Hindawi
Scientific Programming
Volume 2018, Article ID 7548527, 9 pages
https://doi.org/10.1155/2018/7548527

Research Article

Mitigating Interference between Scientific Applications in OS-Level Virtualized Environments

Theodora Adufu and Yoonhee Kim

Department of Computer Science, Sookmyung Women's University, Seoul, Republic of Korea

Correspondence should be addressed to Yoonhee Kim; yulan@sookmyung.ac.kr

Received 13 November 2017; Accepted 21 January 2018; Published 1 March 2018

Academic Editor: Sungyong Park

Recent research and production environments are deploying more container technologies for the execution of HPC applications and for reproducing scientific workflows or computing environments. Research works, however, have not accounted for performance interference when executing corunning applications in containers though the absence of an efficient performance isolation layer cannot guarantee the absence of performance interference among multiple corunning applications which share resources. In this research, we propose an interference-aware scheduling method that mitigates the problem of performance interference based on applications' I/O and CPU usage profiles. The proposed method estimates the amount of interference between various pairs of applications and coschedules them based on estimated interference. We evaluate the proposed method for both Bag-of-Tasks (BOT) scientific applications and scientific workflows and compare our method to the Weighted Mean Method. Our method improves the performance of the target scientific application by coscheduling applications with the least estimated interference ratios.

1. Introduction

Due to their almost zero start-up times, minimal runtime overhead, lightweight feature, and the relatively higher deployment density per physical host [1–3], Linux containers are being deployed as efficient virtual technology solutions in research and production environments and in the cloud [4, 5]. Scientific researchers, in particular, are deploying containers to reproduce scientific workflows [6] and benchmark Virtual Machines [7] and provide distributed storage [8]. Efficient resource allocation to the containers in a manner that guarantees high performance is thus of utmost importance. Correspondingly, performance evaluations of containers by Xavier et al. [9, 10] establish the need for an efficient isolation layer to facilitate resource sharing in HPC environments. From their experiments Xen had better isolation than containers due to nonshared Operating System (OS) indicating potential performance interference when corunning applications in container-based virtualization systems.

Also, existing Container Cluster Managers (CCM) [11–13] allocate physical resources to containers in a manner that maximizes the deployment density per node [3] without considering the cost to performance. Ideally, there should be no interference in performance of corunning containers; however, due to contention for shared physical resources, the performance of a given application varies with the coexecuted applications on the same physical resource. Particularly, as some applications alternate between computation and I/O phases, performance interference is most apparent when corunning applications involve multiple I/O operations. Estimating the interference introduced by different applications concurrently running on a given node and scheduling applications accordingly has become a necessity when aiming to maximize the performance of applications. Research works so far have focused on the performance interference of data-intensive and I/O intensive applications in traditional hypervisor-based virtualization (HPV) environments; however, to the best of our knowledge, this research is the first to propose an interference-aware scheduling scheme for OS-level virtualization technologies.

The proposed method employs a two-phase scheduling strategy to mitigate interference in OS-level virtualized environments. Our proposed scheduler schedules BOT

scientific applications and scientific workflows applications to resources based on the similarities in the CPU usage and I/O usage profiles to minimize the interference from coscheduled applications. In the first phase, it uses information on the CPU and I/O usage of applications to first estimate the interference between the different applications and then coschedules each application with another application of least interference. In the second phase, the applications are matched to container resources using different resource allocation strategies which would further mitigate interference. The proposed method makes the following novel contributions:

(i) It establishes the existence of performance interference during the execution of corunning scientific applications.

(ii) It introduces a two-phase interference-aware scheduler in OS-level virtualization environments to mitigate interference during the execution of applications in containers.

(iii) It estimates interference among applications based on computations using Euclidean distances.

(iv) It coschedules applications in a manner that minimizes contention for same physical host resources and hence minimizes interference.

The rest of this paper is organized as follows: Section 2 presents the related work and Section 3 describes our interference-aware scheduler. In Section 4, we explain our experiment environments and evaluate the results. We finally conclude this paper in Section 5.

2. Related Works

Interference-aware scheduling in virtualized environments is still a novel research area. Existing methods have investigated interference of Virtual Machines (VMs); however, there is little research on interference in OS-level technologies like Linux containers which are currently being deployed in many production environments.

In their investigations on resource and security isolation in container technologies through benchmark experiments, Soltesz et al. [14] demonstrate that container-based systems are twice more suitable for server-type workloads. This finding coupled with the findings of Xavier et al. [9, 10] establish the premise that, for HPC scientific environments, the absence of performance interference cannot be guaranteed.

In hypervisor-based environments, interference-aware schedulers such as DejaVu [15] use an "interference index" to estimate interference between workloads. However, determining the interference index requires profiling of low-level metrics of workloads over many hours which is time-consuming. Also, workload clustering analysis is only used to determine which sets of applications require interference-aware scheduling without estimating the interference introduced by coscheduled applications.

