R-Storm

Resource-Aware Scheduling in Storm

Gheorghiță Mutu
SAI I
Event Based Systems

Abstract

Big data has led to the popularity of real-time distributed stream processing systems like Apache Storm.

Storm lacks intelligent scheduling, as the default round-robin approach is inefficient.

R-Storm is a system that implements resource-aware scheduling within Storm.

R-Storm aims to increase overall throughput by maximizing resource utilization and minimizing network latency.

R-Storm can satisfy soft and hard resource constraints and minimize network distance between components.

Introduction

Challenges in processing large amounts of data quickly in the age of data dominance.

In 2012, 2.5 exabytes of data were created every day, and this has increased to 2.3 zettabytes in 2014.

Storm is an open source distributed real-time computation system.

R-Storm is the first system to implement resource-aware scheduling in Storm.

R-Storm outperforms default Storm in overall throughput and resource utilization.

Efficient scheduling of multiple topologies.

Storm: Distributed Real-time Data Processing Framework

Storm is a distributed processing framework for real-time data processing.

Storm topologies are computation graphs that process live data in real-time.

Storm jobs fragment input data into chunks processed by tasks.

Unlike MapReduce, Storm topologies run forever until they are killed.

Key terms in Storm: Tuples (basic unit of data), Streams (unbounded sequence of tuples), Components (processing operators), Tasks (instantiation of Spout or Bolt), Executors (threads that execute tasks), Worker Process (process spawned by Storm that runs executors).

Storm Components: Spouts and Bolts

Components in Storm are called Spouts and Bolts.

Spout: Source of data streams, emits unbounded number of tuples downstream.

Example: Spout can receive live financial data and transform it into tuples for processing.

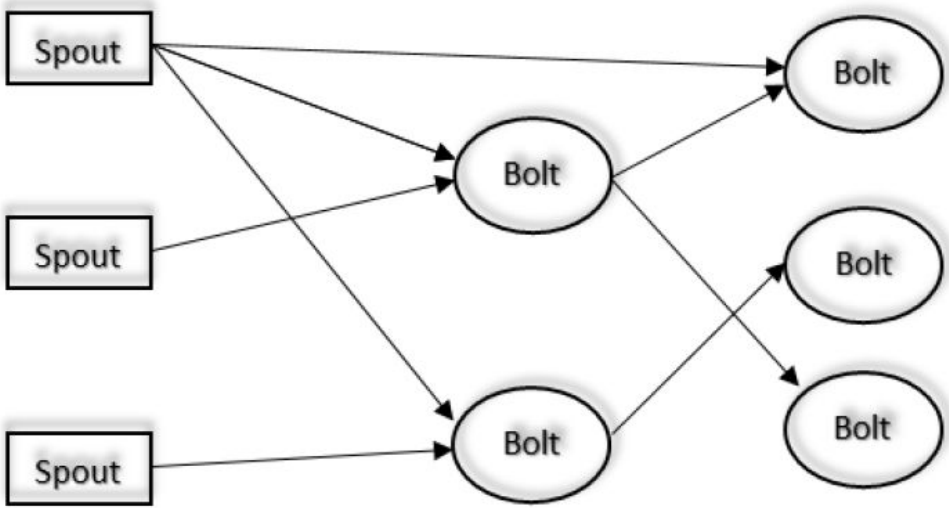
Bolt: Consumes, processes, and potentially emits new data streams.

Bolts can consume input streams from Spouts or other Bolts, perform processing, and emit new streams.

Bolts can filter, aggregate, join, query databases, and perform user-defined functions.

Multiple Bolts can work together to compute complex stream transformations, such as trending topics in tweets from Twitter.

An Example of Storm topology



Storm Cluster: Master & Worker Nodes

Master Node
Schedules tasks among worker nodes and maintains an active membership list.
Nimbus daemon communicates with Zookeeper for consistency and fault tolerance.

