This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

Achieving Fairness-aware Two-level Scheduling for Heterogeneous Distributed Systems

Eunji Hwang, Jik-Soo Kim, and Young-ri Choi, *Member, IEEE*

Abstract—In a heterogeneous distributed system composed of various types of computing platforms such as supercomputers, grids, and clouds, a two-level scheduling approach can be used to effectively distribute resources of the platforms to users in the first-level, and map tasks of the users in nodes for each platform in the second-level for executing many-task applications. When scheduling heterogeneous resources, service providers of the system should consider the fairness among multiple users as well as the system efficiency. However, the fairness cannot be achieved by simply distributing an equal amount of resources from each platform to every user. In this paper, we investigate how to address the fairness issue among multiple users in a heterogeneous distributed system. We present three first-level resource allocation policies of a provider affinity first policy, an application affinity first policy, and a platform affinity based round-robin policy, and two second-level task mapping policies of a most affected first policy and a co-runner affinity based round-robin policy. Using trace-based simulations, we evaluate the performance of various combinations of the first and second level scheduling policies. Our extensive simulation results demonstrate that the first-level policy plays a crucial role to achieve relatively good fairness.

Index Terms—Heterogeneous distributed computing systems, two-level scheduling, fairness, high-throughput computing, many-task computing

1 Introduction

To solve a large-scale problem in various scientific domains such as physics, chemistry, astronomy, and pharmaceuticals, a loosely coupled application which consists of *many* tasks, from tens of thousands to even billion tasks, is commonly used. High-Throughput Computing (HTC) paradigm [1], which focuses on executing loosely coupled applications composed of CPU-intensive tasks, has expanded to Many-Task Computing (MTC) paradigm [2]. In MTC, it is required to execute a huge number of tasks with potentially having a large variance in runtimes and resource usage patterns within a relatively short period of time.

To achieve the desired performance for such loosely coupled many-task applications, we can utilize as many resources as possible from a heterogeneous distributed computing system which is composed of various types of computing platforms or clusters, such as *supercomputers*, *grids* and *clouds*. These heterogeneous platforms are configured differently in terms of hardware and software. Thus, the configurations of each platform on the hardware such as CPU microarchitecture and the amount of memory, the software stack from OS and libraries to the middleware that manages job submission and execution, and the network & storage settings are different from each other.

A heterogeneous (distributed) computing system can use a *two-level scheduling* approach to effectively distribute resources of the platforms to multiple users and map tasks of the users in computing nodes for each platform. In the first level, the computing resources of the platforms are allocated to each user based on a *resource allocation policy*. In the second level, the tasks of the applications submitted by the users are assigned to nodes in each platform based on a *task mapping policy*.

Many-task applications submitted to a heterogeneous computing system have different resource usage patterns. The performance of a many-task application is basically determined by which platform's resources are used to run it. However, with the advance on the multi-core technology, the number of cores per node increases, making multiple tasks from the same application or different applications be executed on the same node. The interference effect caused by co-running tasks, i.e. *co-runners*, on the performance of the application cannot be simply ignored. Therefore, a first level policy and a second level policy can make a decision based on the *affinity of each application to a platform* and the *affinity of each application to co-running applications*, respectively, in order to achieve the maximum performance.

When scheduling heterogeneous resources for workloads of multiple users, service providers of heterogeneous computing systems should consider the fairness among the users as well as the efficiency of the system, especially for production-level systems where many active users are sharing a common infrastructure. However, the fairness cannot be achieved by simply distributing an equal amount of computing resources from each platform to every user. Some users may prefer the resources from the most of the platforms, while other users may prefer the resources from one or two specific platforms. Also, it is possible that a user is allocated preferred resources, but the tasks of the user are placed on nodes with some co-runners that degrade the

- *E. Hwang, and Y. Choi are with the School of Electrical and Computer Engineering, UNIST, Republic of Korea. E-mail: {hwangej88,ychoi}@unist.ac.kr*
- *J. Kim is with Department of Computer Engineering, Myongji University, Republic of Korea. E-mail: jiksoo@mju.ac.kr*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

TABLE 1
Heterogeneous computing platforms used in the experiments

Platform Type	PLSI		Grid	Cloud
Computing Platform	kias.gene (gene)	unist.cheetah (cheetah)	darthvader.kisti.re.kr (darth)	Local cloud (lcloud)
CPU	AMD Opteron 2.0GHz (2 Core)	Intel Xeon 2.53GHz (8 Core)	Intel Xeon 2.0GHz (8 Core)	Intel Xeon 2.0GHz (12 Core)
Memory Size	8 GB	12 GB	16GB	32 GB (2.4GB per VM)
Total # of Nodes	64	61	8	6
Network	1Gbps	1Gbps	1Gbps	1Gbps
Storage	GPFS	GPFS	Separate SE	Local FS
Management SW	LoadLeveler [3]	LoadLeveler [3]	PBS [4]	TORQUE [5]

performance of the user significantly. With the simple fair allocation of the resources, the system ends up having poor fairness in these cases.

In this paper, we investigate how to effectively address the fairness among multiple users in a heterogeneous computing system. Depending on a resource allocation policy used in the first level and a task mapping policy used in the second level, the two-level scheduling mechanism provides different levels of the fairness to the users. Therefore, we examine several first level resource allocation policies, and second level task mapping policies. We then evaluate the performance of various combinations of the first and second level policies, using trace-based simulations, in order to understand how different policies in the first and second levels affect the overall fairness of users. We use traces obtained from experiments of real scientific applications on production-level computing platforms.

Specifically, we propose three first level scheduling policies: a *provider affinity first (PAF) policy* where the cores of each platform are distributed to the users based on the provider's preference to the applications to improve the efficiency of the platform, an *application affinity first (AAF) policy* where the cores of each platform are allocated to the users based on the preference of each user (or application) to the platforms to reduce the execution time, and a *platform affinity based round-robin (PA-based RR) policy* which distributes the cores of the platforms in a round-robin fashion such that each user is assigned cores from her/his most preferred platform in turn. In addition, this paper presents two second level scheduling policies, a *most affected first (MAF) policy* and a *co-runner affinity based round-robin (CA-based RR) policy*. For each platform, with the MAF policy, a user with the worst performance impact by co-runners selects a combination of tasks to be executed together on the same node, while with the CA-based RR policy, every user takes turns to select a combination of co-runners with which the user has the shortest runtime.

