Energy Usage Profiling and Topology-Based Scheduling for Clusters

by

Yangyang Liu

A dissertation proposal submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 16, 2017

Keywords: Cluster, Wimpy Node, Efficiency, Storm, Scheduling Performance

Approved by

Xiao Qin, Professor of Computer Science and Software Engineering
Wei-shinn Ku, Associate Professor of Computer Science and Software Engineering
Bo Liu, Assistant Professor of Computer Science and Software Engineering
Alvin Lim, Professor of Computer Science and Software Engineering
Eric Wetzel, Assistant Professor of Architecture, Design and Construction

Abstract

Energy saving is rapidly becoming one of the hottest topics in technology field within recent decades. With the development of technology, it brings a sheer increasing trend of data and the growth scale of clusters and data centers. Meanwhile, it also raises another essential issue into the path: energy cost.

In the first part of dissertation, we are diving into this key issue and evaluating energy-efficiency based on TPC-W benchmark: a notable web transaction e-commerce benchmark. We simulate the web transaction with different database sizes and collect the energy data by KILL-A-WATT. Also, we deploy this setup on four different cluster systems: PC nodes and wimpy nodes, and two different heterogeneous systems: using PC as front server and wimpy as Database server, and using wimpy as Web server and PC as Database server. Energy result demonstrates different characteristics among them, which can give lightening advice for future works in data center.

In the second part of this dissertation, we propose a novel scheduler for Apache Storm, topology-based scheduler(TOSS for short). Nowadays, our world is undergoing profound challenges in processing a massive amount of data. A handful of computation technologies emerge as a promising computation platform for data intensive processing. Apache Storm is an outstanding open-source platform for large-scale streaming computation, which is widely used in the industry (e.g., Twitter). Performance bottleneck problems encountered in streaming data applications motivate us to investigate scheduling issues in Storm. A key aspect of tuning Storm performance is to decide how to deploy components of a storm application among all available nodes in a cluster. Driven by our observations, we design and implement a new scheduling strategy called TOSS based on application structures. Compared to the existing round-robin scheduler, TOSS not only judiciously handles tight-bind components, but

also balances workloads by introducing a self-tuning mechanism in the deployment stage. We conduct experiments by applying two popular and distinct topologies to evaluate the performance of TOSS. The experimental results suggest that TOSS significantly boost the performance of the round-robin scheduler. In particular, TOSS substantially improves the system throughput of Storm while shortens latency of Storm applications.

Acknowledgments

There is a long list of people that I would express my deep gratitude to. First of all, I would like to express my sincere gratitude to my major advisor Dr.Xiao Qin for the continuous support of my Ph.D study and related research. I've learn a lot about being an qualified student and a good man. His patient guides and continuous supervision are always the beacon to me in the darkness. I could not have imagined having a better adviser and mentor for my Ph.D study.

Besides my adviser, I would like to give my thanks to my co-chair Dr. Ku and rest of thesis committee: Dr Liu and Dr Lim, for their insightful comments and encouragement, which helped me to widen my research from various perspectives.

My sincere thanks also goes to my colleagues at Computer System Lab, in particular, Yuanqi Chen, Ajit Chavan, Shubbhi Taneja, Yi Zhou, Xiaopu Peng, Chaowei Zhang and Jianzhou Mao, for their helps and suggestions in both academic and daily lives.

I am also indebted to my friends I met in Auburn University, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years. Without their precious support it would be much less fun in my Auburn life.

Most of all, I would like to express my deepest gratitude to my wife, Xi Huang, and our parents. They have sacrificed so much for supporting me spiritually and caring my feelings. They are always standing with me for cheering me up throughout the good and bad times.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Energy efficiency and Web service

Although various energy conservation techniques have been proposed to reduce energy cost of clusters, little attention has been paid to energy profiling of web-service applications on heterogeneous clusters. In this study, we analyze the energy consumption of web services running on heterogeneous clusters, where high-performance nodes and wimpy nodes are integrated. We show that wimpy nodes can be deployed to improve energy efficiency of clusters supporting web services in data centers.

The following three factors motivate us to investigate energy usage of web-service applications running on heterogeneous clusters.

- High energy cost of clusters housed in modern data centers.
- Popularity of web services supporting e-commerce applications.
- Feasibility of applying wimpy nodes to conserve energy in clusters.

Motivation 1. Nowadays, there is a growing demand of computing in data centers. Bell *et al.* showed that computing and storage capacity of data centers exponentially increase to fulfill the needs of big data applications [10]. The environment protection agency or EPA claimed that in 2006 data centers consumes approximately 61 billion kilowatt-hours (kWh), which accounts for 1.5 percent of the total U.S. electricity consumption [12]. Recently, a report revealed that the total electricity usage of data centers from 2005 to 2010 dramatically grew compared with the growth from 2000 to 2005 [35].

In the realm of large-scale cluster computing, high performance and low energy cost are indispensable [35]. In the past decade, much attention has been paid to the scalability issues in high-performance clusters. Growing evidence shows that energy efficiency of clusters must be improved to reduce the operation cost of data centers. When the performance of clusters was increased by a factor of 1×10^4, the performance per watt and performance per square foot were only improved by a factor of 300, and 65, respectively [24]. This problem motivates us to address the energy efficiency issues of clusters.

Motivation 2. An increasing number of Web-service applications are running on clusters housed in data centers. Sample web-service applications include YouTube, Facebook, Twitter, and to name just a few. To improve performance, scalability, and reliability of web-service applications, numerous studies were focused on service-oriented system architecture and design models [11]. How to develop energy-efficient web-service applications running on cluster is still an open issue. We start to address this issue by profiling energy usage of web services on clusters.

Motivation 3. In the past decade, a handful of energy-saving techniques have been developed to improve energy efficiency of large-scale clusters. For example, FAWN (Fast Array of wimpy Nodes) is a new cluster architecture with low-power embedded CPUs [6]. Wimpy nodes are also implemented in solving energy utilization problem in big data center [38]. Recent evidence show that equipped with low energy-efficient processors and other low-power components, wimpy nodes are in a position to trade performance for energy savings, even for big data [39]. We are motivated by advanced wimpy nodes to investigate the possibility of substituting conventional high-performance nodes with wimpy nodes to build energy-efficient clusters running web-service applications.

In this study, we conduct energy profiling research on a real-world e-commerce transaction system, where energy profiles inspire developers to design and implement novel energy conservation techniques. We investigate energy efficiency of various hardware configurations, thereby reducing energy consumption by configuring system setups. In our experiments, we

adopt TPC-W benchmark, which mimics the activities of an e-retailer in a controlled Internet commerce environment [44]. We examine a total of four scenarios. The first two scenarios represent homogeneous computing environments, where all nodes are identical. In the first scenario, the cluster system is comprised of all server node. All the nodes in the second scenario are energy-efficient wimpy nodes. The last two scenarios resemble heterogeneous clusters. In the third scenario, the front-end server is a serve whereas the back-end nodes are wimpy nodes. In the last scenario, a wimpy node performs as a front-end node while all the back-end nodes are high-performance servers. We study and compare the energy profiles of the four system configurations.

Our new findings show that wimpy nodes are energy-efficient at the cost of marginally degraded performance. In a three-tier architecture, database servers are busy processing queries under heavy load. After investigating the two heterogeneous configurations, we observe that the wimpy-frontend-server-backend case is a competitive candidate for cluster computing; the server-frontend-wimpy-backend case is the worst one due to wimpy nodes' low computing capacity to handle heavy database load.

The rest of the energy profiling part of the dissertation is organized as following. The Section 1.1 in Chapter 2 summarizes the prior studies related to energy-efficient computing. The challenges and design goal of energy profiling on clusters running commercial transaction are listed in Section 1.1. Chapter 3 includes all the details about the energy profiling research. In the Chapter 3, Section 3.1 illustrates the objectives for our study. Section 3.3 introduces the background and basic information on TPC-W benchmark. The profiling results of homogeneous and heterogeneous configurations can be found in Section 3.4.

1.2 Apache Storm and Scheduling Strategy

As the IT world enters the era of data, massive amount of data introduces challenging big-data problems. Traditional approaches to data processing mainly involves batch data processing techniques on large scale clusters. The batch data processing appears to be an

effective technique for static in-memory data sets. With pressing demands for computing frameworks that facilitate live streaming processing, it becomes indispensable to build data analytic software tools processing sheer amount of data in a timely fashion. Nowadays, the real-time processing platform - *Apache Storm*[56] (a.k.a., Storm) - handles streaming data with fairly low latency, thereby boosting resource allocation efficiency. In our study we propose to develop a structure-based Storm scheduler called *TOSS*, which is capable of accelerating the performance of Storm clusters by shortening process latency. Through an analysis of static structures for Storm applications, the TOSS scheduler leverages a self-tuning parameter to allocate executors in a way to reduce inter-node communication cost.

The following three factors motivate us to propose an efficient scheduler in Apache Storm.

- Inter-node traffic is likely to become a performance bottleneck.
- Scheduling executors represented in form of topologies play a vital role in optimizing performance of storm clusters.
- Reducing rescheduling overhead in the run time.

Motivation 1. We observe that inter-node traffic imposes noticeable impacts on the performance of Storm clusters. The default scheduler in Storm evenly distributes all assignments to workers and slots in a simple round-robin fashion. An immediate advantage of this round-robin approach is to effectively balance load and to avoid any performance bottleneck by equally splitting workload. Unfortunately, the default scheduler in Storm performs poorly due to the ignorance of inter-node network traffics. Overlooking such network load inevitably lead to a long latency in realm of real-time processing. For instance, allocating two executors that extensively exchange information on two processors incurs huge communication cost, which can be significantly reduced by assigning these two executors on the same processor. We address this issue by proposing a scheduler that is aware of inter-node

traffic. The overarching goal of our novel scheduler is to cutback high latency imposed by communication-intensive executors.

Motivation 2. In real-time Storm systems, balancing load across multiple nodes in a cluster plays an essential role in boosting system performance. Transactional topologies or topologies for short provide a strong order on data-processing tasks (a.k.a., executors). Throughout the storm part of dissertation, we use terms tasks and executors interchangeably. Scheduling decisions for executors have a significant impact on overall performance of storm clusters. This observation motivates us to be focusing on designing a new scheduler that judiciously balance workloads to minimize unnecessary network cost while optimizing system through load balancing.

Motivation 3. Our preliminary findings indicate that a balanced schedule may change its gear and become imbalanced due to frequently changing transactional topologies. Any topology update is likely to result in a performance bottleneck in Storm clusters. Rescheduling executors tends to alleviate such a performance problem in a Storm cluster at extra scheduling cost. The frequency of executor rescheduling events largely depends on two factors, namely, scheduling policies and dynamically changing typologies. In this study, we are inspired to suppress expensive rescheduling overhead by reducing the number of potential rescheduling events. We design empirical study to demonstrate that our cost-effective scheduler *TOSS* is capable of speeding up streaming data process by minimizing rescheduling overhead.

In the realm of streaming data processing, low latency time and efficient scheduling are two indispensable goals. In order to leverage available resources in Storm clusters, the default scheduler in Storm clusters evenly distributes the components of a topology on available resource slots in the round-robin fashion. This straightforward approach equally treats all resources, thereby being sufficient to avoid workload bottleneck with uniformly distributed workloads. However, the existing scheduler overlooks inter-node communication overhead as well as static topology structures, which heavily affect the overall throughput and event

processing latency. The goal of our study is to cutback processing latency measured as an interval between an event is generated by a spout and the event is completely processed by traversing the topology structure.

Inspired by the aforementioned three motivations, we design an innovative and efficient scheduling algorithm - *TOSS* - for Storm clusters. TOSS aims at reducing the average event process latency by shortening overall communication costs while balancing load across multiple cluster nodes. The rationale behind the TOSS scheduler is to identify potential edges with heavy communications on a static topology structure. TOSS groups such communication-intensive edges together to be allocated within the same slot governed by a self-tuning parameter called α. For instance, TOSS allocates two components within the same node rather than two separate nodes to reduce communication cost. Furthermore, TOSS dispatches a group of executors to a node with the least workload to well balance load in the Storm cluster. Our experimental results show that the TOSS strategy has an advantage of making judicious scheduling decisions to optimize overall performance in Storm clusters.

The rest of the storm part of dissertation is organized as following. Section 2.2 in Chapter 2 introduces prior studies related to our study. Chapter 4 includes all the details about the storm scheduler research. Section 4.1 reveals the Apache Storm background and technical terms. Section 4.2 illustrates the basic ideas about our innovative scheduler algorithm, and Section 4.3 evaluates the performance comparing to default scheduler by running different benchmarks. In Section 4.4, it offers future work directions. The last Section 4.5 makes the conclusion for our storm study.

Chapter 2

Related Work

2.1 Energy profiling

In the past decade, a wide range of energy conservation techniques were proposed to reduce operation cost of large-scale clusters in data centers. Existing schemes make an effort to make good tradeoffs between energy efficiency and performance. In many cases, high computing capacity comes at the cost of a significant increase of energy cost; reducing energy consumption tends to have adverse impacts on system performance. Existing power management strategies fall into two camps, namely, static power management (SPM) and dynamic power management systems (DPM) [57]. For SPM camp, the main idea of this approach is to utilise low-power components, such as flash memory and low energy consumption in Gordon clusters [15]. While DPM techniques focus on dynamic adjustments of cluster system itself based on current resources. In a notable instance, at the heart of PowerWatch is an internal optimization algorithm that makes use of DVS scheduling to conserve energy in high-performance computing cluster) [3]. Another scheme saves power through dynamic voltage/frequency scaling in parallel sparse applications. [16]. Different from the above studies that were focused on optimizing energy efficiency in clusters, our research pays attention to energy usage profiling of web applications running on clusters.

Web services have increasingly become popular in various domains. A large number of intriguing web applications were developed in the past decade [5]. Jazayeri investigated the trends of web-application usage like searching and tagging [31]. Thanks to the portability and functionality, E-commerce is tagged as a popular trend on the Internet. Elnikety *et al.* proposed an admission control scheme to manage requests issued for e-commerce web services [23]. In paper [49], Ran explored a new web service discovery system with QoS(stands

for quality of service) to speed up adoption rate. Fu *et al.* presented a new set of mechanism to analyze web services interaction in terms of communications among asynchronous XML messages. Unfortunately, those studies have not addressed the energy cost issue in the realm of e-commerce web platforms. Furthermore, the benefits of web service applications cover various IT-unrelated areas. ProSA [62] facilitates researchers to identify recognition of errors for proteins in three dimensional level. The availability of interactive web service largely improves the efficiency of biological computations.

A handful of systems adopt low-power components to improve system energy efficiency. For example, the FAWN architecture integrates low-power embedded CPUs with local flash storage, thereby balancing computation and I/O capabilities to enable efficient and massively parallel access to data [6]. Vasudevan *et al.* tested the performance of FAWN using an array of microbenchmarks [59]. Lang *et al.* evaluated the performance of wimpy clusters under non-wimpy workload. Moreover, Yigitbasi *et al.* proposed an energy efficient scheduling technique for heterogeneous clusters [65]. Patterson *et al.* implemented FAWN in data storage system [47]. Amanjot *et al.* evaluated energy efficiency in FAWN cloud computing data center [33]. Our work is distinctive to the above studies in the way that we investigate the energy efficiency of wimpy clusters under e-commerce workload conditions. We discover that wimpy clusters are promising platforms to energy-efficiently run web applications.

2.2 Storm and scheduler

In this section, we review some outstanding systems and solutions related to data processing. In 2008, Dean and Ghemawat introduced the programming model called map MapReduce [20], which accelerates large dataset processing for real world tasks. Later, plenty of derivatives are developed for high performance computations. Jaliya *et al.* implemented a runtime architecture that expand MapReduce structure for more classes of applications [22]. He *et al.* studied MapReduce and extended capability for graphic data processing on graphics processors [27]. Genome analysis applied functional programming philosophy of MapReduce

to accelerate the DNA analysis computation [43]. In [17], Chu *et al.* used a distinct method to speed up machine-learning algorithms by adapting map-reduce paradigm on multicore computers. Lin *et al.* investigated in real-world application with MapReduce and proposed a framework to handle text intensive tasks in natural language processing. Cloudblast [42] combines the strength of both MapReduce and virtualization for bioinformatics application on Hadoop clusters. Hyrax [41] introduced a cloud computing based variant of Hadoop on Android smart phones. In order to support mobile phone characteristics,Hyrax imports the increment of scale and mobile departure features into Hadoop. HAMA research [52] explored high level matrix computation based on MapReduce framework. In order to optimize throughput with heave traffic, Wang *et al.* applied data locality to schedule map task for better performance [61].