Worker Node
Executes tasks assigned by the Master Node.
Supervisor daemon listens for tasks and can have multiple worker processes for efficient processing.

Storm Topology and Parallelization

Links between components indicate how tuples are passed in a Storm topology.

Components can be parallelized to improve throughput.

User needs to specify parallelization hint for each component to set concurrent tasks.

Tasks from a component can be executed at different physical locations to optimize performance.

Default Storm scheduler places tasks of bolts and spouts on worker nodes in a round-robin manner for distributed processing.

Intercommunication of tasks & Storm machine

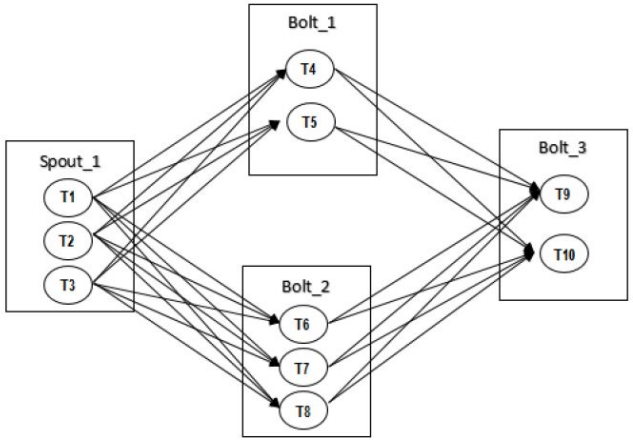


Figure 2: Intercommunication of tasks within a Storm topology

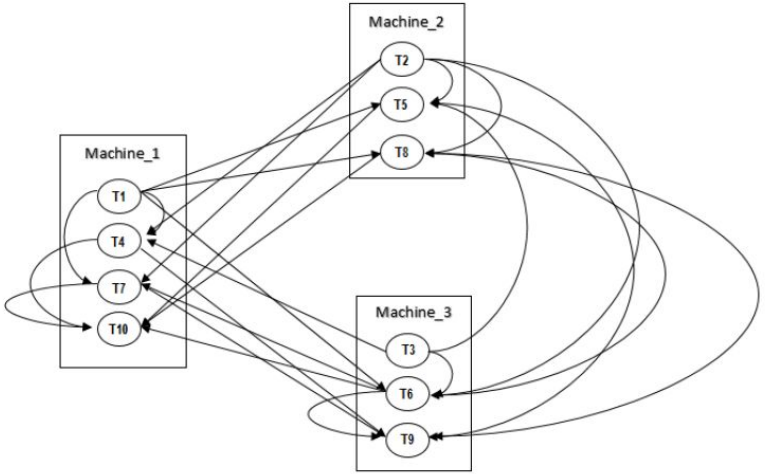


Figure 3: An example Storm machine

Resource-aware Task Assignment

Problem: Assigning tasks to machines while meeting resource requirements.

Three types of resources: CPU, memory, and bandwidth.

Hard constraints must be fully satisfied, soft constraints can be partially satisfied.

User-specified constraints.

Linear programming optimization problem.

Assumes limited budget for resources on each node, and task-specific resource requirements.

Considers resource utilization trade-offs for performance improvement.

Resource-aware Task Assignment

Tasks have CPU, bandwidth, and memory requirements.

Nodes have budget constraints for CPU, bandwidth, and memory.

Goal is to maximize resource utilization, minimize network latency, and stay within budget.

Formulated as a complex variation of Knapsack optimization problem.

Challenges: multiple knapsacks (clusters and nodes), multidimensional knapsack problem (3 resources), and quadratic knapsack problem (successive task assignment).

Efficient approximation algorithms can be utilized for computational feasibility.