TRACON [16] also predicts interference in paravirtualized environments based on the I/O usage of the guest and native driver domains. Interference prediction based on three models, Weighted Mean Method (WMM), the Linear Model (LM), and the Nonlinear Model (NLM) is applied and the application runtime and I/O throughput of the various applications are compared, respectively. With the results of their prediction, incoming tasks are coscheduled to resources with applications of least interference. However, this method was applied to only Virtual Machines (VMs). The proposed method employs a two-phase interference-aware scheduling method which first estimates the interference between applications using clustering analysis and then schedules the applications to suitable OS-level virtual resources in a manner that minimizes contention and hence mitigates interference.

The native Container Clustering Manager for the recently widely adopted Docker containers [17], Docker Swarm [11], employs a manager-agent deployment structure which includes a host that runs a Swarm manager and other hosts which run a Swarm agent each for the management of container clusters. The Swarm manager orchestrates and schedules containers on the hosts according to three (3) scheduling policies: bin-packing, spread, and random [18]. The proposed method adds interference awareness to the existing container placement strategies of Docker Swarm.

3. Interference-Aware Scheduling Based on Clustering Analysis

According to research [16, 19, 20], we consider interference as the change in the relative total execution time of an application due to the execution of concurrently running applications. We propose a two-phase interference-aware scheduling algorithm based on application clustering analysis in this section.

In the first phase of the proposed interference-aware scheduler, applications are coscheduled for execution according to the result of application clustering analysis using the K-means algorithm. After clustering the applications, the coscheduled applications are then scheduled to the suitable resources by a resource scheduler in the second phase of the proposed scheduling procedure. The interference-aware scheduling shows a scheduling procedure for multiple applications waiting to be scheduled unto container resources in a manner that mitigates interference.

First, using profile data of peak CPU and I/O usage, applications are clustered into specified number of clusters using the K-means algorithm [21]. Then, the interference between a selected application known as the target application and other applications in the clusters is computed according to an interference ratio. The application with the least interference is then coscheduled with the selected application and executed according to the container placement strategy indicated by the user. Each application is executed in a different container in all our experiments. The key notations used in this paper are listed in Notations.

3.1. Algorithm 1: Interference-Aware Scheduler. Algorithm 1 starts when a new application joins the queue with profile information on CPU utilization and I/O usage (Line (1)). The scheduler then uses the K-means clustering algorithm

Input: A Queue $\mathbb{Q} = \{x_1, x_2, \ldots, x_n\}$
(1) New application arrives;
(2) **while** ($\mathbb{Q} =! 0$) **do**
(3) Cluster(i)(x_h, c_i) ← Invoke K-means algorithm;
(4) $x_j \leftarrow$ Select$X_j(x_h, c_i)$;
(5) dist(x_j, c_j) ← CalcDist(x_j, c_j);
(6) **for each** x_h in Cluster(i)(x_h, c_i) **do**
(7) dist(x_h, c_j) ← CalcDist(x_h, c_j);
(8) $r_{jh} \leftarrow$ Calc$R$(dist(x_j, c_j), dist(x_h, c_j));
(9) **end for**
(10) **end while**
(11) $x_h \leftarrow$ SelectMin(r_{jh});
(12) **if** $r_{jh} \leq \alpha$ **then**
(13) Perform resource selection;
(14) **end if**
(15) Perform resource selection;
Output: Co-execute x_j, x_h

Algorithm 1: Interference-aware scheduling algorithm.

to group the applications into K clusters. The K-means algorithm partitions data into multiple sets with each containing a unique center or centroid. For a queue containing a set of applications, $\mathbb{Q} = \{x_1, x_2, \ldots, x_n\}$, with initial centroids given as $C = \{c_i \mid i = 1, 2, \ldots, k\}$, the algorithm calculates the distance between each data point and cluster centroids.

Next, the scheduler randomly selects an application from any of the K clusters (Line (4)) known as the target application and calculates the interference ratio between that application and other applications in different clusters (Lines (5)–(9)). The scheduler then selects the application with which the target application has the least interference according to the calculated interference ratio (Lines (11)) and compares the interference ratio to a preset threshold value, α. If the ratio is less than the threshold value, the scheduler then invokes the resource selection method to select a suitable node from the cluster (Lines (13)-(14)).

The method returns nodes for the execution of each pair of applications into containers according to their resource demands. Then the applications are coscheduled to containers on the node accordingly. Otherwise, the system considers other placement strategies of Docker Swarm according to the clustering policies selected on the available nodes.