Our extensive simulation results show that a first level decision on resource allocation strongly affects the fairness of the system, rather than a second level decision on task mapping. Our results also demonstrate that in the first level, the PA-based RR policy which fairly distributes cores of the system to many-task applications while considering the platform affinity of each application provides the best fairness. In the second level, the MAF policy where an application with the high performance degradation by co-runners selects a combination of co-runners as much as possible provides the best fairness. This is because a degree of the co-runner effect on the performance of each application varies widely, so that there is no benefit to make an application with a relatively small co-runner effect determine the mapping of co-runners.

To summarize, the main contributions of this paper are as follows. We first investigate several ways to define metrics for the platform and co-runner affinities, and show the importance of using a refined platform affinity metric which considers both aspects of the platforms and applications to achieve the good performance, especially the fairness. Second, we present three resource allocation policies in the first level, which use the platform affinities of the applications to make a decision. We also propose two task mapping policies in the second level, where a combination of co-runners is selected based on their co-runner affinities. Finally, through the comprehensive simulation study based on the execution traces of real scientific applications, we analyze how different resource allocation and task mapping decisions in the first and second levels affect the performance of a heterogeneous computing system with respect to not only the efficiency but also the fairness.

2 Background

In this section, we first provide a brief description of a heterogeneous computing system and many-task applications, which were used in our prior study [6]. We then summarize the effects of platforms and co-runners on the performance of each many-task application.

2.1 Heterogeneous Computing Systems

A heterogeneous computing system used in our prior study consists of three production-level platforms (of $gene$, $cheetah$ and $darth$) and one private cloud platform ($lcloud$). Each computing platform is configured with different hardware and software, and also has distinguishable characteristics that can affect the execution of tasks. The detailed specifications of the platforms are shown in Table 1. Each type of the platforms used in the experiments with its characteristics is discussed as follows.

PLSI Partnership and Leadership for the nationwide Supercomputing Infrastructure (PLSI) [12] is a supercomputing platform served by Korea Institute of Science and Technology Information (KISTI) [13]. PLSI integrates geographically distributed supercomputing platforms in Korea. Two PLSI platforms of gene and cheetah were used. Every computing node in the platforms shares a global storage system based on GPFS [14], which can cause a potential performance degradation for I/O-intensive applications [15], [16].

Grids A grid platform (darth) operated by KISTI [13] was used. This platform is composed of a computing element (CE), which provides computing resources, and a storage element (SE), which provides storage resources. To execute

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

TABLE 2
Many-task applications used in the experiments

Application	Description	# of Tasks	Input/Output Data (Intermediate Data)	Characteristics
AutoDock [7]	A suite of automated docking tools used for protein-ligand docking	512	8.0MB/3.1KB (-)	CPU-intensive
Blast [8]	Basic local alignment search tool used for genome sequencing	768	1.5GB/1.9MB (-)	I/O-intensive
CacheBench [9]	Benchmark for memory subsystem	-	-/1.4KB (-)	Memory-intensive
Montage [10]	Astronomical image mosaic engine used for assembling images in flexible image transport system format into composite images called mosaics	1024	74.7MB/2.8MB (970.3MB)	I/O-intensive
ThreeKaonOmega [11]	Multiple N-body calculations used to solve a multi-particle production scattering problem in nuclear physics	2304	-/6.3KB (-)	CPU-intensive

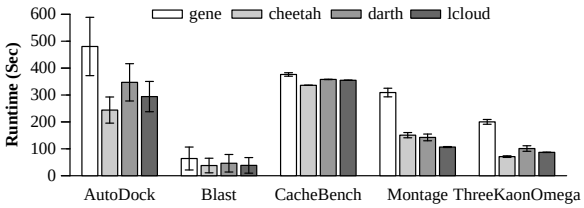


Fig. 1. Effects of different platforms on runtimes

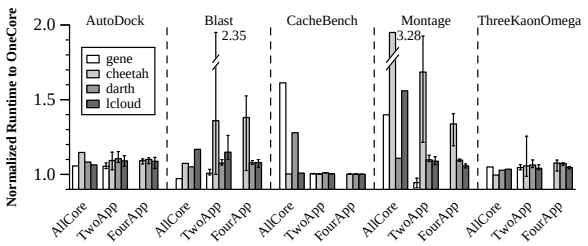


Fig. 2. Runtimes over various co-runner combinations

a task and collect the result of the task in the grid, additional overheads to transfer input/output/executable files between CE and SE are required.

Clouds Our private cloud platform (lcloud) was used. For each VM, its image files are stored in the local disk of each host machine that executes the VM. In the cloud, there are overheads caused by the virtualization layer and resource sharing among multiple VMs running on the same host. In the experiment, there was no on-demand VM provisioning overhead, as we had each worker VM start to run before submitting many-task applications to TORQUE.

2.2 Many-task Applications

In the prior study, four real scientific many-task applications in the areas of pharmaceutics, bioinformatics, astronomy and nuclear physics, and one benchmark that performs heavy memory operations were used to understand and analyze the behaviors of many-task applications on heterogeneous platforms. Each application has different requirements on CPU, memory, and I/O resources, showing a different trend on the performance over the heterogeneous platforms.

Table 2 presents a detailed description of each of the five applications with the sizes of input and output data. Note that due to the limitation on time and resources to run experiments, we used relatively small-scale applications. In the real-world workloads, the above many-task applications except CacheBench are executed with much larger numbers of tasks (for examples, 40,000 queries for Blast [17] and more than one million tasks for Montage [18]). For AutoDock, up to 4,096 IBM BlueGene/P nodes were used to execute it [19]. In case of ThreeKaonOmega, only a subset of tasks from more than two million possible tasks was used. However, we believe that our representative MTC workloads are still enough to show major characteristics of each real many-task application in terms of resource usage patterns on top of heterogeneous distributed systems.

2.3 Effects of Platforms and Co-runners

The performance of the five many-task applications in the heterogeneous computing system was thoroughly studied [6]. Figure 1 shows the average task runtimes of the many-task applications over the platforms. In the figure, the runtime of each application was measured without any co-runners in a node, i.e. with `OneCore` configuration, regardless of the number of cores in the node, and each bar presents the standard deviations of task runtimes. From the results, we can observe the followings. The average runtime and variance of tasks for each application are dissimilar to each other. Also, even though there is a tendency that all the applications except Montage have the shortest runtime on cheetah and have the longest runtime on gene, making cheetah the most preferred platform and gene the least preferred one, the degree of the performance difference over the platforms is quite different from each other. A many-task application can slow down up to 2.82 times in gene, compared to the runtime in its fastest platform.

