applied sciences

MDPI

Article

A Framework for Proactive Resource Provisioning in IaaS Clouds

Yi-Hsuan Lee, Kuo-Chan Huang, Cheng-Hsien Wu, Yen-Hsuan Kuo and Kuan-Chou Lai *

Department of Computer Science, National Taichung University of Education, Taichung 40306, Taiwan; yslee@mail.ntcu.edu.tw (Y.-H.L.); kchuang@mail.ntcu.edu.tw (K.-C.H.); bcs103107@gm.ntcu.edu.tw (C.-H.W.); bcs104102@gm.ntcu.edu.tw (Y.-H.K.)

* Correspondence: kclai@mail.ntcu.edu.tw; Tel.: +886-4-2218-3810

Received: 30 June 2017; Accepted: 26 July 2017; Published: 31 July 2017

Abstract: Cloud computing is an emerging technology for rapidly provisioning and releasing resources on-demand from a shared resource pool. When big data is analyzed/mined on the cloud platform, the efficiency of resource provisioning would affect the system performance. This work proposes a framework for proactive resource provisioning in IaaS (Infrastructure as a Service) clouds to improve system performance. The proposed framework consists of the virtual cluster computing system, the profiling system, the resource management system, and the monitoring system. In this framework, the over-commit mechanism is applied to improve resource utilization. Furthermore, a proactive task scheduling approach is also present to prevent the postponement of tasks in critical stages, especially when the amount of aggregated resources requested by virtual machines exceeds that of available resources on the over-committed physical machines. Experimental results show that the over-commit approach indeed improves the resource utilization. However, when the degree of applying the over-commit approach increases, the burden of this proposed approach also conceivably increases. Therefore, the proposed framework further applies the proactive task scheduling approach to execute the time-critical tasks earlier to shorten the processing time. A small-scale cloud system including 3 servers is built for experiments. Preliminary experimental results show the performance improvement of our proposed framework in IaaS clouds.

Keywords: over-commit; resource provisioning; IaaS; cloud computing; critical path; task scheduling; data stream computing

1. Introduction

Cloud computing is an emerging technology for rapidly provisioning and releasing resources on-demand from a shared pool of configurable computing resources. When the quasi-continuous data stream is captured and collected by a wide variety of devices, these collected data have to be analyzed and mined for identifying significant patterns. As the volume of these collected data grows rapidly, traditional computing systems may not be powerful enough to process these huge-volume and high-velocity data. Therefore, cloud computing becomes a candidate to provide a scalable computing platform for processing these big data. In order to meet the real-time processing requirement of big data streaming applications, cloud computing systems have to provide adequate computing resources to prevent the violation of service level agreements (SLAs). Several new data stream processing engines have been developed recently, e.g., Apache Storm [1], Apache Spark [2], Apache Samza [3], and Apache Flink [4].

Due to the resource virtualization technique, the cloud computing system could provision scalable resources as services dynamically. In the cloud computing environment, several co-hosting jobs or applications share these virtualized resources. However, when lots of service requests come together in a short time or a service has massive resource requirements, the competition of accessing available

resources may result in the unexpected overload or oversubscription situations. Inefficient resource sharing among co-hosted jobs or applications often causes performance degradation. In order to achieve predictable performance, resource provisioning becomes one of the most important issues in developing cloud computing systems.

Fairness is a common goal of resource provisioning approaches, but these approaches generally fail to achieve high resource utilization. When the high resource utilization becomes the main goal, these resource provisioning strategies still have to prevent the violation of other SLAs defined by users. Hence, this study proposes a framework to address high resource utilization under the consideration of the fairness of resource provisioning.

Another important challenge in efficiently using cloud resources is to develop task scheduling algorithms for data stream processing applications. In general, the problem of minimizing makespans by task scheduling has been proven as NP(Nondeterministic Polynomial)-complete. Therefore, most task scheduling approaches adopt heuristic algorithms, because solving such NP-complete problems in the polynomial time complexity is very difficult. Multiple objects could be defined in the task scheduling algorithms to satisfy clients' demands in cloud computing systems, and the minimal makespan is one of major factors in determining system performance.

In this framework, a proactive task scheduling algorithm is proposed for data streaming applications on IaaS clouds. The over-commit mechanism is applied to improve resource utilization, in conjunction with a resource estimating mechanism to reduce the resource over-provisioning. This proposed scheduling approach takes the tasks in critical stages into consideration, and executes these tasks earlier to shorten the makespan of the entire application.

A small-scale cloud system including 3 physical servers is built for experiments. The cloud OS is OpenStack (Grizzly), and Apache Mesos is used to supply the virtual clusters in which virtual resources may be obtained from multiple physical servers. Apache Spark is adopted to be the data processing platform. Three micro-benchmarks are used to evaluate the performance of the proposed approach. Preliminary experimental results demonstrate the performance improvement of the proposed framework.

This paper is organized as follows. Section 2 discusses related works. Section 3 introduces the system framework. The profiling system, the over-commit mechanism, and the critical task scheduling approach are described in some detail in Sections 3.1–3.3. Section 4 presents experimental results. The final section gives the conclusions and future works.

2. Related Works

The technology of the Internet of Things (IoT) [5] allows network devices to sense and collect data from the world around us, and these data could be mined and applied for different applications. In general, the IoT is a network of connected devices which communicate with each other to perform certain tasks. Related applications include location tracking, energy saving, transportation, safety control, and so on. However, the rapid growth of the IoT also increases the rate at which data is generated and results in big data; in the meantime, time-critical data streaming [6,7] applications have the time-limit requirement. Therefore, the cloud computing system with sophisticated resource management is a good candidate to handle these time-critical applications.

Cloud computing [8] applies the virtualization technology to provide diverse services by the pay-as-you-go model. There are various studies on resource management in cloud computing due to its importance. Mashayekhy et al. [9] address the physical resource management problem by considering diverse physical resource types. Their work proposes an approximate winner determination algorithm to decide the provisioning of virtual machines on specified physical machines. Liu et al. [10] propose a resource management framework to guarantee the quality of service (QoS) in cloud computing. When the service workload increases rapidly, their work adopts an aggressive resource provisioning strategy to match the growing performance requirement as soon as possible. Experimental results show that their work could achieve better performance. Wang et al. [11] propose a multi-resource

allocation strategy in the cloud computing system with heterogeneous resources. Their approach ensures that no user prefers the allocation of other users to achieve the fairness of resource allocation. Simulation results show that their approach outperforms the traditional slot-based scheduler.

Baldan et al. [12] propose a methodology to detect the asymmetrical nature of the forecasting problem and to forecast the workload in time series. Several realistic workload datasets from different datacenters are applied to evaluate the system performance. Do et al. [13] propose a profiling system for the decision making of job scheduling and resource allocation. Their approach adopts the canonical correlation analysis method to identify the relationship between application performance and resource usage.

