimize their performance against overloads. The two predominant solutions include data shedding [9] and scaling out the CPU capacity with additional processing nodes [10]. Although efficient *in isolation*, these techniques cannot be used against bursty workload. Data shedding would simply discard all excess load thus rendering any query inadequate to produce accurate results. A scaling out approach adds nodes to process the peak of the burst, however the system remains over-provisioned between peaks thus waisting costly Cloud resources.

It remains an open challenge for data streaming systems how to offer high service availability and to produce "fresh" results in real-time across *any load conditions*. Results' freshness depends on the time window typically associated with data stream processing queries. For example, a query that calculates the *ten most popular topics in Twitter every five minutes* produces new, fresh results every five minutes. To maintain 100% uptime and ensure up-to-date analysis, data streaming systems need to process new, incoming data in near real-time by keeping the latency of the end-to-end data processing just below the query time window.

However, streaming applications can relax the strict time condition of the results' time window and tolerate delays in delivering query results. The previous example query could deliver new results every five minutes and ten seconds. Results can still be meaningful to the end-user and do not significantly compromise the semantics of the query. However, such additional latency in the expectation of the results delivery enables us to uniquely tackle bursty workloads by temporarily storing data in memory for latter processing instead of shedding all the excess load given the systems' computational and storage capacity.

In this paper, we address the following challenging question: how to best store, shed, and process data records in MapReduce-based streaming systems such that both a high service availability and low latency system can deal with harsh load conditions. To such an end, we develop Dslash, a latency-driven controller that keeps and processes as much data as the computational and storage resources allow in order to meet the target latency. To demonstrate our controller, we use Spark Streaming, an emerging MapReduce-based analysis



(a) Load (b) Throughput (c) Latency

Fig. 1: System performance of Spark Streaming experiencing the Slashdot effect.

platform. Dslash implicitly augments the processing speed of the CPUs by improving the task locality and leveraging the distributed memory of the cluster, with a particular focus on peak loads that temporarily exceed the available capacity. The data locality, i.e., tasks executing data blocks stored in local memory, is crucial to maintain satisfactory performance, even at the cost of an extra scheduling delay to execute local data than remote data [11, 12]. We extensively evaluate Dslash on a wide range of workloads in terms of data rates and query types, and Spark Streaming configured with different scheduling policies.

In particular, Dslash has two main features: (i) data placement, which stores the data records directly onto the distributed memory; and (ii) a latency-driven controller, which explicitly sheds data records that cannot meet the latency target, and implicitly delays the processing of data records via the extended memory storage, especially during times of capacity overload. We compare Dslash against vanilla Spark Streaming under two types of queries: one stressing the I/O bandwidth and the other one stressing the computational capacity, under various harsh load conditions. When enabling the placement module, Dslash is able to maintain 100% responsiveness and remarkable data locality, while the vanilla system simply crashes and leads to service outages. Moreover, Dslash effectively uses both CPU and memory capacity to process analyses in a timely manner, while shedding a minimum amount of data.

Our contributions are threefold. First, we develop an effective data shedding and delaying controller to meet the target latency with a minimum of data shedding. Second, we propose a novel data placement design that augments the system storage capacity handling peak loads and improves data locality in batch streaming systems. Third, Dslash agilely uses available CPU and memory resources to maintain high responsiveness in terms of availability and latency.

II. Motivation

Prior to introducing the design of Dslash, we first illustrate the performance impact of the bursty workload, in particular under the Slashdot effect. Figure 1 illustrates the incoming data record load, service outages, and the analysis latency of executing a particular query analysis in Spark Streaming in

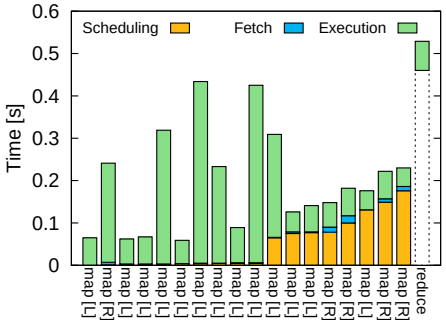


Fig. 2: Job execution breakdown.

subfigures (a), (b), and (c), respectively. The details of the experimental Spark Streaming set up and the queries used can be found in Section III and V, respectively.

One can see that for most of the experiment duration, data records arrive at the rate of 2,000 records per second except a huge spike of 2,000,000 records per second, emulating the Slashdot effect. As the system is not provisioned for such an overload, it becomes unavailable, i.e., throughput drops to almost zero after 200 seconds, shown in Figure 1 (b). Meanwhile, the latency increases from 0.5 seconds to almost 3 seconds, just before the system crashes. Such a huge spike in load collapses the receiver of Spark Streaming, as it does not have enough capacity to process nor to store the data records. In Figure 2, we further present the time-breakdown for an analysis job, consisting of multiple map tasks and one reduce task, represented by rightmost bar. The reduce task creates the final result from the outputs of the map tasks, hence it starts after all map tasks completed. To illustrate locality, we add the label R, and L, denoting map tasks working on remote and local data blocks, respectively. The system spends most of its time on map processing and scheduling delay, defined as the time spent in the queue waiting for an available Spark compute-unit that might have local data. It is worth mentioning that remote tasks consume roughly 11 milliseconds to transfer data from remote to the local memory.

When encountering transient peak loads, Spark Streaming suffers from low responsiveness due to insufficient storage or limited computation capacity. This observation not only

highlights the challenges of managing peak loads but hints that the solution lies in augmenting the CPU and memory capacity momentarily as the target latency permits. Moreover, the high variability of processing times across tasks, scheduling delays and data locality further challenges the management of data.

III. System

In our architecture we consider Spark Streaming [13] as a reference batch streaming system. Spark Streaming is a library on top of the Spark framework [14] which delivers data stream processing capabilities to the Spark engine. In the following text, we first highlight the key components relevant to the generation and execution processes of tasks and jobs. In Figure 3, we illustrate two views of Spark Streaming: the top part shows the logic view of data flow, whereas the bottom part shows the physical view.

