IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS



# Online Scheduling and Interference Alleviation for Low-latency, High-throughput Processing of Data Streams

Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara, *Members, IEEE*

**Abstract**—Data Streams occur naturally in several observational settings and often need to be processed with a low latency. Streams pose unique challenges: they have no preset lifetimes, the traffic on these streams may be bursty, and data arrival rates on these streams can be quite high. Furthermore, stream processing computations are generally stateful where the outcome of processing a data stream packet depends on the state that builds up within the computation over multiple, successive rounds of execution. As the number of streams increases, stream processing computations need to be orchestrated over a collection of machines. Achieving timeliness and high throughput in such settings is a challenge. Optimal scheduling of stream processing computations is an instance of the resource constrained scheduling problem, and depending on the precise formulation of the problem can be characterized as either NP-Complete or NP-Hard. We have designed an algorithm for online scheduling of stream processing computations. Our algorithm focuses on reducing interference that adversely impacts performance of stream processing computations. Our measure of interference is based on stream packet arrivals at a particular machine, the accompanying resource utilization encompassing CPU, memory and network utilization, and the resource utilization at machines comprising the cluster. Our algorithm performs continuous, incremental detection of interference experienced by computations and performing migrations to alleviate them.

**Index Terms**—low-latency stream processing; online scheduling; data intensive computing

## 1 Introduction

Falling costs, improved network connectivity, and the ability to report measurements at increased precision have led to a proliferation of sensors. These devices generate *data streams*, a set of correlated packets, reporting their measurements. The nature of these streams poses unique processing challenges. Streams may have no preset lifetimes or data production rates and can be voluminous. Furthermore, stream processing computations are long running since processing is often tied to the stream lifetimes. Computations also need to retain state and build context that is critical when processing individual stream packets. In this study, the computations that operate on these streams must take into account what has happened in the past. This means the computations are *stateful*, i.e., they build and retain state during execution, and the outcome of processing any stream packet depends on the built up state.

Stateful computations are commonly used in several domains; for example, in health stream processing. In the case of health sensors, vital signs data are reported as physiological streams. These vital sign include blood pressure, thorax extensions, electroencephalograms (EEGs), electrocardiograms (ECGs or EKGs), and pulse oximetry data among others. Stateful computations in health stream processing allow performing trend analysis on the physiological streams prior to determining if an alert needs to be issued. In the case of ECG processing, a stateful computation that retains information about past, recurring, and systemic abnormalities in the ECG waveform may identify an impending health emergency.

The per-packet processing overhead for stream processing may be in the order of a few milliseconds, but packets continually arrive at high rates and must be processed with a low latency. The devices may be configured to report measurements at increased frequencies. For example, in health stream processing settings, this is done to enhance patient surveillance to improve clinical deterioration detection capabilities. If not managed carefully, processing delays can become insidious, resulting in queue buildups at a processing node alongside subsequent buffer overflows, and exhausted memory conditions. Given the data volumes involved, processing must be orchestrated over a collection of machines.

Scheduling processing tasks at scale over a collection of machines is a challenge. Scheduling a single stream processing task is easy; so is scheduling multiple tasks if there are an unlimited number of available machines. For the problem we consider, we have a limited number of machines and a large number of stream processing tasks, with each stream packet representing a unit of work that needs to be completed. The number of tasks will significantly outnumber the number of machines available. This is an instance of the *resource-constrained scheduling problem* – given a set of tasks, a set of machines, and a performance measure, the objective is to assign tasks to machines such that the desired performance measure is maximized. Additional constraints may be specified on the performance measure, such as a bound on the completion time. Depending on the precise formulation of the problem, some have characterized this as NP-Complete and some as NP-Hard [1], [2], [3], [4]. There is no known algorithm for finding an optimal solution in polynomial time; when the problem size grows with increases in machines or tasks, or when additional constraints are imposed, finding the optimal solution is computationally intractable. Stream scheduling must be online, encompassing continuous, incremental scheduling decisions that account for changes in the stream packet arrivals and cluster resource utilizations.

**Challenges:** There are several challenges in accomplishing low-latency, high-throughput processing of streams at scale.

1) *Stateful computations:* Stateless computations are easier to scale, since they can simply be replicated on multiple machines with stream packets being processed in a round-robin fashion and in parallel. In stateful computations, the outcome depends on the built up state.
2) *Stream computations outnumber machines:* The number of stream computations will be multiple orders of magnitude larger than the number of machines available for data processing, especially in Internet-of-Things settings such as smart cities, necessitating the use of a horizontally scalable computing platform [5], [6]. This requires multiple stream

Digital Object Identifier 10.1109/TPDS.2017.2723403

processing computations to be interleaved on the same machine. This brings to the fore the notion of interference either from collocated computations within the same process (internal/endogenous interference) or from collocated external processes on the same machine (external/exogenous interference). Interference leads to increased resource contentions that can preclude high throughput processing. In extreme cases, failing to keep pace with data rates results in queue buildups and processing delays.
3) *Initial placement of computations may become ineffective:* This is due to variability in the resource utilization and the stream processing workload. Variability in the resource utilization profile may be due to external interference and maintenance activities [7]. The stream processing environment may itself contribute to this flux via the removal or addition of stream processing tasks, data induced load imbalance, etc.
4) *Minimizing resource utilization imbalances:* Stream processing over a collection of machines should not introduce imbalances. Specifically, we wish to avoid situations where some machines are overloaded, whereas others are underutilized. Such imbalances may lead to increased interference between computations and will result in lower throughput since a heavily used machine experiences higher context switches, memory pressure, and increased network contention.

**Research Questions:** Achieving low-latency, high-throughput processing of streams at scale requires us to account for stream packet arrivals, resource utilizations, tracking interference between computations, and leveraging these to inform scheduling decisions. Specific research questions that we explore include:

**RQ-1:** *How can we account for stream packet arrivals and their accompanying resource footprints to ensure high-throughput processing?* How can we achieve this while ensuring low latencies per packet? (§ 4.1.1, § 4.1.2)

**RQ-2:** *How can we effectively account for both internal and external interference that impact performance – both latency and throughput – of stream processing computations? Specifically, how can we quantify this interference, and how can these interference scores be used to inform online scheduling of stream processing computations?* (§ 4.1.2, § 4.1.3)

**RQ-3:** *How can we ensure system stability that minimizes oscillations and cascading migrations of computations across machines within the cluster while minimizing utilization imbalances?* Since online scheduling involves migration of computations to alleviate performance bottlenecks, care must be taken so that these migrations do not induce performance problems. We also wish to avoid resource utilization skews where some machines are heavily utilized while others are idling. (§ 4.2)

**Approach Summary:** In this paper we describe our algorithms and an accompanying implementation for online stream scheduling at scale with low latency per-packet while achieving high throughput. Our methodology focuses on reducing interference between stream processing computations in the presence of variability in processing workloads and system conditions. This is achieved through a series of proactive, continuous, and incremental scheduling decisions where computations are migrated to machines with less interference. These migrations help reduce resource utilization imbalances within the cluster while alleviating performance hotspots, both of which improve performance. We have implemented these algorithms in the context of our stream processing engine, Neptune [8].

At the core of our online scheduling algorithm is a data structure called ***prediction rings*** that encapsulates a set of footprint vectors. Prediction rings are used to track the expected resource utilization of a stream computation in the future. Each element in the vector represents the expected resource utilization of a stream computation for a particular time granularity. Resource utilizations in the near future are captured at a fine-grained level while the resource utilizations further out into the future are captured at a coarse-grained level. Prediction rings are populated based on expected stream arrival rates forecast using time-series analysis.

To inform online scheduling of computations, we introduce the notion of an interference score. The interference score is a normalized score that quantifies the expected interference for a stream computation when placed on a particular machine. It is calculated using the prediction ring of a particular computation and the prediction rings of the collocated computations accounting for both the internal and external interference. A computation will be migrated to another machine if there is a significant reduction in interference at the new location compared to its current location. Our migration protocol handles both stateful and stateless computations and ensures the correctness of a stream processing job. The methodology includes mechanisms to counteract oscillations, cascading migrations, and frequent inefficient migrations.

We profile the efficiency of our algorithm based on extensive benchmarks with health stream computations for thorax and ECG processing. Our evaluation metrics include throughput, $99^{th}$ percentile of latency, variance in latency, and resource utilization imbalance within the cluster.

**Paper Contributions:** Contributions of our methodology include the following:

1) Proactive circumvention of internal and external interference by accounting for variability in resource utilization and stream processing workloads.
2) Our prediction rings data structure is compact, memory-resident, and effectively captures data arrivals and the accompanying resource utilization patterns for a computation. These prediction rings can be aggregated to generate compound footprints encompassing the collection of computations at a machine.
3) Prediction rings and interference score calculations allow us to effectively alleviate interference for computations by identifying the most suitable machine to migrate impacted computations to.
4) The online stream scheduling algorithm continually tracks resource utilizations at machines and ensures targeted, incremental, and proactive alleviation of performance hotspots via migrations.
5) Our algorithm minimizes resource utilization imbalances. The processing load is dispersed such that each machine is hosting computations that are less likely to interfere with each other. This prevents performance hotspots and allows us to achieve high-throughput processing of streams.

**Paper Organization:** The remainder of this paper is organized as follows. In section 2, we motivate the necessity of an online scheduling scheme like ours for stream processing. Methodology of our work is presented in section 3. Our online scheduling algorithm is discussed in section 4. The results of our empirical evaluation are presented in section 5. Related work is reviewed in section 6. Section 7 outlines conclusions and future work.