Traditional studies about resource allocation mainly focus on fair resource sharing among virtual machines but rarely consider behaviors of virtual CPUs (vCPUs). Some previous works apply the over-commit approach to improve the resource utilization. Ghosh et al. [14] propose an over-commit mechanism according to the aggregate resource usages by the statistical analysis approach. Their approach tries to improve the resource utilization and to estimate the risks associated with the over-commit approach. Zhang et al. [15] propose a novel virtual machine (VM) migration mechanism in over-committed clouds to balance host utilization. Experimental results show that the number of VM migrations is minimized and the risk of overload is reduced. Chen et al. [16] show the performance degradation of the communication-intensive or I/O-intensive applications in the over-committed situation. Their work proposes a scheduling mechanism specifically for communication/IO-intensive applications, and experimental results show that the performance could be improved in over-committed situations. Juhnke et al. [17] proposes a novel scheduling algorithm for BPEL workflow applications. Emeakaroha et al. [18] present a cloud management infrastructure to provide resource monitoring and management for workflow applications. Janiesch et al. [19] propose an approach for optimizing cloud-aware business process by providing computational resources based on process knowledge. Li and Venugopal [20] present an approach to provision resources and manage instances of web applications for handling volatile and complex request patterns.

In cloud computing, task scheduling is also a very important issue in improving system performance. Gupta et. al. [21], presents a holistic viewpoint to the suitability of high performance computing applications running on clouds. The authors also propose optimizing approaches to improve the performance of HPC applications according to the execution patterns. Simulation results show the significant improvement in average turnaround time. Tasi et al. [22] propose a hyper-heuristic scheduling algorithm to obtain schedules by diversity detection and improvement detection dynamically. Experimental results show that the schedule length could be reduced significantly. Rodriguez and Buyya [23] present a resource provisioning and scheduling approach for scientific workflows to meet users' QoS requirements on clouds. Their approach adopts the meta-heuristic optimization technique to minimize the workflow execution cost with satisfying deadline constraints. Simulation results show the performance improvement of their approach. Kanemitsu et al. [24] propose a clustering-based task scheduling algorithm for applications represented using Directed Acyclic Graph (DAG). The proposed approach adopts two-phase scheduling. In the first phase, the characteristics of the system and applications are considered to determine the number of necessary processors. In the second phase, tasks are clustered and assigned to minimize the schedule length. Experimental results show the performance improvement in terms of schedule length and efficiency. Zhang et al. [25] present an online stack-centric scheduling algorithm for cloud brokers to schedule multiple customers' resource requests. The proposed approach tries to exploit the volume discount pricing strategy, and simulation results present the superiority of their approach. Chen [26] introduces a two-phase approach taking system reliability into consideration. The proposed approach adopts a linear programming strategy to obtain the minimal makespan, and uses a task-duplication strategy to improve system efficiency. Experimental results show the improvement in terms of schedule length ratio, reliability, and speedup.

3. System Framework

As shown in Figure 1, the proposed system framework consists of the data processing engine, the virtual cluster computing system, the cloud platform, and the physical hardware resources. In the virtual cluster computing system, there are the monitoring module, the resource management system, and the profiling module. In the resource management system, there are the over-commit mechanism and the resource estimating module. The virtual cluster computing system provides the computing environment of virtual clusters. The computing node in the virtual cluster is the virtual machine provisioned by the cloud platform. Several Spark data processing systems are executed on the virtual cluster computing system, which are integrated with the proposed proactive task scheduling algorithm aiming for shorter processing time. The historical log of resource usage is generated by the profiling system, and the current resource usage is captured by the monitoring system. All virtual resources are efficiently managed by the resource management system, in conjunction with the resource usage estimating module and the over-commit mechanism.

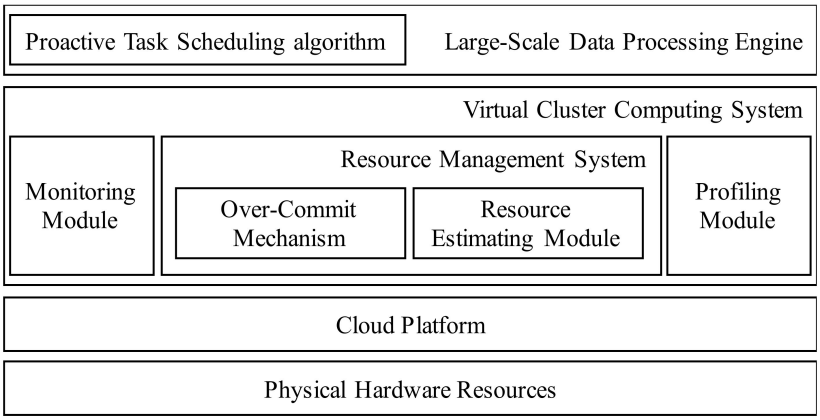

Figure 1. Proposed system framework.

When Apache Spark starts to execute an application, the profiling system first checks whether the profiled log has existed. If the profiled log has not existed, the profiling system would be activated to generate the corresponding profiled log for this newly-arrived application. In the profiled log, Spark could find the execution pattern of this application, such as the number of stages, the number of tasks within each stage, and data dependencies between stages. In the meantime, the resource management system would inform Spark about available resource information, and efficiently manage virtual resources based on the over-commit mechanism to improve the resource utilization. The goal of the resource estimating module is to estimate the resource demand of the launched application according to its historical executing log. Considering the current resource usage captured by the monitoring system, the over-commit mechanism would trigger the resource provisioning beyond the actual requirements in order to improve the resource utilization. The proactive task scheduling algorithm would decide an appropriate sequence to execute stages according to a profiled execution pattern and resource information, and aims to shorten the overall processing time.

3.1. Profiling System

The profiling system is activated to collect the execution pattern for analyzing the execution behavior and workload characteristics of applications. Based on this profiling information, the resource management could adjust resource provisioning to achieve the optimal system performance. In this study, a proposed profiling system is built, dedicated for Apache Spark. However, the approach could be modified to suit other data processing engines. The main purpose of the proposed profiling system is to capture the dataflow relationship of Spark applications in execution, and display the relationship among Spark Resilient Distributed Datasets (RDDs) in the DAG form.

Big data streaming applications are continuously executed in general. Hence, if the execution pattern could be collected in advance, the resource provisioning could be optimized. The proposed profiling system is activated when a Spark application is launched for the first time. The execution patterns, including dependency among RDDs, the level of stages, the size of partitions, and so on, are all recorded in the dot file. According the logged execution pattern, the profiling system could provide some useful information for decision making of resource provisioning, such as the dataflow among RDDs, the task memory usage, I/O access pattern, and so on.

There are three important system components in Spark, including Driver (which is responsible to the creation of SparkContexts for job execution), Executors (which are responsible to the execution of tasks scheduled by Driver), and Cluster Manager (which communicates with the cluster platform). The main components in Spark Driver include SparkContext, DAGScheduler, TaskScheduler, SchedulerBackend, BlockManager, and so on. The profiling system captures information in four different levels during the execution of applications. In the job level, the profiling system captures the information of job.finalStage. The information of stage id, stage.parent and stage.rdd, are collected in the stage level. In the task level, the profiling system captures the information of tasks and taskEndReasons. The data of RDD.id, RDD.dependencies, RDD.type, the location of program codes about RDDs, and other related information are collected in the RDD level.