Figure 2 shows the effects of co-runners on the performance of the five many-task applications. In these experiments, a task of each application is executed in a node with n cores with various combinations. With `AllCore`, n tasks from the same application are executed together. With combinations of two applications (`TwoApp`), the equal number of tasks from each of two applications are executed together. With combinations of four applications (`FourApp`), the equal number of tasks from each of four applications are executed together. In the figure, for each application, the average runtimes in `AllCore`, `TwoApp`, and `FourApp`, normalized to that with `OneCore`, are shown along with a bar with the minimum and maximum normalized runtimes.

From the results, we can easily see that each application suffers from a different level of the performance degradation due to the interference by co-runners, and the co-runner effect of the application appears differently over the different platforms. Also for some applications such as Blast and ThreeKaonOmega in cheetah, the difference between the minimum and maximum runtimes is large, showing that the performance can be strongly affected by the composition of co-runners. In cheetah, when tasks of Blast and ThreeKaonOmega run together with Montage tasks, the negative effect on the performance becomes high. The co-runner effect can degrade the performance of a many-task application up to 3.28 times. To summarize, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

can conclude that both of the platform and co-runner effects cannot be neglected on two-level scheduling of many-task applications in a heterogeneous computing system.

3 Fairness in Heterogeneous Computing Systems

A traditional approach to provide the fairness among multiple users is to distribute computing resources (e.g. CPU cores or CPU cycles) equally to applications submitted by the users. For a heterogeneous computing system composed of different computing platforms, we can simply allocate the equal number of cores from each platform to each application or user[1] to achieve the fairness as discussed in [20]. However, depending on the characteristics of each application, the application has different sets of preferred platforms and co-runners. Some applications mostly run fast enough regardless of used resource types and co-running applications, while other applications may run efficiently only in a small subset of the platforms and/or have good performance only when they are co-located with certain applications. Thus, distributing all the available cores of the system fairly (i.e. equally) to applications does not mean that the equal amount of preferred resources is allotted to each application, and the degree of the performance degradation caused by co-runners is similar for all applications. Such a *simple fairness policy* likely fails to achieve the fairness. For fair two-level scheduling, a first level policy to distribute cores of platforms to applications should consider the affinity of the application to the platforms, while a second level policy to map tasks of the assigned applications to computing nodes of each platform should account the affinity among the applications.

In a heterogeneous computing system, we can compute the ideal performance of an application using *Fastest-OneCore* algorithm. With *Fastest-OneCore*, each application is executed by using the resources from its most preferred platform without any co-runners. To compute the runtime of each application with *Fastest-OneCore*, we first compute a fair share k, i.e. the number of cores to be allotted to each user by the above simple fairness approach. We then compute a hypothetical runtime of the application when it is assigned k nodes from its most preferred platform (where it runs the fastest in the system), and its tasks are executed without any co-runners on each of the assigned nodes. Therefore, this hypothetical runtime is the shortest one for the application by using total k cores in the system.

In this work, we define the fairness of users with a given scheduling algorithm as follows. For each user u, a throughput of the application submitted by u with the scheduler, t_u, is computed and then normalized to its throughput with *Fastest-OneCore*. The scheduling algorithm is considered to be fair if the performance of all users is similarly improved or degraded compared to their ideal performance, i.e., performance with *Fastest-OneCore*. A fairness metric is computed based on the coefficient of variation form [21]:

$$fairness = 1 - \frac{\sigma_t}{\mu_t}$$

where μ_t is the average normalized throughput of all users, and σ_t is the standard deviation of t_u for all users. For a scheduling algorithm that provides a high level of fairness, the variation should be relatively tiny, compared with the average value of the normalized throughputs of the users, making the value of the fairness metric close to one.

In heterogeneous computing systems where diverse resources can be selectively used for each application and its tasks can be carefully mapped to a node with applications of certain types, the fairness should be computed based on the "best possible performance" of each user (application), which is reflected in *Fastest-OneCore*. This is because it is more important how fairly the resource allocation and task assignment are done from the perspective of each application to maximize its performance, rather than simply distributing the resources of each platform evenly to each application.

4 Platform and Co-runner Affinities

To quantify the effects of platforms and co-runners on the performance of many-task applications, we define two metrics of the platform affinity and co-runner affinity for each application. A platform affinity of an application K to a platform P represents how good to run application K on P, while a co-runner affinity of application K to platform P represents a degree of performance degradation of application K on platform P by co-runners. These metrics can be defined in several ways. In this section, we discuss three definitions of the platform affinity and two definitions of the co-runner affinity.

4.1 Platform Affinity Metric

For all the metrics below, a higher metric value means that it is more beneficial to run an application K on a platform P, rather than the other platforms in the system.

Throughput A throughput of a many-task application on a platform with `OneCore` (i.e., the number of tasks processed using one core per hour) can be used for platform affinity as in [20], [22]. It is a simple *raw* metric to show the performance of the application on that particular platform. The throughput of the application is higher as the task runtime of the application is shorter. This metric cannot show the relative importance of a platform for an application, compared to other applications.

Egocentric Platform Affinity An *egocentric platform affinity* (EPA) metric for an application K to a platform P indicates how suitable platform P is to run application K compared to other platforms. For the metric, we compute the performance of K when the equal number of cores from each platform except P is used to complete all the tasks of K to identify the usefulness of P for K. To do so, we compute the average task runtime of K with `OneCore` over all the platforms except P, and normalize it by the runtime of K with `OneCore` on platform P. For example, the EPA of AutoDock on cheetah is 1.532($=\frac{480.42+347.11+294.16}{3}/244.05$), since the runtimes of AutoDock on gene, cheetah, darth and lcloud are 480.42, 244.05, 347.11, 294.16. A metric value which is less than 1 can mean that the runtime of K becomes shorter by not using any resource from P and so P is not so

1. Note that in this work, we use two terms of user and application interchangeably.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

TABLE 3
Platform affinity metrics

Application	Throughput (tasks/hour)				Egocentric Platform Affinity[1]				Reciprocal Platform Affinity[2]			
	gene	cheetah	darth	lcloud	gene	cheetah	darth	lcloud	gene	cheetah	darth	lcloud
AutoDock	7.49	14.75	10.37	12.24	0.614	1.532	0.978	1.214	0.966	1.169	0.918	0.974
Blast	56.28	94.97	77.66	93.58	0.640	1.308	1.009	1.284	1.009	0.997	0.955	1.042
CacheBench	9.57	10.71	10.06	10.15	0.929	1.080	0.994	1.006	1.473	0.854	0.988	0.847
Montage	11.64	23.88	25.28	33.76	0.431	1.234	1.326	1.883	0.684	0.892	1.203	1.484
ThreeKaonOmega	17.98	50.69	35.62	41.44	0.431	1.822	1.181	1.428	0.678	1.341	1.060	1.102

[1]Suitability of P to K, [2]Combined suitability of P to K and K to P, where P is a platform and K is an application.

useful to K. On the other hand, a value which is larger than 1 can mean that the the runtime becomes longer without using resources from P and so P is important to K. This platform affinity metric is similarly defined in [6].