Resource-aware Task Assignment

Problem: Quadratic Multiple 3-Dimensional Knapsack Problem (QM3DKP)

Previous research applied knapsack problem variations to specific contexts

Existing algorithms for knapsack problems have limitations in terms of computational complexity and real-time scheduling requirements

Need for a scheduling algorithm that can handle multiple nodes, resource requirements, and scheduling decisions quickly

Introducing R-Storm as a solution for scheduling tasks within Storm with simplicity and low overhead.

R-Storm Scheduling Algorithm

Optimal solution to resource-aware scheduling in Storm is difficult and computationally infeasible.

Need simpler and effective algorithms for solving knapsack problems, specifically the Quadratic Multiple 3-Dimensional Knapsack Problem.

Observations about Storm's environment: data centers with server racks connected by switches.

Insights on communication latency: inter-rack communication is slowest, inter-node communication is slow, inter-process communication is faster, and intra-process communication is the fastest.

Scheduling algorithm designed based on these insights.

Two core parts, task selection and node selection.

R-Storm Scheduling Algorithm

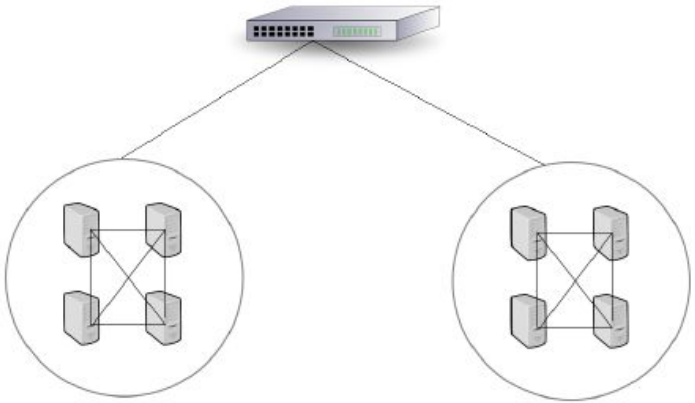


Figure 4: Typical cluster layout

Figure 5: An example node selection in a 3D resource space

Algorithm 1 R-Storm Schedule

```
1: procedure SCHEDULE
2:     taskOrdering ← TASKSELECTION(())
3:     for each Task τ in taskOrdering do
4:         Node n ← NODESELECTION(τ, cluster)
5:         SCHEDULE(τ, n);
6:     end for
7: end procedure
```

Algorithm Overview

Algorithm for scheduling tasks in a Storm topology on a cluster of nodes based on resource availability.

Nodes represented by vectors, where each vector represents resource availability.

Algorithm finds the node closest in Euclidean distance to the resource demand vector of a task, while ensuring hard constraints are not violated.

Tasks of components that communicate with each other have highest priority to be scheduled in close network proximity.

No hard resource constraints are violated.

Resource wastes on nodes are minimized.

Algorithm Overview

Figure 5 shows a visual example of a selected minimum-distance node to a given task in the 3D resource space, while the hard resource constraint (i.e. Z axis) is not violated.

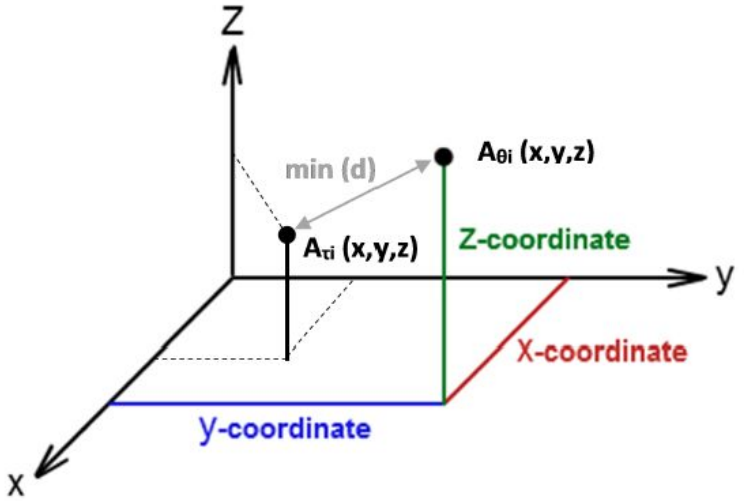


Figure 5: An example node selection in a 3D resource space

Algorithm Overview: Task Selection

Task selection is an important step in scheduling tasks in a Storm topology.