Figure 2 shows the data structure in the dot file for the micro-benchmark SparkTC. In the upper half of the dot file it records the stage information. For example, the stage ID of the first stage is "Stage5", and there are two RDDs with IDs #0 and #1. The second stage, "Stage7", contains six RDDs with IDs #2, #3, #4, #5, #6, and #7. The relationship between RDDs is recorded in the bottom half of the dot file. For example, the transformation of GoGroupedRDD is applied to RDD#2 at line 67 with two corresponded dependencies: one is between RDD#0 and RDD#2, and the other is between RDD#1 and RDD#2. A shuffled relationship between RDDs would be marked as a red line.

Figure 3 presents the dataflow among RDDs for the micro-benchmark SparkTC. The red line indicates the shuffle dependency among RDDs, and the black line indicates other dependency. A larger (or smaller) rectangle represents a stage (or RDD). For a RDD, information including its id, type, and transform instructions are all displayed in the rectangle.

```
digraph {
  node[shape=rectangle]

  subgraph cluster5 {
  graph[labeljust=right]
  label="Stage5"
    0
    1
  }

  subgraph cluster7 {
  graph[labeljust=right]
  label="Stage7"
    7
    6
    3
    2
    5
    4
  }

  2 [
    label="#2: CoGroupedRDD\njoin at SparkTC.scala:67"
  ]
  2 -> 0 [color=red];
  2 -> 1 [color=red];

  3 [
    label="#3: MapPartitionsRDD\njoin at SparkTC.scala:67"
  ]
  3 -> 2;
```

Figure 2. Data structure in the dot file of a profiled SparkTC benchmark.

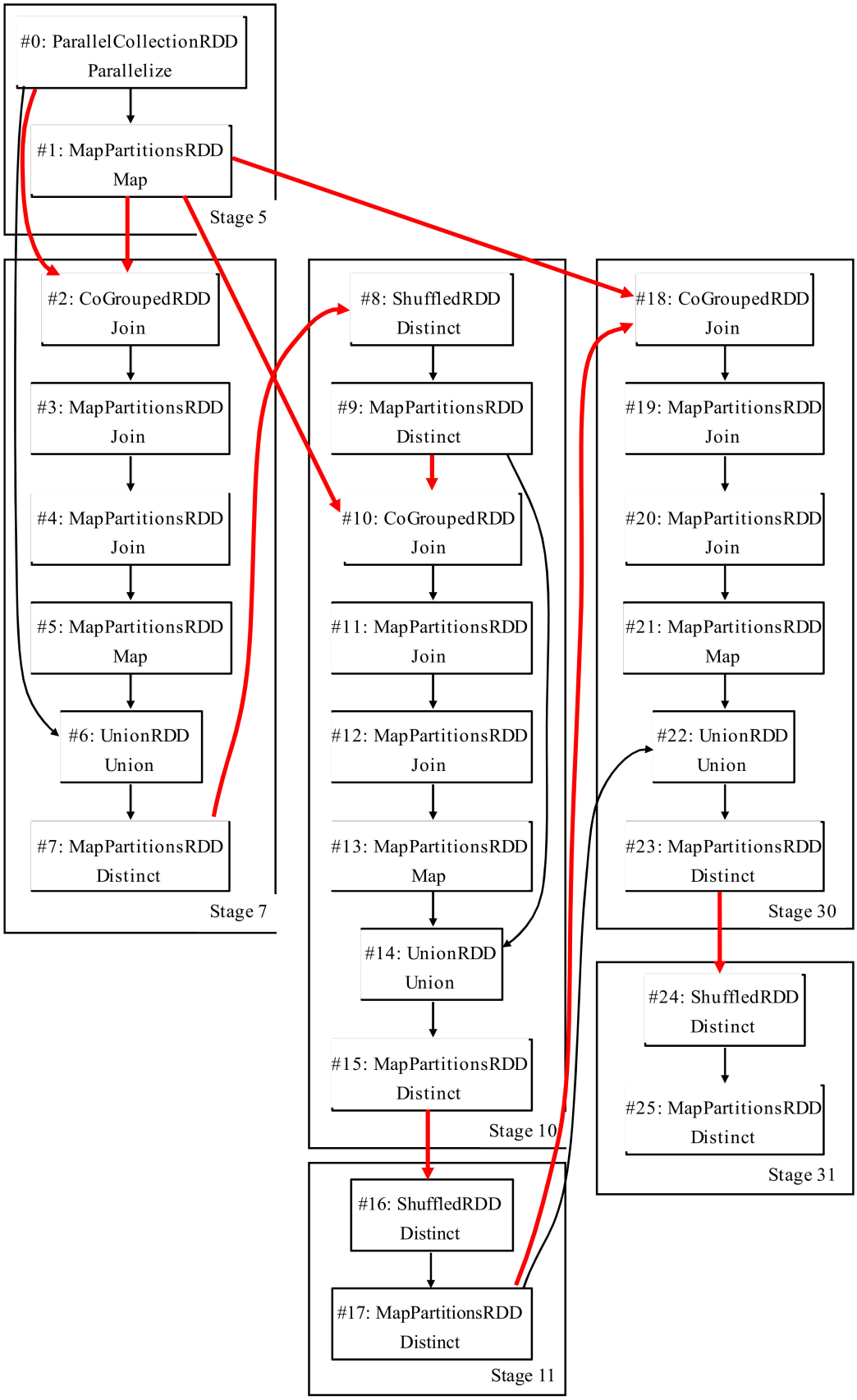

Figure 3. RDD flow of a profiled SparkTC benchmark.

3.2. Over-Commit Mechanism

Resource allocation generally focuses on fairness and efficiency in cloud computing systems. Domain Resource Fairness (DRF), which is the default approach implemented in the Apache Mesos system, tries to obtain the fair allocation of multiple resource types. But the fair allocation usually results in poor resource utilization, which means the optimal resource management could not be achieved. Therefore, this framework proposes an over-commit mechanism for physical resources to enhance the resource utilization.

In virtual cluster computing systems, the master node hosts the processing and storage management services, and data is essentially stored and processed on slave nodes. In general, when a node is added as a slave to the virtual cluster computing system, this slave node has to register and provide its resource information to the master node. During this process, the over-commit mechanism would be triggered, and the slave node would provide dishonest information that it has more physical resources. That is, the number of virtual machines recorded in the master node exceeds the number of virtual machines which all slave nodes could actually provide. Therefore, when an application requests virtual resources for execution, the master node would offer more resources than expected. These over-committed virtual machines have to compete for actual physical resources during execution, and the resource utilization is supposed to be improved.

Although applying the over-commit approach is beneficial for achieving better resource utilization, there are still some risks. When the amount of aggregate resources requested by virtual machines exceeds that of available resources on the over-committed physical machines, the service response time of some requests may be significantly lengthened. Hence, in order to prevent the resource over-provisioning, a mechanism is proposed to estimate an appropriate number of over-committed resources from historical executing logs.