Reciprocal Platform Affinity A *reciprocal platform affinity* (RPA) metric of an application K to a platform P indicates how *suitable* platform P is to run application K compared to the other platforms, and also how *suitable* application K is to be executed on platform P compared to the other applications. For given sets of many-task applications and platforms, this metric can be computed as follows. As the first step, for every platform, we compute the average task runtime of all applications with `OneCore`, and normalize the runtime of each application with `OneCore` by the average runtime of the platform. Then we compute the average of the runtimes of K, which are normalized in the first step, over all the platforms except P, similar to the egocentric platform affinity metric. Finally, we normalize the computed runtime of K to the (normalized) runtime of K on P with `OneCore`. For example, on cheetah, the average runtime of all the applications with `OneCore` is 167.97 seconds. In case of AutoDock on cheetah, its runtime is 244.05 seconds, so that its normalized runtime is 1.453 (=$244.05/167.97$). In the same way, the normalized runtimes of AutoDock on gene, darth, and lcloud are computed as 1.680, 1.744, and 1.669, respectively. Thus, the average normalized runtime of AutoDock, not including that on cheetah, is 1.698, and the RPA metric of AutoDock to cheetah is computed as $1.169 (= 1.698/1.453)$. If the RPA value is high, we can conclude that executing K on P is beneficial from the perspectives of both K and P.

Analysis Table 3 shows the three platform affinity metric values for the applications in the experiments [6]. In all the platforms, the runtime of Blast is the shortest among all applications, so that it has the highest throughput. However, the resource of lcloud is more critical to Montage than Blast, as Blast has high throughputs in other platforms. The throughput metric cannot reflect the comparative importance of lcloud to Montage, but EPA of Montage on lcloud is higher than that of Blast.

When the throughput and EPA are used, the platform rankings of all the applications except Montage are the same as cheetah, lcloud, darth and gene in order from the highest to lowest ones. However, for RPA, the platform rankings of the applications are different from each other. With RPA, CacheBench which has a small variation on runtimes over all the platforms has a high value for gene, and Blast has the highest value for lcloud, as it is more beneficial that other applications like AutoDock and ThreeKaonOmega use the resources of cheetah rather than Blast.

TABLE 4
Co-runner affinity metrics

Application	Raw difference[1] (sec.)				Normalized difference[2] (%)			
	gene	cheetah	darth	lcloud	gene	cheetah	darth	lcloud
AutoDock	26.75	24.05	34.85	25.47	5.57	9.86	10.04	8.66
Blast	0.44	12.80	3.46	4.62	0.69	33.76	7.47	12.01
CacheBench	47.59	0.90	13.30	1.30	12.65	0.27	3.72	0.37
Montage	24.70	106.72	14.01	13.57	7.99	70.78	9.84	12.73
ThreeKaonOmega	9.41	4.42	6.45	3.77	4.70	6.22	6.38	4.34

[1] Average of $C - S$, [2] Average of $(C - S)/S \times 100$, where S is a solorun runtime and C is a runtime with co-runners.

4.2 Co-runner Affinity Metric

For two co-runner affinity metrics discussed below, a higher metric value means that the effect of co-runners on the performance of an application K is stronger, compared to other applications, on a platform P.

Raw Difference A *raw difference* co-runner affinity metric uses the runtime difference (in seconds), of application K between a run with `OneCore` (i.e. solorun) and a run with co-runners on platform P. The co-runner affinity of K to P is computed as follows. For each possible combination composed of some number of applications including K which are assigned some cores of P, we first compute the runtime difference (i.e. the average runtime of tasks with the co-runner combination minus the average solorun runtime). We then compute the average over all the combinations on P. If the value is x, then the execution time of K increases by x seconds on average by running together with other applications on P.

Normalized Difference A *normalized difference* co-runner affinity metric uses the runtime difference, in percentage (%), of application K between a solorun and a run with co-runners, normalized by the solorun runtime. We compute the value of the co-runner affinity of K to P in the same ways as in the raw difference except that we normalize the computed average difference value by the solorun runtime, and then multiply it by 100. If the value is y, then the execution time of K increases by y% on average with co-runners on P.

Analysis Table 4 shows the two co-runner affinity metric values of the applications for each of the platforms. Note that these values are computed based on runtimes of a subset of the tasks of applications (which are randomly selected) with various combinations from the experiments [6]. The normalized difference can reflect a degree of slowdown due to co-runners, compared to a solorun runtime, showing the co-runner effects on the performance of K on P clearly. Also it can show the relative impact of co-runners on the performance of K compared to other applications on P. For example, on gene, the values of the raw difference for AutoDock and Montage are similar, but the performance degradation of Montage is more severe than that of AutoDock, having a higher value of the normalized

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

difference for Montage. Also, for AutoDock, even though the raw differences on gene and lcloud are similar, the performance effect is stronger on lcloud, having a higher normalized difference value on lcloud.

5 Two-level Scheduling Algorithm

In this section, we first discuss the overview of two-level scheduling algorithms, and then present three resource allocation policies which can be used in the first level, and two task mapping policies which can be used in the second level. In a two-level scheduling algorithm, a resource allocation policy basically decides the number of cores to be allocated to a user for each of the platforms. Once the first level scheduling is done, for each platform, a set of the users assigned to the platform and the number of cores allocated to each of the users are decided. In each platform, a task mapping policy then maps tasks of the users to nodes in the platform.

For a many-task application composed of a large number of tasks, the system can employ pilot jobs to reduce scheduling overhead. For each submitted application, the system creates a separate queue to maintain its tasks. For an allocated core to the application, a pilot job is deployed to continuously fetch and execute a task of the application from the queue until there is no task left in the queue [6], [16], [20], [23].