A. Spark Streaming architecture

Spark Streaming comprises two main components as shown in Figure 3: the *receiver* and the *scheduler*. The receiver accumulates incoming data from sources into *blocks* in fixed time intervals. The scheduler further batches many blocks together into *jobs* in separate fixed time intervals.

The logic flow of a query in Spark Streaming is as follows: First, data records are generated from the source at various rates (e.g., 300 data records per ms). Every arriving tuple is stored in the *record queue*. The receiver generates a block by grouping all the records from the record queue every τ ms, e.g., 50ms or 500ms, termed the *block interval*. All generated blocks are stored in the *block queue*. The scheduler (a different scheduler from the one within the Spark engine) creates a job by taking all blocks in the block queue every *job interval*, denoted by β, e.g., every 2s or 20s. The resulting number of blocks per job is then given by β/τ.We note that both job and block intervals are tunable parameters, the optimization of which can be found in [15]. Afterwards, the scheduler processes the analysis jobs that are waiting in the job queue, following the discipline of first-come, first-served (FCFS). The scheduler also allows multiple jobs to be concurrently executed by Spark - in particular, we allow a maximum of four concurrent jobs.

In terms of physical implementation, the receiver physically stores all generated blocks in the block storage located in memory, and keeps only the metadata in the block queue (shown as the dashed line in Figure 3). We term such a storage scheme as the *default placement*. Both Spark and Spark Streaming support a variety of storage engines ranging from the traditional Hadoop filesystem (HDFS) [16] to newer projects such as Tachyon [17]. In Spark Streaming the default choice is `INHEAP`, which stores the data directly in the memory of the Spark nodes. Such a choice is preferable from the perspectives of speedy memory access and short data life cycle, as there is no need for long-term data storage in the streaming paradigm.

After leaving the job queue, each block is executed by a task specified by the scheduler, as shown by the arrow number 5 in Figure 3. Following the concept of the MapReduce paradigm, a Spark job is divided into parallel *tasks*, one for each of the β/τ blocks in the job. Each task occupies one executor corresponding to one or more CPU cores. The discussion of optimal executor configuration is out of the scope of this work. If the number of tasks is to low (less than the number of executors), the system utilization is inefficient due to unused cores. Reducing the block interval increases the number of tasks, but the recommended minimum is 50ms in order to avoid excessive overhead.

When executing blocks that are locally (remotely) stored in the block storage of the same (different) node, we term them local (remote) tasks. The locality of jobs is measured by the percentage of local tasks. When executing on remote blocks, the task essentially copies/shuffles the data block from the remote host to the local one, and then starts processing them. As such, the processing time includes the data fetching time. The default placement, where all blocks are stored in the receiver's storage, suffers from limited storage capacity as well as the low locality.

B. Latency

We define the end-to-end latency at the block level, as the time elapsed from the arrival time to the departing time of a block. As the blocks are processed in jobs, the departing time of a block is thus defined by the job departing time. Due to the nested dependency between blocks and jobs, it is complex to model the end-to-end latency. Figure 4 summarizes the definitions of the different latencies and waiting time considered in this work. The target latency of the analysis is given by L^*. According to the architecture of Spark Streaming, we can roughly divide the end-to-end latency into three parts: queueing time in the block queue, queueing time in the job queue, and the job processing time by the Spark engine. We further introduce the term *job latency* as the time from when a job enters the job queue until it leaves the Spark engine. In a default setting of Spark Streaming, termed *vanilla* Spark Streaming, jobs spend a significant amount of time in the job queue when the system is overloaded, meaning that the record arrival rate exceeds the capacity of the Spark system. Moreover, Spark Streaming can also drop blocks at the block queue when the receiver node is out of memory for its block storage.

The job processing time depends on the amount of data per task and the computation dependencies among tasks. The scheduler offers tasks, which may or may not contain blocks, to available executors. We term the time when tasks are waiting for the assignment of executors as *scheduling delay*. Such a scheduling delay can grow tremendously when the number of tasks is much higher than the number of available executors, resulting in multiple waves of processing. Tasks executing remote blocks, i.e., with no locality, incur a fetching time. To achieve a high data locality, tasks try to get offers of executors that have local data by voluntarily waiting longer than necessary, increasing the scheduling delay [11]. The default threshold for scheduling wait is 3 seconds, which can

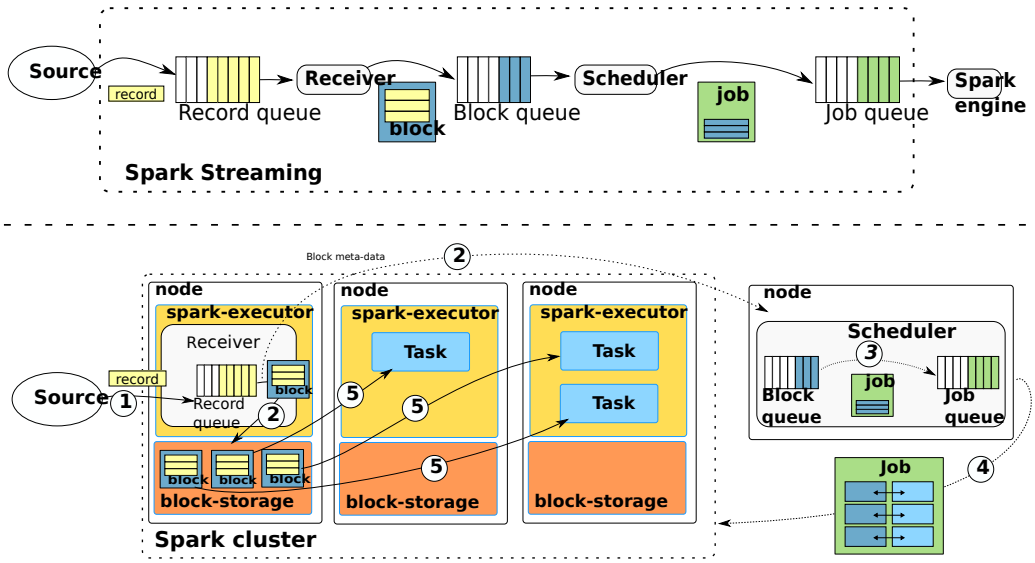


Fig. 3: Architecture of Spark Streaming: logic view (top) and physical view (bottom).