Algorithm 3 provides pseudo-code for the task selection procedure.

Traversal of the directed graph representing the Storm topology starts from spouts using breadth-first search (BFS) traversal.

BFS traversal creates a partial ordering of components, where adjacent components are placed in close succession.

Partial ordering of components is then used to create a partial ordering of tasks.

Ordering tasks based on the partial ordering of components ensures that tasks from adjacent components are scheduled close together, fulfilling the first desired property of the scheduling algorithm.

Algorithm Overview: Task Selection

Algorithm 2 Topology Traversal

```
1: procedure BFSTopologyTraversal(Component root)
2:     queue; % queue of Components
3:     visted; % list of Components
4:     if root == null then
         return null
5:     end if
6:     queue.add(root)
7:     visted.add(root)
8:     while queue is not empty do
9:         Component com ← queue.remove()
10:        for each Component n in com.neighbor do
11:            if visited does not contain n then %check whether visited or not
12:                queue.add(n);
13:                visited.add(n)
14:            end if
15:        end for
16:    end while
       return visited
17: end procedure
```

Algorithm 3 Task Selection

```
1: procedure TaskSelection
2:     components ← BFSTopologyTraversal(root)
3:     while taskOrdering does not contain all tasks do
4:         for each Component c in components do
5:             if c has tasks then
6:                 Task τ ← c.getTask()
7:                 taskOrdering.add(τ)
8:                 c.removeTask(τ)
9:             end if
10:        end for
11:    end while
       return taskOrdering
12: end procedure
```

Algorithm Overview: Node Selection

Node selection is the next step after task selection in scheduling tasks in a Storm topology.

Algorithm 4 provides pseudo-code for the node selection procedure.

For the first task in the topology, the server rack or sub-cluster with the most available resources is selected, and the first task is scheduled on the node in that server rack with the most available resources (Ref Node).

For the remaining tasks, nodes are selected based on the Distance procedure in Algorithm 4, where the bandwidth attribute is defined as the network distance from the Ref Node to each node.

Selecting nodes based on the Distance procedure ensures that tasks are scheduled tightly around the Ref Node, minimizing network latency for tasks communicating with each other.

Algorithm Overview: Node Selection

This process is visually depicted in Figure 5.

Algorithm 4 Node Selection

procedure NODESELECTION(Task τ, Cluster *cluster*)
  $\mathcal{A}_\theta$: set of all nodes θ_i in the n-dimensional space %(n=3 in our context)
  $\mathcal{A}_\tau$: set of all tasks τ_i in the n-dimensional space %(n=3 in our context)
  A_{θ_i}: n-Dimensional vector of resource availability on each node θ_i , such that $A_{\theta_i} \in \mathcal{A}_\theta$
  A_{τ_i}: n-Dimensional vector of resource availability for each task τ_i , such that $A_{\tau_i} \in \mathcal{A}_\tau$
  if global $refNode$ == null **then**
    ServerRack $s \leftarrow$ findServerRackWithMostResources(*cluster*)
    $refNode \leftarrow$ findNodeWithMostResources(s)
  end if
  select A_{θ_j} such that:
  $A_{\theta_j} = min\ d(A_{\tau_i}, A_{\theta_j})\ \forall A_{\theta_j} \epsilon A_\theta$ given $d(\lambda, \lambda^2) \leftarrow$ DISTANCE(τ_i, θ_j), $H_{\theta_j} > H_{\tau_i}$ %for all hard resource constraints
  return θ_j
end procedure
procedure DISTANCE(Task τ_i, Node θ_j)
  $distance \leftarrow weight_m * (m_{\tau_i} - m_{\theta_j})^2 + weight_c * (c_{\tau_i} - c_{\theta_j})^2 + weight_b * (newtorkDistance(refNode, \theta_j))$
  return $\sqrt{distance}$
end procedure