The number of over-committed resources is decided by a parameter called degree of over-commit. A degree of over-commit of 1 means no over-committed resources are provisioned. If the degree of over-commit is k, the number of resources are over-committed k times. In the proposed system framework, a dedicated historical executing log is maintained for each application. Once an application is launched, a pair $P_i(D_i, T(D_i))$ is recorded where D_i is the degree of over-commit and $T(D_i)$ is the execution time. Information recorded in these historical executing logs would be referenced to estimate the appropriate degree of over-commit.

Figure 4 shows the pseudo-code of the proposed resource estimating algorithm. When an application is launched for the first or second time, since the information recorded in the corresponded historical executing log is limited, the appropriate degrees of over-commit D_{app} would be directly set to 2 and 4, respectively. When an application is launched more than twice, D_{app} is estimated according to the binary-search-based approximation algorithm. If the largest D_i that has been set results in the minimal execution time, D_i is simply doubled to provision more over-committed resources. After several iterations, this estimating mechanism is supposed to find a stable D_{app}.

```
Input: P_i(D_i, T(D_i))          /* Recordings in the historical executing log */
Output: D_app                    /* Appropriate degree of over-commit */

1.  Degree ← All D_i recorded in the historical executing log
2.  if |Degree| == 1             /* First launch */
3.      D_app ← 2
4.  elseif |Degree| == 2         /* Second launch */
5.      D_app ← 4
6.  else
7.      Sort Degree in an ascending order
8.      D_j ← arg min T(D_i)    ∀ D_i ∈ Degree
9.      if D_j == max D_i       ∀ D_i ∈ Degree
10.         D_app ← D_j × 2
11.     elseif D_j == min D_i   ∀ D_i ∈ Degree
12.         D_app ← (1 + D_j) / 2
13.     else D_app ← (D_j+1 + D_j-1) / 2
14.     end
15. end
```

Figure 4. Pseudo-code of resource estimating algorithm.

3.3. Critical Task Scheduling Algorithm

In the current Spark system, the default task scheduling algorithm is efficient but intuitive. If there is only one Spark system executing, all resources are allocated to this Spark system, and the default algorithm simply schedules stages using the First-In-First-Out (FIFO) mechanism. If there are multiple Spark systems executing simultaneously, another mechanism is applied to make sure that all Spark systems get fair resource usages. In other words, the default task scheduling algorithm treats all stages ready for execution as equivalent. When a Spark system is informed that an available resource is allocated to it, the ready stage with the smallest id would be picked for execution first. However, in order to shorten the overall processing time, a stage belonging to the critical path should be given higher priority for execution first. Apparently, the default task scheduling algorithm in the Spark system cannot identify the stage belonging to the critical path, hence its overall processing time could probably be improved further.

In the proposed framework, a proactive critical task scheduling algorithm is designed to generate an appropriate sequence to execute stages without violating any data dependencies. The main idea is to execute stages belonging to critical path early, and the goal is to achieve shorter processing time. Figure 5 shows the pseudo-code of proposed proactive critical task scheduling algorithm. When a Spark system starts to execute an application, the execution pattern could be found in the profiled log. Based on data dependencies between stages, each stage S_i would be designated two values; $Rank(S_i)$ and $ET(S_i)$. $Rank(S_i)$ is designated to 0 if stage S_i has no successors. For a stage S_i with successors, its $Rank(S_i)$ is designated to the maximum $Rank(S_j)$ of successors S_j plus one. After the above pre-process, the stage has no predecessor and belongs to the critical path will be designated to maximum rank value. $ET(S_i)$ is set to the longest execution time of a task in stage S_i.

Input: A DAG with N stages
Output: $Assign(S_i)$

1. **if** the profiling log is not existed
2. Activate the profiling system
3. **end**
4. Determine $Rank(S_i)$
5. Determine $ET(S_i)$
6. $Assign(S_i) \leftarrow 0 \quad \forall\ 1 \le i \le N$
7. $Ready \leftarrow$ Stages S_i with no predecessors
8. **for** $k = 1$ to N
9. $T_1 \leftarrow arg$ **max** $Rank(S_i) \quad \forall\ S_i \in Ready$
10. **if** $|T_1| > 1$
11. $T_2 \leftarrow arg$ **max** $ET(S_i) \quad \forall\ S_i \in T_1$
12. **if** $|T_2| > 1$, $T_{select} \leftarrow S_i \in T_1$ with minimum i
13. **else** $T_{select} \leftarrow T_2$
14. **end**
15. **else** $T_{select} \leftarrow T_1$
16. **end**
17. $Assign(T_{select}) = k$
18. $Ready \leftarrow Ready - T_{select}$
19. Update $Ready$
20. **end**

Figure 5. Pseudo-code of proposed proactive critical task scheduling algorithm.

In Figure 5, stages whose predecessors have all been assigned are collected in a ready set. At first, the ready set would contain stages without any predecessors in the given RDD DAG. A loop would then repeat N times, while in each iteration one stage is selected. In each iteration, this task scheduling algorithm finds stages with the maximum rank in the ready set. If there is more than one stage with maximum rank, only stages with maximum ET values would be retained. If there is still more than one stage remaining, the stage with smallest id would be selected in this iteration. After N iterations, an appropriate sequence to execute all stages in the given RDD DAG would be generated.

4. Experimental Results

A small-scale cloud system including 3 servers is built for experiments. Table 1 lists the hardware specification of the 3 servers. The OS of physical machines is Ubuntu 13.04 (www.ubuntu.com), and the hypervisor is KVM 3.5.0-23 (www.linux-kvm.org). The cloud OS is OpenStack (Grizzly, www.openstack.org), and Apache Mesos 1.1.0 (mesos.apache.org) is used to supply the virtual clusters in which virtual resources may be obtained from multiple physical servers. Apache Spark 1.5.0 (spark.apache.org) is adopted as the data processing platform. In this virtual cluster computing system, each virtual machine occupies 4 vCPU, 4 GB RAM, and 40 GB disk. In each virtual machine, the Spark executor occupies one vCPU, and other vCPUs are used to execute tasks of applications. That is, when the virtual machine has m vCPU, the virtual machine could execute $m - 1$ Spark tasks at most.

Table 1. Hardware specification.

IBM Blade Center Model	CPU	Memory	Hard Disk
S22	Intel Xeon E5520*16	26 G	146 GB*2
HS22	Intel Xeon E5620*16	30 G	300 GB*2
HS23	Intel Xeon E5-2609*8	24 G	300 GB*2

The proposed over-commit mechanism is implemented in Mesos. Three micro-benchmarks, SparkTC, SparkLR, and WordCount, are selected for evaluation. SparkTC is used to evaluate the transitive closure on a graph, SparkLR is used to evaluate the classification based on logistic regression, and WordCount is used to count words. In the following, Figures 6–13 present the performance improvement of the over-commit mechanism. The performance improvement achieved by the proactive task scheduling approach is presented in Figures 14–16.