In the past decade, an increasing number of streaming process platforms have been developed to process unbounded streams of data [26]. For example, Filgueira *et al.* developed a data-intensive framework combining the strengths of traditional workflow management systems with novel parallel stream-based dataflow systems to support data-intensive applications across multiple heterogeneous resources [25]. Carbone *et al.* proposed an open-sourced system for both streaming and batch data processing, which is called Apache Flink [14]. Flink presents a high performance framework to execute pipelined fault-tolerant dataflows. In paper [68], Zhou *et al.* proposed a streaming framework to process massive unbounded LiDAR datasets. By modeling LiDAR data as streaming pipeline works, the computation is largely improved with a state propagation mechanism. Egilmez *et al.* presented an analytical framework for QoS(Stands for Quality of Service) enabled streaming applications on openflow networks. Streaming process frameworks largely improve the real-time applications. BiToS [60] is proposed to accelerate BitTorrent(BT) streaming service for P2P content distribution. Other sample applications running on streaming data platforms include, but not limited to, visualization [36], adaptive learning [69], diagnostic analysis [34], and to name just a few.

Among an array of competitive streaming data platforms, *Storm* is a reliable distributed real-time computing system for streams of data [56]. Unlike batch data processing systems such as Hadoop, Storm aims at processing unbounded sequences of data with unstopped components. In addition, Apache Storm enables large scale deployments with advanced capabilities of handling real-time data. For instance, Morales and Bifet implemented the *Samoa* system that makes use of Storm to offer a collection of distributed streaming algorithms for popular data mining and machine learning tasks (e.g., classification, clustering, and regression) [45]. Ranjan elaborated simple instances of large-scale data-stream-processing services by utilizing Storm as a data analytic layer [50]. Cui *et al.* proposed *POS* to simplify the development of real-time streaming applications on Storm clusters [19]. In paper [32], Karunaratne *et al.* demonstrated the way to build micro-clusters to scale Storm clusters on cloud platforms. Two distributed architectures are proposed to execute a new stream clustering algorithm in parallel. Manzoor and Morgan applied Support Vector Machine paradigm in the network intrusion detection research [40]. The detection system utilized Storm to manage the network traffic for security breaches identification. RAPID [58] support interactive data-mining and analytics in real-time fashion on Storm clusters.

A handful of systems adopt efficient schedulers to improves system performance. For example, the *Maui* scheduler is focused on three perspectives - backfilling, job prioritization and, fair sharing [30]. Shaout and McAuliffe designed a batch job scheduler applying a fuzzy logic algorithm to balance loads in a distributed system, thereby maximizing overall performance [53]. Other well-known schedulers have been widely used in various fields in the industry. For instance, the static priority pre-emptive scheduling algorithm is deployed in operating systems governing distributed systems [9]. A new topology-based scaling mechanism was adjusted to eliminate resource usage restriction in Storm clusters [54]. Zaharia *et al.* devised a delay scheduler, which achieves locality and fairness by utilizing a simple way to allocate resources for Hadoop clusters [67]. In [66], delay scheduling and copy-compute

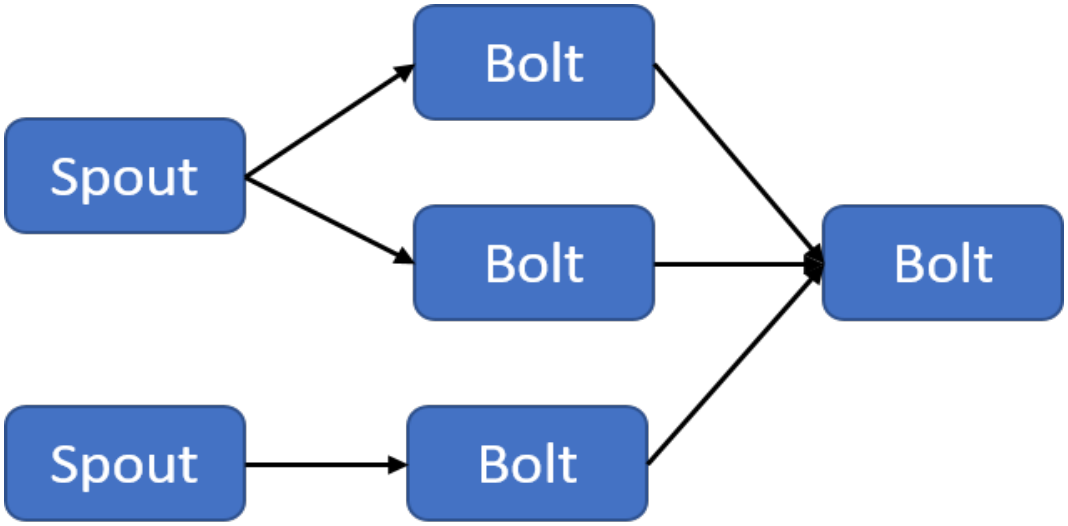

Figure 2.1: An example of topology structures in *Storm*.

splitting scheduling are proposed to improve overall throughput on Hadoop in the industry environment. The aforementioned studies were focused on general-purpose schedulers, which are inadequate for data streaming applications. Unlike the existing schedulers, TOSS proposed in this study is customized for Storm clusters to facilitate parallel data streaming in data centers.

In recent studies, efforts were made to substitute intriguing schedulers for the default round-robin scheduler in Storm [48]. For instance, Peng *et al.* investigated a resource-aware scheduler focusing on resource demands and availability [48]. The adaptive online scheduler built by Aniello *et al.* improves the overall performance in terms of latency and throughput [7]. Moreover, Rychly *et al.* investigated scheduling strategies on heterogeneous Storm clusters [51]. *T-Storm* is an outstanding scheduler that enables fine-grained control to optimize performance [63]. Unfortunately, these schedulers boost system performance at the cost of extra rescheduling overhead. Compared with the above schedulers, one strength of our TOSS is to bypass expensive rescheduling cost by analyzing topology structures. Different from these existing schedulers, TOSS embraces a self-tuning mechanism rather

than manually configuring system parameters. Once the parameters are initialized in TOSS, the parameters can be automatically tuned for changing workload conditions.

Chapter 3

Profiling Energy Usage of Web-Service Applications on Clusters

3.1 Objectives

Although there are a handful of emerging energy-saving techniques, the criteria and challenges still exist. For example, how to comprehensively evaluate energy efficiency using benchmarks like the TPC-W workload remains an open issue. In this chapter, we address the challenges and present our design objectives.

Energy Profiling. Energy performance varies under different workload conditions; this is especially true for the TPC-W benchmark that is mixed with 14 types of requests. An intuitive energy profiling toolkit becomes indispensable to meet the needs of dynamically changing energy performance. As we mentioned in the previous chapter, maintaining low energy cost is one of critical goals in the design of web systems housed in data centers.

There are two immediate benefits offered by energy profiling. First, energy profiling sheds light on energy performance under specific workload conditions; the measured energy in turn becomes a basic premise for a precise energy evaluation of the given setups. Second, energy profiling allows system administrators to make decisions on the configurations of existing setups according to a quantitative analysis. Intuitive energy performance graphs demonstrated by an energy profiling tool can expose the shortcomings and potential risks of an entire system.

Energy Efficiency. The primary goal of our study is centered around energy efficiency. We focus on facilitating web services running on clusters with energy awareness and energy measurement capability, thereby improving the energy efficiency of data centers housing the clusters. In addition to high energy efficiency, maintaining high performance of the clusters plays an essential role in our design. Hence, the best of our design choices is to

make the trade-off between computing performance and energy efficiency in the realm of web services. The energy profiling tool developed in this study is orthogonal to techniques optimizing performance of web services on clusters and; therefore, our toolkit can be readily incorporated into web services on high-performance clusters. We demonstrate that among multiple configurations, our tool offers assistance on selecting a good candidate to optimize both energy efficiency and performance of clusters running web services.

Three-Tier performances. The three-tier architecture is becoming increasingly popular in data centers. Hence, we are motivated to develop an energy profiling tool for clusters powered by the three-tier design (see Section 3.3 for details). Because dynamically changing load of the three-tier architecture imposes distinct burdens on front and back servers, the performance and energy cost trends may vary on these two types of servers. Web servers dispatch data requests to the back-end ones and have jobs displayed, whereas database servers manage data and offer searching services for the web servers. We leverage our tool to analyze workflows within three-tier servers to pinpoint any performance or energy-consumption bottleneck. It is note worthy that our profiling technique can be straightforwardly extended to address the energy efficiency or performance issues in multiple-tier cluster systems.

Wimpy node. Unlike off-the-shelf commodity cluster nodes, wimpy nodes equipped with low-performance wimpy processors offer cost-effective solutions to build large-scale clusters. Not surprisingly, wimpy nodes are unable to accommodate heavy workload normally handled by traditional high-performance nodes. In order to mitigate the high load assigned to wimpy nodes, we have to set a fair load threshold for wimpy nodes to avoid potential break downs. In this dissertation, database load imposed on the wimpy nodes and their counterparts are controlled by modifying parameter ITEM (i.e. the number of items). We conduct an empirical study to tune parameter ITEM in various configurations (see Section 3.4). For example, we discover the upper bound of parameter ITEM for the configuration where wimpy nodes are serving as database servers. This performance tuning process is of importance,

because any workload beyond the upper bound is very likely to crash the wimpy node due to the lack of hardware resources (e.g., main memory).

Heterogeneous setup. In our three-tier architecture (see Section 3.3), we investigate performance and energy efficiency of heterogeneous clusters seamlessly integrating both conventional nodes and wimpy nodes. There are various ways of putting two types of nodes together to form a three-tier web service cluster. The ultimate goal is to configure our heterogeneous cluster to make the best trade-off between high energy efficiency and performance in the arena of web services. Before tuning the performance of web services under a specific configuration, we quantitatively compare four configurations that combine both traditional and wimpy nodes. Next, we optimize energy efficiency and performance by varying other workload parameters like table size used to manipulate load incurred on database servers. Applying the TPC-W benchmark, we discover the best heterogeneous setup to minimize energy cost for web services. To resemble real-life scenarios, we replay web requests on our heterogeneous cluster in accordance with workload sampled and collected from a real-world system.

Open design. We aim to improve the extensibility and expandability of our proposed profiling toolkit. To achieve this goal, we take an open design approach in the sense that our solution makes it easy to add, upgrade, and extend profiling components into the toolkit. For example, our profiling system supports a range of benchmarks other than TPC-W. Our system is applicable to multiple-tier clusters in addition to the three-tier ones. Newly added profiling benchmarks can be seamlessly integrated into our system as add-on modules. A variety of performance metrics are chosen on the fly by system administrators. For example, if one is only concerned with energy efficiency, our tool offers a way to disable the metrics related to performance. In this case, profiling results can be collected and sampled in a fast manner; system administrators are focusing on energy issues without being distracted from any performance measure.

3.2 Framework

3.2.1 The Three-Tier Architecture

Bearing the design objectives (see Section 3.1) in mind, we develop a testbed to study energy efficiency of web services running on clusters. The testbed embraces the multiple-tier architecture design to resemble real-world flexible and scalable systems.

The multiple tier architecture in general and the three-tier architecture in particular are wildly adopted by clusters housed in modern data centers. We build a three-tier cluster coupled with logically and physically separated servers to perform and process web-service applications. Fig 3.1 depicts the layout of the three-tier architecture, where clients in tier one submit requests to front or Web servers in the middle tier, and the requests retrieve data managed by data servers in tier three. In this three-tier design, requests are submitted from client terminal to web servers where business rules are implemented; the data servers are responsible for managing massive amounts of data for web services.

To investigate energy profiles of web services, we deploy the TPC-W benchmark on the three-tier testbed to dynamically emulate real-time e-commerce transactions. The testbed processes the web workload of a book retail website, where multiple web browsers are running to emulate multiple clients [2]. The testbed utilizes emulated browsers (EB) to simulate multiple active users shopping online. The primary matrics of TPC-W are the Web Interactions Per Second or *WIPS*, which has two variants, namely *WIPSb* and *WIPSo*. *WIPSb* represents the *WIPS* performance of the browsing process; whereas *WIPSo* refers to the *WIPS* performance of the ordering procedure.

3.2.2 Metrics and Data Size

We apply these two metrics to study the performance of 14 types of transaction requests (see Table 3.1) in the benchmark. These requests types are grouped into three mix camps, namely, the browsing mix, shopping mix, and ordering mix. The detailed information on the

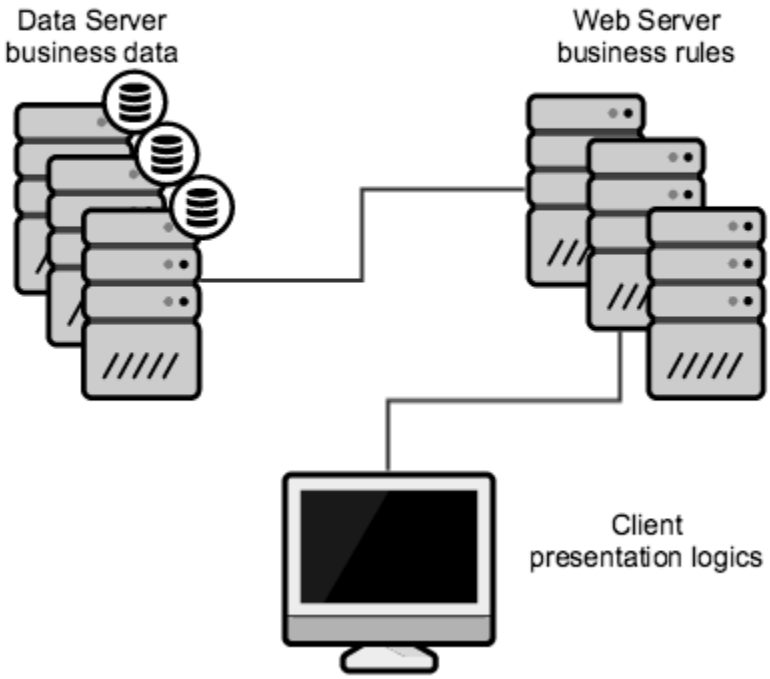

Figure 3.1: In the three-tier architecture, clients submit requests to the web-services; the middle tier consists of front or Web servers; the requests retrieve data managed by data servers in tier three.

three groups of requests can be found in Table 3.1. Regardless of request types, each request is issued by a client that imposes load on front and end servers in the three-tier system. Under the dynamically changing workload conditions, we keep track of the cluster's power consumption as well as system performance.

The important tables stored in the database servers include the ITEM , AUTHOR, CUSTOMER, and ADDRESS tables. The website data are randomly generated and stored into the database. We substantially increase the database size to resemble modern big-data applications running on clusters. We offer analysis on the impact of data size on the energy efficiency of the tested web services. It is worth noting that we manipulate the database size without changing the benchmark's implementation.

Request	**Browsing mix**	**Shopping mix**	**Ordering mix**
Browsing	95.00%	80.00%	50.00%
Home	29.00%	16.00%	9.12%
New Product	11.00%	5.00%	0.46%
Best Seller	11.00%	5.00%	0.46%
Product Detail	21.00%	17.00%	12.35%
Search Request	12.00%	20.00%	14.53%
Search Results	11.00%	17.00%	13.08%
Ordering	5.00%	20.00%	50.00%
Shopping Cart	2.00%	11.60%	13.53%
Customer Reg.	0.82%	3.00%	12.86%
Buy Request	0.75%	2.60%	12.73%
Buy Confirm	0.69%	1.20%	10.18%
Order Inquiry	0.30%	0.75%	0.25%
Order Display	0.25%	0.66%	0.22%
Admin Request	0.10%	0.10%	0.12%
Admin Confirm	0.09%	0.09%	0.11%

Table 3.1: Three types of mixed transactions in the benchmark.

The data size can be increased by adding extra records into any of the aforementioned tables. Among all the database tables, we choose to modify the ITEM table rather than the other tables to increase data size. We make this decision, because the ITEM table is the most frequently accessed table compared with the other counterparts. Appending records into the ITEM table is the most effective way of evaluating the impact of data size on the system's energy efficiency and performance.