## 2 Background And Problem Statement

A stream processing job is usually modeled as a directed acyclic graph of stream operators. A stream operator continuously transforms the data items in a data stream [9]. A stream operator can either be a stream ingestion operator or a stream computation. There may be one or more stream ingestion operators that ingest data streams into the system from external sources. Stream computations implement parts of the stream processing logic. Stream operators are connected through streams that form the edges of the stream processing graph. Sink operators are a special type of stream computation that does not have any outgoing streams; streams flowing into sink operators are called terminal streams [10]. Sink operators make the results of a stream processing job available to external systems such as a visualization system, a persistent storage system, or even another stream processing job.

During the deployment of a stream processing job, multiple instances of a stream operator may be deployed to exploit the parallelism provided by multicore processor architectures as well as distributed computing clusters. This enables data-parallel processing of streams [9]. Streams between operators need to be partitioned to ensure that each instance receives a proportional share of the input streams. The choice of the stream partitioning function depends on the nature of the processing performed at each operator and how the use case is mapped into a stream processing graph in general. Each of these operator instances is executed as a stream processing task.

These tasks need to be deployed within a set of distributed machines for execution where the number of tasks are two orders of magnitude greater. In addition to satisfying the resource matching requirement, the placement plan is further governed by constrains such as collocation and quality of service requirements such as upper bounds on response times. This problem is considered an NP-Hard/NP-Complete problem [1], [2], [3], [4]. Generating the initial placement of stream processing tasks in a distributed setup is a well-studied problem [10], [11], [12], [13], [14]. A naive approach would be to distribute the tasks among the available processes in a round-robin manner. For instance, the default scheduler of Apache Storm [15] follows this approach. A heuristic based scheme would be to estimate the resource requirements of each of the tasks and use a variation of the bin-packing problem to generate the initial placement of the tasks [11]. For instance, Apache Storm's ResourceAwareScheduler [16] follows this approach by allowing users to augment the stream processing job specification with resource requirements for stream operators. Existing work has also relied on the properties of the stream processing graph itself, such as the communication patterns between tasks, in order to derive a more efficient placement plan. One example is to collocate tasks with a higher amount of pairwise communication within a single process [12].

Regardless of the scheme used to generate the initial placement plan, placements are bound to become inefficient over time. This is due to the inherent variability in system conditions and stream processing workloads. The system conditions can vary due to activities of the other applications sharing the resources (e.g., CPU, memory, network bandwidth, etc.), contending for global resources (e.g., network switches), power limits (e.g., CPUs mitigating thermal effects by throttling down), energy management (e.g., power saving modes), and periodic maintenance activities (e.g., periodic log compaction, reindexing of distributed file systems) [7]. Changes in the stream processing workloads occur due to the deployment and termination of stream processing jobs and operators switching between active and dormant phases due to data availability or the satisfiability of other conditions. Data induced load imbalances can also create fluctuations in the computation workloads [7]. For instance, a particular stream partition may be accounting for

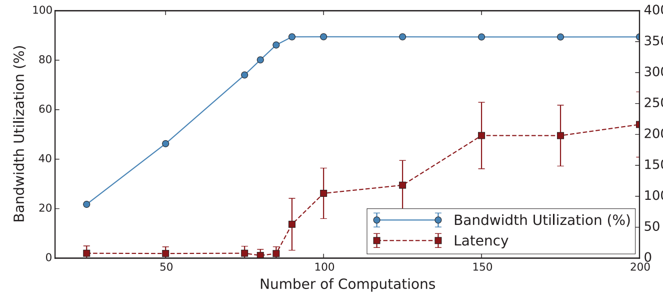



Fig. 1. Bandwidth consumption and processing latency observed at a single machine against the number of collocated computations.

a majority of the stream over time even though the partitions were initially assumed to be balanced (e.g., a monitoring device attached to a patient in a critical condition is configured to take measurements more frequently or flash-crowds responding to certain events via social networks). Additional processing may be triggered at an operator based on characteristics of the data items within a stream (e.g., triggering an alarm upon detecting an anomaly). These are categorized as medium and long-term load fluctuations as opposed to short-term load fluctuations that happen due to the event-based aperiodic nature of stream sources and the transient inconsistencies of the performance of the networking infrastructure [14].

The changes in the workload and system conditions influence a stream computation's level of contention for shared resources with other collocated computations and external processes. Contention for shared resources causes an *interference* in the execution of a stream computation. When caused by collocated computations within the stream processing engine, it is called *internal interference*. When caused by collocated external processes on the same machine, it is called *external interference*. A stream processing computation will experience both internal and external interference in varying degrees throughout its lifetime. Though the motivation behind most heuristic initial placement schemes is to ensure minimal interference across the cluster, it is hard to maintain this property over time due to changes in workload and system conditions as discussed above. Increased contention for shared resources beyond their available capacity can degrade the performance of a stream processing computation, both throughput and latency, because these computations have to wait longer for their share of the shared resource and/or receive a reduced share of resources. Variability in the workload and system conditions can affect individual machines in varying degrees causing resource utilization imbalances that result in different levels of interferences experienced by computations. Computations placed on underutilized machines may experience lower interference levels whereas computations on overutilized machines may experience higher interference levels.

To understand the impact of interference on the performance of a stream processing system, we measured the cumulative performance of a set of stream processing computations when subjected to varying degrees of internal interference. Thorax extension processing computations, explained in § 5.2.1, were used for this experiment. The machines running the stream ingestion operators were adequately provisioned to ensure that they did not become a bottleneck during this scalability test. Stream ingestion and acknowledgment operators were dispersed over a group of 10 machines, whereas the thorax extension processing computations were all collocated on a single machine. The number of concurrent stream processing jobs was increased, which in turn increased the number of collocated thorax extension processing computations; that produced increased internal interference. Figure 1 depicts the bandwidth utilization and the processing latency observed at the machine which hosted the thorax extension processing computations. Incoming traffic that encapsulates thorax readings dominates the bandwidth consumption, which is also a measure of cumulative throughput. Even though the cumulative throughput is expected to increase with the number of stream processing computations, it does not continue to do so beyond a certain point. This is because the node has reached the maximum possible bandwidth utilization at this point; any additional computation placed on the node will interfere with collocated computations with respect to network bandwidth thereby degrading throughput at individual computations. Average end-to-end latency of the cluster also increased mainly due to increased data transfer times between data ingestion operators and data processing computations. Even though the network bandwidth was the first resource to exhaust its capacity for this particular use case, it is possible that any other resource or a combination of resources can become the limiting factor for other use cases. Horizontal scaling [9], [17] and load shedding [9] are two well-studied solutions that are often used in such scenarios. But we argue we must attempt to effectively utilize the available resources *before* provisioning more resources (horizontal scaling) or enabling graceful degradation (load shedding).

This necessitates an online scheduling scheme that is able to continuously adjust the placement of tasks based on the changing workload and system conditions. Such an online scheduling scheme can alleviate hotspots in the cluster and reduce the imbalances in resource utilization. Accomplishing hotspot alleviation and imbalance reduction will improve the performance (throughput and latency) of the stream processing system and reduce the performance variability, especially for latency related metrics. We have implemented such an online scheduling scheme to reduce the interference experienced by computations. Our online scheduling algorithm continuously and incrementally migrates computations to reduce the interference experienced by computations. It ends up moving computations away from overutilized nodes towards the underutilized nodes that have spare capacity to host additional computations while also alleviating hotspots within the cluster. Our algorithm complements any existing initial placement algorithms and focuses on the dynamic online scheduling of stream processing jobs under varying loads and system conditions. Even though, our dynamic online scheduling scheme preserves the necessary QoS guarantees through improved resource utilization, the system may need to scale out if the workload demands more system capacity. Also for certain applications, load shedding is a viable alternative where the input streams are sampled. Even though this impacts the accuracy of the results (hence considered unsafe), it is acceptable for certain types of applications [9]. If the online scheduling algorithm repeatedly fails to reduce the interference experienced by a computation, either of those schemes can be triggered.

## 3 System Overview

We have validated our methodology (§ 4) for online scheduling in the context of our stream processing system, Neptune [8]. Here, we discuss the key components of the system, their responsibilities, and interactions with each other.

A Neptune cluster comprises a set of *worker nodes* running on a cluster of interconnected physical or virtual machines.

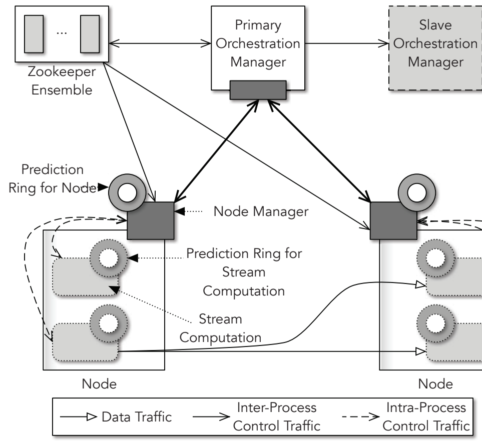



Fig. 2. System architecture depicting key components and interactions.

Each worker node is an independent JVM process that concurrently executes a set of stream processing computations. Each worker has a *node manager* that supervises execution of computations assigned to the node. Initial placement and online scheduling of tasks in the cluster is managed by a separate process called the *orchestration manager*. These entities are depicted in Figure 2. Node managers send periodic status updates to the orchestration manager incorporating both node and task status using a data structure called a *Prediction Ring*. At the orchestration manager, these periodic updates are used to infer the global state that informs scheduling decisions to migrate tasks and mitigate resource imbalances. The orchestration manager coordinates a migration (as explained in § 4.2) by communicating with node managers at the new node, current node, and upstream nodes to redirect data streams through the exchange of control messages. The *control plane* is logically independent from the *data plane* that carries data streams that are input to stream processing computations. This helps to reduce the end-to-end latency involved in processing control messages without being affected by queuing delays and backpressure if the same channel is used for both types of traffic.