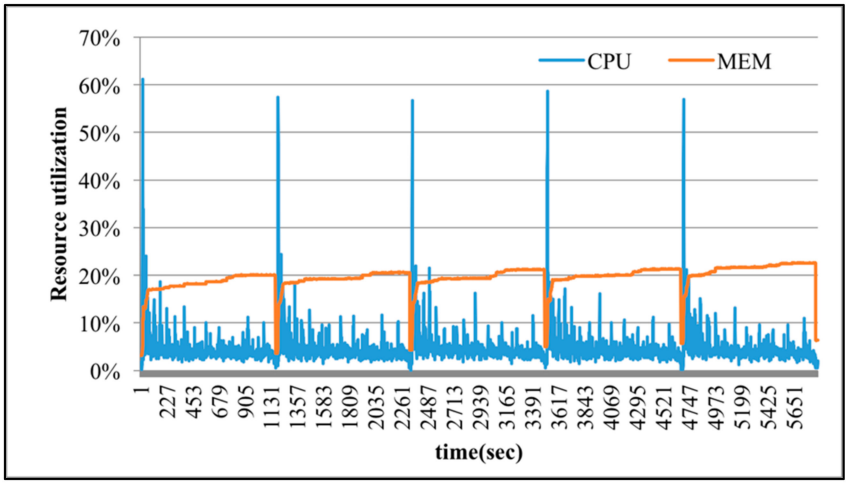

Figure 6. Resource utilization of executing SparkTC without over-commit.

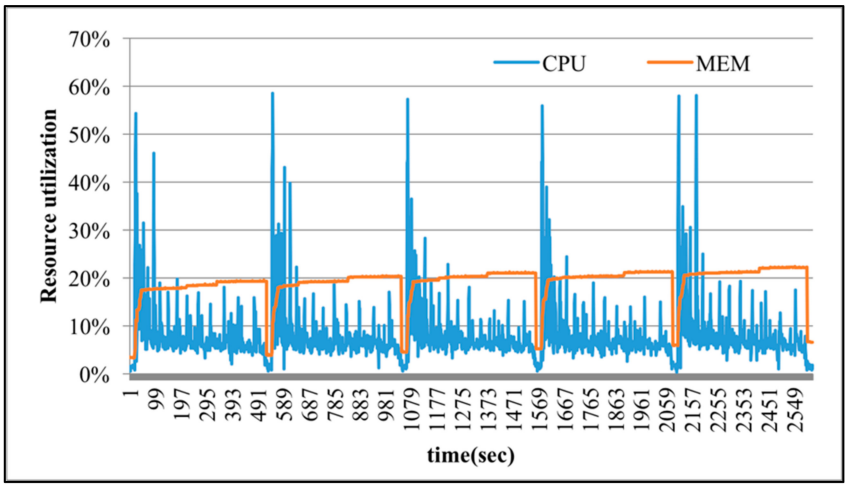

Figure 7. Resource utilization of executing SparkTC when over-commit with 8 vCPU demands.

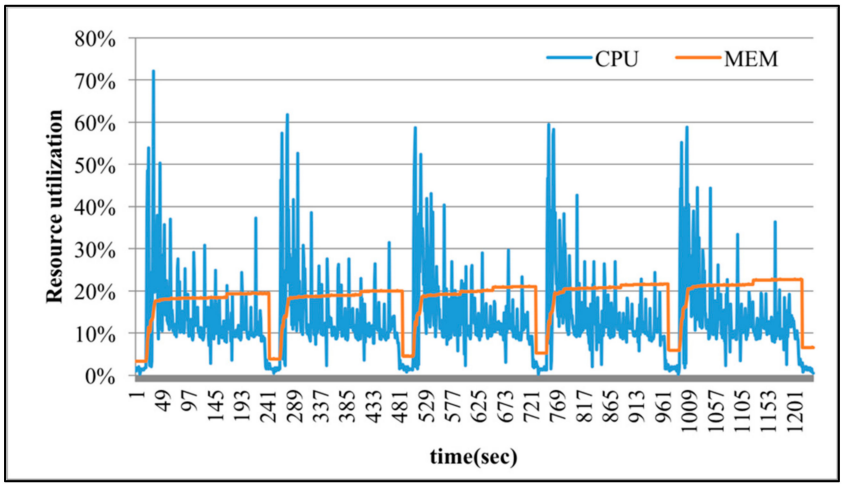

Figure 8. Resource utilization of executing SparkTC when over-commit with 16 vCPU demands.

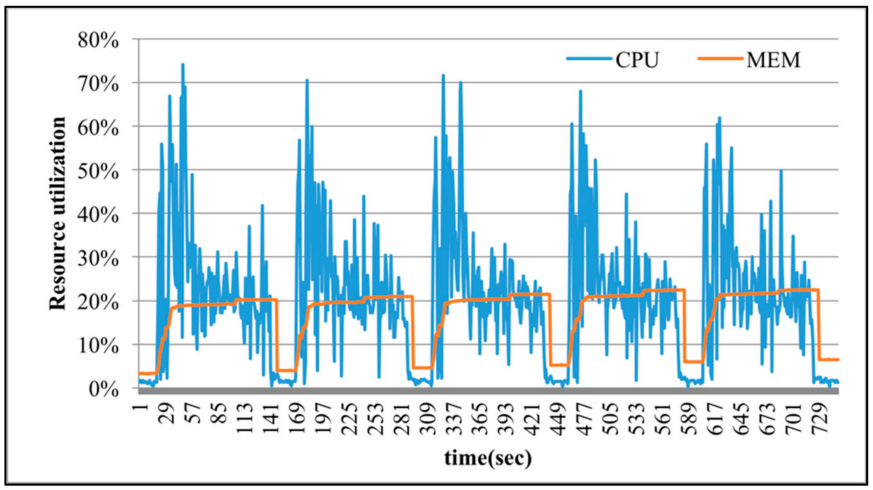

Figure 9. Resource utilization of executing SparkTC when over-commit with 32 vCPU demands.

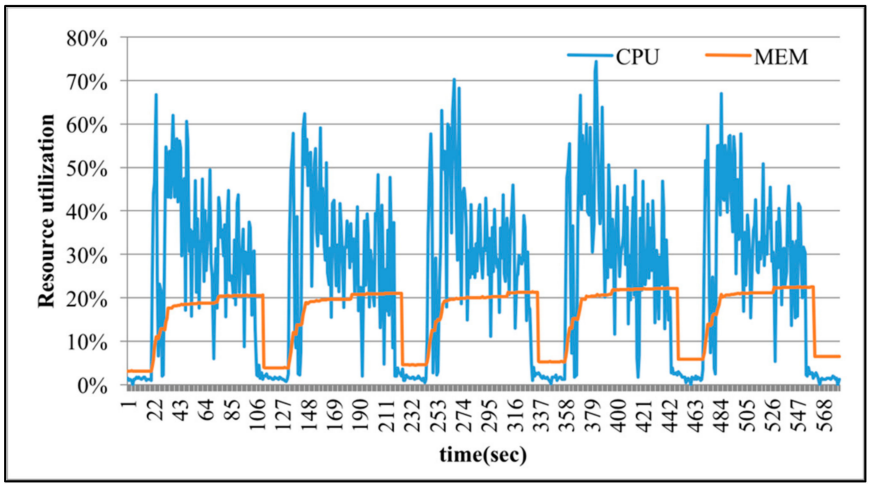

Figure 10. Resource utilization of executing SparkTC when over-commit with 64 vCPU demands.