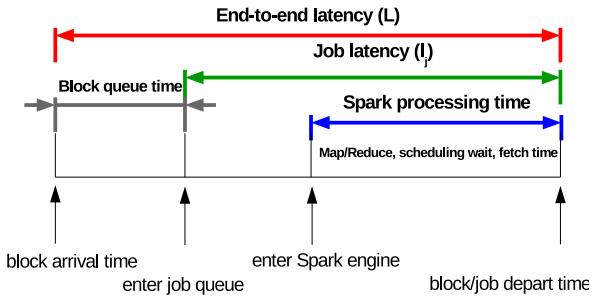


Fig. 4: End-to-end latency breakdown.

be adjusted to try to improve the trade-off between locality and scheduling delay.

IV. DSLASH

The objective of Dslash is to store and process records as fast as the target latency, computation and storage capacity of batch streaming system permits, especially during the transient overloads. To such an end, Dslash comprises three modular components: Round-robin data placement, a shedding controller, and a delaying controller. The round-robin placement evenly distributes every arriving block in block storage of all Spark nodes, whereas the default placement only uses the receiver's block storage. At the beginning of every job interval, the shedding and delaying controllers, shown in Figure 5, directly communicate with the scheduler, which actuates their control actions. On a high level, Dslash works as follows. Depending on the job interval and the observed job latency, the delaying controller decides the amount of data records that can enter the job queue and be processed by the Spark engine without overloading the system. This information, along with the latency target, is provided as input to the shedding controller, which sets an age threshold for data blocks. The data block queue contains the freshly arrived blocks, as well as delayed blocks from the previous rounds of processing. Any data blocks older than the age threshold are dropped from the block queue by the shedding actuator, and the remaining blocks are provided as input to the delaying actuator. The delaying actuator forwards items up to the maximum set by the delaying controller, whereas any remaining blocks are kept at the head of the data block queue. The admitted data blocks are scheduled and processed by the Spark Streaming engine, which at the end feeds back the observed job latency to the delaying controller.

The modular design of Dslash is motivated by its flexibility for different systems, but also the inherent complex nature of batch streaming systems. Indeed, determining a controller model of complex systems such as Spark is no mean feat, especially when encountering time-varying load dynamics.

A. Round-robin placement

The goal of the Dslash placement is to mitigate the drawbacks of the default placement by increasing the data locality as well as the buffer memory used to store the blocks. We modify the Spark Streaming receiver such that when the blocks are created, they are stored directly at all available nodes in a round-robin fashion. Consequently, we term the Dslash placement *round-robin placement*. In particular, we modify the `INHEAP` block storage of Spark to build a distributed memory that is readable and writable from all the nodes in the cluster. Our patch is based on the `Block Transfer Service`, a built-in feature of Spark that allows transferring of blocks between nodes, and the patch requires only minor modifications of the Spark codebase.

The advantages of the proposed placement come directly from the augmented storage space and smart distribution of

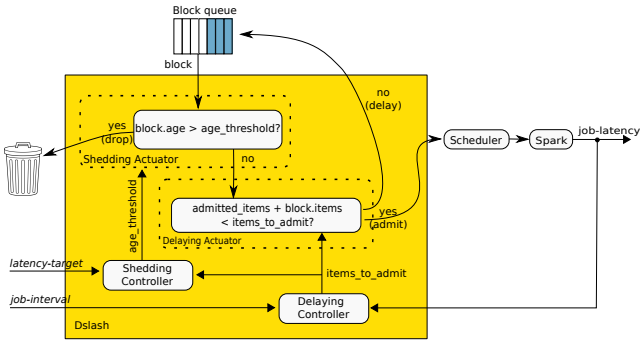


Fig. 5: Dslash: controller overview.

data blocks. Clearly, under round-robin placement, the storage space of blocks upon their arrival amounts to the entire cluster, reducing the risk of reaching the storage limit of the receiver. Moreover, as the blocks are pre-distributed evenly across all nodes, the Spark scheduler can distribute the tasks to available executors with higher probability of having local data, hence improving the locality.

We note that a similar result can be achieved by enabling the replication option of the `INHEAP` storage engine. However, such an option suffers from two drawbacks: (i) it doubles the data storage requirement, reducing the memory storage for fresh data, and (ii) the operation of replicating the data consumes more CPU power than distributing the data as done in our round-robin placement.

B. Delaying controller

The goal of the delaying controller is to determine the amount of data, n, that can be admitted at every job interval, β, so that the Spark processing engine is not overloaded. In other words, all the blocks dispatched by Dslash can be completed by the Spark engine every job interval, i.e., the job latency (l_j) is less than the job interval.