To ensure failure resiliency, a secondary instance of the orchestration manager is run in parallel in the active replication mode. The secondary replica is actively synchronized with computation placement information after deploying new jobs or at the end of a migration in addition to a job's physical deployment plan: tasks and the data flow between tasks. The remainder of the state can be reconstructed through periodic updates from node managers within a time interval less than or equal to the periodicity of state update messages after the secondary orchestration manager has taken over the role of the primary. Neptune uses Zookeeper [18] for metadata management. We leveraged the same Zookeeper ensemble for leader election of orchestration manager nodes to appoint and discover the primary orchestration manager.

Using its global knowledge of the entire system, the centralized orchestration manager can make efficient scheduling decisions. Alternatively, it is possible to use a more decentralized approach such as, peer-to-peer or cluster-to-cluster, where nodes will arrange themselves as a virtual network [19]. By making scheduling decisions based on local knowledge encompassing a subset of the nodes, this provides a more scalable and failure resilient model at the expense of efficient resource utilization. Such an approach facilitates work-stealing [20] as opposed to the work-pushing approach employed by our methodology, which is more stable and introduces lower communication overhead.

## 4 Online Scheduling Algorithm

Our algorithm closely resembles the MAPE loop used in autonomous systems, which is widely adapted for implementing elastic and dynamic systems [21]. It comprises four phases: monitoring (M), analysis (A), planning (P) and execution (E). We use *prediction rings* during analysis and planning phases.

### 4.1 Prediction Rings

Prediction rings track data stream arrivals for a given stream processing computation. The data structure is then used to track and predict a computation's expected resource utilization. We use prediction rings to compute an *interference score* that quantifies the impact of placing an additional stream computation alongside other collocated computations on a machine.

#### *4.1.1 Data Structure*

The prediction rings data structure consists of multiple footprint vector "rings", each implemented as a circular buffer. Each element of a ring represents the expected resource utilization during a given discrete time window. The value stored in each time window can vary by the goals of the application, provided it is some metric indicating the amount of resources required by the computation. For instance, the rings may be biased towards memory utilization, penalizing higher memory consumption. In this study the prediction rings are biased towards tracking CPU and network bandwidth consumption.

The number of rings and the resolution of the time windows within each ring are configurable. Having a multiring data structure allows us to capture arrival patterns (and expected resource utilization) at both fine and coarse-grained levels. Each ring radiating outwards represents progressively increasing time frames; the ring is bigger, with larger sectors each of which is at a coarser grained resolution. The innermost rings capture expected packet arrival rates at fine-grained resolutions. During an update to the prediction ring, the current window pointer is set to the current time, and each following clockwise window indicates the expected utilization at an increasingly distant future time. Furthermore, moving from the inner rings to the outer rings represents moving further out into the future. As such, each ring has a static offset equal to the total time capacity of the preceding inner rings. The window offset for any additional outer rings begins immediately after the last offset of the preceding inner ring. As wall time progresses, the current window pointer advances to a future clockwise window. Previous windows become invalid and are filled with a new prediction for the far future.

A conceptual view of a prediction ring with 3 rings is depicted in Figure 3. The innermost ring contains 16 $1250ms$ windows accounting for the next $20s$ from the current time $t'$. Similarly the middle ring accounts for the time period of [$t' + 20s$, $t' + 41s$) using 12 windows of $1750ms$ resolution. In our reference implementation we have used a prediction ring with 3 rings. Each ring from innermost ring to the outermost ring contained 30 windows with window lengths of $1s$, $2s$ and $3s$ respectively. Together, these three rings account for the next 3 minutes from the current time.

Prediction rings are designed to be compared against each other. Furthermore, the data structure is amenable to aggregation; i.e., we can take a collection of prediction rings and aggregate them into a single combined footprint vector. This is done by summing the window contents of each individual ring. The combined vector can then be part of a pairwise interference calculation with another computation, eliminating the need to individually check for interference with each existing computation. During aggregation operations, prediction rings are aligned with each other by shifting their current window pointers which indicates the time each prediction ring was last updated to the current timestamp.

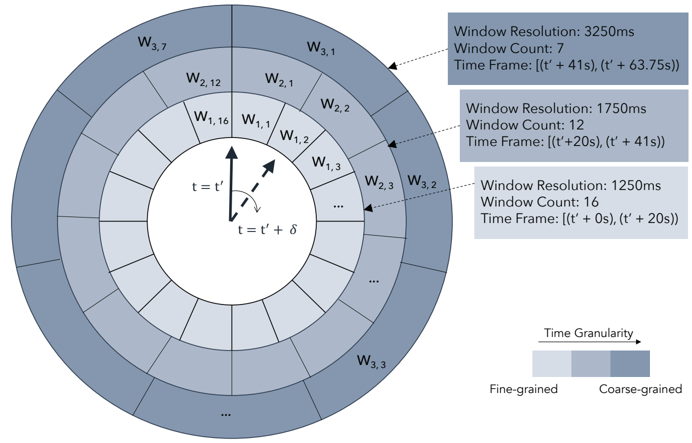



Fig. 3. A conceptual view of a prediction ring. The ring tracks how many packets are expected in a given time window. The arrow points at the current window. Clockwise windows represent the future. The number of rings as well as the resolution is configurable.

#### *4.1.2 Populating Prediction Rings*

A prediction ring is updated periodically to reflect operating conditions. These updates invalidate past windows. It is also possible that current values of future windows have become obsolete due to changes in the workload and system conditions. Updates to a prediction ring are performed in two steps:

1) Initial value assignment with predicted message arrivals
2) Projecting the normalized resource consumption using the predicted input rates

For the first step, we use exponential smoothing, a time-series prediction model, to predict the message arrival rate for a computation. Exponential smoothing relies on the entire set of past observations but assigns exponentially decaying weights for older values. This is different from other moving average models where an equal weight is assigned to every past observation [21]. We use the triple exponential smoothing method that has a seasonal component ($\beta$) in addition to a smoothing constant ($\alpha$) and a trend component ($\gamma$) [22]. More specifically, we use the Holt-Winters method of exponential smoothing for predicting the message arrival rates for a computation based on prior arrival rates. The average message arrival rate calculated over a sliding window is used as the input for building the time series model because it eliminates short-term variations in arrival rates arising due to shared, overloaded network resources and other optimizations such as application-level buffering.

There are two challenges when using the time series prediction model mentioned above. The triple exponential smoothing model requires data gathered from at least two seasons in order to produce predictions with higher accuracy. This creates a cold start problem due to unavailability of observations at the beginning of the execution of a stream computation. Possible solutions to this problem would be to feed the model with observations gathered offline or to collect data during the

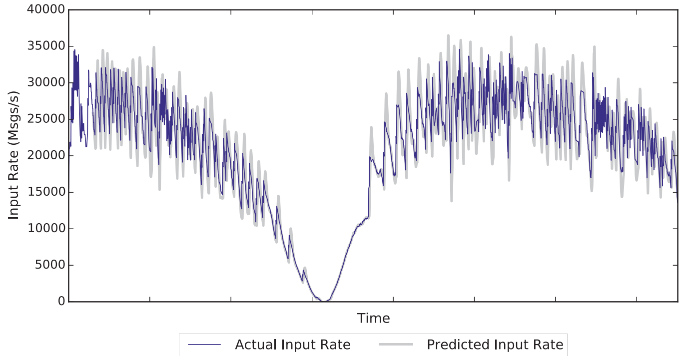



Fig. 4. Predicted message arrival rate using Holt-Winters model vs. actual message arrival rate for a single computation.

execution of a computation and start predictions only when a sufficient number of observations are collected. Using data gathered offline can be error prone because the input rates observed by a computation (which may be different from the rate at which its upstream computation emits data) is heavily dependent on its operating environment. So we opted for the latter approach to address the cold start problem. Until enough observations are recorded for the time series prediction model, we use a simple moving average model over a sliding window of past observations to approximate the next value. The second challenge when using the Holt-Winters model is the necessity to adjust the smoothing parameters ($\alpha$, $\gamma$ and $\beta$) over time if there is a significant change in the environment that invalidates the current model [23]. Due to changes in the system conditions (e.g., changes in the number of collocated computations), the time-series model may become inefficient. We address this issue by recalculating the smoothing parameters using only the latest observations if the prediction error exceeds a certain threshold for a consecutive number of prediction cycles. Figure 4 shows the predicted value vs. actual value of message arrival rates for a single computation. The message rate of the simulated input stream follows a recurring pattern that is effectively captured by our time-series model.

Our methodology does not preclude the use of other time-series prediction schemes such as ARIMA [24], artificial neural networks or genetic algorithms [23]. Artificial neural networks and genetic algorithms provide very accurate predictions in highly dynamic systems but require prolonged training times. We chose exponential smoothing because it satisfies several of our requirements: fast training, accuracy, compactness and quick evaluations.

The predicted input rates are then transformed to reflect the predicted resource utilization of the computation. A prediction ring will undergo a series of manipulations during this process to ensure that the calculated resource utilization values are normalized. Normalizing computations and host machines is necessary for fair comparisons to find a better location for a computation. We now discuss the rationale behind the multipliers used for normalizing prediction rings of computations.

- *Processing time per message:* This is a measure of a computations CPU requirements. This also accounts for the heterogeneity of computations and data induced additional load.
- *Message size:* This is a measure of the computation's bandwidth requirements when used together with its input rate.

Once the prediction rings of individual computations are transformed, they are summed to generate the prediction ring for the resource. Then a series of multiplications are performed on the resulting aggregated ring in order to reflect the resource utilization of the node.

- *Normalized load average of the host machine:* This is calculated by dividing the load average of the last minute by the number of CPU cores as a measure of how saturated the host is.
- *Excess bandwidth utilization:* This reflects the bandwidth consumed in excess of the preferred upper limit.
- *1 - fraction of load average caused by the node:* This is a measure of the CPU-wise external interference on the process.
- *1 - fraction of bandwidth utilization incurred by the node:* This is a measure of the network bandwidth-wise external interference on the process.