The scheduling algorithm needs to be invoked when there are some changes in the system such as the submission and termination of an application, the addition and removal of resources to and from the system. On redistributing cores to applications, it may need to re-deploy a pilot job to another node. In this case, the system adopts the non-preemption mechanism to let running tasks complete before the re-deployment of the pilot job [6], [16], [20]. Also, for the system, we assume that some services for job submission and account management [16], data management to transfer input/output files to/from computing platforms [24] are provided to run many-task applications.

```
1: struct User{
2:     int task; // num of remaining tasks in queue
3:     tuple comb[c][m]; // c is num of combinations of co-runners
4:     int share[m]; // num of allocated cores for each platform
5:     double CA[m]; // dynamic co-runner affinity metric
6: }
7: struct Platform{
8:     int avCores; // num of available cores
9:     double PA[n]; // platform affinity of each user
10: }
```

In the two-level scheduling algorithm, the above data structures of $User$ and $Platform$ are used, where n is the number of users, and m is the number of platforms. For users, attribute $task$ is set to the number of remaining tasks in the job queue of the user as input. The initial value of $task$ is the total number of tasks for a many-task application. Attribute $comb[k][j]$, where k is in 0 ... $c-1$, and c is the total number of combinations of users, is to store a tuple $< cSet, cVal >$, where $cSet$ is a set of users in that combination k, and $cVal$ is a (task) runtime of the user with combination k on platform j, and this information is provided as input. Attributes $share[j]$ and $CA[j]$, where j is 0 .. $m-1$, are used to store the number of allocated cores for a platform j to the user, and a dynamically computed co-runner affinity of the user on platform j, respectively, and their initial values are all 0.

For a platform, attribute $avCores$ is the number of cores to be allocated to users, and initially its value is the number of all available cores in the platform. Attribute $PA[i]$, where i is 0 .. $n-1$, is used to store a platform affinity value of each user to the platform.

5.1 First Level Resource Allocation Policies

5.1.1 Provider Affinity First and Application Affinity First policies

In this section, we discuss two first level policies, called a *Provider Affinity First* (PAF) policy and an *Application Affinity First* (AAF) policy, which allocate the resources of $f$avored platforms with respect to a resource provider and a user application, respectively, to each application.

Algorithm 1 Affinity First Policy (first-level)

```
1: Input: U: User[n];
2:        P: Platform[m];
3: Var: d: int; // initially 0;
4:      Z: int[n][m]; // initially 0, favored platform matrix
5:      S: int[m]; // initially 0, num of assigned users for each plat.
6: Compute P[j].PA[i] of each plat. j for user i with remaining tasks;
7: while SumAvailCores(P) > 0 ∧ SumTask(U) > SumShare(U) do
8:    Set Z[i][j] to 1 if i favors j for each user i and platform j;
9:    If no selection was made in line 8 at all, assign some user(s) to a
      platform with available cores;
10:   S := SumColumn(Z);
11:   for each i in U in increasing order sorted by U[i].task − Sum(U[i].share) do
12:      for each j in P in decreasing order sorted by P[j].PA[i] do
13:         if Z[i][j] > 0 then
14:            d := Min(U[i].task − Sum(U[i].share), P[j].avCores / S[j]);
15:            Z[i][j], S[j] := 0, S[j] − 1;
16:            U[i].share[j] := U[i].share[j] + d;
17:            P[j].avCores := P[j].avCores − d;
18:         end if
19:      end for
20:   end for
21: end while
```

Algorithm 1 presents the overview of PAF and AAF policies. In the algorithm, we first compute the platform affinity values for each user with remaining tasks to all the platforms with available cores. This step is only needed for RPA metric, since the values of the throughput and EPA for each application are computed by only using its runtimes over the platforms, and so they have no need to be updated during the execution (unless a new platform is added or an existing platform is removed to/from the system). Thus, at each scheduling invocation, the platform affinity values can be dynamically changed. In a process of allocating cores to users, based on each policy, we select *favored* platforms for users (line 8), and if none of platforms are selected at all by any users, we assign each platform with remaining cores to some user(s) (line 9).

Next, we distribute the cores of the platforms to the users based on the assignment computed as the above. Basically for the available cores of each platform, we allocate the equal number of the cores to each of the assigned users (line 11-20). However, we do not need to allot more cores than the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

demand of a user. Thus, we allocate the cores of a platform to each user in increasing order sorted by the number of tasks which still need to be allocated cores. In this way, all the available cores can be assigned to a user who requires more cores. Also, for each user, we assign cores from a platform with a higher platform affinity value first to the user. After this process, it is possible that the available cores of some platforms have not been distributed yet, and some users still need to get more cores. In this case, we repeat the above process. Note that in each step of the algorithm, we only consider a user who needs to get more cores and a platform that has available cores.

Provider Affinity First When selecting favored platforms of users (line 8), for each platform, we first sort the users in decreasing order by their affinities to the platform. We compute that a platform provider favors a user, if the user is within the highest (i.e., top) k% users for the platform. Note that k should be an integer which is less than or equal to 50, since this is a threshold to differentiate favored users.

Note that in case of PAF, k% of users with remaining tasks are selected for a platform. Thus, if the number of the users reduces, only a small number of users are assigned to the platform, which will make it hard to find good combinations of users in the second level. A computed allocation matrix in one run is provided as input for the next run, and based on this information, we make a user continuously execute its tasks on a previously assigned platform.

Application Affinity First For the favored platform selection, for each user, we first sort the platforms in decreasing order by the platform affinity values of the user. We then consider a platform as a favored one of the user if the platform is within the highest k% platforms for the user.

5.1.2 Platform Affinity based Round-Robin Policy

Algorithm 2 Round-Robin Policy (first-level)

1: **Const**: D_{unit} // unit of allocating cores, $D_{unit} \geq 1$
2: **Input**: U: User[n];
3: P: Platform[m];
4: **Var**: r, d: int; // initially 0;
5: $fairShare$: int[n]; // fair share of each user
6: Compute $P[j].PA[i]$ of each plat. j for user i with remaining tasks;
7: Compute $fairShare[i]$ of each user i with remaining tasks;
8: **while** $Sum(fairShare[i]) > SumShare(U)$ **do**
9: **for** i in $0..n-1$ where $fairShare[i] - Sum(U[i].share) > 0$ **do**
10: Get a plat. r, where $P[r].avCores > 0$, with max PA for $U[i]$;
11: $d := Min(D_{unit}, fairShare[i] - Sum(U[i].share), P[j].avCores)$;
12: $U[i].share[r] := U[i].share[r] + d$
13: $P[r].avCores := P[r].avCores - d$;
14: **end for**
15: **end while**

Algorithm 2 presents a *Platform Affinity based Round-Robin* (PA-based RR) policy. Like PAF and AAF policies, the PA-based RR policy computes the platform affinities of users dynamically at each run when RPA metric is used. The algorithm computes a fair share of cores per user. Basically, for the total available cores in the system, the same number of cores is assigned to each user as a fair share, but when some user has less number of remaining tasks than the fair share, the unneeded cores are equally distributed to the users who need more cores to run their tasks. Then the available cores from the heterogeneous platforms are assigned to users in a round-robin fashion. For each turn of a user, the user selects a platform with the highest platform affinity value among the platforms that still have available cores, and up to D_{unit} core(s) can be allotted to the user.