We model the system as dominated by the costs associated with the data gathering and the data processing, respectively. These costs model a complex interplay between the data and the components of the systems which cannot be readily measured and which dynamically evolve over time depending on the data arrival rates. To overcome the drawbacks of estimating these two costs off-line via system identification techniques [18], we resort to and adopt a feedback controller presented in [19]. We thus model the job latency l_j at a given time t as $l_j(t) = n(t)\ c_g(t) + n(t)\ c_s(t)$ where $n(t)$ is the data admitted, and $c_g(t)$ and $c_s(t)$ are the data gathering and processing costs, respectively. To further simplify the model, we combine $c_g(t)$ and $c_s(t)$ into $c(t)$, i.e. $c(t) = c_g(t) + c_s(t)$, hence:

$$l_j(t) = n(t)\ c(t). \tag{1}$$

To ensure the system capacity is well utilized without being overloaded, the optimal amount of data $n^*(t)$ is given by Equation 1 when $l_j(t) = \beta$:

$$\beta = n^*(t)c(t). \tag{2}$$

The goal here is to get to $n^*(t)$ provided that $c(t)$ is unknown and time-varying. Since the system allows a new control decision to be made at the beginning of every job interval, we set the control period equal to β and consider a simple integral controller:

$$n(t+\beta) = n(t) + p\ e(t-\delta(t)),$$

where p is a positive constant, $e(t) \triangleq n^*(t) - n(t)$ is the error from the optimal value at time t, and $\delta(t)$ models a time-varying bounded delay in the feedback loop, i.e. $0 \leq \delta(t) \leq \delta_{max}, \forall t$.

Using Equations 1 and 2, we rewrite the error as:

$$e(t) = \frac{\beta}{c(t)} - n(t) = \frac{\beta - c(t)\ n(t)}{c(t)} = \frac{\beta - l_j(t)}{c(t)}, \tag{3}$$

which depends on $c(t)$. If $c(t)$ would be known and, more importantly, constant, our controller would be a simple integrator. However, here $c(t)$ is unknown and depends on the time-varying data arrival rate. Taking into account the delay of the latency measures and substituting $c(t)$ by inverting Equation 1 our controller becomes:

$$n(t+\beta) = n(t) + p\ \left(\beta - l_j(t-\delta(t))\right)\ \frac{n(t-\delta(t))}{l_j(t-\delta(t))}, \tag{4}$$

which will be stable by an appropriate choice of a small enough p. To simplify and reduce oscillations in the controller, we approximate

$$\frac{n(t-\delta(t))}{l_j(t-\delta(t))} \approx \frac{n(t)}{\beta} \tag{5}$$

and account for stability when $n(t-\delta(t)) < n(t)$ and $l_j(t-\delta(t)) > \beta$ by choosing a control gain q such that

$$q\ \frac{n(t)}{\beta} \leq p\ \frac{n(t-\delta(t))}{l_j(t-\delta(t))}.$$

Hence the controller in Equation 4 becomes

$$n(t+\beta) = n(t) + q\ \left(\beta - l_j(t-\delta(t))\right)\ \frac{n(t)}{\beta} \tag{6}$$

where q must be chosen such that the controller remains stable. To set q, we observe that Equation 6 can be rewritten as

$$\begin{aligned} n(t+\beta) &= n(t) + q\ \left(\beta - l_j(t-\delta(t)\right)\ \frac{1}{c(t-\delta(t))} \\ &= n(t) + q\ e(t-\delta(t)) \end{aligned}$$

by replacing $n(t)/\beta$ by $1/c(t-\delta(t))$ based on Equation 5 and using the definition of $e(t)$ in Equation 3; and consider the following proposition from [20]:

Proposition 1: A system of the form:

$$x(t+1) = x(t) + k\ e(t-\delta(t))$$

for $0 \leq \delta(t) \leq \delta_{max}, \forall t$ is stable provided that the control gain k is selected such that

$$0 < k < \frac{2}{2\ \delta_{max} + 1}.$$

Even if the controller in Equation 6 is non linear, from Proposition 1, it can easily be deduced that by choosing q small enough, the system is stable provided that the corresponding linear system is stable. We further note that $\delta(t)$ can be practically set to zero in a stable system with a small control gain.

C. Shedding controller

While the delaying controller decides the optimal amount of data, $n^*(t)$, the selection of blocks is determined by the shedding controller. At the beginning of a job interval, it first estimates the maximum queuing time $w^*(t)$ at the block queue for unprocessed blocks such that the target of end-to-end latency, $L*$, can be met. Fresher blocks having queue time lower than $w^*(t)$ can be processed at the current job interval or delayed at the queue in a FCFS manner depending on $n^*(t)$, whereas older blocks, i.e., having waiting times exceeding $w^*(t)$, are dropped.

We derive $w^*(t)$, based on the simple relationship between the end-to-end latency and job latency, l_j. From Figure 4, we know that the end-to-end latency L of a block is given by

$$L = w + l_j, \tag{7}$$

where w denotes the queueing time in the block queue, and l_j is the job latency during which the block is batch processed.

As the delaying controller is triggered at every job interval, we also employ the slotted design for the shedding controller, and thus introduce a time index to Equation 7. At the beginning of every job interval, say at time t, we can write $w(t)$ from Equation 7 as

$$w(t) = L(t) - l_j(t). \tag{8}$$

To derive $w^*(t)$ that satisfies the end-to-end target latency, we first substitute $L(t)$ by L^* in Equation 8. Moreover, we propose to estimate $\widehat{l_j}(t)$ by considering the current decision of the delaying controller, $n(t)$. Combining with Equation 1, we thus derive $\widehat{l_j}(t) = n(t)\ \widehat{c}(t)$, where $c(t)$ is estimated from the last measured job latency and control decision, i.e, at time $t - \beta$:

$$\widehat{c}(t) = \frac{l_j(t-\beta)}{n(t-\beta)}.$$

Finally, at the beginning of every job interval the shedding controller computes $w^*(t)$ as:

$$w^*(t) = L^* - n(t)\frac{l_j(t-\beta)}{n(t-\beta)},$$

and uses it as a threshold to discard the blocks waiting at the block queue or forward them to the delaying controller.

D. Observation on end-to-end latency

We present an observation to decide appropriate target values for the end-to-end latency targets.