3.2.3 The Software Framework

Fig 3.2 illustrates the detailed software framework of our testbed. The emulated browsers running on the first tier proactively issue a large number of web-service requests. In our experiments, we create 100 threads running in the client tier. Each thread follows the mixed workload specification (see Table 3.1) to randomly issue requests to the web servers in the middle tier. If the threads are not in the process of generating requests, we make the threads sleep. In doing so, we are able to manipulate the sleeping time to dynamically control

Tables	Fields	**Default Size**	**Modified Size**
ITEM	I_TITLE	60	1200
	I_PUBLISHER	60	500
	I_DESC	500	2000
AUTHOR	A_FNAME	20	200
	A_LNAME	20	200
	A_MNANE	20	200
CUSTOMER	C_UNAME	20	400
	C_UPASSWD	20	400
	C_LNAME	20	400
ADDRESS	ADDR_STREET1	40	400
	ADDR_STREET2	40	400
	ADDR_CITY	30	300
COUNTRY	—	—	—
ORDERS	—	—	—
ORDER_LINE	—	—	—
CC_XACTS	—	—	—

Table 3.2: In the modified column, we changed the numbers to expand the database size. In this case, database server pays more energy cost in processing requests.

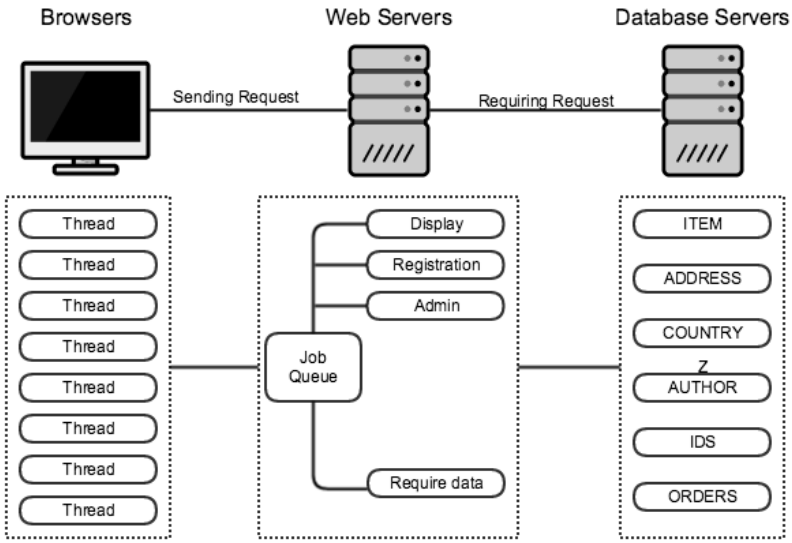

Figure 3.2: The software framework about the TPC-W benchmark.

the workload conditions. For example, a long sleeping time leads to light load; short sleep intervals give rise to heavy workload.

The web servers in the middle tier play a manager's role in terms of request dispatching. The web servers are capable of balancing the load across to the data servers by evenly distributed requests, as long as data have sufficient replicas stored on multiple data servers. The tree-tier system keeps all arrived requests in a central request queue in the web servers residing in the middle tier, which treats requests based on their types. For example, if requests intend to retrieve data in the back-end servers, then the web severs dispatch the requests to the corresponding database servers that are managing the requested data. In this case, each database server places requests transferred from the web servers into a local queue. The multiple queue governed by a group of database servers in tier three allow each tier-three server to enforce its local scheduling policy.

Interestingly, some requests are directly processed by the web servers (e.g., through JBoss) without being transmitted to the database servers. Samples of these requests include

Hardware	
Computer	HP-Mini-5103
CPU	Intel(R) ATOM(TM) CPU N4555 @ 1.66GHz *2
Memory	1.9 GB
Network	Broadcom n wireless radio
Disk	155.3GB disk
Software	
Operating System	Ubuntu 12.04 LTS Linux Kernel 3.11.0-26
Web Server	Jboss 4.0.2
Database	MySQL 5.1.72

Table 3.3: Hardware and Software configurations for Wimpy nodes

displaying the home page and customer registration. It is undoubtedly true that these types of requests have no energy and performance impact on database servers.

To improve overall system performance, the web servers make use of local caches to store popular and recently accessed data retrieved from the database servers. The caching mechanism helps in reducing load imposed on database servers in tier three while shortening request response times. It is arguably true that caching improves the energy efficiency of the database servers thanks to reduced workload by the web servers.

3.3 Setup

This section presents the experiment setup of the web-service cluster. We first describe the hardware and software configurations in Subsection 3.3.1. Next, we outline the experimental methodology in Subsection 3.3.2.

3.3.1 Configurations and Toolkit

We delineate the experiment setup, where four configurations are evaluated. We deploy the PC and wimpy nodes as the web server and database server in our three-tier architecture (see Subsection 3.2.1), respectively. Thus, the four setup configurations include:

- **Homogeneous Cluster 1:** PC Web Server and PC Database Server.

Hardware	
Computer	HP Proliant ML110 G6
CPU	Intel(R) Celeron(R) 450@2.2GHz
Memory	2 GB
Network	1 GigaBit Ethernet network card
Disk	WD-500GB Sata disk([2])
Software	
Operating System	Ubuntu 12.04 LTS Linux Kernel 3.11.0-26
Web Server	Jboss 4.0.2
Database	MySQL 5.1.72

Table 3.4: Hardware and Software configurations for PC nodes

- **Homogeneous Cluster 2:** Wimpy Web Server and Wimpy Database Server.
- **Heterogeneous Cluster 1:** PC Web Server and Wimpy Database Server.
- **Heterogeneous Cluster 2:** Wimpy Web Server and PC Database Server.

In our first configuration setup, the web and database servers are performed by the traditional PC nodes. Then, we deploy the wimpy nodes (see the description in the next paragraph) into the cluster in the other three configurations. Thus, in the second configuration, all the servers are supported by the wimpy nodes. In the third configuration, web servers are powered by the PC nodes, whereas the database servers are supported by the wimpy nodes. The web and database servers in the last configuration are performed by the wimpy and PC nodes, respectively.

We refer to the first two configurations as homogeneous setups (see the results in Subsections 3.4.1 and 3.4.2); whereas the third and fourth configurations are considered as heterogeneous setups.

Please note that wimpy nodes employed in our system contain cost-effective and energy-efficient processors. Although wimpy nodes are slower in performance than traditional PC nodes, wimpy nodes offer energy savings compared to its PC counterparts. Table 3.3 and 3.4 outline the hardware and software configurations of the PC and wimpy nodes. Regardless

Specifications	
Rating	110-130 Volts AC 60Hz
Load max	15 Amps, 1800 Watts
Unit power consumption	Under 0.5 Watts
Time Display	
Accumulated ON time	0 seconds to 9999 days
Energy Display	
Measured voltage range	110-130 Volts AC
Measured current range	0.000-15.000 Amps
Measured power	0.0 - 1800 Watts
Measured frequency range	0-60 Hz
Overload threshold	Max. 1800 Watts
Cost Display	
Accumulated electricity usage	0.00-9999 killowatt hour(kWh)
Selectable price/kWh	00.00-9.99
Accumulated cost/kWh	0.00-9999
Operating environment	0^oC - 50^oC
Storage environment	-10^oC - 50^oC
Weight	6.5Oz (185g)

Table 3.5: Detailed specifications of the Weanas electricity usage monitor.

of the node types (i.e., PC or wimpy), the JBOSS and Mysql software packages are installed on the server and database nodes, respectively. We make two types of nodes share the same software setup, because we are focusing on the energy savings provided by the hardware configuration (i.e., wimpy node).

We make use of the Weanas electricity usage monitor to measure energy cost in terms of kWh (Killwatt-Hour) for each configuration. The detailed specifications of the monitor are summarized in Table 3.5. We connect one Weanas monitor to the web servers and one to the database servers to keep track of energy consumption in parallel.

3.3.2 Experimental Methodology

In all the four aforementioned configurations, clients keep randomly sending requests governed by the scripts (see, for example, the Browsing Mix in Table 3.1). Approximately 95% requests in the browsing mix are browsing requests, which involve searching and displaying contents. We concurrently run a hundred threads to simulate distinct clients or customers, which can help to resemble real-world web service scenarios. We also follow the specification of the TPC-W benchmark [44] by determining the sleep time interval between two consecutive client requests. This way allows our testbed to simulate a web-service cluster facilitating e-commerce services.

While we are simulating requests handled by the web-service cluster in the three-tier architecture, the electricity monitors are incorporated to measure energy cost. To increase the measurement accuracy of the monitors, we conduct each experiment for a minimal duration of two hours. This accumulative energy measures are collected as a total energy cost E with the unit of kWh. Let N denote the total number of requests issued to the web-service cluster in each experiment. We measure the energy efficiency in terms of energy consumption per request. Such a measure can be derived by the total energy consumption divided by the total number of requests. Thus, we apply (3.1) to quantify energy cost of each configuration under various workload conditions. In the subsequent sections, we use the term *energy cost* and EP interchangeably.

$$EP = \frac{E}{N}. \tag{3.1}$$

The EP metric represents the energy cost with a fixed number of requests. A large EP value indicates high energy cost in the web-service processing. In this study, we aim at reducing the overall energy consumption in an entire process rather than in a fixed time period. To achieve this goal, we collect CPU utilization to quantitatively measure performance. We observe a node's behavior and performance by keeping track of CPU utilization. In the next

section, (see Section 3.4), we make use of these two metrics to evaluate the energy efficiency and performance of web-service applications on clusters.

3.4 Evaluation

Now we are in a position to present experimental results of the tested web-service cluster configured by four setups (see also Subsection 3.3.1). In Subsections 3.4.1 and 3.4.2, we evaluate the energy efficiency and performance of the homogeneous PC and wimpy clusters. The comparison between homogeneous PC and wimpy clusters can be found in Subsection 3.4.3. Finally, we present the experimental results of the heterogeneous clusters in Subsection 3.4.4.

3.4.1 Homogeneous Clusters

Our primary aim is collecting from experiments and graphing the dynamic figures of the whole setup, only one case is not enough to get the whole picture. We are also trying to change the database size by modifying ITEM number, which is a frequently used table in this mix. We keep the same ITEM contents in the ITEM table but simply populate more items into ITEM table for a different traversal time. In terms of ITEM numbers, we test energy with 1×10^4, 2×10^4, 3×10^4, 4×10^4 and 5×10^4 numbers. The energy trends are demonstrated in Fig. 3.3. Another thing needs to be monitored is the performance for job handling, so we keep monitoring and recording the CPU utilization every 60 seconds. Fig. 3.4 is telling the trends about the CPU utilization on front and back servers, this will be a clear overview for burdens on two nodes.

Fig. 3.3 shows that populating more item numbers gives rise of increased energy cost in both web and database servers. When the number of items is below 4×10^4, the Web server's energy consumption is higher than that of the database server. This trend is reasonable, because the JBoss middleware imposes extra energy cost to the Web server. Note that JBoss - playing an application server role - facilitates the communication between clients and the database. The front Web server receives requests; the Web server also processes any job

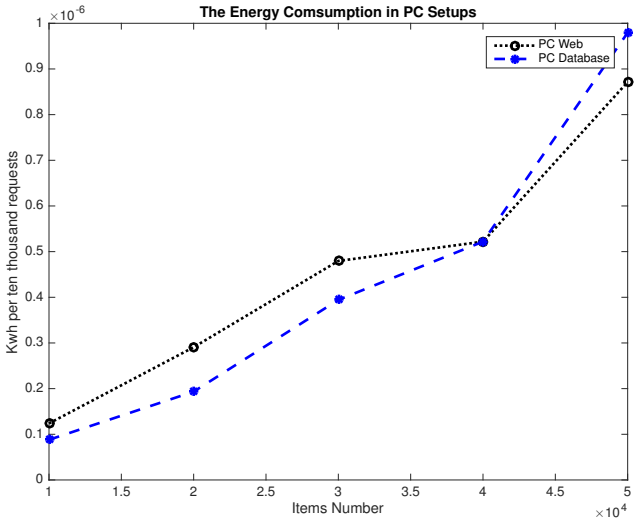

Figure 3.3: The energy trends of the web and database servers of a conventional PC cluster.

that does not access data from the backend server (e.g., such requests include home page accesses and ordering display). Processing these types of requests lead to an increased energy consumption in the Web server even when the number of items is relatively small (i.e., fewer than 4×10^4).

When the number of items is larger than 4×10^4, further increasing $ITEM$ dramatically pushes up the energy consumption of the database server. As result, the energy consumption of the database server exceeds that of the Web server. The significantly increased energy consumption at the database server is attributed to the increased CPU utilization. For example, Fig. 3.4 reveals that the database server's CPU utilization is consistently growing when we increase the number of items; in contrast, the CPU utilization of the Web server is slightly dragged down with an increasing $ITEM$.

Fig. 3.4 also shows that regardless of the $ITEM$ value, the CPU utilization of the database server is always higher than that of the Web server. When the $ITEM$ becomes large, the CPU utilization gap between the database and Web servers widens. In what follows, we explain the reasoning behind such a widened CPU-utilization gap.

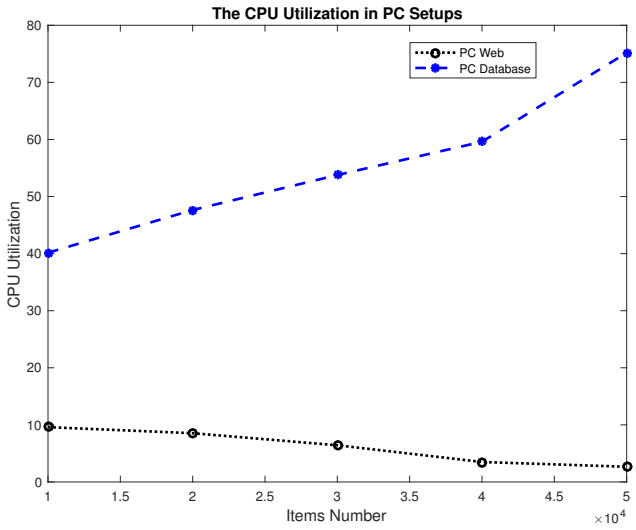

Figure 3.4: Impact of the table size on CPU utilization of the web and database servers of a conventional PC cluster.

Our analysis suggests that the database server's high CPU utilization is contributed by the nature of the browsing mix requests, which demand excessive CPU computing resources in the back-end nodes. We observe that 95% of the total requests belong to one of the browsing mix requests, among which the product-detail, search-request, and search-results requests are proactively accessing data residing in the back-end database. Adding more items in the database enlarges the item table size, which inevitably extends item-table traversing time. The long traversal times in turn lead to slow request response times of the Web server, thereby keeping an increased number of requests in a queue in the database server. Consequently, the front-end Web server maintains low CPU utilization thanks to (1) the database's slow response and (2) a long waiting queue at the back-end server.

When the $ITEM$ value reaches 5×10^4, the CPU utilization of the database server goes close to 80%, whereas the Web server's CPU utilization almost drops down to zero. The CPU-utilization trends of the Web and database servers provide evidence for the fact that the database server consumes more energy than the Web server, when the number of items is larger than 4×10^4.

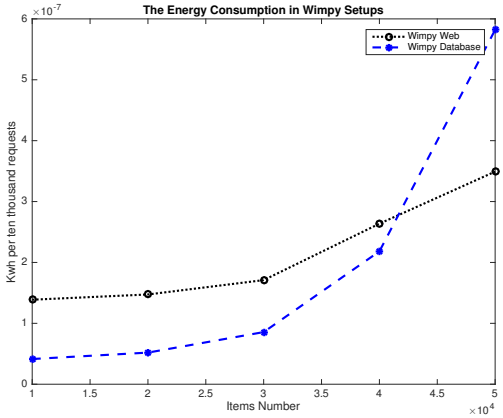

Figure 3.5: The energy trends of the web and database servers of a homogeneous wimpy cluster.

3.4.2 Homogeneous Wimpy Clusters

Now we are in a position to investigate energy efficiency of a cluster comprised of homogeneous wimpy nodes. A salient characteristic of the wimpy nodes lies in their high energy efficiency by the cost of performance. Not surprisingly, the wimpy cluster exhibits lower energy consumption than the conventional PC cluster; the request processing time of the wimpy cluster is longer than that of the PC cluster. Nevertheless, the wimpy cluster achieves good energy performance or *EP* measured in terms of energy cost per ten thousand requests. In what follows, we show evidence of such energy performance trend of the wimpy cluster.