Normalizing the prediction ring of the machine is more involved than normalizing prediction rings of individual computations. The original prediction ring of the machine, calculated by summing up the prediction rings of the individual components, is preserved for aggregations with prediction rings of computations in order to calculate interference scores (as explained in the next section). The resulting prediction ring from the aggregation operation is then normalized using multipliers discussed above. These multipliers, captured using various monitoring tools, are valid only for a short duration of time because they are dependent on the stream rates and the load profile of external processes. So instead of multiplying the values in windows of every ring, the multiplication operations are applied only to the windows of the innermost ring.

TABLE 1
Notation used in interference score calculation algorithm.

| | |
|---|---|
| $score$ | Interference score |
| $ring$ | Current ring number |
| $ringCount$ | Number of rings in the prediction ring |
| $dist$ | Distance to window pointed by window pointer |
| $ringOffset$ | Offset to first window in a ring |
| $p$ | Window pointer |
| $ringSz$ | Number of windows in a ring |
| $ru$ | Resource usage score of a window in the nodes's prediction ring without computation |
| $rw$ | The value in the window in a prediction ring of the node |
| $rs$ | Weighted resource usage Score of a window in the nodes's prediction ring without computation |
| $n$ | Interference score difference amplifier |
| $cpuFrac$ | Fraction of the available processor cores |
| $bwFrac$ | Fraction of the available bandwidth |
| $ringRes$ | Resolution of the window |
| $cu$ | Resource usage score of a window in the node's prediction ring with computation |
| $cw$ | The value in a window in the computation's prediction ring |
| $cs$ | Weighted resource usage score of a window in the node's prediction ring with computation |
| $totalDist$ | Total length of time represented by the entire prediction ring |

### 4.1.3 Using Prediction Rings to Quantify Interference

We use the notion of *interference scores* to inform migration decisions. The interference score is a floating point value that indicates the degree of interference between computations. The larger the score, the greater the degree of interference. There are two main properties that we wanted in our interference score.

- *Property-1: Identifying how soon an interference is likely to occur* - We accomplish this by assigning less weight to more distant interferences. This counteracts prediction errors too far out into the future.
- *Property-2: Ability to reflect the load on a given node* - This is to ensure that computations contribute only slightly to the score if the node is lightly loaded for a given window and contribute much more significantly if the load exceeds available resource capacity.

Fig. 5 depicts pseudocode for our interference score algorithm. The notation used in the algorithm is defined in Table 1. The usage score for each window is computed with and without the existence of the computation being considered for migration. In order to calculate the prediction ring of node without a computation, we subtract the computation's prediction ring from the node's prediction ring. We use the non-normalized version of node's prediction ring in this subtraction. The difference is then normalized as explained in § 4.1.2. The difference between these weighted usages scores is added to the final interference score so that the score effectively reports only the impact of placing the new computation on the resource *and* is not unduly weighted by previously placed computations. This also means that the interference score operation is asymmetric. Checking how much a single computation interferes with all computations on a resource will produce a different score than checking how much the many computations interfere with the single computation, which is logical behavior for an interference scoring algorithm. The parameters $cpuFrac$ and $bwFrac$ in the interference score calculation take into account the slack we set aside to accommodate bursty traffic. By setting these upper resource consumption thresholds, we are trying to achieve both an efficient and a safe resource consumption across the cluster similar to most dynamic systems that support workload migrations and horizontal scaling.

The $dist$ indicates how many milliseconds into the future the window at position $p$ is, growing as more windows are processed. Dividing by the total millisecond capacity of all rings, $totalDist$, allows the result to be scaled down based on the distance into the future. For each window in the prediction rings, the utilization reported by that window is scaled and contributes to the final score, thus satisfying *Property-1* that is expected in the calculated interference score.

*Property-2* is achieved by exponentiating the score components with an integer greater than 1 – this exponent is called the interference score difference amplifier ($n$). This allows the score contribution to grow quickly as more computations are assigned to a time window. This, in turn, differentiates between placements resulting in collisions involving two computations versus collisions involving three computations, with fewer collisions being more desirable. Another reason arises from the tendency for computations with a high arrival rates to unnecessarily produce high interference scores, even if few collisions occur.Exponentiating allows the many windows without collisions to contribute only slightly, while allowing the occasional collision to contribute appropriately based on the severity of the collision.

$score \leftarrow 0$
**for** $ring = 0$ to $ringCount$ **do**
  $dist \leftarrow$*ringOffset*
  **for** $p = 0$ to $ringSz$ **do**
    $ru \leftarrow .5 \times rw[p-1] + rw[p] + .5 \times rw[p+1]$
    $rs \leftarrow ru^n/(cpuFrac * bwFrac)^n/ringRes$
    $cu \leftarrow ru + .5 \times cw[p-1] + cw[p] + .5 \times cw[p+1]$
    $cs \leftarrow cu^n/(cpuFrac * bwFrac)^n/ringRes$
    $score \leftarrow score + (cs - rs) \times (1 - dist/totalDist)$
    $dist \leftarrow dist + ringRes$
  **end for**
**end for**
**return** $score$

Fig. 5. Pseudocode for computing interference scores.

## 4.2 Migrating Computations Using Interference Scores

Prediction rings and interference scores are eventually used for online scheduling where computations are migrated to nodes where they are subject to less interference and improved performance. The steps involved in the migration of a computation are depicted in Figure 6 in chronological order. There are three periodic tasks every node manager executes:

1) Update prediction rings of individual computations
2) Calculate the prediction ring for the node and send it to the orchestration manager
3) Calculate interference scores for individual computations

During the third task, the computation that records the highest interference against the rest of the computations — exceeding a predefined threshold in consecutive evaluations — is chosen for the next migration attempt. If there is such a computation, the node manager sends a migration request to the orchestration manager. Migration requests are piggybacked with the periodic prediction ring update messages sent by the node manager. A migration request contains the prediction ring of the computation chosen for migration and the interference score it recorded against the rest of the computations[$I_c$].

Upon receiving a migration request, the orchestration manager identifies the best possible node for the computation. The orchestration manager calculates interference scores individually for each node (except for its current host node) using their prediction rings and the prediction ring of the computation. If the minimum resulting interference score ($I_n$) is significantly lower than the interference score reported at its current location ($I_c$), then a migration is initiated. Otherwise a rejection response to the migration attempt is sent back to the current node; this can also be used to inform provisioning of new nodes or horizontal scaling in a cloud setting. To decide whether to initiate the migration, the percentage reduction in interference for the impacted computation ($\Delta I$) is calculated as follows.

$$\Delta I = \frac{(I_c - I_n)}{I_c} \times 100\%$$

If the percentage reduction is greater than a configurable threshold, a migration is initiated. For example, the default threshold in our implementation was set to 5%.

The first step of the migration is to deploy an empty instance of the computation, i.e., without any state, in the new location. Once the deployment of the empty instance is complete, the upstream computation needs to be paused until the current state of the computation is successfully migrated to the newly deployed instance. Instead of completely pausing the entire upstream computation, it temporarily stops emitting messages to the stream connected to the computation being migrated while continuing to emit messages to other streams. The messages destined to the paused stream are buffered in memory. If the memory consumed by the buffered messages exceeds a certain threshold, the upstream computation completely pauses to ensure that the performance of the other collocated computations are not affected due to increased garbage collection activities. Pausing the stream from the upstream computation ensures

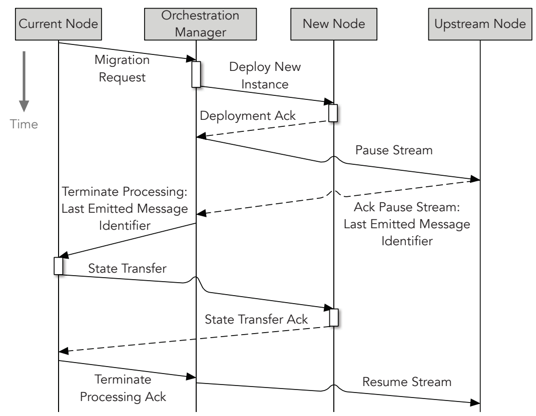



Fig. 6. Sequence diagram depicting a computation migration.

*safety*, in other words preserving the correctness of the stream processing job during the migration [9]. Pausing is required because we have used point-to-point communications in Neptune to optimize for high throughput settings [8]. Switching to Neptune's publisher-subscriber communication mode can completely eliminate the need for pausing of the stream albeit at reduced throughput. The pub-sub mode will make the stream packet flow asynchronous and improve the mobility of stream computations as advocated in [25], [26]. After pausing, in its acknowledgment to the orchestration manager, the upstream computation includes the sequence number of the last message emitted to the paused stream. This information is relayed back to the stream computation by the orchestration manager. The stream computation will wait until it has completed processing this particular message before moving to the next phase ensuring that no messages are left unprocessed during the migration. Next, the computation being migrated will serialize its state and send it over to the fresh instance of the computation placed at the new location. Upon processing this message, the new instance of the computation restores its state. Once the new instance is ready to process messages, the upstream computation will first play the buffered messages and resume its regular operations. The computation at the old location is then terminated permanently.

#### 4.2.1 *Ensuring System Stability*

Migrations incur a significant overhead mainly because they interrupt the regular operation of the upstream computations and pause the processing of a subgraph for the duration of the migration. The throughput of the stream processing job drops for a while, and the latency will show a sudden spike when processing the buffered messages. Hence triggering a migration should be done only if the expected performance-gains outweigh this temporary degradation in performance. There are some built-in measures in our implementation to reduce unnecessary migrations.