For a batch streaming system in its steady state, the worst arrival time for a new block is just after the scheduler admits a job. This block must wait up to β time before entering the job queue. Moreover, we define the system being non-overloaded, if it is able to process all admitted blocks in the job queue within the job interval, i.e., β. Combining these two observations, the worst end-to-end latency at a non-overloaded system is approximately $L = 2\beta$. This highlights that, to avoid excessive drops in the aforementioned worst case scenario, the delay controller requires a target end-to-end latency $L^* > 2\beta$.

V. Evaluation

In this section, we evaluate the proposed latency-driven controller, Dslash, on Spark Streaming and compare its performance against a vanilla Spark system configured with default values, including the block placement policy, and with no admission control policy applied. Our aim is to show that Dslash is able to achieve high service availability as well as target latency under various challenging overload conditions and with different types of analysis queries. To further emphasize the efficacy of the different components of Dslash, we evaluate Dslash with different combinations of its three modules: round robin placement, and shedding and delaying controllers. We first detail the experimental setup and present the overall performance of Dslash against vanilla spark. Afterwards, we highlight the improvements in handling peak loads due to the proposed round robin placement and data shedding at high load. Finally, we show the additional effect on the reduction of data drop attributed to efficiently delaying data blocks.

A. Experimental setup

We implement our solution in Spark Streaming version 1.3, deployed on a testbed of 16 virtual machines. The hosting physical servers are equipped with two Intel Xeon E5-2690 at 2.9 GHz totaling 32 hyperthreaded cores, 96 GB of RAM and 10G network connectivity, whereas VMs are equipped with 2 virtual cores and 4 GB of RAM, corresponding to t2.medium EC2 instances. The VMs are run with qemu/KVM as hypervisor on top of Debian 7. A separate physical server is used to inject the workload into the Spark VM cluster. Out of the 16 VMs, one acts as Spark master, while the other 15 act as Spark workers, each offering 2 cores. The Spark master coordinates the Spark cluster, whereas the workers host the Spark executors running the Spark tasks.

1) Spark parameters and metrics: Spark and Spark Streaming offer a large number of parameters that can be used to tune the system performance. In the following we present the main ones that differ from their default values in our experimental setup: configuration of spark executors, and blocks/jobs generation.

We configure each Spark executor with 2 cores, for a total of 30 cores. Moreover, given that each VM has 4 GB of memory, we set the Spark worker memory limit to 2.7 GB and the executor memory limit to 1 GB, with up to 100 MB reserved for block storage. We store data in memory, i.e., set the storage level to MEMORY_ONLY, since this is the fastest option. In terms of generating blocks and jobs, we set the jobs interval, β, based on the query as described in SectionV-A2, and use 40

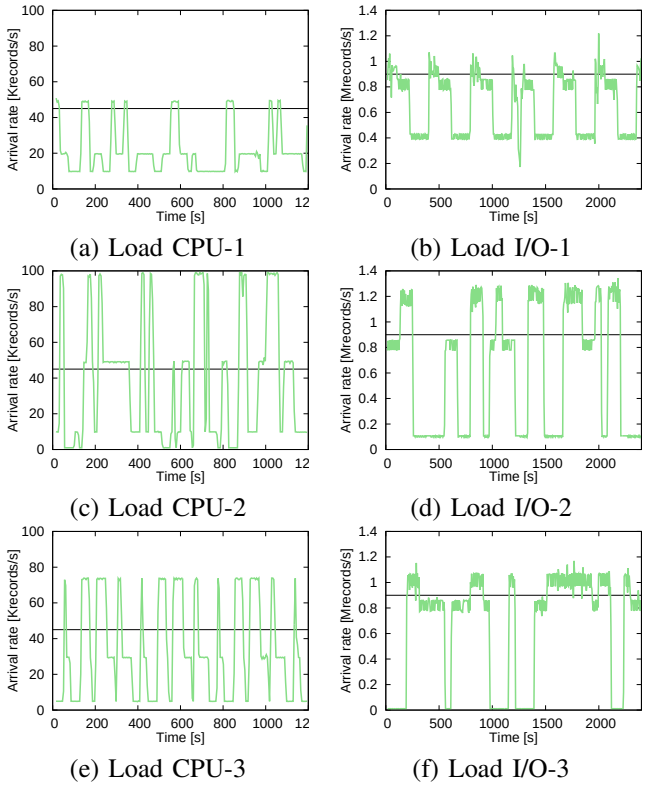
as a ratio for number of blocks per job to determine the block interval, τ. Note that such a ratio is slightly greater than the number of executors, i.e., 30, to better use the parallelism of the entire system. Last, to test the robustness of Dslash under Spark engines with different configurations of locality, we also vary the wait time thresholds in the scheduling policy.

2) Analysis loads: We test the system with two different queries, where both queries emulate the log analysis of a server farm, monitoring the overall system status. In more detail, we count the number of different log messages grouped by three severity levels and message codes, and further compress the output to save space. The resulting MapReduce jobs are similar to the popular WordCount benchmark [21]. The CPU-intensive query performs some additional filtering operations on the type of errors, increasing its computational complexity over the I/O-intensive query. The I/O-intensive query has a job interval β of 20 seconds, whereas the CPU-intensive query has a job interval of 2 seconds. Following the observation in Section IV-D, we set the target latencies slightly above twice the job interval to allow room for the delaying controller to act: i.e., 45 seconds and 6 second for the I/O-intensive and CPU-intensive query, respectively. Based on the job intervals, we also vary the wait time thresholds of the locality-aware scheduler. In both cases we use four different values: i.e., $\{0, 1000, 2000, 3000\}$ s with the I/O-intensive query, and $\{0, 100, 250, 500\}$ ms with the CPU-intensive query.