In Figures 6–10, SparkTC is executed 5 iterations successively to demonstrate the variations of physical resource utilization. The numbers of nodes and edges in executing SparkTC are set to 100

and 150. When the degree of over-commit is 1, which means there is no over-committed resource provisioning, each virtual machine has 4 vCPUs and each vCPU corresponds to a physical CPU core. The execution time is 5611 s as shown in Figure 6, because the number of tasks that Spark submitted to execute in parallel is very limited. In Figures 7–10, the degree of over-commit is set to 2, 4, 8, and 16, respectively. That is, each virtual machine is over-committed to 8, 16, 32, and 64 vCPUs, respectively, but the number of physical CPU cores is still 4. This over-commit setting lets Spark believe there are many available resources, and then submits more tasks to compete for physical CPU cores. As shown in Figures 7–10, when the degree of over-commit increases, the execution time of SparkTC is reduced to 2509, 1189, 727, and 573 s. Apparently, applying the over-commit mechanism makes the CPU utilization increase and results in shorter execution time. In the meantime, the CPU utilization of executing SparkTC without over-committed resources is about 10%. When the degree of over-commit increases, experimental results show that the CPU utilization also increases accordingly. For example, in Figure 10, the range of CPU utilizations of executing SparkTC with 64 over-committed vCPUs is about 20% to 60%. However, although the amount of provisioned resources exceeds the actual needs in executing SparkTC, the utilization of memory is rarely changed as the provisioned resource increases. The reason is that SparkTC is a computing-sensitive application and the amount of required memory is stable.

Figure 11 shows the execution time of SparkTC when the degrees of over-commit are 1 and 16 (i.e., without and with (64 vCPU) over-committed resources). In this figure, the number of nodes remains 100, and the number of edges are 150, 300, 600, 900, and 1200. Since the number of nodes is fixed, when the number of edges increases, the graph topology of SparkTC changes from sparse to dense. The main step in micro-benchmark SparkTC is to check whether there is a path from node x to node z through node y. When the graph topology becomes denser, it is reasonable that the probability of finding such paths increases and the execution time of SparkTC decreases. Figure 11 also shows that the execution time after applying the over-commit approach is much shorter than without applying over-committed resources.

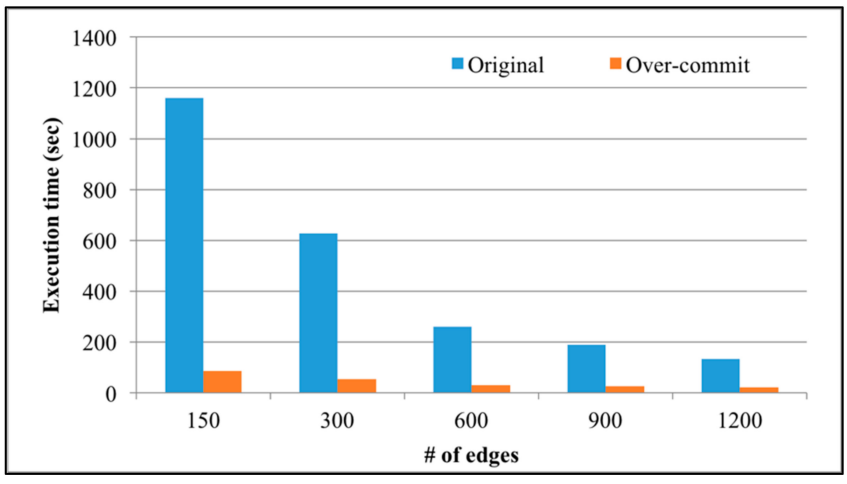

Figure 11. Execution time of executing SparkTC without over-committed resources and with 64 over-committed vCPUs.

Figure 12 shows the execution time of SparkLR when the degrees of over-commit are 1 and 16 (i.e., without and with (64 vCPU) over-committed resources). The number of data points in SparkLR is 100,000, the scaling factor is 0.7, and the number of dimensions are 8, 16, 64, and 128. As shown in Figure 12, the execution time of SparkLR is rarely changed between various numbers of dimensions, and even slightly increases when the number of dimensions is 128. This is because the total number of data points is fixed, and each dimension would contain less data points as the number of dimensions increases. Nevertheless, after applying the over-commit approach the execution time is still improved.

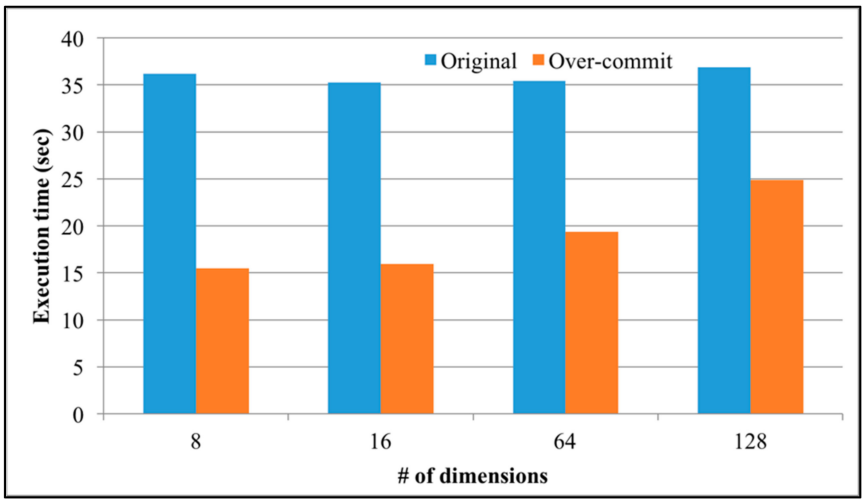

Figure 12. Execution time of executing SparkLR without over-committed resources and with 64 over-committed vCPUs.

Figure 13 shows the execution time of WordCount when the degrees of over-commit are 1 and 16 (i.e., without and with (64 vCPU) over-committed resources). The size of datasets in WordCount are 1, 3, and 5 GB. As shown in Figure 13, the execution time of WordCount increases as the size of datasets increases, with or without over-committed resources. The execution time after applying the over-commit approach clearly outperforms that without applying the over-commit approach.