Fig. 3.5 and 3.6 show energy consumption and CPU utilization of the wimpy cluster running Web-service applications. Interestingly, both the energy-cost and CPU utilization trends of the wimpy cluster are similar to those of the conventional PC cluster.

Fig. 3.6 reveals that when the table size exceeds 3×10^4 items, the database server's CPU utilization approaches 100%. Under the mixed request workload, the database server is a serious performance bottleneck when the table size is huge. Unlike the wimpy cluster, the PC cluster's CPU utilization is relatively low at a large table size (e.g., 3×10^4 items) thanks to the high-speed performance of the database server. This group of experiments

indicate that the wimpy cluster is incapable of handling heavy load. When the table size is set as large as 4×10^4 and 5×10^4, the saturated database server starts preventing the web sever from passing the submitted requests to the database server. Consequently, the CPU utilization of the web server becomes very low (e.g., 20%) due to the large table size (e.g., 4×10^4 and 5×10^4). In the PC-cluster case, the CPU utilization of the web server drops down to almost 0 when the table size is set to a large number.

3.4.3 Wimpy vs. PC Clusters

Now we compare energy efficiency between wimpy and PC clusters running web-service applications. To make fair comparisons, we keep software configuration and benchmark applications identical on the tested wimpy and PC clusters.

Figs. 3.7 and 3.8 show the energy efficiency of wimpy and PC clusters from the perspective of web servers and database servers, respectively. We observe that regardless of the wimpy or PC servers, the overall energy cost increases with the increasing number of items. This trend is consistent with the previous experimental results.

Importantly, both wimpy web and wimpy database servers are significantly more energy efficient than their PC counterparts. For example, when the number of items is set to 5×10^4, the wimpy web server reduces the energy consumption of the PC web server by 60.2%; the wimpy database server reduces the energy consumption of the PC database server by 39.8%. The results suggest that the wimpy cluster offers an energy-efficient three-tier architecture solution to Web-service applications.

The downside of the wimpy cluster, however, lies in its performance under heavy workload (e.g., the number of items goes beyond 5×10^4). In the case of e-commerce transactions, customers are expecting to promptly receive response of their ordering and searching requests; any delay to the response can lead to a huge amount of profit loss (see, for example, [1]). To optimize a cluster running business web applications, one has to make a good tradeoff between energy efficiency and performance under given workload conditions. With

respect to system performance, the wimpy cluster is capable of handling requests in a timely manner if the database size is small; the wimpy cluster's performance inevitably becomes unacceptable when it comes to large database tables.

For small e-business systems processing small-scale data, wimpy clusters are an ideal energy-efficient computing platform to cut operation cost in data centers. wimpy clusters can offer energy savings to large-scale web systems, where workload is relatively low (e.g., non-peak hours). In contrast, PC clusters are superior to wimpy clusters when web-service workload becomes heavy (e.g., during holiday seasons and peak hours), because wimpy clusters are unlikely to handle the request burstiness.

3.4.4 Heterogeneous Clusters

Overall Performance Comparisons

In this subsection, we aim to propose an energy-efficient way of building heterogeneous clusters by investigating the energy efficiency of two types of heterogeneous web-service clusters. Please refer to Subsection 3.3.1 for these two configurations of heterogeneous clusters running web applications. We will answer the following two intriguing questions.

- **Question 1:** Are heterogeneous clusters more energy efficient than homogeneous clusters?
- **Question 2:** What type of heterogeneous cluster is energy efficient in nature?

We build two heterogeneous clusters. In the first configuration is comprised of PC web and wimpy database servers; whereas the second heterogeneous system contains wimpy web and PC database servers. For each heterogeneous cluster, we measure energy consumption and CPU utilization under web-service workload conditions. Due to the unbalanced energy and performance efficiency between PC and wimpy servers, the tested heterogeneous clusters exhibit distinctive energy and performance behaviors compared with homogeneous counterparts.

Fig. 3.9 shows the energy efficiency of the heterogeneous configuration with the PC web and wimpy database servers. In this group of experiments, we increase the number of items (i.e., ITEM) from 1×10^4 to 5×10^4 with an increment of 5×10^4. Not surprisingly, we observe that the energy consumption per request of the PC web server is significantly higher than that of the wimpy database server. Nevertheless, the energy-efficiency trends of the PC web and wimpy database servers are very similar. In other words, the energy consumption per request slightly goes up when the number of items is increasing. Generally speaking, the results plotted in Figs. 3.10 and 3.9 are consistent; thus, the wimpy server is more energy efficient than the PC server regardless of their role (i.e., web or database servers). Unlike the trends shown in Fig. 3.9, the energy-efficiency trends plotted in Fig. 3.10 are seemingly unpredictable. For example, when the number of items is increased from 1×10^4 to 2×10^4, the energy consumption per request goes up; if we further change the number of items from 2×10^4 to 3×10^4, the energy consumption drops. In what follows, we elaborate the rationale behind such intriguing trends.

PC Web and Wimpy Database Servers

In the scenario of the first heterogeneous setup (i.e., PC web and wimpy database servers), the job queue in the PC Web server is almost empty during the course of the entire testing experiment. The evidence can be found in Fig. 3.11, which reveals that the CPU utilization of the PC web server stays at a very low level. Thus, the load imposed on the PC web server is light, making the PC web server sit idle for the majority of the time.

In contrast, Fig. 3.11 illustrate that CPU utilization of the wimpy database server rises dramatically, indicating that the wimpy server becomes heavily loaded when the number of items is large. For example, when ITEM varies from 1×10^4 to 5×10^4, the wimpy database server's CPU utilization increases from approximately 10% all the way up to 90%. Such CPU-utilization trend pictures a fact that the performance bottleneck lies in the back-end wimpy nodes. The performance gap between these two types of servers largely depends on

the computing capacity of the internal processors. Specifically, the PC's high performance enables web servers to process multiple requests within a short time period, which is not the case for wimpy processors.

Upon the arrivals of requests, the front-end PC server easily handles all the displaying requests. All the other requests requiring detailed item information are placed by the PC web server into a waiting queue, which is connected to the back-end wimpy database server. The wimpy server is unable to keep the pace with the fast speed of the PC server; this problem is worsened when the wimpy server's processing time of each request (e.g., data traversing time) is large. Under heavy workload conditions, the communication between the PC web and wimpy database servers is blocked, meaning that a significant number of requests are waiting at the front-end server due to the saturated wimpy server. In the presence of the slow wimpy servers, only a limited number of requests are handled by the wimpy database server; nevertheless, the displaying and ordering requests are independently processed by the PC web server thanks to the cached data.

An important conclusion drawn from Fig. 3.9 is that the first heterogeneous system is energy-efficient under light load (e.g., ITEM is smaller than 2×10^4). The energy efficiency of the PC web and wimpy database server noticeably drops under heavy workload (e.g., ITEM is larger than 3×10^4). The energy consumption per request of the two types of servers is almost flat before the wimpy database server becomes a performance bottleneck. Once the wimpy server is overly loaded, the energy efficiency of both web and database systems is downgrading.

Wimpy Web and PC Database Servers

Now we offer an analysis on the energy efficiency of the second heterogeneous system powered by wimpy web and PC database servers. Fig. 3.10 reveals that the second heterogeneous system exhibits an unpredictable energy efficiency when it comes to web-service

applications. The high-performance PC database server is capable of handling a large number of requests issued by the front-end wimpy server, implying that the PC database server never becomes the system performance bottleneck at the back end.

Fig. 3.12 and Fig. 3.11 show that the CPU utilization of the PC database server marginally grows from 2% to 5% when the number of items varies from 1×10^4 to 3×10^4. Then, the database server's CPU utilization jumps to 40% when the number of items is as large as 5×10^4. One attributing factor is that the local cache in the wimpy web server is relatively small, which is insufficient to cache a massive amount of data for future requests (e.g., browsing and searching). Previously cached data are likely to be repeatedly evicted from the wimpy server in order to accommodate recent requests.

Unlike the first heterogeneous system, the second heterogeneous system has different CPU trends incurred by the wimpy web server. Fig. 3.12 evidently indicate that the wimpy web server's CPU utilization slightly increases when the number of items increases, while CPU utilization decreases in other three configurations. It is arguably true that the overall performance of the second heterogeneous cluster largely depends on the front-end wimpy node. Requests accessing the database are placed in a job queue residing in the PC server, waiting for responses from the database hosted on the PC server. The wimpy front-end node has to independently handle displaying and ordering requests without interacting with the database server.

In the three-tier web-service architecture, the system reliability and stability partially depends on the front-end web server. The wimpy web server is responsible for (1) managing a job queue, (2) receiving calls from browsers, (3) sending data requests to the web server, (4) receiving data from end node, and (5) processing data. When the wimpy web server's utilization reaches its peak level, the wimpy node is unable to instantly handle incoming requests. As a result, the database server's CPU utilization drops correspondingly.

3.4.5 Put It All Together

Fig. 3.13 illustrates the energy cost trends of all the four tested configurations. We draw the following three observations by comparing the four configurations.

First, the cluster coupled with the wimpy web and PC database servers delivers the worst performance among all the solutions. If improving energy efficiency is the sole purpose, the homogeneous wimpy cluster is undoubtedly a winner under all workload conditions. In the case of a homogeneous cluster, the processing capacity of front-end servers is on par with that of back-end servers.

Second, although the two heterogeneous systems have identical hardware components, they exhibit distinct performance and energy efficiency. The system equipped with the wimpy web and PC database servers becomes the worst choice, whereas the PC-web-wimpy-database system is considered a competitive candidate. In these two heterogeneous computing environments, the performance bottleneck lies in the wimpy nodes. In contrast, the PC database servers handle heavy load at the cost of high energy consumption.

Last, when it comes to overall system performance, the PC-web-wimpy-database system is a good choice. The only downside for such a configuration is that the energy efficiency is low when data size is small due to unbalanced load between the PC web and wimpy database servers. Fortunately, when the data size is large (e.g., the number of items increases to 4×10^4 and 5×10^4), this heterogeneous system can not only speedup the request process compared with the two homogeneous clusters, but also be more energy efficient than the homogeneous PC cluster.

3.5 Summary and Future work

3.5.1 Summary and Future Work

Energy-efficient clusters equipped with wimpy nodes offer ample opportunities to conserve energy (see, for example, FAWN - fast array of wimpy nodes [6]). FAWN can well

balance computation and I/O capabilities, processing distributed data with parallel accesses. In the case of our second heterogeneous cluster, we can improve the performance of the front-end server by replacing the single wimpy node with a wimpy cluster. The wimpy cluster's parallel processing capability tends to fill the performance gap between the front-end and the back-end servers.

Evidence shows that for complex data processing workloads, a large-scale wimpy cluster may not be as energy-efficient as a small-scale traditional cluster [37]. This challenging issue opens up the research arena of heterogeneous cluster computing. We believe that job scheduling policies, heterogeneity configurations, workload profiling, and scaleup analysis are promising ways to optimize performance of heterogeneous clusters powered by energy-efficient wimpy nodes. For example, a prior study shows that mixing low-power systems and high-performance ones can energy-efficiently handle diverse workloads with various service-level agreements [18].

Among all the four tested configurations, the cluster containing PC web and wimpy database servers makes the best tradeoff between performance and energy efficiency. To speedup back-end processing performance, one may increase the number of wimpy nodes of the wimpy cluster serving as a database system. Large-scale wimpy clusters have a high reliability compared with their low-scale counterparts. With a large-scale wimpy database cluster in place, we plan to design a node partitioning strategy to dynamically partition the wimpy cluster into a set of small-scale sub-clusters. We also will develop a scheduler to dispatch requests to the multiple sub-clusters in a way to maximize energy savings while maintaining good performance.

Recall that Fig. 3.13 shows that the heterogeneous cluster with PC web and wimpy database servers consumes more energy than the two homogeneous systems. In the future, we will address this drawback by investigating a heterogeneous computing setup where the back-end database servers are powered by both energy-efficient wimpy and high-performance PC servers. In our design, we plan to have the wimpy nodes store data items that optimize

the energy efficiency of the wimpy database servers; PC database nodes manage data items that make PC servers offer energy savings. A challenging issue we will address in the future study is to investigate what data access patterns are beneficial to wimpy and PC database nodes in terms of energy efficiency. This future investigation allows us to further improve the overall energy efficiency and performance of the heterogeneous database clusters.

3.5.2 Summary

In this study, we focused on evaluating energy efficiency of web servers running on homogeneous and heterogeneous clusters. We investigated the energy profiles of a real-world e-commerce transaction system deployed on clusters. We applied the TPC-W benchmark, which mimics an e-retailer on the Internet, to study the energy efficiency of various cluster configurations. Among the four investigated system setups, the first two represent homogeneous computing environments, whereas the other two are heterogeneous systems. We showed that reducing energy consumption can be achieved through configuring cluster supporting web services. Our energy profiling results are expected to inspire developers to design and implement novel energy-efficient clusters in data centers.

If energy efficiency is the first priority, our evidence shows that homogeneous wimpy clusters are undoubtedly a winner under all the tested Web workload conditions. We confirmed that in the heterogeneous computing environment, the wimpy nodes are becoming the system performance bottleneck. Between the two heterogeneous systems, the PC-web-wimpy-database system is a competitive candidate, whereas the system equipped with the wimpy web and PC database servers is the worst choice. An important observation concluded from this study is that the PC-Web-wimpy-database system makes a good tradeoff between energy efficiency and performance.

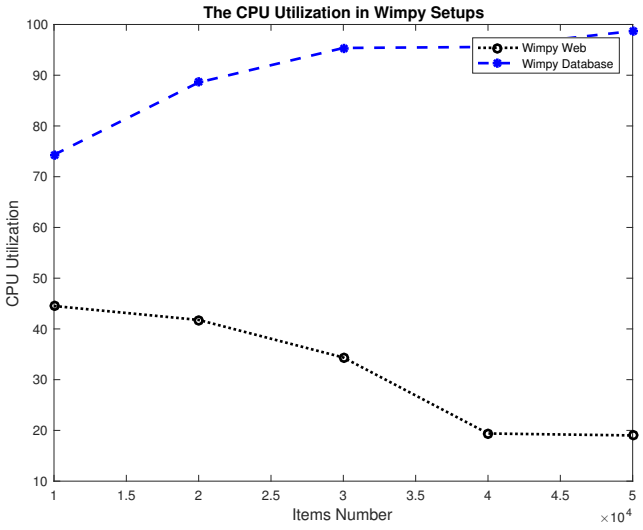

Figure 3.6: Impact of the table size on CPU utilization of the web and database servers of a homogeneous wimpy cluster.

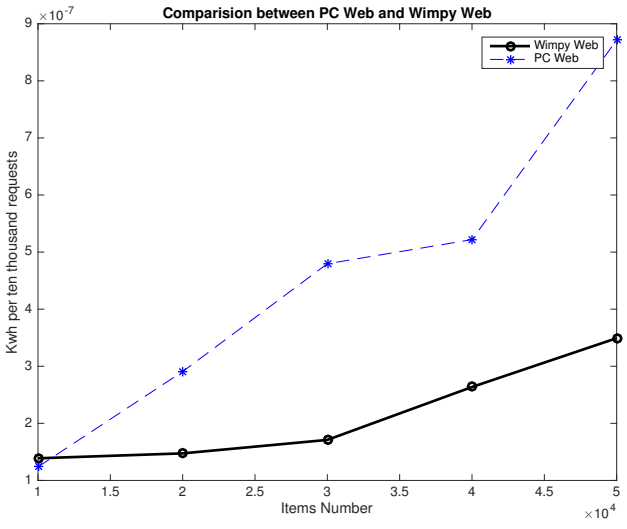

Figure 3.7: We are putting two web servers data in the same chart, in order to clearly provide the picture of the major differences.