3.2. Interference Detection and Interference Ratio, r_{jh}. The proposed approach clusters all the applications in the queue into K clusters using profile data on their peak CPU and I/O usage. To calculate the interference ratio between the randomly selected application and other applications in other clusters, our approach computes a two-dimensional Euclidean distance between the selected application $x_h(a, b)$ and the centroid of its cluster $C_i(c, d)$ and that of other applications $x_j(a, b)$ and the centroid of the target application $C_i(c, d)$ according to (1). With the result, an interference ratio is calculated according to equation (2) and the results are used to select container resources for the execution of coscheduled applications.

$$\text{CalcDist}\left(x_h, C_i\right) = \sqrt{(a-c)^2 - (b-d)^2}, \tag{1}$$

$$\text{Calc } R = \frac{\text{dist}\left(x_h, C_i\right)}{\text{dist}\left(x_j, C_i\right)}. \tag{2}$$

3.3. Algorithm 2: Resource Selection Method. Interference-aware scheduling methods for scientific applications aim at minimizing the performance overheads introduced by coscheduling applications on a particular physical resource. It is therefore important that, in a container cluster, the resource that minimizes the likelihood of interference is selected. At the second phase of the proposed scheduling method, we adopt the placement strategies of Docker Swarm to schedule container resources to applications. The Swarm manager orchestrates and schedules containers on the hosts according to three (3) scheduling policies: *bin-packing, spread, and random* [18]. The *bin-packing strategy* chooses physical resources based on the highest number of containers running on that resource whilst the *random scheduling strategy* does not consider the number of running containers. The *spread strategy*, on the other hand, schedules applications to physical resources with the least number of containers running on them. We adopt the spread strategy as our default strategy with the assumption that the number of containers reflects the amount of resource contention for resources in the system.

When the resource selection method is called, the resource scheduler looks for available nodes and considers the container placement strategy indicated by the user. When the default strategy indicated is the spread strategy as used in our experiments, the node with the least number of containers is returned (Lines (4)-(5)). On the other hand, when the bin-packing strategy is used, the available node with the highest number of containers is returned (Lines (8)-(9)). With the random strategy, any available node is returned (Lines (11)-(12)). The method returns the results of executing the applications in the containers to the scheduling algorithm.

4. Experiments

Scientific applications are classified into Bag-of-Tasks (BOT) and scientific workflows [22] represented as Direct Acyclic Graphs (DAG). In this section, we describe the experiment environments and establish performance interference when corunning BOT applications and scientific workflows in OS-level virtualized environments. We then compare the proposed method to the Weighted Mean Method for BOT scientific applications. We also experiment which of the three (3) container placement strategies of Docker Swarm best mitigates performance interference for both BOT scientific applications and scientific workflows.

4.1. Experiment Environments. We use Docker containers, a lightweight virtualization solution for fast creation and

Input: NodeList $\mathbb{N} = \{(n_i) \mid i = 0, 1, \ldots, m\}$, Co-scheduled applications $[x_h(\min_C, \min_R)]$, $[x_j(\min_C, \min_R)]$
(1) Set resource available == true;
(2) Set default == spread;
(3) **if** resource available == true **then**
(4) Node, $n_i \leftarrow$ FindNode($\min_{\text{cont}}$);
(5) $\text{cont}_h \leftarrow$ Create$[x_h(\min_C, \min_R)]$;
(6) **else**
(7) **if** default == binpack **then**
(8) Node, $n_i \leftarrow$ FindNode($\max_{\text{cont}}$);
(9) $\text{cont}_h \leftarrow$ Create$[x_h(\min_C, \min_R)]$;
(10) **else**
(11) Node, $n_i \leftarrow$ FindNode(ran_{cont});
(12) $\text{cont}_h \leftarrow$ Create$[x_h(\min_C, \min_R)]$;
(13) **end if**
(14) **end if**
(15) Results $\leftarrow$ Execute(x_h);
Output: Execution Results

Algorithm 2: Resource selection method.

Table 1: Profiles of applications characteristics.

Application	I/O Read (KB)	I/O Write (KB)	CPU usage%
Update-Heavy	2,600	55	97.4
Read-Only	1,890	1	82.2
Read-Modify-Update	1,150	10	40.6
Melt 3500	44	18	100

execution of applications independent of a hypervisor layer. To manage the scheduling of Docker containers, we deploy Docker Swarm. Our experiment environment consists of two nodes on the same local network with one node serving as both the Docker Swarm Manager node and an execution node and the other node serving as an execution node only.

The nodes in our system have a total RAM of 8 GB and 12 CPU cores for the manager node and total RAM of 8 GB and 4 CPU cores for the other execution node, respectively. Each of the server machines are operated by Trusty Tahr Operating System. To validate the accuracy of our results, we deploy Ubuntu 14.04-based container images which have been preinstalled with the scientific applications used for the experiments.

4.2. Performance Interference in Bag-of-Tasks (BOT) Scientific Applications. We first validate the postulation that there is performance interference in containers when corunning BOT scientific applications using an illustrative experiment. For this experiment, we consider scientific applications with four different workloads. Three (3) of the workloads are variations of YCSB applications, each with a different I/O and CPU usage profile, whilst the other application is Melt, a variation of LAMMPS [23]. The workloads and their I/O characteristics are briefly described as follows.