5.2 Second Level Task Mapping Policies

Algorithm 3 Task Mapping Policy (second-level)

1: **Input**: U: User[n];
2: P: Platform[m];
3: **Var**: U_{done}: set of Users; // initially $null$;
4: **for** each j in P **do**
5: Set U_{done} to $null$;
6: Add $U[i]$ to U_{done} for each $U[i]$ in U if $U[i].share[j] = 0$;
7: **while** $|U_{done}| < n$ **do**
8: Compute $U[i].CA[j]$ of each $U[i]$ in U where $U[i] \notin U_{done}$; // only for MAF
9: Place some combination(s) of tasks on available nodes in platform j;
10: Update $share$ of users and U_{done} according to above placement;
11: **end while**
12: **while** $U[i].share[j] > 0$ for some i **do**
13: Assign possible random comb. of $U[i]$ to an available node in platform j;
14: Decrease $share$ for each user in random comb. accordingly;
15: **end while**
16: **end for**

An overview of a task mapping policy used in the second level is given in Algorithm 3. For each of the platforms, we find a co-runner combination of the users and place the combination on a node based on a task mapping policy (line 9). If only a few number of shares (or pilot jobs), which are not enough to make any combination by a task mapping policy, from some of the users remain, we make a random combination and assign it to a node (line 12-15).

5.2.1 Most Affected First Policy

A *Most Affected First* (MAF) policy allows a user who suffers from co-runners severely to select a combination of users which will be placed together on a computing node. For each of the platforms, we first compute the co-runner affinity value of each user assigned to the platform dynamically by considering only the users who are assigned to the platform and still can be used to make a combination (line 8). We then search the user with the maximum co-runner affinity (whose performance can be most degraded by co-runner tasks) to the platform. For the selected user i', we find a possible combination c with the smallest runtime (i.e., $cVal$). We then attempt to map this combination to a node as much as possible in the platform until the remaining number of shares for some of the users in c becomes insufficient to create a whole combination (i.e. $U[i].share[j] < \frac{P[j].nCore}{|U[i'].comb[c][j].cSet|}$ for some $U[i]$ in c where $P[j].nCore$ is the number of cores per node in platform j) (line 9). We also decrease the shares of the users in c and update U_{done} which is a set of users who are allocated all or nearly all of the shares for the platform accordingly (line 10). If there are users whose tasks have not been mapped yet, we repeat the above process to assign a new co-runner combination from the remaining users to nodes. (A similar algorithm was discussed in in [6].)

Note that if two or more users have the same maximum co-runner affinity value because the users execute the same

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

application or they happen to have the same value, the algorithm can work in a round-robin fashion for those users similar to the below policy.

5.2.2 Co-runner Affinity based Round-Robin Policy

In a *Co-runner Affinity based Round-Robin* (CA-based RR) policy, for each platform, all the users who are assigned to the platform take a turn to select a combination and map the selected combination up to P_{unit} node(s). Each user selects one of the possible combinations, which has the smallest runtime of the user with co-runners, on the platform (line 9). Similar to MAF policy, if the share of a user becomes zero, or too small to make a combination, then the user will be added to U_{done} (line 10). This round-robin scheduling process is repeated until all users belong to the U_{done}.

5.3 Discussion

We analyze the time complexity of the proposed first and second level policies in terms of the number of users (i.e. n) in the system, since in general the number of platforms (i.e. m) is relatively smaller than n and it does change often. We then discuss how to measure the platform and co-runner affinities of many-task applications.

5.3.1 Time Complexity of the First Level Policies

PAF and AAF To compute the time complexity of PAF and AAF, we assume that k is 50. The time complexity of computing the platform affinity of n applications is $O(n)$ in line 6. In PAF, at the first iteration of the while loop in line 7, all the platforms are selected by some user(s). Once a platform is selected, all the available cores of the platform are assigned to the user(s) (who favor the platform), unless all the users are allocated all of the needed cores. An additional iteration may be needed for the platform, if the total number of remaining tasks for the user(s) is less than the number of available cores for that platform. If only one user who still needs more cores is left, but there are one or more platforms with available cores, no selection occurs in line 8. In this case, the platforms can be assigned to the user as favored ones (line 9). Thus, the additional iterations are bounded by n. The complexity for line 8 is $O(n \log n)$ and that for line 11-20 is $O(n \log n)$. The complexity of each iteration for the while loop (in line 7) is $O(n \log n)$. Hence, the complexity of PAF is $O(n^2 \log n)$.

In AAF, at each iteration of the while loop in line 7, at least $\lfloor \frac{m}{2} \rfloor$ platforms are selected as favored ones in line 8. If there is only one platform with available cores, a user or users cannot make a selection in line 8. In this case, the platform can be assigned to a user whose platform affinity value to the platform is higher than or equal to the median platform affinity value over all the users who need more cores (line 9). Therefore, there can be additional iterations bounded by n, similar to PAF. The complexity for line 8 is $O(n)$ and that for line 11-20 is $O(n \log n)$. The complexity of each iteration for the while loop is $O(n \log n)$. Therefore, the complexity of AAF is $O(n^2 \log n)$ (= $O((\log \frac{m}{2} + n) \times n \log n)$).

PA-based RR The time complexity of computing fair shares for users is $O(n \log n)$ (in line 7). For PA-based RR, each user who needs more cores is assigned up to D_{unit} cores at a time in one iteration of the while loop (in line 8). The maximum number of iterations for the while loop is bounded by $\frac{Sum(P.avCores)}{D_{unit}}$ under the assumption that the total number of available cores in each platform and the number of tasks for each user are multiples of D_{unit}. The complexity of each iteration for the while loop is $O(n)$. Thus, the time complexity of PA-based RR is computed as $O(n \log n)$ (= $O(n + n \log n + \frac{Sum(P.avCores)}{D_{unit}} \times n)$).

5.3.2 Time Complexity of the Second Level Policies

We compute the complexity of mapping cores to users for each platform (i.e., the complexity of line 5-15).