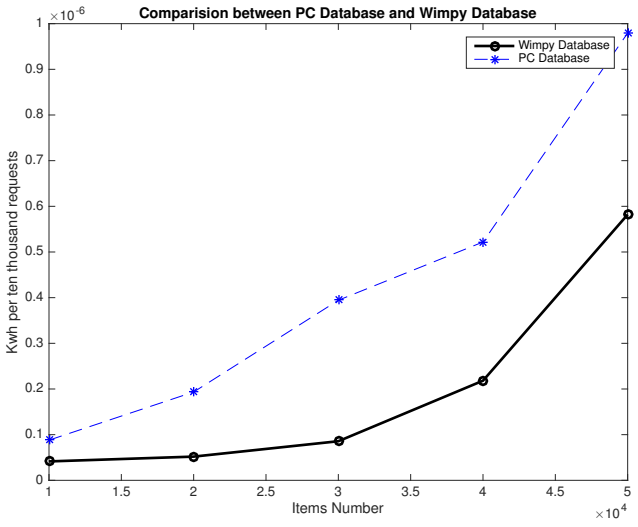

Figure 3.8: This is the graph for Database energy comparison from two setups.

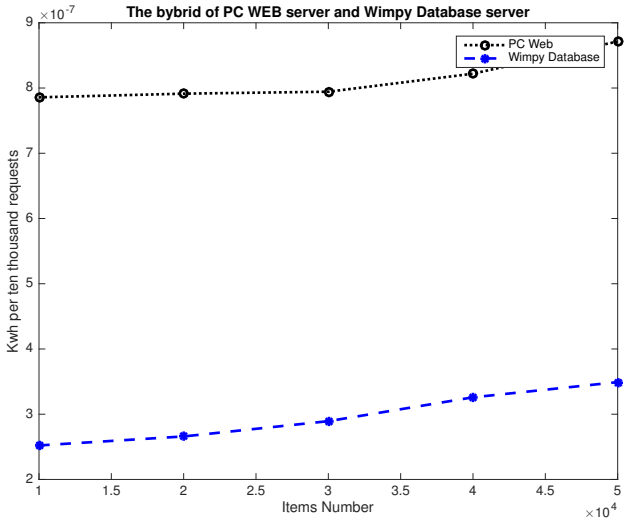

Figure 3.9: This is the energy performance for heterogeneous system with pc web and wipy database.

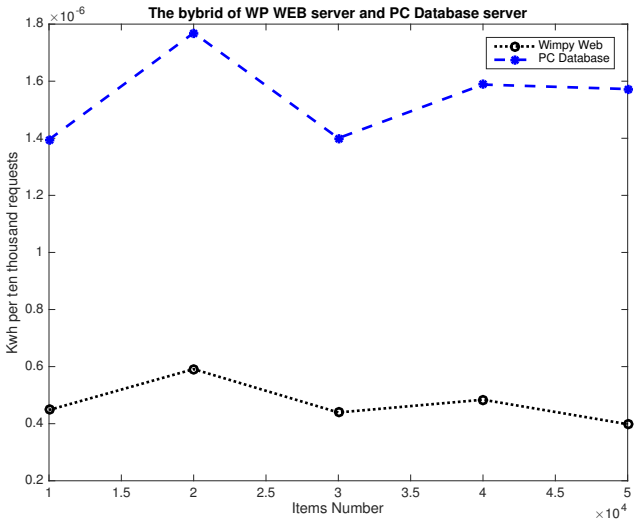

Figure 3.10: This is the energy performance for heterogeneous system with wimpy web and pc database.

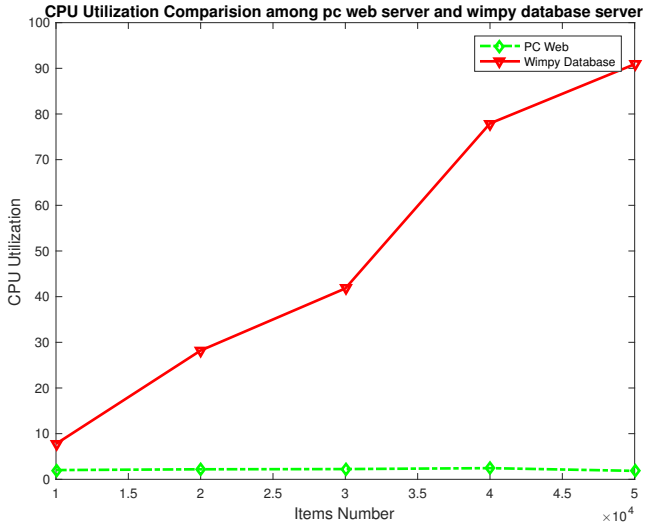

Figure 3.11: This is CPU utilization for pc web and wimpy database.

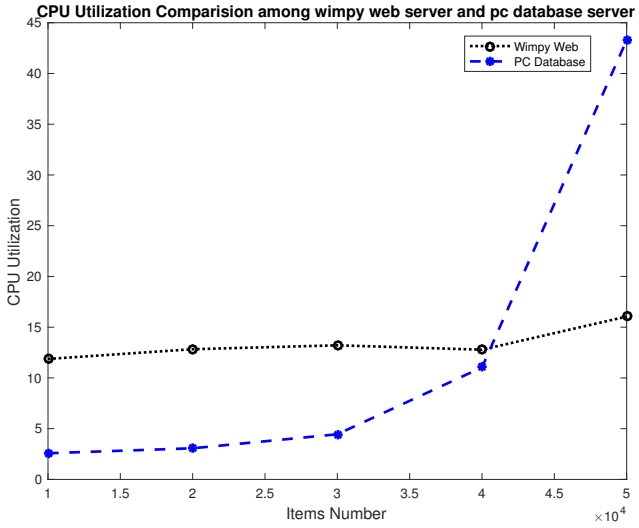

Figure 3.12: This is CPU utilization for wimpy web and pc database.

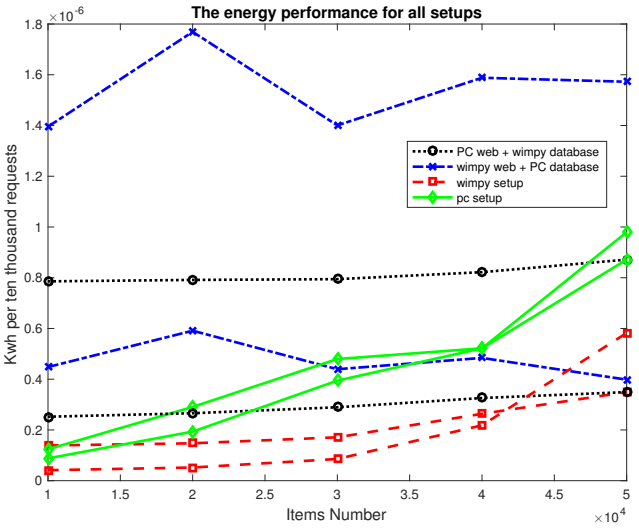

Figure 3.13: An overall comparison for all setups.

Chapter 4

TOSS: Topology-based scheduler on Apache Storm

4.1 Background

4.1.1 Topology Structures

Storm performs as a reliable distributed and fault-tolerant framework to process streaming data in a real-time fashion. Storm provides a high-level abstraction for a streaming application; the abstraction is referred to as *topology*. A Storm topology intends to organize a logic data-flow view governing how data is processed. Storm jobs typically work in a similar way as that of Map-Reduce jobs running on Hadoop clusters; storm jobs acquire non-stopped streaming data sets as an input while continuing their execution until being intentionally killed.

Fig. 2.1 illustrates a sample topology structure in Storm. A topology contains two major components, namely, *spout* and *bolt*. Spouts play the role of data collectors. A spout is responsible for wrapping actual data generators and emitting an unbounded number of data referred to as *tuple*, which follow a predefined path in the topology. In contrast, a bolt encapsulates a specific processing logic (e.g., filtering and mapping tuples) handling streams delivered from spouts. In general, multiple bolts cooperate one other to cope with complicated stream transformations that may require multiple steps. Each component (i.e., spout or bolt) is executed as a thread in Storm cluster; such a processing component is defined as an executor in storm. In a streaming application, a complete topology may consist of multiple collaborative spouts and bolts. Owing to tight collaborations among various components, communication patterns are likely to provoke heavy communications within a cluster.

The Storm architecture is constitutive of two types of nodes - master node and worker node. The master node, performing as a cluster coordinator, is liable to manage resource availability. Additionally, the master node maintains an active- membership list to ensure reliable fault-tolerant processing across worker nodes. The master node utilizes Apache *Zookeeper* [28] to manage a list of available nodes in the cluster; meanwhile, the master node relies on the *Nimbus daemon* process to perform the coordination. Worker nodes run as executor containers in a cluster. Each worker node is configured with a limited number of slots, each of which curbs the maximum number of resource allocations in execution. A worker's service and health is constantly monitored by a daemon process named *supervisor*. Once an unhealthy event occurs (e.g., temporary hardware fault and process failure), the work node will report to Nimbus offering fault tolerance by immediately handling the error. Fig. 4.1 depicts a master node and worker nodes in a storm cluster, where Nimbus, supervisors, and executors collectively processing storm tasks.

4.1.2 Scheduling Mechanism

By default, the round-robin scheduler dispatches executors to evenly distribute workload among computing nodes. Initially, the scheduler iterates through all the executors in a predefined topology and; then, the scheduler allocates assigned executors into available slots in a storm cluster. Because the round robin strategy handles all processes in a circular order without any priority, all worker nodes share almost equally process the same number of executors. Fig. 4.2 elucidates that this cyclic executive allocation eventually leads to two executors in each slots. The round-robin scheduler is compelling thanks to its simplicity and starvation-free features, which cause the least scheduling overhead in storm clusters. Unfortunately, the round-robin scheduler introduces a potential performance bottleneck due to unbalanced workload distribution. In other words, keeping the equal number of executors across computing nodes doesn't necessarily guarantee load balancing.

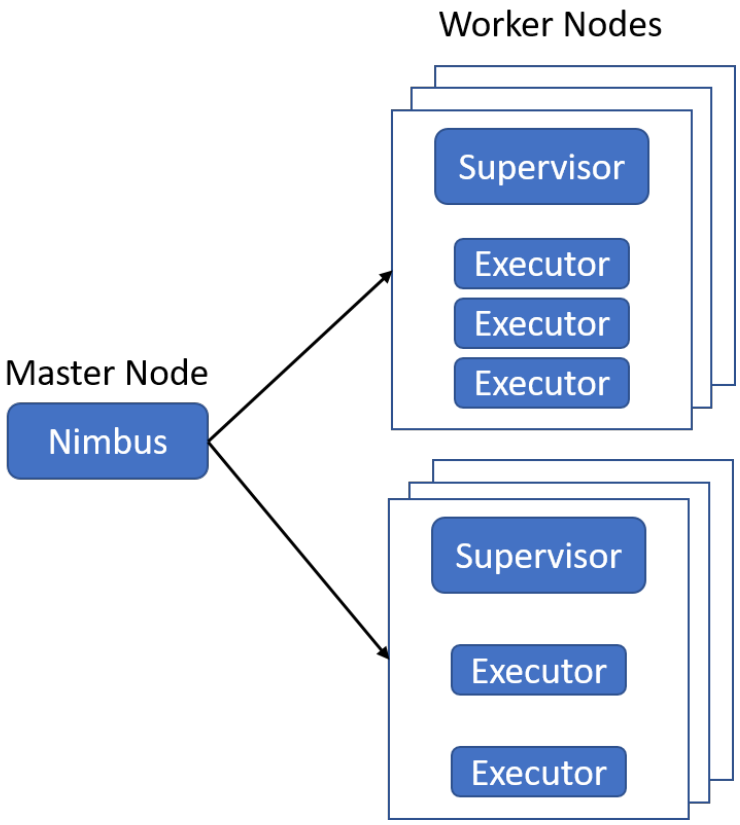

Figure 4.1: A master node and worker nodes in Storm.

Storm internally provides an interface *IScheduler* for any customized scheduling implementation. Once our TOSS scheduler is implemented and configured, Storm will detect the TOSS configurations and substitute TOSS for the existing round-robin strategy. Making allocation decisions according to static structures, TOSS is able to resolve deployment solutions without rescheduling executors at run time. On the client side, when a topology is submitted to the master node, critical information embedded in the submission include (1) initial custom parameter and (2) initial component connectivity (see Section 4.2 for details). Given the submission information, the TOSS scheduler is allowed to gauge explicit structures, followed by optimizing scheduling outcomes.

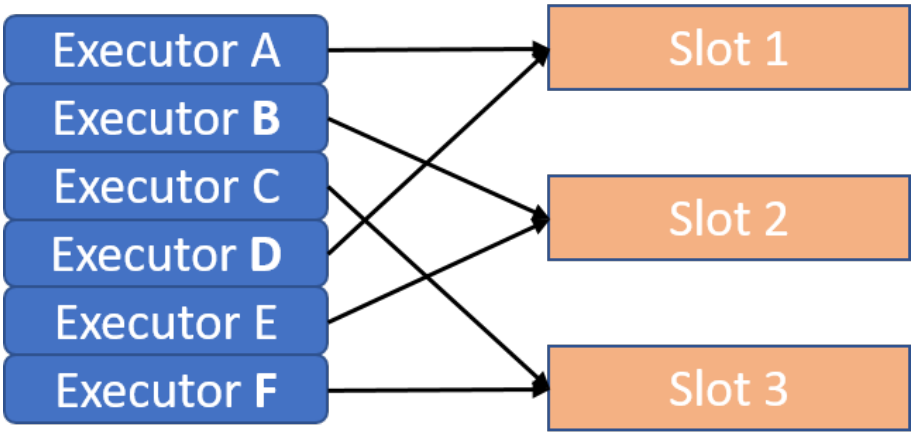

Figure 4.2: A simple example of round robin allocation strategy.

4.2 System Design

As a streaming-data processing platform, a Storm cluster's overall throughput is deeply dependent of event processing latency. We start this section by offering a deep analysis on two contributing factors of computation latency (see Section 4.2.1). Next, we design an algorithm to alleviate performance bottleneck incurred by the factors (see Section 4.2.2).

4.2.1 Pinpoint Performance Bottleneck

Prior to the design of TOSS, let us discover how to deploy tasks on Storm clusters in a way to enhance throughput and to lower latency. It is arguably true that the following two factors heavily affect computing latency in the field of streaming data process.

- Communications among a group of executors may trigger heavy network traffic.
- An structure analysis of Storm topology demands flexible parameter configurations.

In general, computing components in Storm applications fall into two camps, namely, *spout* and *bolt*. Each component constitutes of executors, which are the basic allocation units in Storm clusters. Fig. 4.3 illustrates a detailed example of inter executor communications. In Fig. 4.3, the communication patterns among the eight executors can be modeled by a

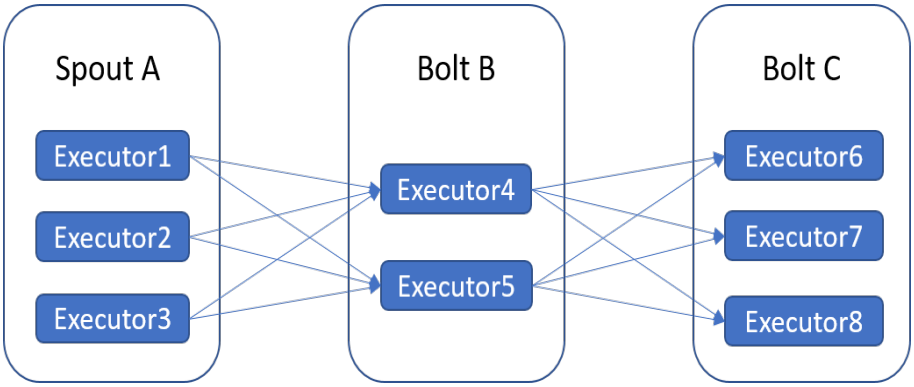

Figure 4.3: Communications among multiple executors before executor allocation.

directed acyclic graph or *DAG* for short. In this example, a linear data flow generated from spout A is past through Bolt B, which in turn collaborates with Bolt C that completes the entire process.

Each component in Storm topology is comprised of multiple concurrent running executors, which contain the same processing logic as that of the component. Those executors follow the identical data flow predefined in the topology; however, the executors may be allocated and executed on various processors or worker nodes. Inter-executor communication cost varies dramatically, heavily depending on allocation strategies.