Implementation

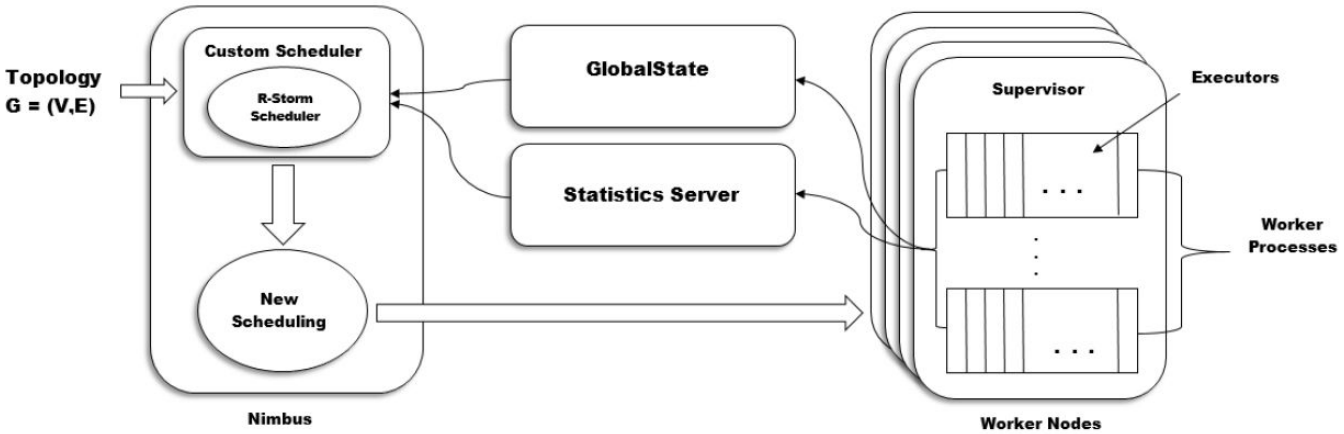


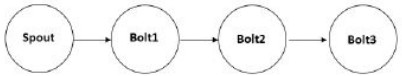
Figure 6: R-Storm Architecture Overview.

Implementation

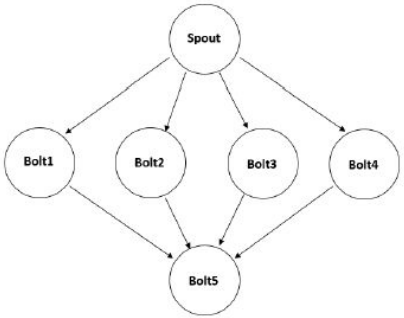
R-Storm has three modules:
- **StatisticServer**: collects statistics in the Storm cluster for evaluative purposes
- **GlobalState**: stores important state information regarding scheduling and resource availability
- **ResourceAwareScheduler**: custom scheduler that implements the core R-Storm scheduling algorithm and initializes the other two modules.

There is also a list of APIs for the user to specify the resource demand of any component and the resource availability of any physical machine.