MAF For each iteration of the while loop (in line 7), at least one user is inserted to U_{done}. Thus, an upper bound on the number of iterations of the while loop is n. For each iteration, the policy needs to compute the co-runner affinity for each of users (in line 8), and to search all possible combinations of a selected user with at most n users on a platform composed of nodes with k cores each (in line 9). For a user, the total number of possible combinations of co-runners is ${}_{n+k-2}C_{k-1}$, i.e., the total number of selecting $k-1$ users from n users with repetition, since that user should be selected at least once. Thus, the complexity of each iteration for the while loop is $O(n^k)$(= $O(n^k + n^{k-1})$), making the complexity of the policy $O(n^{k+1})$.

CA-based RR For CA-based RR, each user who is assigned to a platform takes a turn to place a combination of co-runners up to P_{unit}. For each user, it searches possible combinations of co-runners with at most n users on a node with k cores each (in line 9). Therefore, the complexity of each iteration for the while loop is $O(n^k)$. The maximum number of iterations for the while loop is bounded by $\frac{N_{node}}{P_{unit}}$, where N_{node} is the total number of nodes in the platform, under the assumption that N_{node} and the number of tasks for each user are multiples of P_{unit}. Hence, the time complexity of CA-based RR is computed as $O(n^k)$.

Discussion on the second level complexity The complexity becomes huge if the number of cores per node is large. In practice, to reduce the complexity of the algorithm, we can consider mapping at most two users to each node, regardless of the number of cores per node. When two users are assigned to a node, the cores of the node are evenly distributed to the users. In this case, the value of k becomes 2 and so the complexities of MAF and CA-based RR become $O(n^3)$ and $O(n^2)$, respectively. Note that in many previous scheduling algorithms which reflect the effect of co-runners, only pairwise co-location of applications on a node is considered due to the intricacy of understanding the effect of a large number of co-runners on the same node [25]–[27]. Furthermore, with regard to MAF, the co-runner affinity values of users for a platform are recomputed for each iteration of the while loop in order to exclude users who cannot be used for mapping anymore. However, in most of cases, the updated values are similar to the previous ones. If we compute the co-runner affinity values only once (after line 6), the complexity of MAF becomes $O(n^k)$. For pairwise co-location, it is $O(n^2)$. We will discuss how the above approaches to reduce the complexity affect the performance in Section 6.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing

TABLE 5
Default resource configuration

	gene	cheetah	darth	lcloud
# of nodes	300	75	75	50
# cores per node	2	8	8	12
# total cores	600	600	600	600

TABLE 6
The number of tasks in default workload

	AutoDock	Blast	CacheBench	Montage	ThreeKaonOmega
# tasks	34,600	248,800	31,230	72,950	112,420

5.3.3 Measuring Platform and Co-runner Affinities

To compute the platform and co-runner affinity metrics of an application, we need the (average) task runtime of the application with `OneCore`, and its task runtime with each possible combination on every platform. In this work, we perform off-line profiling to measure these runtimes in advance, which incurs overhead. However, at runtime, for a small subset of tasks of each submitted application, we can measure the runtimes with `OneCore` and with the possible combinations on some dedicated nodes of every platform. Based on the measured runtimes of the tasks in the subset, we can compute its platform and co-runner affinity values, and continuously get feedback to update the values, as more tasks of the application are executed (which is similar to adaptive runtime profiling techniques [28], [29]).

6 Results

6.1 Evaluation Methodology

We have simulated our two-level scheduling algorithms using a trace-based simulator in C++. The trace of task runtimes was obtained from the experiments of real scientific applications running on top of production-level computing platforms [6]. For `TwoApp` and `FourApp` combinations, runtimes of a subset of tasks for the applications were measured. Therefore, for each application, the regression analysis was used to estimate a runtime of a task with some co-runner combination from its runtime with `OneCore`, and the estimated runtimes are used for the simulations as in [6]. Note that we did not have any exclusive access to the production-level systems of gene, cheetah, and darth, and so we could not replace their scheduling software for evaluating our policies. Since we could not use each platform exclusively during our experiments, it may have been influenced by other applications co-running at that time. However, the effects of different hardware configurations, software stack, and network & storage settings of real platforms were still reflected in the trace. Therefore, the overall performance trend of our algorithms measured by the simulator based on application execution trace will be retained in a real production-level environment.

The default resource configuration of a heterogeneous computing system is given in Table 5. The total number of cores in the system is 2,400. The number of tasks for each application used in the default workload is given in Table 6. The total number of tasks in the workload is 500,000. To generate the workload, the tasks of the applications from the trace were replicated. In the default workload, the numbers of tasks for the applications were decided such that if the equal number of cores from each platform is assigned to each of the applications and each task is executed with `OneCore`, the makespan of each application to complete all of its tasks is similar to each other.

TABLE 7
Metrics and policies used in our simulations

Feature	First Level	Second Level
Metrics	• Throughput (Thr.) • Egocentric Platform Affinity (EPA) • Reciprocal Platform Affinity (RPA)	• Raw Difference • Normalized Difference
Policies	• Fairness • Platform Affinity First (PAF) • Application Affinity First (AAF) • Platform Affinity based Round-Robin (PA-based RR)	• AllCore • Random • Most Affected First (MAF) • Co-runner Affinity based Round-Robin (CA-based RR)

TABLE 8
Simulation scenarios

Feature	Default	W/o Plat.	S/L Plat.	W/o App.	S/L App.
# of scenarios	1	4	8	5	5
# of plat. used	4	3	4	4	4
# of app. used	5	5	5	4	5

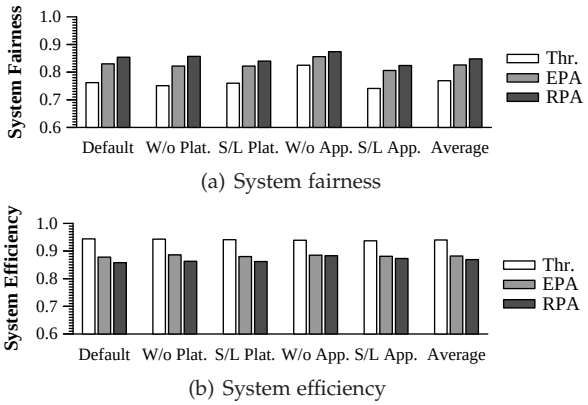
In each simulation run, all applications are submitted at the same time. The simulator was implemented such that if multiple pilot jobs of an application attempt to fetch and execute a task from the queue of the application at the same time, the task is assigned to one of the pilot jobs randomly. Thus, we ran 100 simulation runs and computed the average value for each of the simulation results.