Fig. 4.4 shows a representative example of intern-executor communication after a resource-allocation decision is made. The directed data flow is determined in storm topology (see Fig. 4.3). It is worth mentioning that communication cost in a pair of two executors may have a wide discrepancy compared with other executor pairs. For instance, in Fig. 4.4, the communication traffic between executors 1 and 4 spawns heavier cost than that between executors 1 and 5, the communication of which are handled within a thread or a process rather than across two nodes.

It is noteworthy that predicted workload based on static structures may differ from run-time workload. In particular, most of the topology-based schedulers (see, for example, the offline scheduler [36]) configure a linear parameter in order to determine the topology structure. Due to the lack of feedback from the run-time workload, the topology-component

partitions governed by the linear parameter are far from the optimal level. Such sub-optimal partitioning results in run-time lead to serious rescheduling overhead, which in turn incurs performance penalties for the topology-based scheduling strategies. Therefore, it is important to device a novel approach to accurately determine an initial linear parameter.

4.2.2 Design and Implementation

Now we are positioned to elaborate the design of *TOSS* - the topology-based scheduling algorithm. TOSS improves performance by reducing the communication overhead while taking advantage of distributed computation power of storm clusters. *TOSS* works in two phases to achieve this goal. In the first phase, *TOSS* analyzes static topology structure and partitions executors in a way to minimize the communication overhead between executors. In the next phase, TOSS utilizes previously stored run-time workload information to estimate current workload, followed by adjusting the linear parameter to control executor partitions without relying on run-time rescheduling.

Recall that (see Section 4.2.1) inter-executor communication is one of the major factors adversely impacting optimal scheduling in storm clusters. Therefore, it is of importance to suppress communication overhead during the course of making scheduling decisions. Although inevitable, the communication overhead is largely dependent of executor locations. In a storm cluster, there are three types of communication overhead, namely, inter-node communication, inter-process communication, and inter-thread communication. Our preliminary findings suggest that (1) inter-thread communication incurs the lowest overhead, (2) inter-process communication causes higher overhead, and (3) inter-node communication exhibits the worst overhead. Given a topology, an ideal strategy is to allocate executors bearing heavy communication load to the same processor rather than multiple processors or nodes.

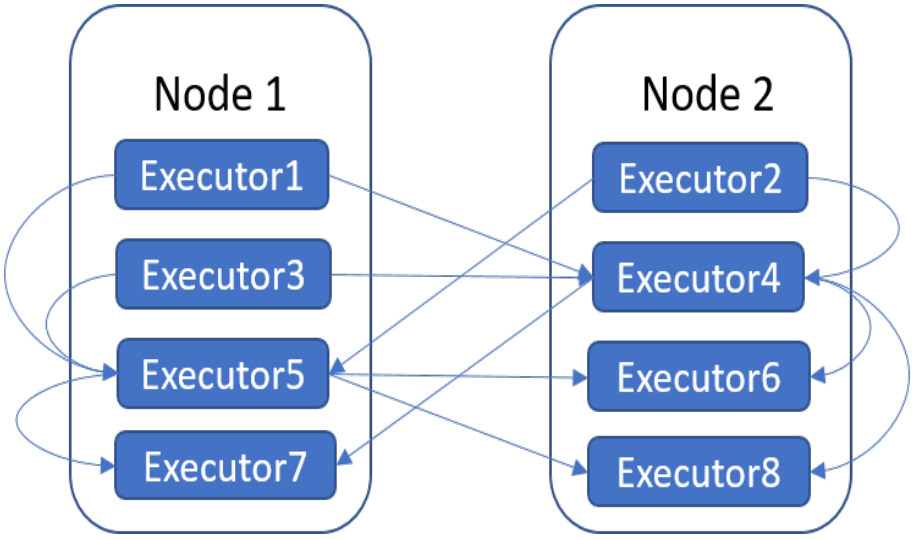

Figure 4.4: Communications among multiple executors after executor allocation.

We do not imply by any means that minimizing communication overhead guarantees improved performance. For example, a scheduling solution that aims to exclusively minimizing communication overhead may prefer to allocate all the executors to a single processor, which inevitably becomes a performance bottleneck due to the lack of processor- and node-level parallelism. Such an allocation strategy may prove to be beneficial in situations, where computation workload is dominated by communication overhead. However, when a topology consists of computation-intensive workloads, the strategy can take no advantage of parallelism due to the bottleneck problem where a single processor ends up being responsible for all computational workload. By utilizing fundamental load-balancing principles, *TOSS* makes tradeoff between high throughput and low latency in the storm clusters.

Table 4.1 summarizes important notation and symbols used throughout this part. The *TOSS* scheduling policy is divided into two main phases: partitioning and allocation. The partitioning phase takes a set of topologies (Ω) and the corresponding set E of executors. *TOSS* first analyzes the static topology structure to discover edges (e.g., between executors e_i and e_j) with heavy communication load. We refer such edges as hot edges. Because *TOSS*

Table 4.1: Notation and symbols.

Notation	Description
N	The set of all nodes in storm cluster
n_i	The ith worker node in storm cluster
Γ_i	The set of all exectuor for allocation in node n_i
α_i	THe maximu ratio of all executors for assignment θ_i
E	The set of executors for allocation
e_i	The ith executor waiting for allocation
Ω	The set of all topologies
ω_i	The ith topology for allocation
Θ	The set of assignment for allocation
θ_i	The assignment on node i
L_i	The actual load of node i
u_i	The CPU utilization of node i
s_i	The CPU speed of node i
$T_{i,j}$	The traffic overhead between executor i and executor j

analyzes topology structures without run-time workload, hot edges are pinpointed by merely investigating the connectivity among a group of components rather than run-time traffic.

In order to explicitly express connections among all the components, *TOSS* represents a topology as a directed acyclic graph or DAG. *TOSS* traverses through the entire DAG by implementing a graph traversal algorithm (e.g., Depth First Search [55] and Graph-Based traversal [13]), followed by partitioning executors into a couple of nodes centered around hot edges. In doing so, *TOSS* clusters hot edges within local nodes to avoid heavy communication traffic. While partitioning executors into an array of slots, *TOSS* strives to maximum resources available on the node (i.e., maximum allowable executor allocation α_i for assignment θ_i) to prevent the overloading issue (i.e., a single node becomes a performance bottleneck).

After partitioning is completed, TOSS generates a set Θ of assignments, which specific how to assign executors (e.g., e_i) to nodes (e.g., n_j). In TOSS, each assignment contains a group of executors to be dispatched to the same node or processor. Next, TOSS seeks available nodes to run executor groups. To balance workload, TOSS is aware of the current workload condition among all the nodes in a Storm cluster. The run-time workloads prior

to deploying topology ω_i could be locally collected from worker nodes; then, TOSS stores the workload in a central database for the purpose of future workload estimation. Assuming that the Storm cluster is homogeneous in nature, we argue that it is intuitive to measure workload using CPU utilization, which can be acquired through a Java API - *getThreadCpuTime(threadID)*. In a heterogeneous cluster, however, it is inaccurate to measure load using CPU utilization as a sole indicator. This is because a high-speed node with high CPU utilization may sustain better computation performance than low-end node with lower CPU utilization. In the heterogeneous computing case, the CPU speed should be incorporated into load predictions.

Let L_i denote node n_i's load incurred by executors. We use u_i to denote the CPU utilization of node n_i. We denote s_i as the CPU speed of node n_i. In the case of multicore processors, the CPU speed is quantified as a product of the number of cores and the single-core speed. Hence, we measure the workload L_i of node n_i as

$$L_i = u_i \times s_i. \tag{4.1}$$

Apart from cutting back communication traffic, load balancing is a second design goal in TOSS. To quantitatively measure the load balancing performance, one may introduce load deviation among all the node in a cluster. Minimizing such a load deviation achieves good load balancing performance. The load-balancing goal can be achieved by minimizing the maximal load among all the nodes.

Problem Statement. The problem can be formally stated as follows. Given a cluster of n nodes each of which has load of L_i, TOSS partitions and assign executors in the way to minimizing the maximal load among all the nodes. Thus, we have

$$\text{Minimize: } \max_{1 \le i \le n}(L_i). \tag{4.2}$$

When it comes to inter-executor traffic measurement, TOSS collects the rate at which executor e_i receives tuples from executor e_j $(i \neq j)$. The rate unit is the number of tuples per second or *tuples/sec*. Let us denote $T_{i,j}$ as the inter-executor traffic between executor e_i and e_j.

We measure the total inter-node traffic by aggregating the traffic of events exchanged among executor e_i and e_j $(e_i \in \Gamma_p,\ e_j \in \Gamma_q,\ p \neq q)$ deployed on distinct nodes, where Γ_p denotes the set of executors dispatched in node n_p. The objective of TOSS with respect to allocation is to minimize the inter-node traffic. Thus, we have

$$Min \sum_{e_i \in \Gamma_p,\ e_j \in \Gamma_q,\ p \neq q} T_{i,j}\ (i \neq j) \tag{4.3}$$

For a single assignment, the allocation has to satisfy constraint α_i:

$$\theta_i \leqslant \Theta\ *\ \alpha_i \tag{4.4}$$

where θ_i represents the number of executors in the allocation assignment on node i, and Θ denotes the total number of executors defined in the topology ω_r. The α_i denotes the the maximum ratio of all executors that can be allocated for assignment θ_i. The value of α_i varies anywhere between 0 to 1; we have $\sum_{i \in N} \alpha_i\ =\ 1$.

The algorithms 1 and 2 implement the TOSS scheduling strategy. Algorithm 1 acquires parameter α by running self-tuning algorithm. By utilizing the graph traversal algorithm DFS, TOSS searches for a chain in the topology structure. A chain is defined as a series of executors connected by data flow one after the other, for instance, executor e_1, executor e_5 and executor e_6 constitute a chain in Fig 4.3. The responsiblity of the chain search is to group the executors with communications, enabling TOSS to reduce the inter-executor traffic overhead by grouping executors into the same slot. DFS is a preferred graph traversal algorithm thanks to its characteristics to explore as far as possible along each branch before

backtracking. After the partition phase is accomplished, TOSS is capable of generating allocation assignments.

Algorithm 2 is responsible for allocating assignments Θ into available slots or nodes. The principle followed by Algorithm 2 is to seek a node with the lowest run-time workload. By utilizing Eq. 4.1, TOSS is able to collect all run-time workload from all the nodes in a storm cluster. Next, applying *PriorityQueue*, TOSS assigns load to a node with the minimum workload. This phase aims to balance the workload among all the nodes, whereas the first phase intents to reduce inter-executor communication cost.

```
procedure TOSS;
Input  : Initial parameter set α, topology set Ω for allocation
Output: Allocation assignment Θ for each executors
Self-tuning parameter set α;
E ← total number of executor for allocation;
for topology ω_i ∈ Ω do
    for parameter α_i ∈ α do
        E_i ← E * α_i;
        θ_i ← empty set;
        Runs DFS algorithm traverse, finds one chain.;
        C ← head of the chain;
        while θ_i.size() < E_i do
            θ_i add e_k ∈ E in C;
            C ← next component with executor for allocation;
            if C reaches the end of chain then
                C ← back to head;
            end
        end
        Add θ_i into Θ;
        if C remains executor for allocation then
            Leave for next assignment computing iteration;
        end
    end
end
```

Algorithm 1: TOSS scheduling algorithm: partition phase

The objective of the self-tuning mechanism is to optimize the system parameters (e.g., α_i) in a way to evenly dispatch workloads among all the nodes. In particular, given the parameter set $\{\alpha_1, \alpha_2, ..., \alpha_m\}$, the actual workloads are supposed to be evenly split as

procedure TOSS;
Output: Executor allocation on all nodes
Collects runtime workload data;
$pq \leftarrow$ priorityQueue contains all current runtime workload for nodes N;
for *assignment* $\theta_i \in \Theta$ **do**
    $n_i \leftarrow pq.pop()$;
    assign θ_i to node n_i;
end

Algorithm 2: TOSS scheduling algorithm: allocation phase

$\{\frac{W}{M}, \frac{W}{M}, ..., \frac{W}{M}\}$, where $W = \sum_{n \in N} L_n$ and M is the number of parameters in the α set. In order to balance overall running workload, we devise a self-tuning system to judiciously reduce number of executors for heavily loaded assignments, while increasing the number of executors for assignments where load is relatively light. Hence, we employ the linear regression technique [46] to utilize gradient descent for parameter tuning. Gradient descent, an iteratively optimize algorithm, offers an optimal value for a function. In the case of TOSS, the optimal value can be envisioned as evenly distributed workload $\frac{W}{M}$. By keeping track of workload $\{L_1, L_2, ..., L_m\}$ from prior assignment, we can deduct the parameter tuning formula:

$$\alpha_i \coloneqq \alpha_i - \xi \times (\frac{L_i - \frac{W}{M}}{W}), i \neq m \tag{4.5}$$

and

$$\alpha_m = 1 - \sum_{k \in M, k \neq m} \alpha_k \tag{4.6}$$

where ξ is the parameter that controls the tuning rate. It is required to configure a proper value for tuning rate ξ to avoid slow tuning or "hill climbing". Additionally, it guarantees that the new value of α_i locates in the range of $[0, 1)$. Practically, a proper tuning rate is able to improve performance after deploying several topologies.

Fig. 4.5 depicts the system design of the TOSS scheduler. Before making scheduling decisions, TOSS proactively collects all run-time workloads from the worker nodes. In the

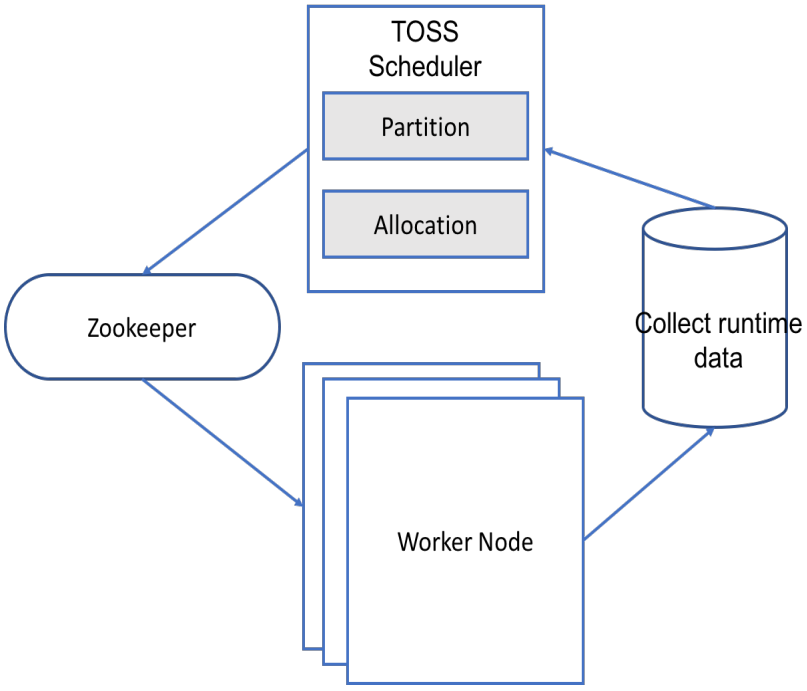

Figure 4.5: The System Design of the TOSS scheduler.

partitioning stage, TOSS is in a position to tune allocation parameters for achieve balanced load. To facilitate parameter tuning, a database is maintained by TOSS to (1) store runtime workloads and (2) keep track of workloads from prior assignments. After wrapping up the partitioning and allocation phases, TOSS embark on dispatching all executors to all the nodes while waiting for the next round of new topology assignments.

4.3 Evaluation and Experimental Results

In this section, we focus on performance of evaluation for the TOSS scheduling algorithm. We validate the TOSS performance by comparing TOSS with the default round-robin scheduler handling various topology types.

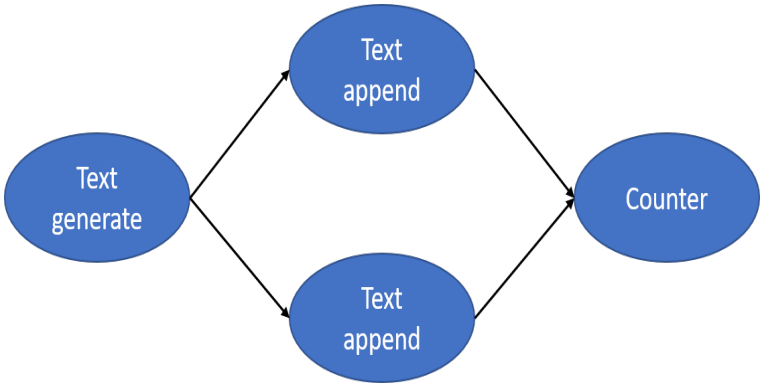

Figure 4.6: The diamond strucutre for network intensive topology.