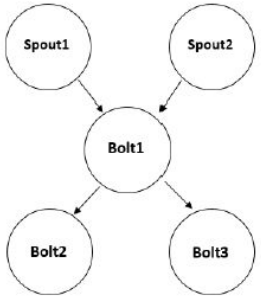
Evaluation



(a) Layout of Linear Topology

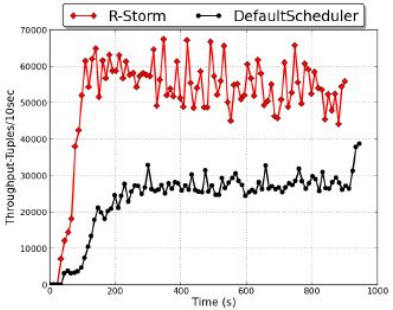


(b) Layout of Diamond Topology

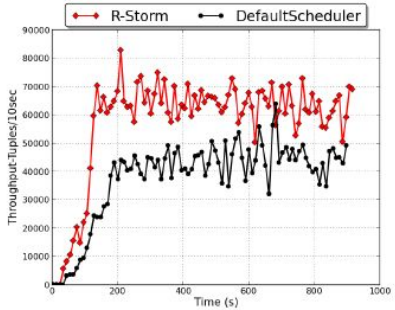


(c) Layout of Star Topology

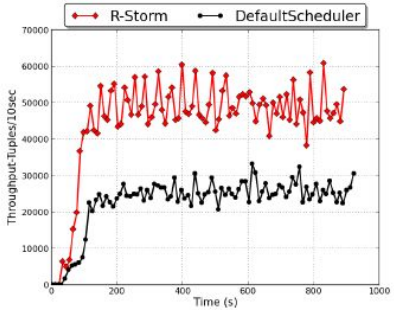
Figure 7: Layout of Micro-benchmark Topologies



(a) Linear Topology



(b) Diamond Topology



(c) Star Topology

Figure 8: Experimental results of Network-bound Micro-benchmark Topologies

Evaluation: Experimental Setup

The aim is to evaluate the scheduling algorithm in a simulated real-world environment.

Emulab, a network testbed, was used to run experiments.

The Emulab setup consists of a total of 13 machines, with one designated as the master node and the other 12 as worker nodes.

To emulate inter-rack communication latency, two VLANs were created, with each VLAN holding 6 machines.

The latency cost of inter-rack communication is 4ms for a round trip time.

Each machine runs on Ubuntu 12.04 LTS with a single 3GHz processor, 2GB of RAM, 15K RPM 146GB SCSI disks, and is connected via 100Mbps network interface cards.

Evaluation: Network Resource Bound

Micro-benchmark topologies configured to be network resource-bound

Overall throughput limited by network speed, not computation time

R-Storm outperforms Storm's default scheduler

Results show around 50%, 30%, and 47% higher throughput for Linear, Diamond, and Star Topologies respectively

R-Storm minimizes network communication latency by colocating communicating tasks on the same machine or server rack

Evaluation: Computation Time Bound

The overall throughput of micro-benchmark topologies is limited by computation time spent at each component.

Each component is configured to conduct a significant amount of arbitrary processing.

R-Storm efficiently schedules tasks closely together to maximize CPU utilization and minimize resource waste.

The cluster used consists of 12 machines partitioned into two racks.

Storm's default scheduler schedules executors on all 12 machines regardless of the actual computation each executor will use.

By using R-Storm, the user can provide hints on how much CPU computation each instance of a component in a topology needs.

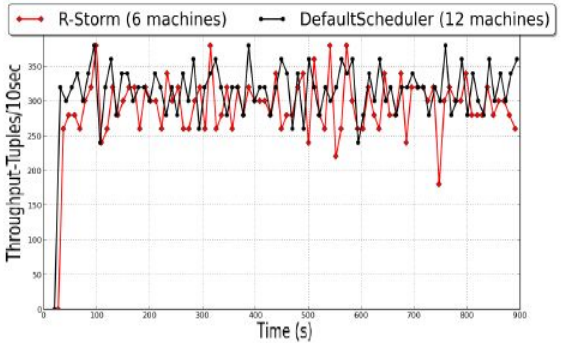
R-Storm only uses as many machines as needed to satisfy user-specified resource requirements.

R-Storm achieves the same level of throughput while using only a portion of the total number of machines.