Yahoo Cloud System Benchmark (YCSB) [24], a basic benchmark for cloud systems, includes a set of six (6) core workloads that define the running of data-intensive applications in virtual environments. These benchmarks represent applications with varying I/O intensities and I/O access patterns. For these experiments, we deploy three (3) of the workloads, the *Update-Heavy, Read-Only, and Read-Modify-Write workloads*, due to their various I/O intensities and I/O access patterns. The I/O access pattern of Update-Heavy Workload and Read-Only Workloads take the form of random reads whilst the Read-Modify-Write workloads access I/O sequentially as the application reads the database.

Melt, a variation of a molecular dynamics code known as Large-scale Atomic Molecular Massively Parallel Simulator (LAMMPS) [23], is used for simulating different types of particle behaviors. Melt simulates the rapid melting of a 3d LJ system and accesses I/O resources in a bursty manner with maximum I/O reads taking place at the beginning and end of the executions.

Using Glances profiling tool [25], we obtain data of the I/O and CPU usage by each of the above scientific applications when run alone as shown in Table 1. For each characteristic, the data represented is based on the peak value obtained for each application during their execution lifecycle. We evaluate only two characteristics, CPU and I/O, per application and perform clustering analysis based on them.

In the experiment, we select the Update-Heavy workload as our target application and assume that only two containers can run on the same node simultaneously. Thus in this experiment scenario, we coexecute the Update-Heavy workload with Read-Only workload, YCSB Read-Modify-Update workload, Melt (3500), and another YCSB Update-Heavy workloads, respectively, under the same conditions.

Table 2: Application clusters and their interference ratios.

Cluster	Application	I	WMM
Cluster 0	Update-Heavy	1.00000	0.00023
Cluster 0	Read-Only	0.00052	0.43790
Cluster 1	Read-Modify-Update	0.00267	0.85731
Cluster 1	Melt	0.00015	1.48595

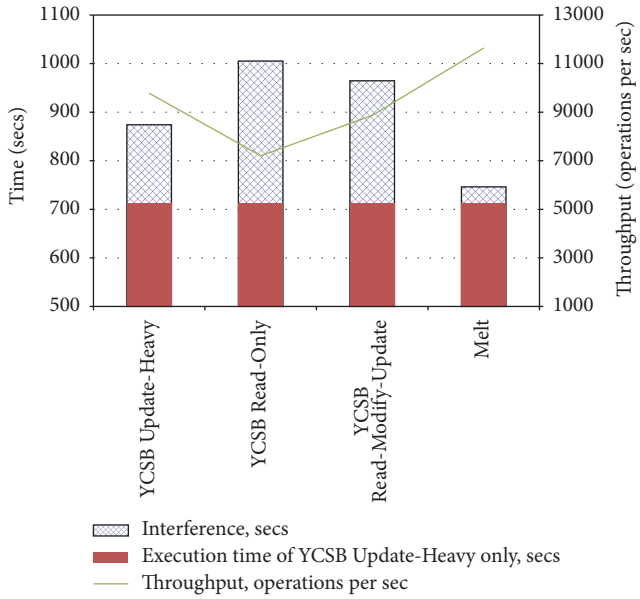

Figure 1: YCSB Update-Heavy workload with other applications.

We present the results of the change in execution times of the Update-Heavy workload when corun with other workloads as well as the change in throughput in Figure 1.

Figure 1 shows the changes in execution time of Update-Heavy workload from Cluster 1, when coexecuted with different applications. The execution time of YCSB Update-Heavy in each run is compared to the execution time of Update-Heavy when executed alone in a container on the physical node. In this experiment, the change in the execution time is reflective of the amount of performance interference due to coscheduling with other applications.

From the results, YCSB Update-Heavy workload with Read-Only update workload relatively experiences the most interference of about 293 seconds when coexecuted whilst YCSB Update-Heavy workload experiences the least interference of 34 seconds when coexecuted with Melt application. We attribute these variations in the execution times to contention for same I/O resources. Particularly, YCSB Read-Only workloads take a longer time to process data requests and retrieve the required data from the database, thus increasing the time spent on I/O operations. In effect, the YCSB Update-Heavy workload spends more time contending for I/O resources shared with YCSB Read-Only workloads. Consequently, the throughput for YCSB Update-Heavy workload when coexecuted with Read-Only workloads is the least due to relatively longer execution time. This validates our hypothesis that there is performance interference among containers on the same physical resource. The results also show that it is imperative to estimate the performance interference between colocated applications during the coscheduling of applications.

4.3. Comparing the Proposed Method and the Weighted Mean Method (WMM). We evaluate the relationship between the proposed interference ratio and the execution times of some Bag-of-Tasks applications and compare our results with results obtained from scheduling using Weighted Mean Method and a noninterference-aware method based on the bin-packing scheduling method of Docker Swarm.

Weighted Mean Method [16] also uses Euclidean distances to determine weights of each application according to the Principal Component Analysis (PCA). Some research works [16, 26] use the similarity between workload characteristic vectors of applications and account for noise, redundancy, and similarity in applications characteristics by using the principal components of a vector. It then uses the reciprocal of their distances as the weights to get the predicted response.