Using a *dynamic threshold function* at the orchestration manager is one such measure. As discussed before, a migration is triggered only if the expected percentage reduction of interference ($\Delta I$) is greater than a certain threshold. This threshold value is dynamically adjusted based on the state of the system: if there is a significant variation in resource utilization within the cluster, it is set to a lower value and vice versa. This dynamic threshold function encourages migrations when there is a significant resource imbalance in the cluster, even if there is a small improvement in interference. In our implementation, we have used a simple step function that sets different threshold values based on the variance in the interference scores reported for a computation (targeted for migration) against every node.

Another stability measure that we leverage is to force a *cooling down period* [21] on nodes after they have participated in a migration, either as the source or destination. During this period, such nodes are not allowed to either trigger any migrations nor are they considered a candidate to receive computations from other nodes. The cooling down period also provides time for the monitoring system to capture changes that occurred during the previous migration allowing time series models to stabilize and recalibrate if necessary. This also reduces the number of migrations triggered due to unreliable prediction rings.

Since the computation with the highest interference score at a node is picked for the next migration attempt, the orchestration manager may not be able to find a better node. The migration request will be rejected and possibly will continue to get rejected in successive attempts. Such successive rejections will prevent the node from making any progress towards alleviating the hotspot. As a countermeasure, if a migration request for a particular computation is rejected then it will not be scheduled for migration for some time. By moving computations with less interference to better alternative locations, it may reduce the interference at the current node. If it does not reduce the interference of the original computation as expected, then it is an indication that the system either needs to be scaled out horizontally or start load shedding.

## 5 Empirical Evaluation

### 5.1 Experimental Setup

The benchmarks reported here were performed in a cluster comprising 54 physical machines connected over a 1 Gbps LAN. Each machine is a HP-DL60 server (Xeon E5-2620 CPU and 16 GB RAM) running Fedora 23 and Oracle Java 1.8.0_65. Primary and secondary instances of the orchestration manager were running on dedicated machines. A three node Zookeeper ensemble with each Zookeeper server running on a dedicated machine was used. Stream ingestion operators were scheduled

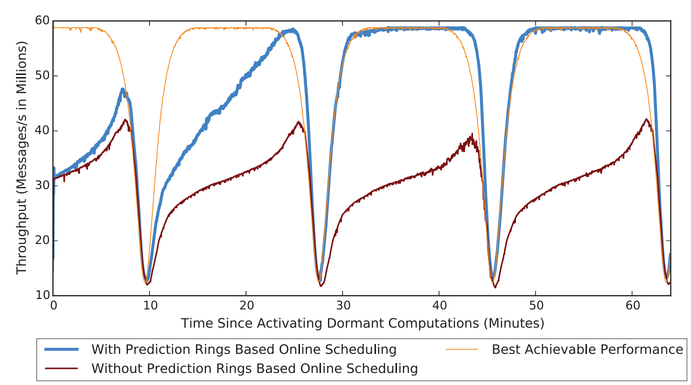



Fig. 7. Cumulative throughput of the cluster over time with **variable input rates** under **internal interference**. When migrations are enabled, the throughput gradually increases as hotspots are alleviated. Its peak performance is approximately equal to the near-ideal performance.

to run in an adequately provisioned setting with 33 dedicated machines ensuring that ingestion operators do not become a bottleneck during the experiments. Stream computations were scheduled to run on group of 15 dedicated physical machines that did not overlap with the machines allocated for stream ingestion operators. Each physical machine was running a single Neptune process. A central statistics server was used to gather various cluster-wide benchmark metrics: stream processing performance, resource utilization at individual nodes, and migration activity data. Having a centralized statistics server helped us accurately analyze how different metrics varied over time across the cluster.

### 5.2 Stream Processing Use Cases

Our empirical evaluations are performed with two use cases from the health stream processing domain: thorax extension and ECG processing using the Pan-Tompkins algorithm. In order to simulate high throughput streams, the records from these datasets were ingested at much faster rates than the actual message rates. This did not affect the correctness of the stream computations, because the timestamp value encoded in the record itself was used for processing instead of the ingestion timestamp or the wall clock time.

Thorax extension processing was more CPU intensive than ECG processing. Due to its high throughput, ECG processing computations were creating more strain on the network bandwidth. Due to their smaller in-memory state, none of them caused significant memory pressure. Stream processing graphs of both these use cases have the same structure. A single stream processing job contains a stream ingestion operator, a stream computation operator which implements the thorax extension or ECG processing logic and a sink operator that is used primarily for measuring end-to-end latencies. The stream ingestion operator and the sink operator were collocated on the same Neptune node allowing us to measure the end-to-end latency without being affected by clock synchronization and skew issues. The workloads for these benchmarks comprised a *mixture of these stream processing jobs* with a ratio of 1:1. Though real-world health stream processing use cases will only be executed on dedicated clusters due to their critical nature, we have used them to evaluate the efficacy of our online scheduling scheme in both shared and dedicated cluster setups.

#### *5.2.1 Thorax Extension Processing*

The thorax monitoring computation we use here is designed to act as a backend for a visual monitoring application. It retains the last 10 seconds of chest expansion and contraction data in memory of 6 patients, while also maintaining a running average, minimum, and maximum values seen. The thorax extension dataset we use was gathered by Dr. J. Rittweger at the Institute for Physiology, Free University of Berlin [27].

#### *5.2.2 ECG Processing with Pan-Tompkins Algorithm*

Our ECG computations process ECG waveform data from 10 ICU Patients that is available as part of the MIMIC dataset from physionet.org [28], [29]. An ECG monitors the heart's electrical activity, which drives the expansion and contraction of heart muscles based on the generated polarity.

We preprocessed ECG waveforms using the well-known Pan-Tompkins algorithm to detect the QRS complex [30]: this includes bandpass filtering, differentiation, and integration. Since ECG waveforms need to have all its frequency components within the 5-15 Hz range, the waveform is bandpassed to filter out undesired frequency components and then differentiated to attenuate the higher variations and squared to remove negative components. We then used integration to identify the peaks of the squared wave. Integrated signal peak points are used to find the QRS locations, and hence the distance between two QRS complexes, and the amplitude of the QRS is the same as that of the bandpassed wave.

### 5.3 Internal Interference: Alleviating Resource Imbalances

The objective of this set of experiments is to profile how effectively our prediction ring based online scheduling algorithm alleviates resource imbalances caused by internal interference. Such situations can arise when the workloads are unevenly distributed among nodes. We simulated an uneven workload across the cluster by activating a set of dormant computations on a select subset of nodes. We deployed 2250 stream processing computations across 15 machines (150 computations per node). Only 20% of those computations (450) were active at the beginning. After a while, we activated the remaining 80% dormant computations on 5 of those machines (600) which created a total of 1050 active computations across the cluster. After dormant computations become active, (as intended) there was a significant imbalance in the cluster where 33% of nodes became performance hotspots.

To contrast with the achievable near-ideal performance, we evenly distributed the stream processing workload within the cluster. Specifically, it placed 1050 computations evenly across the cluster with each node executing 70 active stream computations from the very beginning. This represented the best possible placement where the workload is evenly distributed and there are no imbalances in resource utilization. Over time, our online scheduling algorithm should be able to achieve a placement closer to this near-ideal distribution through migrations even when the placements are initially highly imbalanced.

Our evaluation metrics include: cumulative throughput of the cluster, $99^{th}$ percentile and standard deviation of the end-to-end latency, and resource utilization of the cluster.

The experiment was conducted for both fixed rate input streams as well as variable rate input streams. In order to generate a stream with a variable message rate, the stream ingestion operator employs a load profile that defines the message emission rate over an outgoing stream at any given time. The load profile is a function that takes the time elapsed since it was activated as the input variable to calculate the stream emission rate. Fixed rate input streams were used primarily for assessing latency related metrics. This is because the end-to-end latency of a stream packet is also governed by the message rate on that stream. If the message rate is high, the latencies tend to be higher due to prolonged queuing delays awaiting access to resources such as CPU time and network buffers.

Figure 7 depicts the cumulative throughput of the cluster observed over time with variable rate input streams. This experiment relies on the time-series models to predict the stream rates and migrate computations accordingly. Due to the bandwidth-bound nature of the stream processing use cases, computations at crowded nodes were underperforming mainly due to heavy bandwidth interference from collocated computations. As our online scheduling moves computations over to nodes with less interference, individually they start to perform better, resulting in increased cumulative throughput. The cumulative throughput improved by 48.89% compared to the original peak throughput values, and near-ideal performance is achieved after alleviating the hotspots. Next, we repeated the same benchmark with fixed rate streams in order to understand the impact of our scheduling scheme on end-to-end latency. Figure 8 depicts the performance of the system over time with and without our prediction ring based online scheduling alongside a comparison with the near-ideal performance achievable. Using the $99^{th}$ percentile, we have evaluated how the long-tail latency improved as the computations are moved away from overutilized nodes. The predictability of measured latencies is evaluated using standard deviation, which is a measure of