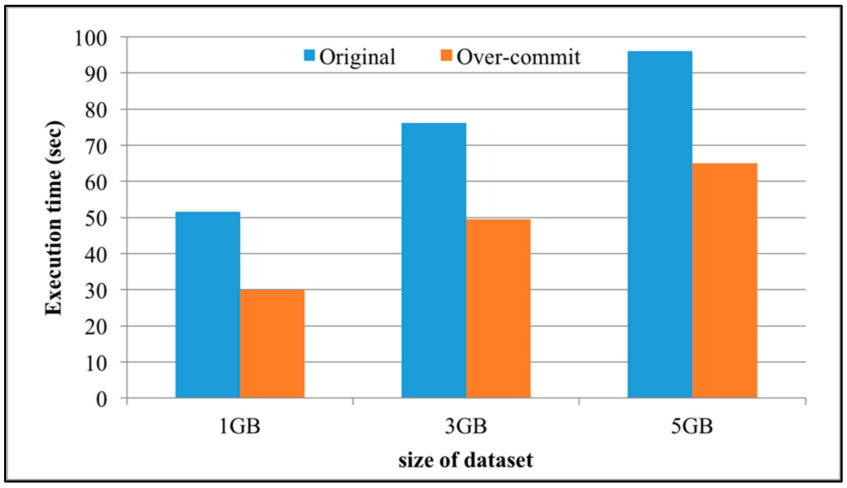

Figure 13. Execution time of executing WordCount without over-committed resources and with 64 over-committed vCPUs.

Figure 14 shows the execution time of executing SparkTC using either the default FIFO approach or the proposed proactive task scheduling (PTS) approach. The number of nodes is 100, and the number of edges is 150, 300, 600, 900, and 1200. As shown in Figure 14, when the number of edges increases, the execution time of SparkTC decreases in both approaches, and the proactive task scheduling approach always outperforms the default FIFO approach. The percentages of performance improvement using the proactive task scheduling approach are 16.09%, 15.32%, 15.17%, 13.24%, and 7.58%, when the numbers of edges are 150, 300, 600, 900, and 1200, respectively.

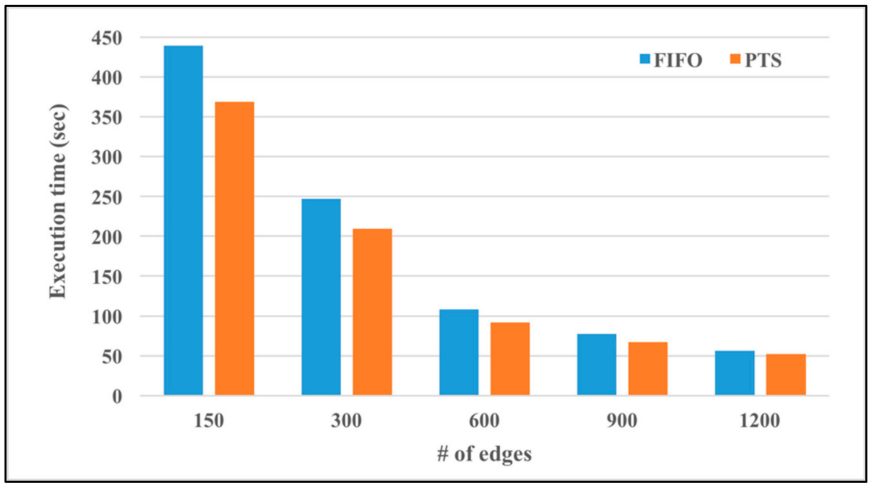

Figure 14. Execution time of executing SparkTC by FIFO and proactive task scheduling (PTS) approaches.

The execution time of SparkLR using FIFO and PTS approaches are presented in Figure 15. The number of data points is 100,000, the scaling factor is 0.7, and the number of dimensions are 8, 16, 64, and 128. In Figure 15, the execution times using PTS are only slightly worse than that using default FIFO approach in all dimensions. The main reason is that the dataflow of SparkLR is fully distributed, and the number of tasks in each Spark stage is at most one. Thus, the proactive task scheduling approach doesn't have many chances to reschedule the sequence of tasks in each Spark stage in order to shorten the processing time.

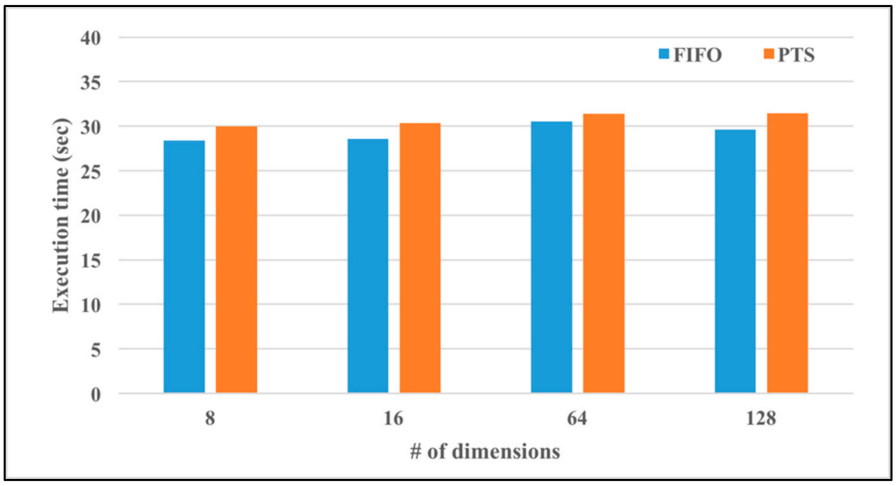

Figure 15. Execution time of executing SparkLR by FIFO and by PTS approaches.

Figure 16 shows the execution time applying FIFO and PTS approaches in executing micro-benchmark WordCount. The sizes of the datasets are 1, 3, and 5 GB, and the execution time applying both FIFO and PTS increases with the size of the dataset. Nevertheless, the execution time achieved by the proactive task scheduling approach still outperforms that of the FIFO approach. The detailed percentages of improvement are 12.8%, 14.52%, and 17.63%, when the sizes of datasets are set to 1, 3, and 5 GB, respectively.

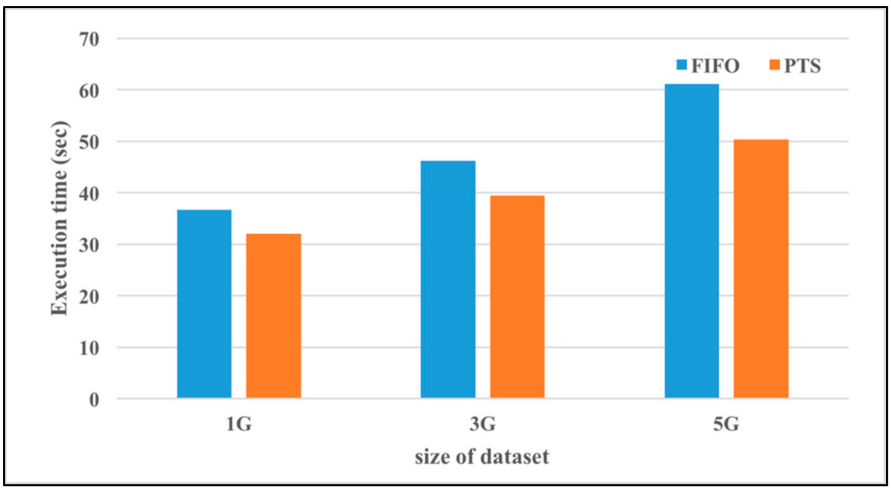

Figure 16. Execution time of executing WordCount by FIFO and PTS approaches.