We use a custom load generator written in the Go programming language to stream records emulating log entries composed of a severity level, message code, timestamp, and host. The load generator runs on a separate physical machine to prevent it from being the bottleneck of the system. Preliminary tests show that the load generator can easily achieve up to 3 Mrecords/s using only a fraction of the 10G link between the load generator and the Spark cluster. To emulate different loads and to stress the system, the load generator randomly alternates over time between a set of different constant record rates. Figure 6 presents six load scenarios, three for each query. The black lines indicate the estimated system capacity for the two queries, making it easy to identify the overload regions.

3) Metrics of interest: We define the system availability as the percent of time the system is up and fully functional. The dropping metric is measured as a percentage of records discarded with respect to the records injected. As Dslash is composed of three main modules: *p*lacment, *s*hedding, and *d*elaying, we add additional subscript of $\{p, s, d\}$ to Dslash when referring to different versions of Dslash in the remainder of the evaluation section. For example, $Dslash_{psd}$ denotes all three modules enabled, whereas $Dslash_{ps}$ represents only the placement and shedding modules enabled.

We evaluate different versions of Dslash defined by the combinations of: (i) two choices of query analysis, (ii) three kinds of data rates (shown in Figure 6), and (iii) four different values of scheduling delay thresholds in the Spark engine. To ease the comparison across different queries, we sometimes present the normalized latency defined as the latency divided by the latency target in percent. Due to the large number of evaluations, we focus on presenting the average values across different system scenarios.



(a) Load CPU-1 (b) Load I/O-1

(c) Load CPU-2 (d) Load I/O-2

(e) Load CPU-3 (f) Load I/O-3

Fig. 6: Analysis load time series: three for the CPU-intensive query (left) and three for the I/O-intensive query (right).

B. Vanilla system: default Spark Streaming

We run the different experiments outlined previously and summarize the results across the vanilla Spark Streaming system, or simply $Vanilla$ system hereon, our shedding controller $Dslash_{ps}$ and our shedding-delaying controller $Dslash_{psd}$, in Figure 7. The top row shows the results for the CPU-intensive query, whereas the bottom row is for the I/O-intensive one. For each metric we present the mean across all experiments via the bar, and its variation via the whiskers.

We can see that the vanilla system is outperformed by both controllers for both queries. The reason is that all loads present challenging (close to) overload regions which make the vanilla system suffer from two main drawbacks. The first one is that the latency is not bounded, i.e., when the system is overloaded, data accumulates in the receiver and the latency rapidly grows. The high latency is clearly visible for the CPU-intensive query in Figure 7 (a). Clearly, the mean of the normalized latency is well above our target, i.e., above 100%., and highly varying as indicated by the whiskers. Such a high latency can also be explained partially by the lower mean locality, shown in Figure 7 (d). In contrast, Dslash is able to achieve a high and stable locality, thanks to round-robin placement.

The second drawback is that the accumulated data, if excessive, can lead to failures due to the lack of free memory

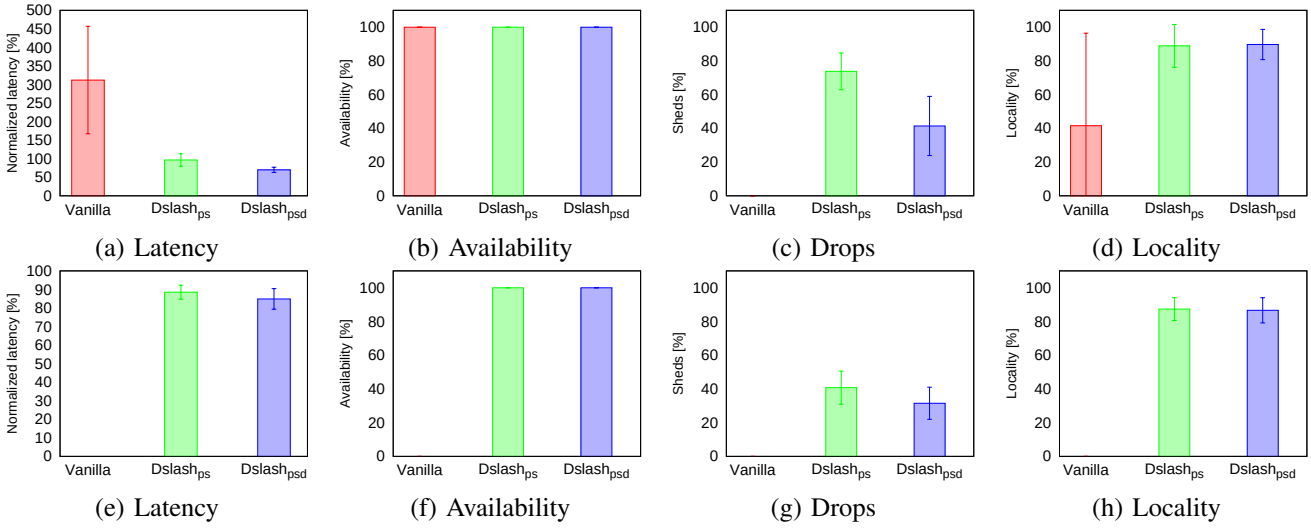


Fig. 7: Performance comparison among $Vanilla$, $Dslash_{ps}$, and $Dslash_{psd}$: the top row for the CPU-intensive query and the bottom row for the I/O-intensive query.

and consequent data drops, in particular observed for the I/O-intensive query. As a result, the system becomes unavailable as shown in Figure 7 (f). Indeed, we observe two types of failures. The first type happens when the Spark engine processes the data. A detailed analysis of the fault is outside of the scope of this paper, but preliminary analysis indicates a race condition caused by the Spark Streaming receiver deleting a block, on which the Spark scheduler later on assigns tasks. Consequently, this results in an exception at the processing task that cannot be recovered since the data is no longer available in the system. The second type directly involves the Spark Streaming receiver component. In this case no more data is admitted into the system. The worst situation is created by the heavier data rates of the I/O-intensive query scenarios, which always lead to failures. Hence the data regarding $Vanilla$ is missing in Figure 7 (e), (f), (g), and (h).