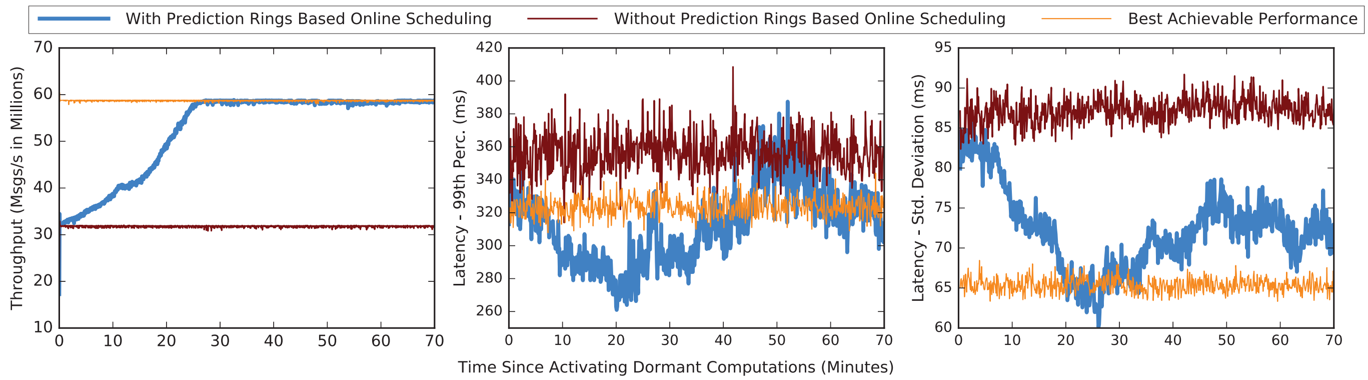



Fig. 8. Variation of cumulative throughput, $99^{th}$ percentile of latency and standard deviation of latency of the cluster over time with **fixed rate** streams under **internal interference**. The performance of the online scheduling algorithm is compared with the near-ideal performance (computations are evenly distributed across the cluster) and the initial performance (the setup without the scheduling algorithm).

the variability in the recorded latency values. Latency related metrics for even scheduling (near-ideal performance) and for the setup without the online scheduling algorithm demonstrate a relatively steady series of observations. The latencies exhibit a high initial variation when the online scheduling algorithm is running, especially when there is a large number of migrations taking place due to message buffering at upstream computations. However, once the system reaches a steady state (with a smaller number of migrations), we observe improved latency with respect to both the $99^{th}$ percentile and standard deviation. We observed a 5.24% improvement in the $99^{th}$ percentile and 15.85% improvement in the standard deviation of the end-to-end latency mainly due to reduced communication delays experienced by packets when processing is moved to nodes with less saturated links. Similar to the previous benchmark with variable rate streams, **there is a significant improvement in throughput of 83.93% compared to the setting without online scheduling.** Table 2 summarizes the performance improvements for different metrics with respect to near-ideal and initial performance for the aforementioned benchmarks.

Using the data gathered by monitoring individual nodes during the benchmark with variable rate streams, we evaluated how the resource utilization of individual nodes changed over time. The objective of this evaluation is to observe how well our scheduling algorithm can alleviate resource imbalances present in the cluster. Resource utilization was measured based on the normalized CPU load average of the process (provided by `OperatingSystemMXBean` class of Java 8) and bandwidth utilization as a percentage of the available bandwidth. Memory utilization was not considered as part of the resource utilization because our stream processing use cases did not introduce a significant memory pressure; however, our methodology facilitates incorporation of memory utilization when appropriate. Figure 9 shows how the resource utilization at individual nodes was changing over time after the dormant computations were activated. Table 3 lists the mean and standard deviation for bandwidth utilization percentage and CPU load average at different points in time. Initially, the nodes where the dormant computations were activated show significantly higher resource utilization compared to the rest of the nodes, which results in a resource imbalance within the cluster. As the online scheduling algorithm moved computations away from these hotspots, gradually the resource utilization across the cluster became more consistent and even. This can be clearly observed by how the standard deviation reduced over time. Also the average resource utilization increased as our online scheduling algorithm attempted to spread the workload evenly. This also improved the system throughput significantly as evident from our previous benchmarks (Figure 7, Figure 8, Table 2).

### 5.4 External Interference: Alleviating Hotspots

The objective of this benchmark is to evaluate the effectiveness of our online scheduling algorithm when a subset of nodes are affected by external interference. Each node is allocated the same number of stream processing computations, and 33.3% of the nodes were subjected to external interference. External interference was simulated using a separate process that

TABLE 2
Summary of performance improvements provided by our online scheduling under **internal interference**. For throughput, positive is better whereas for latency metrics, negative is desirable.

| **Use case:** Stream Computations with *Variable* Rate Streams | | |
|---|---|---|
| Metric | Deviation from Initial Perf. | Deviation from Near-Ideal Perf. |
| Throughput | +48.891% | -0.001% |
| **Use case:** Stream Computations with *Fixed* Rate Streams | | |
| Metric | Deviation from Initial Perf. | Deviation from Near-Ideal Perf. |
| Throughput | +83.932% | -0.413% |
| Latency - $99^{th}$ perc. | -5.241% | +4.768% |
| Latency - Std. Dev. | -15.845% | +12.672% |

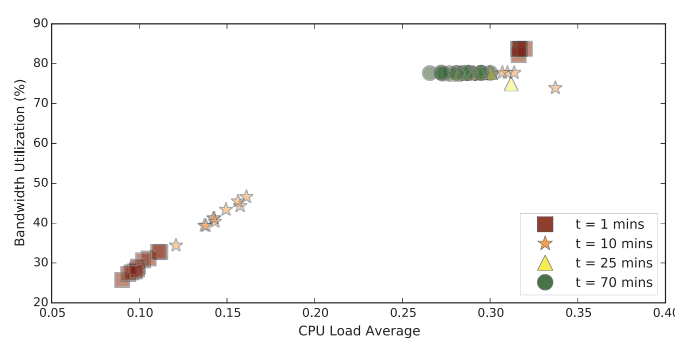



Fig. 9. Resource utilization of machines at different points in time after the dormant computations are activated.

TABLE 3
Resource utilization of individual machines over time since the activation of dormant computations.

| Time Elapsed | CPU Load Average | | Bandwidth Utilization (%) | |
|---|---|---|---|---|
| | Mean | Std. Dev. | Mean | Std. Dev. |
| 1 min | 0.1727 | 0.1023 | 47.3592 | 25.5610 |
| 10 mins | 0.2012 | 0.0805 | 53.3550 | 16.9064 |
| 25 mins | 0.2919 | 0.0082 | 77.4808 | 0.6902 |
| 70 mins | 0.2836 | 0.0098 | 77.6729 | 0.0642 |

TABLE 4
Summary of performance improvements provided by our online scheduling under **extenal interference**. For throughput, positive is better whereas for latency metrics, negative is desirable.

| Metric | Deviation from Initial Perf. | Deviation from Near-Ideal Perf. |
|---|---|---|
| Throughput | +1.300% | -0.376% |
| Latency - $99^{th}$ perc. | -85.0041% | +8.2079% |
| Latency - Std. Dev. | -82.8989% | +19.2485% |

generated significant CPU and network bandwidth pressure. Similar to the previous benchmarks cumulative throughput, $99^{th}$ percentile and standard deviation of the observed latencies were used as the evaluation metrics. As the near-ideal performance, the same number of computations were executed on a cluster of equal size without any external interference. Online scheduling algorithm was disabled when measuring the near-ideal performance. Thorax and ECG processing computations with fixed rate streams were used; but a fixed variability was introduced to the message rate as it closely simulates most real world streams.

Figure 10 plots the variation of the three metrics over time. By migrating computations away from the nodes with external interference, **our algorithm is able to recover 77.25% of the lost throughput due to external interference**. Computations migrated away from nodes with external interference benefit from less contention for network bandwidth, which is the main reason behind the throughput improvement. We could observe significant improvements in latency related metrics with our scheduling algorithm. **Both the $99^{th}$ percentile and the standard deviation showed an improvement of over 82%.** This is mainly due to reduced waiting times experienced by computations for their CPU and network bandwidth shares at nodes with less interference after the migration. The improved performance is still slightly less than the maximum achievable performance (setup without any external interference) because

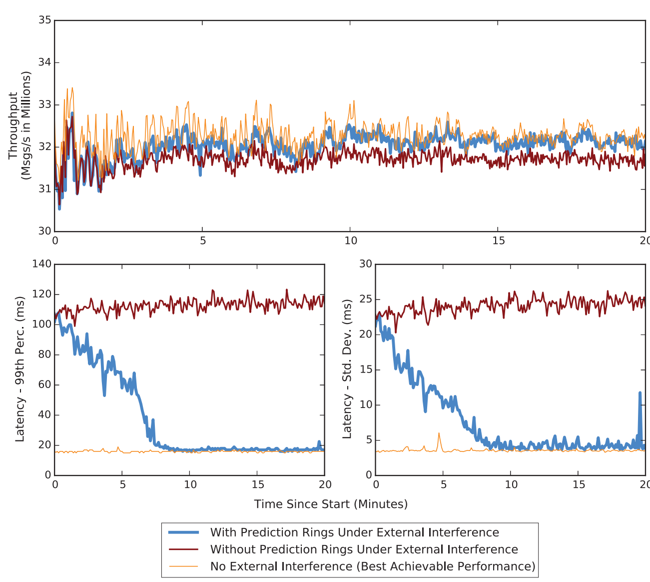



Fig. 10. Variation of cumulative Throughput, $99^{th}$ percentile of latency and standard deviation of latency of the cluster over time when 33.3% of nodes in cluster is subjected to **external interference**.

regardless of migrating computations away from the nodes affected by external interference, system resources are still shared between two groups of processes— Neptune nodes and interfering processes. Hence it is not possible to completely recover from performance degradation. A summary of performance improvement compared to the initial setting and the near-ideal setting is available in Table 4.

### 5.5 Evaluating the Stability of the System

Measures taken to maintain system stability by ensuring that only the migrations yielding significant improvements are allowed are discussed in § 4.2.1. We evaluated the effectiveness of these measures using the variable stream rate benchmark.