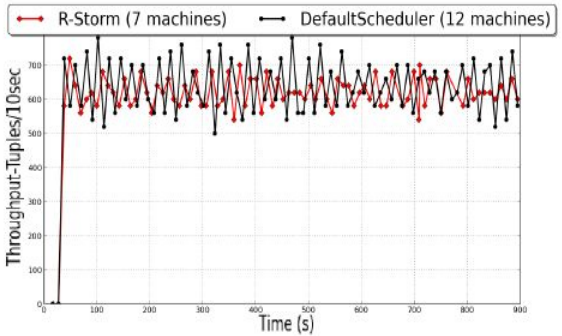
Figure 9 displays the timeline graphs for the throughput of Linear, Diamond, and Star topologies.

Figure 10 compares the average CPU utilization of machines used in the cluster when scheduling using Storm's default scheduler versus R-Storm.

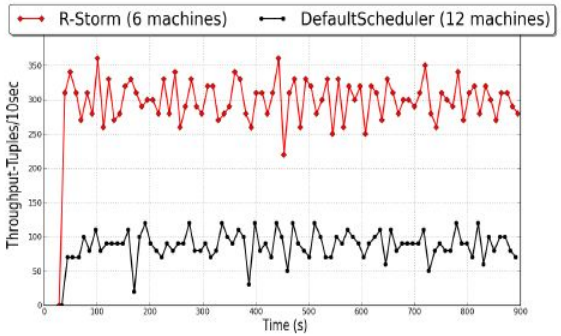
Evaluation: Computation Time Bound



(a) Linear Topology



(b) Diamond Topology



(c) Star Topology

Figure 9: Experimental results of Computation-time-bound Micro-benchmark topologies

Evaluation: Computation Time Bound

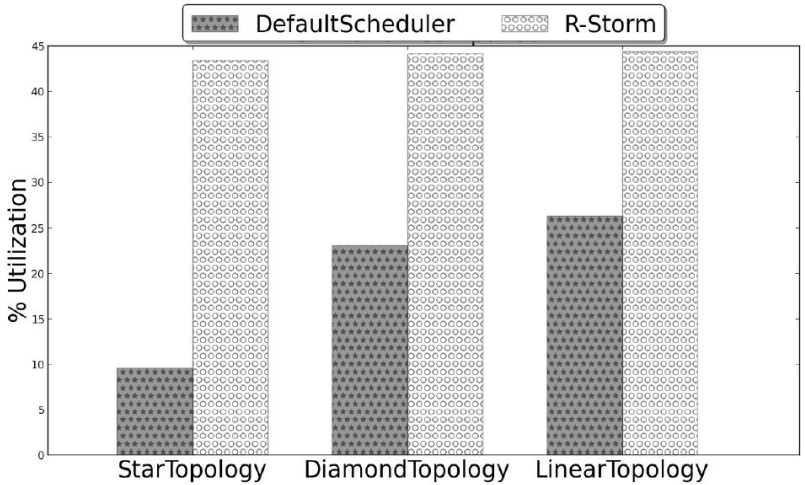


Figure 10: CPU Utilization Comparison

Micro-benchmark topologies were created to evaluate R-Storm's performance: Linear, Diamond, and Star topologies.

Performance of a topology is mainly influenced by CPU and network resources.
Storm topologies can be categorized into those bounded by network resources and those bounded by computation time.

Results of network resource-bound micro-benchmark topologies are presented first, followed by computation time-bound micro-benchmark topologies.

Evaluation: Computation Time Bound

R-Storm creates schedulings that better utilizes machines and performs just as well as Storm's default scheduler even when using fewer machines.

For the Linear topology, the average CPU utilization when using R-Storm is 69% higher than when using Storm's default scheduler.

For the Diamond topology, the average CPU utilization when using R-Storm is 91% higher than when using Storm's default scheduler.

For the Star Topology, R-Storm had much higher throughput than Storm's default scheduler even when using half of the machines.