5. Conclusions and Future Work

This study proposes a framework for proactive resource provisioning to address the resource provisioning problem for data streaming applications in IaaS clouds. The proposed framework consists of the virtual cluster computing system, the profiling system, the resource management system, and the monitoring system. In this framework, the over-commit mechanism and the proactive task scheduling approach are proposed. A small-scale cloud system including 3 servers is built for experiments. Preliminary experimental results show that the over-commit approach indeed improves the resource utilization, and the proactive task scheduling approach yields a reduction in execution time. In general, the over-commit mechanism performs well in all three micro-benchmarks. The proactive task scheduling approach is only effective when a Spark stage contains more than one task. As the number of tasks in each Spark stage increases, the performance of the proactive task scheduling approach becomes better and better.

In the future, we will address the proactive resource allocation algorithm and conduct more experiments in IaaS clouds. Although, due to the scale of our cloud system, the preliminary experimental results may not show significant improvement. We will further extend the experiment with more factors, such as resource heterogeneity, application parallelism, and system scale in the near future.

Acknowledgments: This study was sponsored by the Ministry of Science and Technology, Taiwan, R.O.C., under contract numbers: MOST 103-2218-E-007-021 and MOST 103-2221-E-142-001-MY3.

Author Contributions: Yi-Hsuan Lee conceived and designed the experiments; Cheng-Hsien Wu and Yen-Hsuan Kuo performed the experiments; Kuo-Chan Huang analyzed the data; Kuan-Chou Lai finalized the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Apache Storm. Available online: http://storm.apache.org/ (accessed on 25 June 2017).
2. Apache Spark. Available online: https://spark.apache.org/ (accessed on 25 June 2017).
3. Apache Samza. Available online: http://samza.apache.org/ (accessed on 25 June 2017).
4. Apache Flink. Available online: https://flink.apache.org/ (accessed on 25 June 2017).
5. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, Applications. *IEEE Commun. Tutor.* **2015**, *17*, 2347–2376. [CrossRef]
6. Stonebraker, M.; Çetintemel, U.; Zdonik, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Newsl.* **2005**, *34*, 42–47. [CrossRef]

7. Xu, L.; Peng, B.; Gupta, I. Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In Proceedings of the 2016 IEEE International Conference on Cloud Engineering, Berlin, Germany, 4–8 April 2016; pp. 22–31.
8. Mell, P.; Grance, T. *The NIST Definition of Cloud Computing*; NIST Special Publication 800-145; National Institute of Standards and Technology: Gaithersburg, MD, USA, September 2011.
9. Mashayekhy, L.; Nejad, M.M.; Grosu, D. Physical Machine Resource Management in Clouds: A Mechanism Design Approach. *IEEE Trans. Cloud Comput.* **2015**, *3*, 247–260. [CrossRef]
10. Liu, J.; Zhang, Y.; Zhou, Y.; Zhang, D.; Liu, H. Aggressive Resource Provisioning for Ensuring QoS in Virtualized Environments. *IEEE Trans. Cloud Comput.* **2015**, *3*, 119–131. [CrossRef]
11. Wang, W.; Liang, B.; Li, B. Multi-Resource Fair Allocation in Heterogeneous Cloud Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 2822–2835. [CrossRef]
12. Baldan, F.J.; Ramirez-Gallego, S.; Bergmeir, C.; Herrera, F.; Benitez-Sanchez, J.M. A Forecasting Methodology for Workload Forecasting in Cloud Systems. *IEEE Trans. Cloud Comput.* **2016**. [CrossRef]
13. Do, A.V.; Chen, J.; Wang, C.; Lee, Y.C.; Zomaya, A.Y.; Zhou, B.B. Profiling Applications for Virtual Machine Placement in Clouds. In Proceedings of the 2011 IEEE 4th International Conference Cloud Computing, Washington, DC, USA, 4–9 July 2011; pp. 660–667.
14. Ghosh, R.; Naik, V.K. Biting off Safely More than You Can Chew: Predictive Analytics for Resource Over-commit in IaaS Cloud. In Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, 24–29 June 2012; pp. 25–32.
15. Zhang, X.; Shae, Z.Y.; Zheng, S.; Jamjoom, H. Virtual machine migration in an over-committed cloud. In Proceedings of the 2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, 16–20 April 2012; pp. 196–203.
16. Chen, H.; Jin, H.; Hu, K.; Huang, J. Scheduling overcommitted VM: Behavior monitoring and dynamic switching-frequency scaling. *Future Gener. Comput. Syst.* **2013**, *29*, 341–351. [CrossRef]
17. Juhnke, E.; Dörnemann, T.; Bock, D.; Freisleben, B. Multi-objective scheduling of BPEL workflows in geographically distributed clouds. In Proceedings of the 2011 IEEE 4th International Conference Cloud Computing, Washington, DC, USA, 4–9 July 2011; pp. 412–419.
18. Emeakaroha, V.C.; Maurer, M.; Stern, P.; Labaj, P.P.; Brandic, I.; Kreil, D.P. Managing and optimizing bioinformatics workflows for data analysis in clouds. *J. Grid Comput.* **2013**, *11*, 407–428. [CrossRef]
19. Janiesch, C.; Weber, I.; Kuhlenkamp, J.; Menzel, M. Optimizing the Performance of Automated Business Processes Executed on Virtualized Infrastructure. In Proceedings of the 2014 47th Hawaii International Conference on System Sciences, Waikoloa, HI, USA, 6–9 January 2014; pp. 3818–3826.
20. Li, H.; Venugopal, S. Using reinforcement learning for controlling an elastic Web application hosting platform. In Proceedings of the 8th International Conference on Autonomic Computing, Karlsruhe, Germany, 14–18 June 2011; pp. 205–208.
21. Gupta, A.; Faraboschi, P.; Gioachin, F.; Kale, L.V.; Kaufmann, R.; Lee, B.S.; March, V.; Milojicic, D.; Suen, C.H. Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Trans. Cloud Comput.* **2016**, *4*, 307–321. [CrossRef]
22. Tsai, C.W.; Huang, W.C.; Chiang, M.H.; Chiang, M.C.; Yang, C.S. A Hyper-Heuristic Scheduling Algorithm for Cloud. *IEEE Trans. Cloud Comput.* **2014**, *2*, 236–250. [CrossRef]
23. Rodriguez, M.A.; Buyya, R. Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds. *IEEE Trans. Cloud Comput.* **2014**, *2*, 222–235. [CrossRef]
24. Kanemitsu, H.; Hanada, M.; Nakazato, H. Clustering-Based Task Scheduling in a Large Number of Heterogeneous Processors. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 3144–3157. [CrossRef]
25. Zhang, R.; Wu, K.; Li, M.; Wang, J. Online Resource Scheduling Under Concave Pricing for Cloud Computing. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1131–1145. [CrossRef]
26. Chen, C.Y. Task Scheduling for Maximizing Performance and Reliability Considering Fault Recovery in Heterogeneous Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 521–532. [CrossRef]