Dynamically adjusting the threshold for the expected reduction of interference ($\Delta I$) is one measure to ensure system stability. Figure 11 shows how the threshold is dynamically adjusted based on resource utilization imbalances within the cluster as indirectly captured by the variance in the interference scores for the impacted computation at nodes. Figure 12 shows how the number of migrations completed in successive, non-overlapping two-minute intervals is changing over time. As seen in Figure 11, the variance in interference scores recorded against individual nodes decreases gradually over time. This is indicative of the alleviation of resource utilization imbalances that were present at the beginning. There is also a decrease in the number of migrations over time (as shown in Figure 12) mainly due to the adjustment of the threshold, the deciding factor for initiating a migration, to higher values at later stages. Our benchmarks demonstrate that our algorithm encourages aggressive migrations when there is a significant imbalance in the resource utilization among nodes until the resource utilization in the cluster reaches a reasonably consistent state. Also, in the case of computations with variable rate streams, there will be continuous attempts for migrations. This is because of the different degrees of interference expected by computations due to the variable input rates, even though the amount of work performed per message is similar. This can be seen in Figure 12 from the relatively low number of migrations taking place after the initial aggressive scheduling period. In this benchmark, the average completion time for a migration is 54.68 ms (std. dev. = 65.70 ms). The time required to complete a migration is dominated by the state transference phase and the backlog clearance phase of the current computation tasks.

### 5.6 Profiling the Runtime Overhead for Online Scheduling

Running our prediction rings based online scheduling algorithm incurs additional processing and memory overheads. Periodic execution of prediction ring updates, interference score calculation, and maintaining the prediction ring data structures in memory are the primary contributing factors to this additional overhead. We monitored the memory consumption

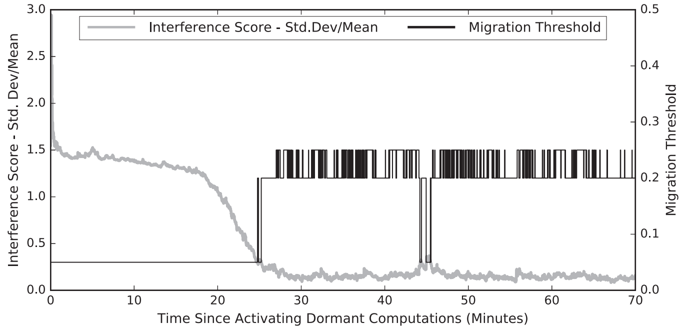



Fig. 11. Dynamically adjusted threshold of expected reduction of interference ($\Delta I$) for triggering migrations over time.

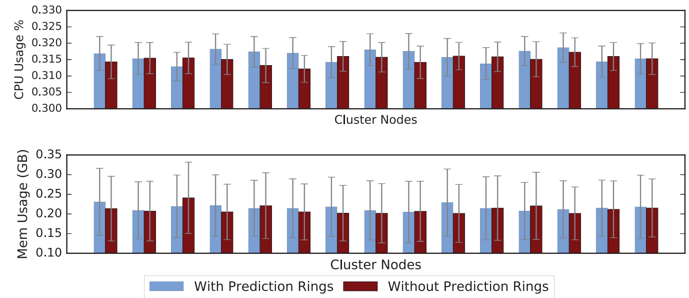



Fig. 13. CPU and memory overhead of running prediction ring algorithm and maintaining relevant data structures.

and CPU utilization of nodes when our scheduling algorithm was running and contrasted it with regular Neptune operations with the online scheduling algorithm disabled. An equal number of computations were placed with fixed input rates on each node in both cases, and their CPU and memory consumption was measured periodically. We have calculated the mean CPU and memory utilization per node using these metrics. In order to maintain a fixed number of computations at a node when the online scheduling algorithm is running, we disabled migration triggers at the orchestration manager. Node managers were still executing their periodic tasks of updating prediction rings, calculating interference scores, and sending periodic status updates to the orchestration manager. So this benchmark does not capture the additional resource utilization caused when a migration is triggered that includes: mainly processing of a few additional control messages and serialization/deserialization overhead when transferring, and restoring the state of the migrated computation. We posit that this is still a valid comparison because it captures the processing and memory overheads caused by all periodic monitoring and reporting operations.

Figure 13 shows the average CPU and memory utilization at each node with and without the online scheduling algorithm. We observed a high standard error in the memory utilization readings due to periodic garbage collection cycles. Single tail two sample t-tests were performed to check if our online scheduling algorithm caused a significant overhead. CPU utilization has increased slightly due to the online scheduling algorithm (p-value $= 0.03896$, $\alpha = 0.05$) and there was no significant evidence to suggest that the memory utilization has increased (p-value $= 0.08924$, $\alpha = 0.05$). There was approximately a 0.33% increase in CPU utilization, which we believe is acceptable given that it did not disrupt the regular execution of computations. Further, this benchmark substantiates our claim: prediction rings are lightweight and do not introduce significant memory pressure.

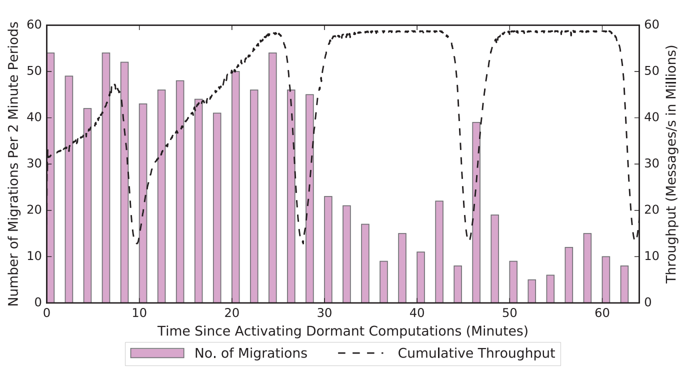



Fig. 12. Number of migrations (over a window of 2 mins) over time.

## 6 Related Work

Current state of the art stream processing systems support different scheduler implementations that are used for the initial placement of tasks in the cluster. Besides allowing users to implement custom schedulers, Apache Storm [15] includes a set of built-in schedulers such as the EvenScheduler, IsolationScheduler and ResourceAwareScheduler [16]. EvenScheduler, which is the default scheduler of Apache Storm, distributes stream computation tasks across cluster nodes in a round-robin manner. With the ResourceAwareScheduler which was intially implemented on top of R-Storm [11], the initial scheduling plan is derived based on the CPU, memory and network requirements of each Storm Spout and Bolt and resource availability of nodes as manually set by the user. In R-Storm, memory is considered a hard constraint that is always fully satisfied, whereas CPU and network are considered soft constraints which may not always be fully satisfied. These requirements are matched with the resource availability of each node using a Euclidean distance function; also tasks that communicate with each other are attempted to be scheduled with minimum network distance. Apache Flink [31] attempts to collocate tasks on a execution pipeline (a sequence of tasks through which the data flows) to a single slot in a task manager (equivalent to a Neptune node). Spark Streaming [32], being a micro-batch based stream processing system follows a different task scheduling scheme from the continuous operator systems mentioned earlier. Instead of allocating a task to a node for the lifetime of the stream processing job (assuming no dynamic scheduling), in Spark Streaming tasks are short-lived and are allocated at the beginning of each interval to calculate output RDDs for that interval. During this task allocation, it attempts to preserve the data locality, assign adjacent operators to a single task, and avoid shuffling data across the network. Xing et al. [14] proposes an initial placement algorithm for operators that is resistant to short-term load fluctuations. The expected load at each operator is modeled as a linear function of stream input rates and selectivity which is then used for operator distribution in the cluster based on two heuristics: equal load distribution and avoiding creation of bottlenecks when multiple operators are collocated. In fact, this algorithm is complementary with dynamic scheduling algorithms like ours as the authors suggest because together they can withstand short, medium and long-term load fluctuations. In such a setup, the initial placement is derived using a static algorithm, and the stream processing runtime will automatically switch to an online scheduling algorithm (such as ours) during runtime.

In continuous query evaluation over streams, round robin, chain [33], adaptive broadcast disk scheduling [34], highest normalized rate [35], and query class scheduling [36] are commonly used techniques for scheduling operators for execution within a single machine. Scheduling of new stream processing jobs in a setup where jobs are continuously arriving and departing is discussed in [10], [13], [37]. SODA [10] is an epoch scheduler designed for System S. It ensures the processing elements executing at a given time receive an adequate fraction of resources when the number of jobs in the system constantly varies assuming that the system is overloaded most of the time. This is achieved by accepting only a subset of jobs that are being submitted, selecting a data flow plan, and placing processing elements of the new jobs based on an objective function that optimizes the utilization of processing power and network interface cards of nodes and network links in the system. Unlike our study, this work focuses on initial scheduling of processing elements of new queries based on the current state of the system. It also rejects the jobs that it cannot accommodate.

Imai et al. [17] have used actors as the unit of workload migration to implement application-level workload migration. Virtual machines with less resource utilization will continuously trigger actor migrations through work-steal requests. Our system differs from this work mainly due to its proactive approach to trigger migrations. Further, they support dynamic sizing of resource pools via opportunistic VM creation and termination. Our approach assumes a fixed size resource pool but can be extended to support dynamic scaling. FUGU [38] employs a rebalancing scheme to maximize resource utilization by migrating operators between hosts upon addition or removal of queries in a complex event processing setting. FUGU migrates operators from underutilized hosts and terminates those hosts in order to improve overall resource utilization whereas in our system, computations are migrated in order to alleviate any resource imbalances in the system. In FUGU, rebalancing is triggered when queries are removed or a new host is spawned as a result of adding a new query. In our system, we continually attempt to alleviate resource imbalances caused not only due to a change in the number of concurrent stream processing jobs (equivalent to number of queries in a complex event processing system), but also due to fluctuations in both the workload and in system conditions. There has been recent work on developing network-traffic aware continuous scheduling schemes [12], [39], [40], [41] for Apache Storm [15] in order to reduce the latency by reducing the amount of network communication in a given stream processing application. Computations that communicate the most (hot edges) in a stream processing graph are identified by monitoring the communication between each computation pair, and there are continuous attempts to migrate such pairs into the same process or two processes running within the same physical node. They also ensure that the worker processes are not overloaded by taking into account the performance requirements of the computations identified through continuous profiling. T-Storm [39] goes one step further by trying to consolidate worker processes in order to reduce the number of worker processes required to run a given workload. Chatzistergiou et. al [40] improves the above task allocating approach by attempting to collocate the majority of the tasks of two communicating computations (called groups). Fischer et al. [41] uses a graph partitioning algorithm, METIS [42], to partition the query processing graph to reduce the number of messages communicated through the network. These approaches are reactive and focus on improving the latency, whereas our approach is proactive, relying on time-series analysis, and improves both throughput and latency.