4.3.1 Performance Metrics

We start our experiments by paying attention to the network-sensitive benchmark, which generates an excessive number of communication tuples among components with light computation overhead. Next, we evaluate the TOSS performance on a general topology that resemble the characteristics of a broad classic topologies. We apply two fundamental metrics to conduct performance evaluation:

- **Latency:** the latency experienced by events to traverse an entire topology.
- **Throughput:** the average throughput that tuples pass through all bolt processes.

We implement the proposed TOSS in Storm 0.8.2 obtained from the Storm's repository on GitHub [8]. All the experiments are carried out on a five-node Storm cluster (i.e., four worker nodes and one master node). The master node plays the role of resource manager, hosting the Nimbus and Zookeeper services. Each worker node hosts four slots with 2.40 GHz Intel Xecon CPU and 1.90 GB RAM. The entire testbed environment is installed on top of Ubuntu Linux 12.0 connected by a 1Gbps network.

4.3.2 Network-intensive topology

We start our testing with a simple topology that consists of one spout and three bolts. The simple benchmark is developed based on the Yahoo storm performance/stress

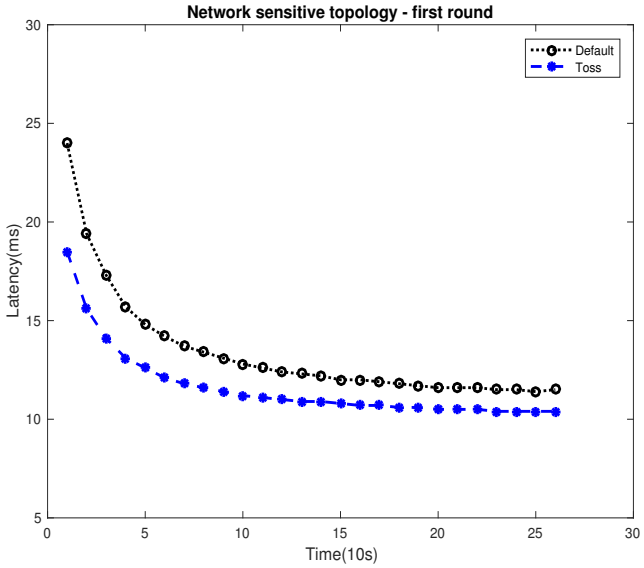

Figure 4.7: The throughput comparison between default scheduler and first TOSS run.

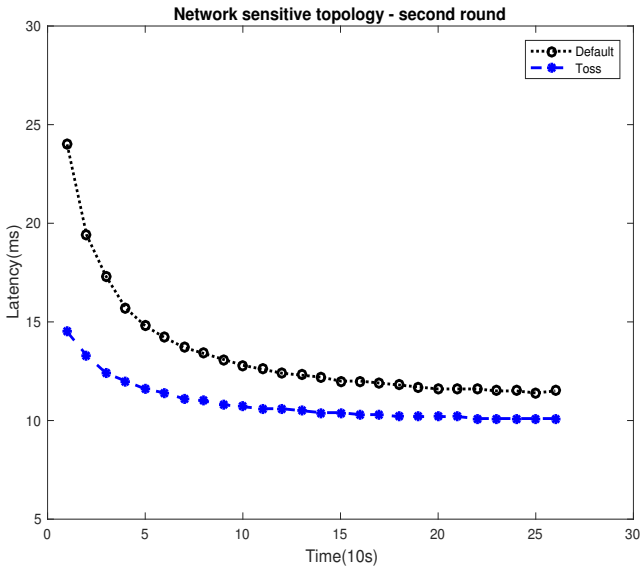

Figure 4.8: The throughput comparison between default scheduler and second TOSS run.

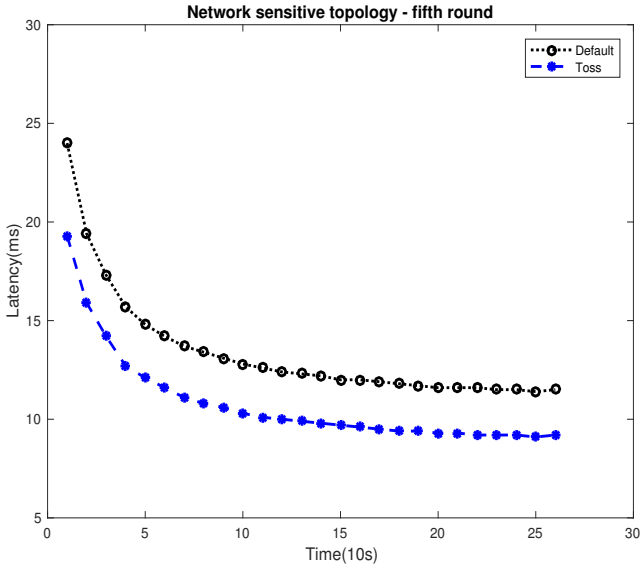

Figure 4.9: The throughput comparison between default scheduler and fifth TOSS run.

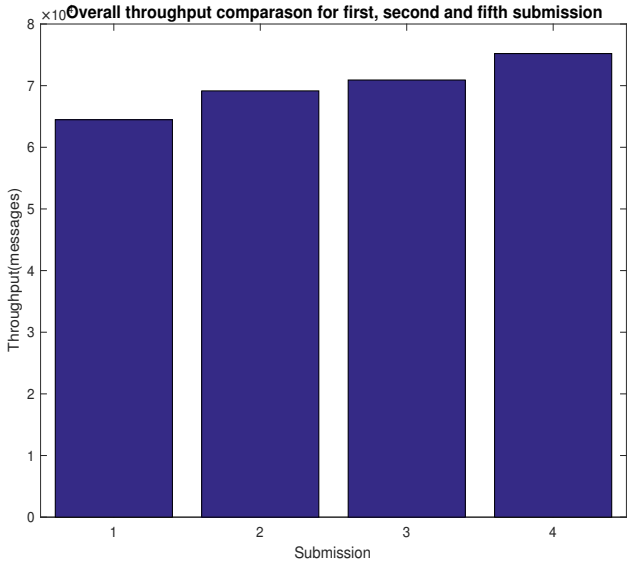

Figure 4.10: The throughput comparison between default scheduler and multiple runs TOSS.

test [64][29]. The entire structure is diamond structure with massive communication load and light computing loads(Shown in the Fig 4.10). The spout repeatedly generates random strings of a fixed size of 10K bytes as input tuples, which go through two middle bolts with little changes. Two middle bolts append a single character in the tail of a received input. The last component in the topology is a counter bolt, which counts the number of received tuples and outputs the counter value when a tuple is processed in the tail. Because the process logic in the bolts is straightforward and simple, the measured throughput heavily depends on the network connectivity in the Storm cluster. Our experiments are running for an average of 5 minutes when the measured latency and average throughput become stabilized and converged.

Recall that TOSS incorporates the self-parameter-tuning mechanism. To assess the effectiveness of the parameter-tuning mechanism, we repeatedly run TOSS to process each topology. We evaluate latency and throughput by comparing multiple executions of TOSS treating the same topology. Before processing each topology, the prior topologies are killed. Observing the TOSS performance in a sequence of processes with respect to each topology, we are able to fully evaluate the behavior of our self-parameter-tuning mechanism in TOSS.

Fig. 4.7, Fig. 4.8 and Fig. 4.9 illustrates the latency of the network-intensive application (a.k.a., *SOL*) governed by TOSS and the default scheduler (i.e., the round-robin scheduler), respectively. We conduct the experiments by submitting the topology multiple times to the Storm cluster. We ignore the latency measurements in the first time window of 20 to 30 seconds due to the high latency caused by executors deployment and initialization. We refer to such an initial interval as a cold start.

The initial constrain vector $[\alpha_1, \alpha_2, ..., \alpha_m]$ is set to $[0.1, 0.2, 0.1, 0.2, 0.1, 0.2, 0.1]$. For example, when α_1 is set to 0.1, the maximal number of executors to be grouped within assignment 1 is 10% of the total number of the executors.

Fig. 4.7 shows latency results of the TOSS and round-robin schedulers under the first topology submission. We observe that the cold start phase lasts for approximately 50 seconds

for both TOSS and round-robin schedulers. During the cold start phase, the latency drops dramatically. The executors are deployed and initialized during the cold start phase, after which the average latency become stabilized. For example, after time 150 seconds, the latency of the round-robin scheduler is settled down at around 11.6 ms, whereas the latency of our TOSS stays at the level of 10.4 ms. Compared to the round-robin scheduler implemented in the Storm cluster, TOSS noticeably reduces the average latency by 10.34%.

It is noteworthy that the initial constraint vector doesn't resemble the actual workload condition of the network-sensitive topology. This problem can be solved by our self-tuning mechanism (see also Section 4.2.2 for details). Fig. 4.8 reveals that after the parameter set (i.e., initial constraint vector) is automatically tuned, the measured latency under the second submission has been significantly shortened. More specifically, the results in the second experiment indicate that the self-tuning system in TOSS is capable to reduce the latency from 11.6 ms to 10.1 ms, which represents a latency reduction of 12.93% compared to the default round-robin scheduler. Fig. 4.9 demonstrates that after the fifth run, the average latency of the Storm application managed by TOSS drops to 9.2 ms, which is 20.69% lower than that of the existing round-robin scheduler.

4.3.3 Rolling WordCount topology

WordCount is a simple MapReduce application running on clusters [20]. The application counts the number of occurrences of each word in a given input set on multiple computing nodes, thereby largely boosting performance with parallelism. Different from the WordCount application on Hadoop clusters, we test a streaming version of the advanced WordCount application referred to as *Rolling WordCount* in our experimental [29]. Different from the conventional WordCount, Rolling WordCount applies a sliding window to pick a valid input range. Any text out of the window is considered as an expired input. Compared with the simple WordCount, Rolling WordCount gains more performance benefits from data

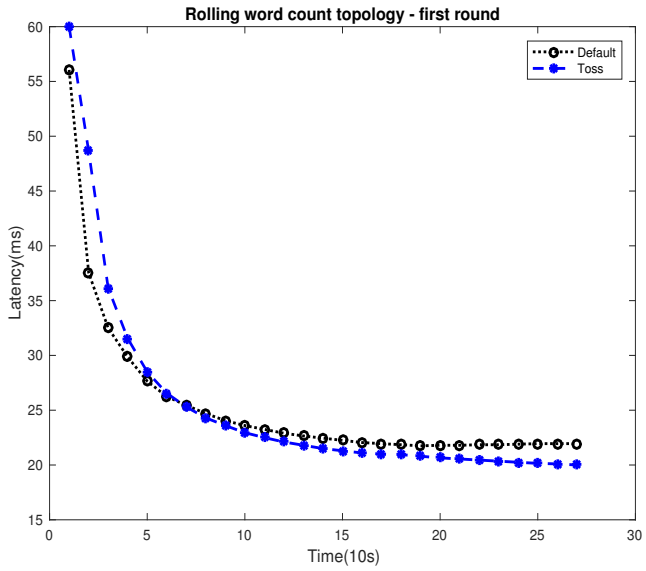

Figure 4.11: The throughput comparison between default scheduler and first TOSS run.

streaming platforms. We conduct a set of experiments using Rolling Word Count, the structure of which is a linear topology with one spout and three bolts. The spout is responsible for originating a massive amount of text context by reading a large local text file. The text readers repeatedly feeds text input to a split bolt, which is charge of splitting each line into words to be pushed into a rolling count bolt. The rolling count bolt increments counters based on distinct input word tuples. Then, the count bolt utilizes the sliding window to filter results, passing all outputs to the last bolt. The last stage of the topology is a file writer bolt, which writes output data into a local file.

The spout consists of four executors, whereas the split bolt and the count bolt each has eight executors. We configure the initial constrain vector $[\alpha_1, \alpha_2, ..., \alpha_m]$ as the one used in the network-intensive topology (see Section 4.3.2. Additionally, we set the sliding window size to one minute.

Fig. 4.11 and Fig. 4.12 shows the latency comparison between our TOSS and the existing round-robin scheduler. After the cold start phase, the latency of the round-robin scheduler stabilizes at approximately 21.95 ms. During the first submission, TOSS reduces the latency

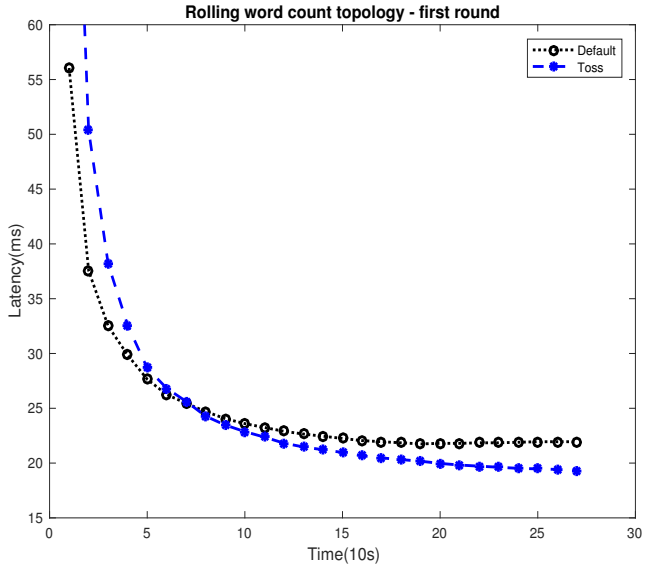

Figure 4.12: The throughput comparison between default scheduler and second TOSS run.

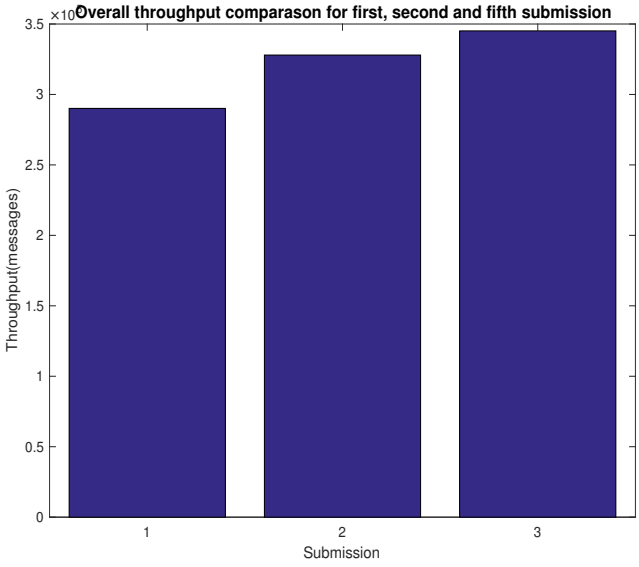

Figure 4.13: The throughput comparison between default scheduler and multiple runs TOSS.

of the round-robin scheduler by 8.5%, which is not as impressive as the performance gain in the case of the network intensive topology (see Section 4.3.2). The rationale behind this observation is that the initial submission with manually configured constraints shortens the average latency by focusing on tight-binding components. The first submission emphasizes on suppressing network traffic overhead, which is not on par with the computing load in the rolling WordCount topology. Nevertheless, TOSS successfully improves the streaming performance in terms of lowering latency. In the subsequent submissions, TOSS is capable of forecasting workload and partitioning all the components into a reasonable number of groups. Fig. 4.12 reveals that the performance is accelerated by tuning the parameter sets to lower latency. The multiple consecutive submissions offer further opportunities to decrease latency and boost throughput. Fig. 4.13 shows that TOSS is able to boost the Storm performance by raising throughput by 12.05% and 18.97% for the first and the second submissions, respectively.

4.4 Discussions

The experimental results elaborated in Section 4.3 indicate that TOSS is conducive to handling various topologies on a streaming data processing platform. Figs. 4.7, 4.8, 4.9, 4.11 and 4.12 illustrate that TOSS exhibits significant performance advantages in terms of lowering latency through an entire topology. In this section, we shed light on a few promising ways of further optimizing the performance of Storm schedulers powered by TOSS.