The average CPU utilization of the cluster when using R-Storm is 350% better than that of when using Storm's default scheduler.

A topology's performance may not necessarily scale with the number of machines.

Without adjusting the parallelism of components, a topology's throughput will reach a ceiling at which adding more machines will not improve performance.

The performance of a topology is more closely related to what resources are needed, and scheduling a topology among an unnecessary number of machines can also cause an increase in communication latency.

Evaluation: Yahoo Topologies: PageLoad and Processing Topology

Two topologies from Yahoo! Inc. were evaluated: Page Load and Processing.

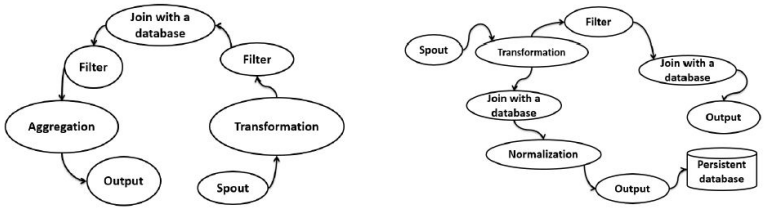
These topologies process event-level data from advertising platforms for real-time analytical reporting.

Figure 11a and 11b show the layouts of the Page Load and Processing topologies, respectively.

Experimental results show that R-Storm outperformed Storm's default scheduler.

Figure 12 displays the results, with R-Storm showing 50% and 47% better overall throughput for Page Load and Processing, respectively.

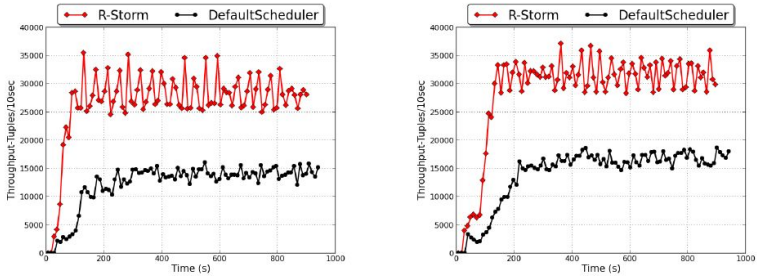
Evaluation: R-Storm on Industry Topologies



(a) Layout of Page Load Topology

(b) Layout of Processing Topology

Figure 11: Production Topologies Modeled After Typical Industry Topologies



(a) Experiment results of Page Load Topology

(b) Experiment results of Processing Topology

Figure 12: Experiment results of Industry Topologies

Evaluation: Multi-topology Performance

The experiment evaluates the performance of R-Storm when scheduling multiple topologies in a cluster.

A 24-machine cluster is used, separated into two 12-machine subclusters.

Both the Yahoo! PageLoad and Processing topologies are scheduled by R-Storm and Default Storm.

R-Storm outperforms default Storm in terms of throughput for both topologies.

The Processing topology experiences terrible performance when scheduled by default Storm, with an average overall throughput near zero.

The average throughput of the PageLoad topology is around 53% higher when scheduled by R-Storm compared to default Storm.

The average throughput of the Processing topology is orders of magnitude higher when scheduled by R-Storm compared to default Storm.

Conclusion

Storm is an open-source platform for processing streams of data in real-time.

Storm's scheduling mechanism is inadequate, scheduling tasks in a round-robin fashion with disregard to resource demands and availability.

R-Storm implements resource-aware scheduling within Storm, satisfying both soft and hard resource constraints and minimizing network distance between components that communicate with each other.

R-Storm achieves 30-47% higher throughput and 69-350% better CPU utilization than default Storm for micro-benchmark topologies and 50% higher overall throughput for Yahoo! Storm topologies.

R-Storm performs much better when scheduling multiple topologies than default Storm.

The concepts and algorithms used in R-Storm are applicable to other distributed data stream processing systems that have a DAG-based data processing model and should yield similar improvement as in Storm