## 7 Conclusions & Future Work

Our methodology for online stream scheduling encompasses algorithms and data structures that have a low memory and processing footprint when they reside in the critical path of processing streams. We treat stream processing computations as black boxes; we do not perform code inspection and rely only on externally observable features relating to stream arrival patterns and resource usage footprints to inform decisions. We now revisit our research questions.

**RQ-1:** Our online stream scheduling accounts for packet arrival rates at computations, the accompanying resource footprints, and the strains they place on machines. The prediction ring data structure and online scheduling algorithm allows us to account for packet arrival rates on streams at millisecond resolutions. Our use of exponential smoothing to perform short-term, time-series analysis of arrival patterns ensures memory compactness and fast evaluations without compromising on the accuracy needed to circumvent interference.

**RQ-2:** Prediction rings can proactively identify internal interference by tracking packet arrivals at both fine and coarse-grained scales. Tracking changes in resource utilization at a machine allows us to account for external interference from collocated processes. Alleviating external interference allows us to cope with situations wherein the quality of a machine degrades over time either due to collocated resource-intensive processes or maintenance. Prediction rings are amenable to aggregation (for all computations on a machine) and pair-wise comparisons to compute interference scores. Together, prediction rings and interference scores, allow identifying computations that are most impacted by interference and also the machines that are best suited to host them. Our empirical benchmarks demonstrates an 84% improvement in throughput by alleviating internal interference while improving latency related metrics by more than 82% by alleviating external interference.

**RQ-3:** The system targets resource utilization imbalances allowing the processing load to be amortized over a collection of machines. Continuous, incremental, and targeted migration of interference impacted (and low performing) computations allows processing to be effectively dispersed over a collection of machines leading to reduced resource imbalance. Since resource imbalances are alleviated, it is less likely that queues and processing hotspots to emerge – improving throughputs and reducing per-packet delays and the corresponding variance in these delays. The methodology also incorporates safeguards to damp oscillatory behavior where computations are continually migrated. Only the most impacted computations are migrated and that too only to those machines that have the necessary resource slack and where they do not introduce increased internal interference for existing computations.

In our future work, we will explore using reinforcement learning to inform our online stream scheduling. This will involve cost assignments including rewards for migration decisions that improve throughput and alleviate utilization imbalances while penalizing those that degrade performance.

**Acknowledgements:** This research is supported by the NSF Computer Systems Research Program (CNS-1253908).

## References

[1] M. R. Garey *et al.*, "Resource constrained scheduling as generalized bin packing," *Series A Journal of Combinatorial Theory*, vol. 21, no. 3, pp. 257–298, 1976.

[2] D. Fernández-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1427–1436, 1989.

[3] M. Wall, "A genetic algorithm for resource-constrained scheduling," Thesis, 1996.

[4] J.-K. Kim *et al.*, "Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling," *IEEE Transactions Parallel and Distributed Systems*, vol. 19, no. 11, pp. 1445–1457, 2008.

[5] A. V. Dastjerdi *et al.*, "Fog computing: Helping the internet of things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

[6] J. Gubbi *et al.*, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[7] J. Dean *et al.*, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[8] T. Buddhika *et al.*, "Neptune: Real time stream processing for internet of things and sensing environments," in *IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2016, pp. 1143–1152.

[9] M. Hirzel *et al.*, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, p. 46, 2014.

[10] J. Wolf *et al.*, "Soda: an optimizing scheduler for large-scale stream-based distributed computer systems," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2008, pp. 306–325.

[11] B. Peng *et al.*, "R-storm: Resource-aware scheduling in storm," in *Proc. of the ACM Middleware Conference*, 2015, pp. 149–161.

[12] L. Aniello *et al.*, "Adaptive online scheduling in storm," in *Proc. of the ACM DEBS*, 2013, pp. 207–218.

[13] J. Ghaderi *et al.*, "Scheduling storms and streams in the cloud," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, 2015, pp. 439–440.

[14] Y. Xing *et al.*, "Providing resiliency to load variations in distributed stream processing," in *Proc. of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 775–786.

[15] "Apache Storm," https://storm.apache.org, 2015.

[16] "Apache Storm - Scheduler," http://storm.apache.org/releases/1.0.1/Storm-Scheduler.html, 2015.

[17] S. Imai *et al.*, "Elastic scalable cloud computing using application-level migration," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 91–98.

[18] P. Hunt *et al.*, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[19] K. E. Maghraoui *et al.*, "The internet operating system: Middleware for adaptive distributed computing," *Journal of High Performance Computing Applications*, vol. 20 (4), pp. 467–480, 2006.

[20] R. D. Blumofe *et al.*, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46 (5), pp. 720–748, 1999.

[21] T. Lorido-Botrán *et al.*, "Auto-scaling techniques for elastic applications in cloud environments," *University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, vol. 12, p. 2012, 2012.

[22] NIST, "Engineering Statistics Handbook - Triple Exponential Smoothing," http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc435.htm.

[23] N. Wagner *et al.*, "Time series forecasting for dynamic environments: the dyfor genetic program model," *IEEE transactions on evolutionary computation*, vol. 11, no. 4, pp. 433–452, 2007.

[24] G. E. Box *et al.*, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[25] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." DTIC Document, Tech. Rep., 1985.

[26] C. A. Varela *et al.*, *Programming Distributed Computing Systems: A Foundational Approach*, 2013.

[27] J. Rittweger, "physiodata," 2000.

[28] M. Saeed *et al.*, "Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database," *Critical care medicine*, vol. 39, no. 5, p. 952, 2011.

[29] A. L. Goldberger *et al.*, "Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000.

[30] J. Pan *et al.*, "A real-time qrs detection algorithm," *Biomedical Engineering, IEEE Transactions on*, no. 3, pp. 230–236, 1985.

[31] "Apache Flink," https://flink.apache.org/index.html, 2015.

[32] M. Zaharia *et al.*, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proc. of the 4th USENIX HotCloud*, 2012.

[33] D. Carney *et al.*, "Operator scheduling in a data stream manager," in *Proc. of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 838–849.

[34] L. Al Moakar *et al.*, "Adaptive class-based scheduling of continuous queries," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. IEEE, 2012, pp. 289–294.

[35] M. A. Sharaf *et al.*, "Efficient scheduling of heterogeneous continuous queries," in *Proc. of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 511–522.

[36] L. A. Moakar *et al.*, "Class-based continuous query scheduling for data streams," in *Proc. of the Sixth International Workshop on Data Management for Sensor Networks*. ACM, 2009, p. 9.

[37] Z. Han *et al.*, "Elastic allocator: An adaptive task scheduler for streaming query in the cloud," in *IEEE Service Oriented System Engineering (SOSE), Intl. Symposium on*. IEEE, 2014, pp. 284–289.

[38] T. Heinze *et al.*, "Elastic complex event processing under varying query load." in *BD3@ VLDB*. Citeseer, 2013, pp. 25–30.

[39] J. Xu *et al.*, "T-storm: Traffic-aware online scheduling in storm," in *ICDCS*. IEEE, 2014, pp. 535–544.

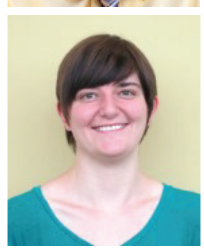
[40] A. Chatzistergiou *et al.*, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *CIKM*. ACM, 2014, pp. 1579–1588.

[41] L. Fischer *et al.*, "Network-aware workload scheduling for scalable linked data stream processing," in *Proc. of the 2013 Intl. Conference on Posters & Demos Track-Vol:1035*. ceur-ws.org, pp. 281–284.

[42] G. Karypis *et al.*, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.


**Thilina Buddhika** Thilina Buddhika is a Ph.D. candidate in the Computer Science department at Colorado State University. His research interests are in the area of real time, high throughput stream processing specifically targeted to environments such as Internet of Things (IoT) and health care applications. Email: thilinab@cs.colostate.edu

**Ryan Stern** Ryan is a Ph.D. candidate in the Computer Science department at Colorado State University. His research interests are in the area of visual analytics with an emphasis on generating real-time views of voluminous datasets. This involves coping with issue such as representativeness, memory residency, and page-fault avoidance. Email: rstern@cs.colostate.edu



**Kira Lindburg** Kira Lindburg is a software engineer at Hewlett Packard Enterprise. Her research interests are in the area of enterprise storage systems with an emphasis on fault tolerance. Email: kjlind@cs.colostate.edu

**Kathleen Ericson** Kathleen Ericson is an Assistant Professor of Computer Science at the University of Tennessee Martin. Her research interests are broadly in the area of Distributed Systems and Machine Learning. Kathleen received her B.S degree from Drexel University and her M.S. and Ph.D. degrees in Computer Science from Colorado State University. Email: ericson@cs.colostate.edu

**Shrideep Pallickara** Shrideep Pallickara is an Associate Professor in the Department of Computer Science and a Monfort Professor at Colorado State University. His research interests are in the area of large-scale distributed systems, specifically cloud computing and streaming. He received his Masters and Ph.D. degrees from Syracuse University. He is a recipient of an NSF CAREER award. Email: shrideep@cs.colostate.edu