Since both vanilla and $Dslash_p$ systems fail to cope with peak loads, we do not present any results comparing vanilla Spark to $Dslash_p$.

C. Dslash

As shown in Figure 7, in contrast to the performance of $Vanilla$, both $Dslash_{ps}$ and $Dslash_{psd}$ keep the latency below the target, i.e. normalized latency below 100%, with slightly higher margins for the CPU-intensive query load scenarios. At the same time system failures are avoided as indicated by the system availability equal to 100% (see Figure 7 (b) and (f)). Considering the percentage of dropped records, $Dslash_{psd}$ outperforms $Dslash_{ps}$ controller as shown in Figure 7 (c) and (g). In terms of locality, i.e., Figure 7 (e) and (h), both controllers obtain very similar results.

To better understand these results and the differences between the two controllers, we continue our analysis in two parts. First, we focus on the effects of the round-robin placement adopted in both controllers. Second, we dig into the benefits of enabling delaying records on top of the shedding controller. To this end, we group our extensive experimental results by query type and control policies accordingly.

TABLE I: Default vs. round-robin placement: mean values across scenarios.

Query	**Placement**	**Avail.** [%]	**Locality** [%]	**Fetch time** [ms]	**Sched. wait time** [ms]
I/O-int.	Default	4.5	N/a	N/a	N/a
	RR	99.3	85.2	9.8	2.3
CPU-int.	Default	94.2	57.0	5.6	653.9
	RR	100.0	90.6	1.2	70.0

1) Placement: Here we focus on the impact of round-robin placement against Spark's default placement, showing the benefits of high availability and high locality for the I/O-intensive and CPU-intensive query, respectively. In particular, we select two group of results: Dslash with round-robin placement, i.e., $Dslash_{ps}$, and $Dslash_{psd}$; and Dslash with the default placement, i.e., $Dslash_s$, and $Dslash_{pd}$. Note that the second group of results are not summarized in Figure 7. The mean values by groups are summarized in Table I.

For the I/O-intensive query, we observe that the default placement is not able to cope with the input load arrival, and results into frequent outages due to insufficient memory space at the Spark Streaming receiver. As the shedding and delaying controllers only act after the block is processed by the receiver, Dslash without round-robin placement also suffers from low service availability, similar to $Vanillia$ systems in Figure 7. On the contrary, round-robin placement allows storing of the data blocks across all nodes, hence taking advantage of the augmented memory. It thus increases the resilience to overloads both in magnitude and time, reflected by availability of above 99% for the I/O-intensive query.

For the CPU-intensive query, as the memory pressure is

lessened due to the lower data rates, both placement policies can reach high system availability. However, the resulting locality ratios are very different: We observe that the locality ratio improves from 57% to 90%, i.e., almost all tasks are executed on the same node where the data is stored under the round-robin placement. Under default placement, only the task executing on the same node as the receiver can be local. The benefit of accessing the data locally is immediately reflected by the fetch time and the scheduling wait time in Table I. As the data is local, round-robin placement reduces the fetch time by almost 5x, from 5.6 ms to 1.2 ms compared to the default placement. Moreover, the average scheduling wait also reduces by almost one order of magnitude, i.e., from 653 ms under the default placement to 70 ms under round-robin. We note that as the mean values are collected across experiments with different scheduling wait thresholds, the default placement can achieve a higher locality ratio when such scheduling wait values increase.

D. $Dslash_{ps}$ versus $Dslash_{psd}$

Here, we focus on the differences and advantages offered by the shedding and delaying controller $Dslash_{psd}$ over the shedding-only controller $Dslash_{ps}$. To such an end, in addition to the end-to-end latency, we also present the processing time at the Spark engine, the scheduling wait time, and the block queueing time in the block queue, termed as the *block delay*. Table II presents the mean values of the experiments grouped by the two controllers and two queries.

The shedding law is the same in both controllers, i.e., both admit the same amount of data into job queue, be it in normal load or overload conditions. The difference between the two controllers lies in the fact that $Dslash_{psd}$ allows the excessive records to wait at the block queue as long as their estimated latency is within the target, whereas $Dslash_{ps}$ sheds all the excessive records. We thus expect $Dslash_{psd}$ to result in higher processing times, higher scheduling wait time, less record drops, and higher block delay than $Dslash_{ps}$, with the following reasoning:

Record drop: As $Dslash_{psd}$ takes advantage of delaying the data processing in face of excessive loads, the amount of record drops thus reduces, especially effectively during the transient overloads. This is well reflected in the number of dropped records, which is consistently lower for $Dslash_{psd}$, i.e., 30% and 50% less for the I/O-intensive and CPU-intensive query, respectively.

Spark engine processing time: By shedding less data, $Dslash_{psd}$ increases the data processing loads. The average processing time of the Spark engine thus increases as a direct consequence of the increasing loads. Moreover, the average scheduling wait time also increases because each task runs longer and resources are freed later.

Block delay: In the case of $Dslash_{ps}$ we expect this mean to be half of the job interval since we have uniform record inter-arrival times, and at the end of each job interval the queue is always completely emptied. Indeed, looking at the table the measured values match quite well, i.e., 10.1 s and 1.0 s for the I/O-intensive and CPU-intensive query, respectively. $Dslash_{psd}$ does not empty the block queue at each job interval, but accumulates data over time as long as the latency target permits. The resulting block delay increases by 13.7% and 19.8% for the I/O-intensive and CPU-intensive query, respectively.

Since the aforementioned times contribute to the end-to-end latency in a complex dependent manner, it is difficult to predict their performance for adhering to end-to-end latency targets. In the case of the CPU-intensive query, $Dslash_{psd}$ achieves slightly higher mean latency, but a slightly lower 95^{th} percentile. In the case of the I/O-intensive query, the same trend is observed, but the differences are more exacerbated: the mean is reduced by 43.6% and 95^{th} percentile is increased by 16.2%.