In Table 2, we present the details of the estimated interference calculated using both the proposed method and the Weighted Mean Method. Both estimations are based on the CPU usage and I/O of four (4) workloads of scientific applications when executed with different parameters. We choose Update-Heavy workload as our target application and conduct the ensuing experiments with it. The table contains the interference ratio of all the applications in the queue waiting to be scheduled to resources in the system. From the results, applications in Cluster 0 have the highest amount of interference with Update-Heavy workload.

In this experiment, we submit 1 Update-Heavy workload, 1 Melt application, and 1 Read-Only workload and coexecute them in three runs using three scheduling strategies: the proposed interference-aware method, Weighted Mean Method, and no-interference scheduling methods. The no-interference method, which does not account for interference, is based on Docker Swarm's bin-packing strategy and chooses resources based on the highest number of containers running on that resource.

In each run, the proposed method schedules both Update-Heavy workload and Melt on the same node because Melt has the least interference of 0.00015 compared to 0.00052 when scheduled with Read-Only workload. The Weighted Mean Method however schedules both Update-Heavy workload and Read-Only workload on the same node because Read-Only workload has the least interference of 0.43790 compared to 1.48595 when scheduled with Melt. The no-interference scheduling method which does not consider

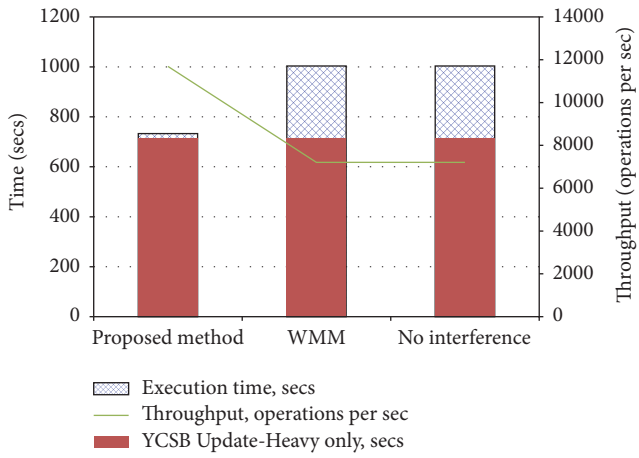

Figure 2: YCSB Update-Heavy with different scheduling strategies.

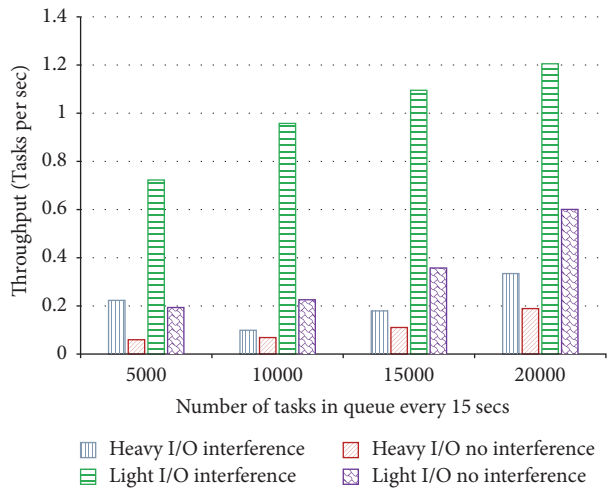

Figure 3: Throughput for tasks using interference awareness.

interference ratio between applications also coruns Update-Heavy workload with Read-Only workload which has a relatively higher interference ratio of 0.00052 according to our proposed method.

Figure 2 describes the scheduling result showing execution times of YCSB Update-Heavy workload when coexecuted with other applications using different scheduling strategies. For each scheduling method, we obtain the execution time of the workflow, YCSB Update-Heavy workload, and compare the results. From the results, the execution time of our proposed method is least at 732.89 seconds. However, the execution time of the no-interference and WMM is same at 1003.45 seconds since they coexecute the same applications.

The WMM model predicts interference by calculating the Euclidean distances between the data points in the clustering space and chooses three nearest data points whose reciprocal is used as the weights to get the predicted response. This does not guarantee an accurate prediction when the applications to be scheduled have highly variable differences in I/O intensities. The proposed method, however, guarantees a more accurate relationship between the prediction and the actual performance interference of applications because it predicts interference based on the distances between the target application and the application considered for coexecution.

4.4. Interference-Aware Scheduling for Dynamic Workloads. In the previous scenarios, the proposed method is able to improve the execution time for which the Update-Heavy workload application is executed for static workloads. In other words, all the applications were present in the queue at the time of the scheduling. In this section, however, we examine the scheduling of a dynamic workload since in real scenarios, tasks arrive dynamically. Thus, a simulation experiment using CloudSim [27], which considers the arrival of different tasks every 15 seconds, is conducted in this section to show the improvement in throughput for both Heavy I/O tasks and light I/O tasks by the proposed interference-aware method. We modify cloudlets to have Heavy I/O tasks and light I/O tasks according to their use of resources.