In our study, we adopted the gradient descent technique to tune the parameter set. The constrain vector $[\alpha_1, \alpha_2, ..., \alpha_m]$ intuitively resembles computing workloads after a few tuning processes are accomplished. Admittedly, the parameter tuning mechanism deserves further improvements. A handful of alternative parameter-tuning approaches are likely outperform the gradient descent method. For instance, Eiben and Smit proposed a conceptual framework for the purpose of parameter tuning. The framework - built on a three-tier hierarchy - utilizes an evolutionary algorithm to generate well-tuned parameter [21]. Nguyen *et al.* optimized

the artificial bee colony algorithm by applying a parameter tuning mechanism [4]. Arcuri and Fraser performed the largest empirical analysis on parameter tuning for *SBSE* (i.e., Search-based software engineering). Although the existing parameter tuning strategies may not perfectly cater to our TOSS scheduler, we are inspired by these strategies to take the parameter tuning mechanism in TOSS to the next high level. The drawback of the existing parameter tuning mechanisms lies in their incompetence in tuning automation. The goal of parameter self-tuning is to enable schedulers to automatically configure constrain vectors without a manual tuning procedure. Unfortunately, the gradient descent scheme depends on parameter ξ to control tuning rate, which is not managed by the current tuning system. Such a shortcoming can be resolved by replacing gradient descent with an advanced parameter tuning techniques.

We intend to improve TOSS by further investigating the constrain vector $[\alpha_1, \alpha_2, ..., \alpha_m]$, which assists TOSS to partition executors into assignments. The parameter set - reflecting runtime workload, allows TOSS to balance computation cost among all the nodes in a cluster. The self-tuning system calibrates workload discrepancy between actual workload and average workload. The self-tuning mechanism is unable to change the size of parameter set. In our study, the size of parameter set m leads to the total number of assignments. If the self-tuning mechanism overlooks the importance of set size m, such a constant size may fail in giving rise to optimal topology deployments in some cases. Concerning the flexibility of the self-tuning mechanism, we seek to exploit novel ideas of dynamically adjusting size m for the constrain vector in TOSS.

4.5 Summary

Storm is an emerging technology in the field of streaming data process. The wide range of use cases motivate us to propose an efficient scheduler for better performance. In the storm part of the dissertation, firstly, we introduced a background of data processing and provided three motivations for better scheduling performance. Furthermore, we listed bunch

of related researches conducted by scholars. The followed sections show the challenge associated with existing scheduler. Then we presented the design, implementation and evaluation of TOSS comparing to default round-robin scheduler. The gist of TOSS is to reduce the latency by grouping the executors with tight bind executors and to balance workloads by forecasting workload with self-tuned parameters. In the evaluation section 4.3, we conducted real experiments in a Storm cluster using stream application such as SOL(network sensitive benchmark) and rolling word count. Experiment results show that our TOSS can achieve roughly 20% latency decrements after multiple rounds. Additionally, average throughput can be boosted by around 24%. Moreover, we provide more room for further optimization based on TOSS. We firmly believe that the suggestions we purposed in Section 4.4 is able to improve the efficiency and flexibility.

Chapter 5

Conclusions and Future Work

In this dissertation, we have profiled energy usage of four distinct setups on web service clusters and proposed a new topology-based scheduler for Apache Storm clusters. In this chapter, we make the conclusion of dissertation by illustrating our main contributions and providing future works for extending our researches. Section 5.1 summaries all the main contributions in our researches. In the next Section 5.2, we provide future visions about the extensions and combinations of our past researches.

5.1 Main Contributions

As the IT world enters the era of data, the amount of data that's being created and processed on a global level is becoming inconceivable. The amount of data leads to the increasing demands of data center scale. While building up data centers, two main issues are concerned: computation performance and energy efficiency. The major goal of scaling data center is to speed up the data process. Emerging technologies are applied to scale out data center for high computation performance. Nevertheless, the increment of data center scale reveals an critical issue: high energy cost. Maintaining low operational of big data centers is becoming one primary challenges for building data centers. In order to address these two major concerns, in this dissertation, we proposed two mechanisms to address energy cost and performance issues respectively.

5.1.1 Energy Cost Profiling of Web-Service on Clusters

In the first part of our dissertation, we investigated the energy cost and computation performance of various combinations among pc nodes and wimpy nodes. Wimpy node is an

outstanding technology which is in the position of trading part of computation power to low energy cost. The major goal is to inspect the potentials of different setups and to apply each setup for different situation. Four setups we investigated are listed as following:

- Homogeneous pc setup
- Homogeneous wimpy setup
- Pc web server and wimpy database server
- Wimpy web server and pc database server

Our experimental results revealed characteristics of all four setups. In our dissertation, we provide ideas to adjust the composition of data cluster for different primary goals. After energy cost evaluation with TPC-W benchmark, we make the conclusions that comparing to traditional nodes, the wimpy node owns the advantage in terms of energy consumption. On the other side, traditional pc nodes dominate wimpy nodes for handling heavy computation workloads. No matter how the data center is built, wimpy nodes are undoubtedly becoming the performance bottleneck.

For heterogeneous setup investigation, two setups show contrary candidacy for building an energy efficient data center. The combination of pc web server and wimpy database server is a competitive candidate, whereas the setup with wimpy web server and pc database server provides the worst performance for both computation performance and energy consumption.

5.1.2 Topology-based Scheduling Policy for Apache Storm Cluster

Streaming data processing is beneficial in most scenarios in people's daily lives. Apache Storm is an outstanding tool with excellent performance and fault tolerance. The second part of dissertation introduces a brand new scheduling policy for Apache Storm Cluster by reducing communication cost and balancing the workloads with a self-tuning parameter mechanism.

Our scheduler(Short as TOSS) algorithm contains two major phases: partition and allocation. The first phase is responsible to split the topology structure into different assignments by defining the groups with potential communications. A set of parameters $[\alpha_1, \alpha_2, ..., \alpha_m]$ is utilized by partition phase for limiting the maximum number of executors which can be allocated in each assignment. The parameter constrains are initialized by developers and later tuned by self-tuning system. The purpose of self tuning is to reflect the actual workloads for topology. In the second phase, the allocation algorithm collects all runtime workloads from nodes in the cluster. Based on the running workloads, TOSS picks the set of nodes with relatively low workloads and allocate assignments to nodes one by one. The allocation phase benefits the workload balance in the Storm clusters.

We have evaluated our implemented scheduling policy with two storm benchmark: network sensitive topology and rolling word count topology. Our experimental results show that our scheduler TOSS achieves significant performance improvement with lower latency and higher throughput comparing to default round-robin scheduler.

5.2 Future Work

After addressing the challenges on both hardware and software levels, we came across several interesting problems that are not solved yet. In this section, we provide some open issues which require further investigations. Furthermore, we present some ideas that combine and extend our researches for future work.

5.2.1 Integrate Scheduler on Energy Efficient Clusters

In the dissertation, we investigated researches in energy efficiency cluster setup and topology based scheduler on hardware and software level respectively. The purpose of the researches are focusing on energy efficiency and computation performance. There is strong possibility that integrates these two ideas into one data cluster. The drawback of our energy

profiling research is that we stand in the position of trading part of computation performance for lower energy consumption. This disadvantage can be offset with the integration of scheduler, since our TOSS scheduler is capable to improve computation performance. The combination of cluster setup choices and scheduler is able to reduce the power consumption without sacrificing computation performance.

However, this simple integration ignores the different complexities of two clusters, for the simple reason that web service applies three-tier architecture, while TOSS is implemented in Storm cluster with master-slaves structure. The incoherence can be corrected by adjusting two scheduler on both web servers and database servers. By following the same scheduling policy, the master node is able to dispatch working executors for web server groups and database server groups separately. Additionally, a request balance layer may be required for balancing request sent from client to web servers. The power consumption produced by the cluster with balanced workloads is higher than the energy cost consumed by the cluster with overloaded nodes.

5.2.2 Machine Learning Mechanism

Recall that TOSS applies a self-tuning mechanism by exploiting gradient descent methodology, the gist of tuning system is to explore the precise parameter sets which reflects the actual workloads in the cluster. Essentially, TOSS incorporates the self-tuning mechanism to predict the workloads about next topology for load balancing.In order to achieve workload prediction, some other machine learning methodologies are able to improve self-tuning performance. There are plenty of existing machine learning algorithms to acquire more precious parameters with data collected from last assignment allocations. We firmly believe that the performance and efficiency of scheduler can be further improved by employing machine learning technologies to analyze topology structure.

Moreover, machine learning methods can be applied in the cluster setup building as well. The purpose of cluster energy profiling is to seek corresponding cluster setups for different

workloads with better balance between performance and energy cost. We firmly believe that an effective workload analysis tool is able to seek a proper cluster setup. Machine learning algorithms facilitate us to continuously analyze workloads and predict the matched cluster setup. It is worthwhile to split a data center into small groups of clusters with distinct setups. For instance, a data center can be split into two camps: a micro homogeneous wimpy cluster, and a micro heterogeneous pc web and wimpy database cluster. An additional layer is responsible to apply machine learning algorithms and to feed requests into corresponding micro clusters. By doing so, all micro clusters handle different groups of requests with distinct characteristics. Therefore, the cluster is able to conserve part of energy cost, while the performance is kept in a relatively high level.

5.3 Conclusion

The dissertation has presented energy usage profiling and topology based scheduler for data centers. The experimental results show the improvement provided by our approaches. The first part of the dissertation provides four distinct cluster setup to match different cluster requirements. By following the energy profiling cost, we moved our research to the infrastructure layer and proposed a topology based scheduling policy. TOSS scheduler owns advantages with lower latency and higher throughput comparing to default round-robin scheduler. In particular, the self-tuning parameter system we implemented keeps improving performance by predicting actual workloads for submitted topology. Moreover, based on our investigations, future works is able to extend our existing researches by combining advantages of the two researches. In a scenario where the cluster requires both low energy consumption and relatively high computation performance, the cluster designer is able to adjust TOSS scheduling policy for task scheduling, while the cluster setup researches helps designers to pick a matched cluster setup for different workloads. Furthermore, machine learning algorithms facilitates developers to improve efficiency and effectiveness of data process. There

are still some rooms to further improve the decrements of energy cost and increment of computation performance.

References

[1] Amazon 2000. http://www.fool.com/news/2000/wmt001127.htm.

[2] Tpc-w benckmark specification. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf.

[3] *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[4] B. Akay and D. Karaboga. Parameter tuning for the artificial bee colony algorithm. *ICCCI*, 2009:608–619, 2009.

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services*. Springer, 2004.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 1–14, 2009.

[7] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[8] Apache. Storm github repository, 2015.

[9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[10] G. Bell, T. Hey, and A. Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[11] A. Brown, S. Johnston, and K. Kelly. Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation*, 2002.

[12] R. Brown et al. Report to congress on server and data center energy efficiency: Public law 109-431. *Lawrence Berkeley National Laboratory*, 2008.

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[15] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 217–228, 2009.

[16] G. Chen, K. Malkowski, M. T. Kandemir, and P. Raghavan. Reducing power with performance constraints for parallel sparse applications. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*, 2005.

[17] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[18] B. Chun, G. Iannaccone, G. Iannaccone, R. H. Katz, G. Lee, and L. Niccolini. An energy case for hybrid datacenters. *Operating Systems Review*, 44(1):76–80, 2010.

[19] B. Cui, J. Jiang, Q. Huang, Y. Xu, Y. Gui, and W. Zhang. Pos: A high-level system to simplify real-time stream application development on storm. *Data Science and Engineering*, 1(1):41–50, 2016.

[20] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[22] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM, 2010.

[23] S. Elnikety, E. M. Nahum, J. M. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 276–286, 2004.

[24] W. Feng and K. W. Cameron. The green500 list: Encouraging sustainable supercomputing. *IEEE Computer*, 40(12):50–55, 2007.

[25] R. Filgueira, R. F. da Silva, A. Krause, E. Deelman, and M. Atkinson. Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science. In *Proceedings of the 7th International Workshop on Data-Intensive Computing in the Cloud*, pages 1–8. IEEE Press, 2016.

[26] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016.

[27] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010.

[29] Intel-hadoop. Storm benchmark, 2016.

[30] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[31] M. Jazayeri. Some trends in web application development. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 199–213, 2007.

[32] P. Karunaratne, S. Karunasekera, and A. Harwood. Distributed stream clustering using micro-clusters on apache storm. *Journal of Parallel and Distributed Computing*, 108:74–84, 2017.

[33] A. Kaur and A. Kaur. Energy management models for efficient cloud environment: A review. *International Journal of Advanced Research in Computer Science*, 6(6), 2015.

[34] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, et al. Towards analytics aware ontology based access to static and streaming data. In *International Semantic Web Conference*, pages 344–362. Springer, 2016.

[35] J. Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[36] D. Kumar, J. C. Bezdek, S. Rajasegarar, M. Palaniswami, C. Leckie, J. Chan, and J. Gubbi. Adaptive cluster tendency visualization and anomaly detection for streaming data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(2):24, 2016.

[37] W. Lang, J. M. Patel, and S. Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN 2010, Indianapolis, IN, USA, June 7, 2010*, pages 47–55, 2010.

[38] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo. A performance study of big data on small nodes. *PVLDB*, 8(7):762–773, 2015.

[39] M. Malik and H. Homayoun. Big data on low power cores: Are low power embedded processors a good fit for the big data workloads? In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, pages 379–382, 2015.

[40] M. A. Manzoor and Y. Morgan. Network intrusion detection system using apache storm. *Probe*, 4107:4166, 2017.

[41] E. E. Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, Carnegie-mellon univ Pittsburgh PA school of computer science, 2009.

[42] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE, 2008.

[43] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[44] D. A. Menascé. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, 2002.

[45] G. D. F. Morales and A. Bifet. Samoa: scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16(1):149–153, 2015.

[46] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[47] H. R. Patterson III, Z. Wang, and M. L. Huang. Data reconstruction in distributed data storage system with key-based addressing, Feb. 25 2016. US Patent 20,160,055,054.

[48] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[49] S. Ran. A model for web services discovery with qos. *ACM Sigecom exchanges*, 4(1):1–10, 2003.

[50] R. Ranjan. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014.

[51] M. Rychly et al. Scheduling decisions in stream processing on heterogeneous clusters. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*, pages 614–619. IEEE, 2014.

[52] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[53] A. Shaout and P. McAuliffe. Job scheduling using fuzzy load balancing in distributed system. *Electronics Letters*, 34(20):1983–1985, 1998.

[54] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti. A topology-based scaling mechanism for apache storm. *International Journal of Network Management*, 27(3), 2017.

[55] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[56] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[57] G. Valentini, W. Lassonde, S. U. Khan, N. Min-Allah, S. A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, H. Li, A. Y. Zomaya, C. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, and P. Bouvry. An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, 16(1):3–15, 2013.

[58] M. Vanni, S. E. Kase, S. Karunasekara, L. Falzon, and A. Harwood. Rapid: real-time analytics platform for interactive data-mining in a decision support scenario. In *SPIE Defense+ Security*, pages 102070L–102070L. International Society for Optics and Photonics, 2017.

[59] V. Vasudevan, D. G. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient cluster computing with FAWN: workloads and implications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking, e-Energy 2010, Passau, Germany, April 13-15, 2010*, pages 195–204, 2010.

[60] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos: Enhancing bittorrent for supporting streaming applications. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–6. IEEE, 2006.

[61] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Transactions on Networking*, 24(1):190–203, 2016.

[62] M. Wiederstein and M. J. Sippl. Prosa-web: interactive web service for the recognition of errors in three-dimensional structures of proteins. *Nucleic acids research*, 35(suppl_2):W407–W410, 2007.

[63] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544. IEEE, 2014.

[64] Yahoo. Storm performance test, 2015.

[65] N. Yigitbasi, K. Datta, N. Jain, and T. Willke. Energy efficient scheduling of mapreduce workloads on heterogeneous clusters. In *Green Computing Middleware on Proceedings of the 2nd International Workshop*, page 1. ACM, 2011.

[66] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical report, Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, 2009.

[67] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[68] Q.-Y. Zhou and U. Neumann. A streaming framework for seamless building reconstruction from large-scale aerial lidar data. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 2759–2766. IEEE, 2009.

[69] I. Zliobaite and B. Gabrys. Adaptive preprocessing for streaming data. *IEEE transactions on knowledge and data Engineering*, 26(2):309–321, 2014.