VI. Related Work

In contrast to the design of continuous operators, recent developments of batch streaming systems, such as Spark Streaming [22, 7], take advantage of the computational capacity offered by the MapReduce paradigm, but also provide a scalable and fault-tolerant substrate. There are a huge number of parameters that offer great potential for improving performance, e.g., by adjusting the number of job sizes to achieve the target latency [15]. However, coping with transient overloads is yet a critical issue to be investigated for such systems, due to their fast development and complex system stacks.

To optimize the job latency in data stream processing systems, two types of control knobs, i.e., adjusting the capacity and the loads, have been considered to cope with persistent and transient overloads. Scaling out the resources according to the loads is a costly but effective option [10] particularly for persistent overloads. Admission control by shedding data according to the capacity can effectively mitigate the transient overloads, either in a probabilistic random fashion [23] or for certain types of operators, such as joins [9]. In [19] a feedback controller is presented to control the queueing time and ultimately the job latency to match user-specified targets. Nonetheless, shedding data incurs not only financial loss but also users' dissatisfaction. Indeed, shedding and delaying workloads has been well discussed in the area of energy management [24] and network buffer management [25], showing that delaying workloads is challenging to optimize and strongly coupled with storage constraints.

Our controller Dslash combines the idea of resource augmenting and admission control by delaying the unprocessed data in memory and thus shedding a minimum amount of data. Moreover, Dslash explores the benefit of data locality inherent to batch streaming systems.

VII. Conclusion

In this paper, we propose Dslash, a latency-driven data controller, for batch streaming systems where data is analyzed in a real-time fashion by a MapReduce-like engine, i.e., Spark Streaming. In particular, we focus on the peak loads that overwhelm the system capacity, resulting in service unavailability

TABLE II: The efficiency of delaying controller: Shedding vs. shedding-and-delaying controller.

Query	Policy	Latency mean [s]	Latency p95^{th} [s]	Processing time [s]	Scheduling wait time [s]	Dropped records [M#]	Block delay [s]
I/O-intensive	$Dslash_{ps}$	42.03	60.61	22.26	1.99	365.51	10.09
	$Dslash_{psd}$	41.34	61.56	24.32	2.68	255.53	11.46
CPU-intensive	$Dslash_{ps}$	5.37	18.85	1.23	0.05	25.20	1.01
	$Dslash_{psd}$	4.15	10.62	1.43	0.09	13.10	1.22

and violations of the analysis latency requirements. Dslash features intelligent placing of data records into distributed memory, shedding of out-of-date records, and processing of the data in accordance with the application target latency and system capacity. By delaying the record processing and achieving high data locality, Dslash can make more efficient use of the computational capacity of the entire system. Extensive evaluation on Spark Streaming shows that Dslash achieves 100% service availability, adheres to the application latency target, and sheds only a small amount of data across a large number of query and load conditions, in contrast to vanilla Spark Streaming.

REFERENCES

[1] J. K.-S. Liew and G. Z. Wang, "Twitter Sentiment and IPO Performance: A Cross-Sectional Examination," *Available at SSRN 2567295*, 2015.

[2] T. Sakaki, M. Okazaki, and Y. Matsuo, "Tweet analysis for real-time event detection and earthquake reporting system development," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 919–931, 2013.

[3] R. Krikorian, "New Tweets per second record, and how!" https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, Accessed: 2015-12-16.

[4] "Slashdot Effect," https://en.wikipedia.org/wiki/Slashdot_effect, Accessed: 2015-07-31.

[5] "IBM InfoSphere Streams," http://www-03.ibm.com/software/products/en/infosphere-streams/, Accessed: 2015-07-31.

[6] "Apache Storm," http://storm.apache.org/, Accessed: 2015-07-31.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc of NSDI*, 2010, pp. 313–328.

[8] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *Proc of SOSP*, 2013, pp. 423–438.

[9] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *Proc of SIGMOD*, 2003, pp. 40–51.

[10] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Procof the SIGMOD*, 2013, pp. 725–736.

[11] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc of Eurosys*, 2010, pp. 265–278.

[12] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," in *Proc of INFOCOM*, 2013, pp. 1609–1617.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc of HotCloud*, 2010.

[14] "Apache Spark," http://spark.apache.org/, Accessed: 2015-07-31.

[15] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of SoCC*, 2014, pp. 16:1–16:13.

[16] "Apache Hadoop Distributed File System," http://hadoop.apache.org/, Accessed: 2015-07-31.

[17] "Tachyon," http://tachyon-project.org/, Accessed: 2015-07-31.

[18] Y. Tu, S. Liu, S. Prabhakar, and B. Yao, "Load shedding in stream databases: a control-based approach," in *Proc of VLDB*, 2006, pp. 787–798.

[19] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch, "Overload Management in Data Stream Processing Systems with Latency Guarantees," in *Proc the Feedback Computing*, 2012.

[20] C.-Y. Kao and L. Bo, "Simple stability criteria for systems with time-varying delays," *Automatica*, vol. 40, no. 8, pp. 1429–1434, 2004.

[21] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *ICDE Workshop*, 2010, pp. 41–51.

[22] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *Proc of SOSP*, 2013, pp. 423–438.

[23] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proc of VLDB*, 2003, pp. 309–320.

[24] C. Wang, B. Urgaonkar, Q. Wang, and G. Kesidis, "A hierarchical demand response framework for data center power cost optimization under real-world electricity pricing," in *Proc of MASCOTS*, 2013, pp. 305–314.

[25] A. Vishwanath, V. Sivaraman, and G. N. Rouskas, "Anomalous loss performance for mixed real-time and TCP traffic in routers with very small buffers," *IEEE/ACM Trans. Netw.*, vol. 19, no. 4, pp. 933–946, 2011.