We begin the execution with 50000 tasks and increasingly submit tasks at an interval of 15 seconds. We increase the job size from 50000 tasks to 100000 tasks, 150000 tasks, and 200000 tasks for both types of tasks. As soon as the jobs arrive, interference is calculated and the applications are rescheduled according to the interference-aware method. This translates to an increased number of tasks being executed within a given period of time since tasks are scheduled in a manner that reduces interference.

Accordingly, the throughput of all the applications in the system is relatively higher than for jobs which are scheduled with the no-interference-aware method:

$$\text{Throughput}, T = \frac{\text{Number of applications}}{\text{Execution time}}. \quad (3)$$

The threshold for the interference-aware method is randomly set at 0.2 and throughput T is calculated according to (3) as the number of tasks completed during an execution, per the time of execution. As jobs with different characteristics are executed, their throughput and the number of jobs executed are compared using the proposed method and the no-interference scheduling method.

From the results in Figure 3, the execution of tasks using both the proposed method and a no-interference method decreases in throughput for increasing number of applications due to increasing time taken to execute the applications as a result of interference and an increasing number of tasks which were not scheduled for execution due to the set interference ratio. However the throughput of the proposed method is slightly higher due to the reduced time taken for execution as a result of I/O interference mitigation. Also, Heavy I/O tasks for both scheduling methods have a lower throughput since most of the execution time of these tasks are spent performing I/O operations instead of computations. This adds to the fact that including interference awareness in scheduling decisions helps improve the throughput and the performance of applications.

Table 3: Applications' profiles, clusters, and their interference ratios.

Cluster	Application	Memory usage (MB)	CPU usage%	Interference ratio, r
Cluster 0	CFD 512 KB	403,230,720	206.7	0.000432
Cluster 1	Melt 500	23,183,360	100	0.0194
Cluster 1	Melt 1000	23,314,432	100	0.0191
Cluster 1	Melt 1500	23,445,504	100	0.01886
Cluster 1	Melt 2000	23,576,576	100	0.01858
Cluster 1	Melt 2500	23,707,648	100	0.01831
Cluster 1	Melt 3000	23,838,720	100	0.0186
Cluster 1	Melt 3500	23,969,792	100	0.01781
Cluster 1	Peptide 500	32,460,800	100	0.0093
Cluster 1	Peptide 1000	33,382,400	99.99	0.0089
Cluster 1	Peptide 1500	34,213,888	99.99	0.0085
Cluster 1	Peptide 2000	35,606,528	100	0.00797
Cluster 1	Peptide 2500	36,810,752	100	0.00754
Cluster 1	Peptide 3000	37,613,568	100	0.00727
Cluster 1	Peptide 3500	38,817,792	100	0.00691
Cluster 2	GALFA 5	14,708,736	132.9	1
Cluster 2	GALFA 10	14,716,928	126.2	0.9535
Cluster 2	GALFA 15	14,745,600	121.01	0.8199
Cluster 2	GALFA 20	14,766,080	119.5	0.745
Cluster 2	GALFA 25	14,782,464	117.9	0.6949
Cluster 2	CFD 2 KB	13,967,360	201.4	0.2928
Cluster 2	CFD 32 KB	14,098,432	205.2	0.3796

4.5. Performance Interference for Scientific Workflow Applications. We also validate interference awareness for coexecuting scientific workflows using similarities between CPU and memory usage profiles. For this experiment, we select 1 scientific workflow, Montage GALFA [28], and three (3) BOT applications, CFD [29], Melt, and Peptide [23]. We cluster these applications into three clusters based on the CPU and memory usage profiles when the applications are coexecuted with Montage GALFA as shown in Table 3.

Montage GALFA, a data-intensive workflow application, is an Astronomical Image Mosaic Engine for creating mosaics using multiple astronomical images. In this paper, the Montage GALFA application shrinks five (5) data cubes by averaging different number of planes (5, 10, 15, 20, 25) and then aggregates them into a mosaic [30] following three major steps.

The second target application is an aerodynamic variation of a *Computational Fluid Dynamics (CFD)* simulation application used for 2-dimensional Euler unsteady flow analysis. In this experiment, we deploy different meshes with sizes 2 KB, 32 KB, and 512 KB, respectively.

The third and fourth target applications, *Melt and Peptide*, are variations of LAMMPS. Whilst Melt simulates the rapid melting of a 3d LJ system, Peptide simulates the granular particle pour and flow of both 2D and 3D systems. We iteratively run coscheduled applications which have shorter execution times such as Melt and Peptide applications throughout the experiments to maintain fairness.

Table 3 describes the result of performing clustering analysis for the applications using the proposed approach. From the analysis, the applications are grouped into 3 clusters. Cluster 1 has the highest number of applications (15) whilst Cluster 2 has 7 applications and Cluster 0 has only 1 application, respectively. We choose Montage GALFA (5) as our target application from Cluster 2 and conduct the ensuing experiments with it. The table also contains the interference ratio of all the applications in the queue waiting to be scheduled to resources in the system. From the results, applications in Cluster 2 have the highest amount of estimated interference with Montage GALFA (5 planes) using the proposed method.

Figure 4 shows the changes in execution time of GALFA workflow application, in Cluster 2, when coexecuted with applications from different clusters, C_0, C_1, and C_2, respectively. The interference ratio of the applications relative to C_2 is 0.00043, 0.01781, and 1, for C_0, C_1, and C_2, respectively. The execution time of C_2 in each run is compared to the execution time of C_2 when executed alone in a container on the physical node. In this experiment, the difference in the execution time is considered as the amount of performance interference due to coscheduling with another application.

From the results, C_2 experiences the most interference of about 63 seconds when executed with a similar application from the same cluster whilst C_2 experiences the least interference of 4 seconds when coexecuted with C_0. This also validates our hypothesis that there is performance interference among containers on the same physical resource and proves that our proposed interference ratio accurately predicts the interference among applications from different clusters. The results also show that it is imperative to

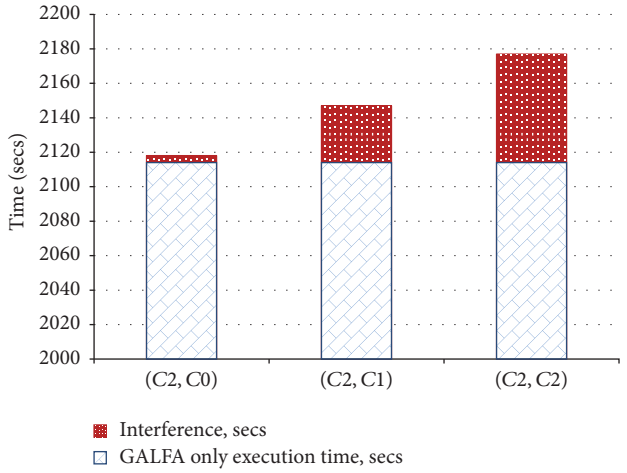

Figure 4: Montage GALFA (5 planes) with other applications.

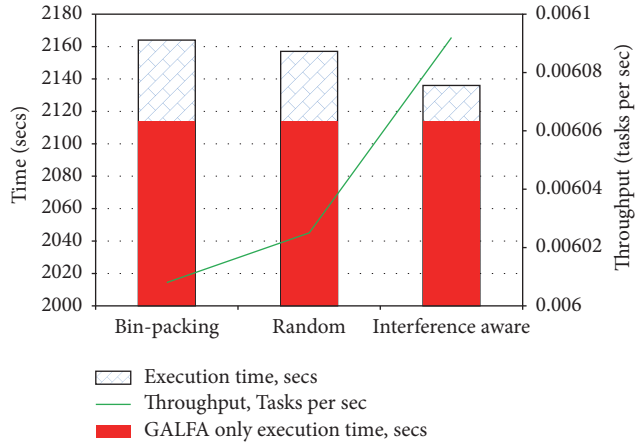

Figure 5: Montage GALFA (5 planes) with different scheduling strategies.

clearly understand the performance relationship between colocated applications during the coscheduling of applications.

4.6. Performance Interference for Montage GALFA Using Different Scheduling Strategies. In this experiment, we submit 1 Montage GALFA (5 planes) workflow, Melt (3500) application, and 1 Peptide (3000) application and coexecute them in three runs using three scheduling strategies: bin-packing, random, and the proposed interference-aware scheduler. The bin-packing strategy chooses resources based on the highest number of containers running on that resource and so schedules all the applications on the same node. The random scheduler and the proposed method, however, schedule both Montage GALFA (5 planes) and Peptide (3000) on the same node. The proposed method considers the interference ratio between the applications in making the scheduling decision and chooses Peptide (3000) which has a ratio of 0.00728 relative to 0.0181 of Melt (3500) applications. Due to the differences in execution times of both applications, Melt (3500) and Peptide (3000) applications are run iteratively in 20 runs to cover the span of time for which Montage GALFA (5 planes) is executed.

Figure 5 describes the scheduling result of our interference-aware scheduling algorithm, as shown in Section 3. For each scheduling method, we obtain the execution time of the workflow, Montage GALFA (5 planes), and compare the results. From the results, the execution time of our proposed interference-aware method and the random method is the least at 2117 seconds whilst that of bin-packing strategy is the highest at 2177 seconds. This is because the bin-packing strategy schedules all the applications on the same node without considering the contention for shared resources. Also from Figure 5, our method is able to improve the throughput of Montage GALFA application by 4.58% relative to the throughput when executed alone. This also shows that our method is able to mitigate interference.

5. Conclusions

In this paper, we propose an interference-aware scheduling method for both BOT and scientific workflows in different experiments. For BOT applications, the proposed method is compared with the Weighted Mean Method and evaluated for dynamic workloads whilst for scientific workflows the methods efficiency is evaluated for three different container placement strategies. From each of our experiments, we establish the fact that there is performance interference among applications when coexecuted using containers on the same node. Our proposed method however reduces the amount of interference when compared with other methods.

In the future we further investigate different interference prediction approaches for applications run in OS-level virtualized environments.

Notations

$\mathbb{Q}$:	A queue with a set of applications waiting to be scheduled
x_h:	An application in a queue $\mathbb{Q}$ with profile data on resource usage and an application in Cluster(i)
Cluster(i)(x_h, c_i):	An ith Cluster containing x_h data points and centroid c_i
x_j:	Another data point in Cluster(i) selected for coexecution with x_h
c_i:	The centroid of Cluster(i)
r_{jh}:	The interference ratio between x_j and x_h
SelectX_j(x_h, c_i):	Randomly selecting an application from a cluster
CalcDist(x, c):	Calculating the distance between data point and centroid
CalcR(x, c):	Calculating interference between two points
SelectMin(r_{jh}):	Selecting the application with the least interference ratio
α:	A threshold value for interference

default:	Container placement strategy indicated by the user
$\mathbb{N}$:	A list of nodes in the node cluster
n_i:	An ith node from the list of nodes in the node cluster
cont_h:	An hth container on node(i)
FindNode():	Returns available node according to the parameter indicated
min_{cont}:	Node with minimum number of containers
max_{cont}:	Node with maximum number of containers
ran_{cont}:	Randomly selected available node.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (Ministry of Science and ICT) (NRF-2015M3C4A7065646).

References

[1] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?" in *Proceedings of the 17th Asia-Pacific Network Operations and Management Symposium (APNOMS '15)*, pp. 507–510, Republic of Korea, August 2015.

[2] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proceedings of the 15th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*, pp. 171-172, USA, March 2015.

[3] W. Li, A. Kanso, and A. Gherbi, "Leveraging Linux containers to achieve High Availability for cloud services," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E '15)*, pp. 76–83, USA, March 2015.

[4] Amazom Elastic Containers, https://aws.amazon.com/ecs/.

[5] Jelastic, https://jelastic.com/.

[6] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[7] B. Varghese, O. Akgun, I. Miguel, L. Thai, and A. Barker, "Cloud benchmarking for performance," in *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom '14)*, pp. 535–540, Singapore, December 2014.

[8] H. Yoon, M. Ravichandran, and K. Schwan, "Distributed Cloud Storage Services with FleCS Containers," *Open Cirrus Summit*, 2011.

[9] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '13)*, pp. 233–240, March 2013.

[10] M. G. Xavier, M. V. Neves, and C. A. F. D. Rose, "A performance comparison of container-based virtualization systems for MapReduce clusters," in *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '14)*, pp. 299–306, Italy, February 2014.

[11] "Docker Swarm, clustering for docker," https://docs.docker.com/swarm/.

[12] Apache Mesos, http://mesos.apache.org/.

[13] Google Kubernetes, http://kubernetes.io/.

[14] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the Eurosys Conference*, pp. 275–287, Portugal, March 2007.

[15] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and B. Bianchini, *DejaVu: accelerating resource allocation in virtualized environments*, 2012.

[16] R. C. Chiang and H. H. Huang, "TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments," in *Proceedings of International Conference the IEEE Transactions on Parallel and Distributed Systems*, 2011.

[17] Docker, http://www.docker.com.

[18] "Docker Swarm rescheduling policies," https://github.com/Docker/swarm/blob/master/experimental/rescheduling.md.

[19] F. Nadeem and T. Fahringer, "Predicting the execution time of grid workflow applications through local learning," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, USA, November 2009.

[20] Y. Zhang, W. Sun, and Y. Inoguchi, "Predicting running time of Grid tasks based on CPU load predictions," in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID '06)*, pp. 286–292, Spain, September 2006.

[21] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1, pp. 281–297, University of California Press, 1967.

[22] R. Duan, R. Prodan, and L. Xiaorong, "Multi-Objective Game Theoretic Scheduling of Bag-of-Tasks Workows on Hybrid Clouds," *IEEE transaction on cloud computing*, vol. 2, no. 1, 2014.

[23] Melt, http://lammps.sandia.gov/doc/Section_example.html.

[24] Yahoo Cloud Benchmark, https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[25] Glances, https://www.maketecheasier.com/glances-monitor-system-ubuntu/.

[26] K. Younggyun, K. Rob, B. Paul, B. Mic, W. Zhihua, and P. Calton, "An analysis of performance interference effects in virtual environments," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[27] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. de Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[28] Montage, http://montage.ipac.caltech.edu/.

[29] CFD, https://www.cfd-online.com/Wiki/Main_Page.

[30] J. E. G. Peek, C. Heiles, K. A. Douglas et al., "THE GALFA-HI survey: Data release 1," *The Astrophysical Journal Supplement Series*, vol. 194, no. 2, article 20, 2011.