For comparison of the first level policies, we have also used the simple fairness policy. This policy allocates the cores of each platform equally to every user, but if the number of tasks for a user is smaller than the total number of cores assigned to the user, the remaining cores of the user are distributed equally over other users who need more cores. For the second level, we have used two additional policies, `AllCore` and `Random`. `AllCore` (which assigns the tasks of the same application to a node) can be commonly used in a production-level cluster without knowing other applications running on the cluster. `Random` randomly maps tasks of the applications to a node, which represents the average performance of the task mapping since the tasks of the applications are executed with various combinations. In our simulations, we randomly select a possible combination of applications among the combinations that we have the trace of task runtimes. For PAF and AAF, k is set to 50. By default, D_{unit} for PA-based RR and P_{unit} for CA-based RR are assigned to 1.

Along with the fairness metric defined in Section 3, we also measure a system makespan normalized to that when the fairness policy is used in the first level and `AllCore` is used in the second level, for the system efficiency. Note that unlike the fairness metric, *Fastest-OneCore* which computes a hypothetical runtime of each application on its fastest platform without co-runners cannot be used to evaluate overall performance of the system to support multiple many-task applications.

We evaluate the performance of two-level scheduling algorithms of different combinations of the first level and second level policies over various scenarios. Table 7 summarizes all the platform and co-runner metrics, and all the first level and second level policies considered in our evaluation with their abbreviations. We analyze how the metrics and policies perform over different compositions

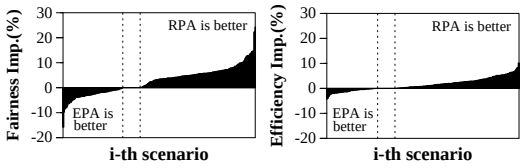
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing



(a) System fairness

(b) System efficiency

Fig. 3. Performance over various platform affinity metrics



(a) System fairness (b) System efficiency

Fig. 4. Performance difference caused by EPA and RPA metrics

of a heterogeneous computing system regarding the types and amounts of resources. We also investigate their performance over various workloads of multiple many-task applications, which have different resource requirements. Thus, each scenario varies depending on the number of platforms, the number of cores in each platform, the number of applications, and/or the number of tasks composed of each application. Total 23 scenarios are categorized into five types as follows.

- Default: The default workload given in Table 6 is executed with the default resource configuration given in Table 5.
- W/o Platform (*W/o Plat.*): The default workload is used, but the resources from one of the four platforms are not used at all.
- Small/Large Platform (*S/L Plat.*): The default workload is used. However, the number of cores for one of the four platforms is either reduced to half of or doubled from that in the default configuration, becoming either a "small" or "large" platform. For the rest of the platforms, the default resource configuration is used.
- W/o Application (*W/o App.*): The default resource configuration is used, but only four applications out of the five applications are submitted.
- Small/Large Application (*S/L App.*): The default resource configuration is used, but the numbers of tasks for the applications vary. For a "small" application, the number of tasks is reduced to half of that in the default workload, while for a "large" application, the number of tasks becomes twice that in the default workload. For each workload in this type, one of the five applications is selected as a small application, while the remaining applications become large applications.

Table 8 shows the number of scenarios, and used platforms and applications for each of the scenario types.

6.2 Effects of Different Platform and Co-runner Affinity Metrics

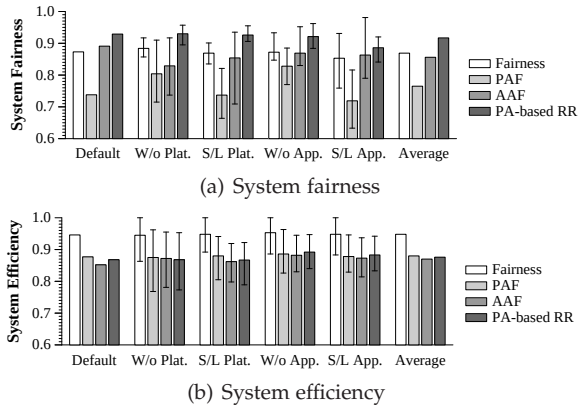
Figure 3 shows the system fairness and efficiency by different platform affinity metrics of the throughput, EPA and RPA. For each result, we consider all combinations of the first level policies which use a platform affinity metric, i.e. all except the fairness policy, with every second level policy. (Note that if a fairness value is closer to one, we achieve higher fairness, while if an efficiency value is smaller, we achieve higher efficiency.) Each of the results is the average value over all the executions in each scenario type.

When the throughput is used with PAF, all available resources of the system are allocated first to Blast and ThreeKaonOmega which have shorter runtimes than the other applications, and until Blast and ThreeKaonOmega are done, the other applications suffer from starvation. The throughput metric does not present the resource preference among the applications properly, having the lowest system fairness and efficiency. With EPA, in many scenarios, some application is allotted more resources than the other applications except PA-based RR policy, whereas with RPA, in most cases, the applications are assigned a similar number of the cores from the system in total. Therefore, on average, the system fairness of RPA is 10% and 3% higher than those of the throughput and EPA, while the efficiency of RPA is 8% and 2% higher than those.

Figure 4 compares the performance of EPA with that of RPA over all the scenario runs with different combinations of the first and second level policies (except the fairness policy in the first level). For each of the runs, we compute how the fairness or efficiency with RPA is better than that with EPA. In the figure, the results of all the runs are sorted in increasing order of performance improvement, and value 0% means that the resource allocation and task assignment by EPA and RPA are done in the same way. For 59.7% of the scenarios, the fairness with RPA is higher up to 24.3% than that with EPA, while for 64.3%, the efficiency with RPA is higher up to 10.2% than that with EPA. Note that in some cases of *S/L Plat.* scenarios, when EPA is used, all the applications except Montage receive a larger number of cores from their fastest platforms or a smaller number of cores from their slowest platforms, ending up having a higher fairness value than that with RPA.

For the co-runner affinity, the two metrics of the raw difference and normalized difference can make a performance difference only for MAF policy in the second level. We evaluate all combinations of MAF with every first level policy. In most of scenarios, we observe that the most affected application is the same, or the same co-runner combination is selected, regardless of which metric is used. There are certain combinations of co-runners, which degrade the performance significantly such as co-running Montage with other applications on cheetah, and co-running Montage and Blast on gene. However, the resource allocation is done by the first-level policies except the fairness policy in a way that such co-runner combinations are unlikely occur. For example, the cores of cheetah is not assigned to Montage. Therefore, even for some cases in which different co-runner

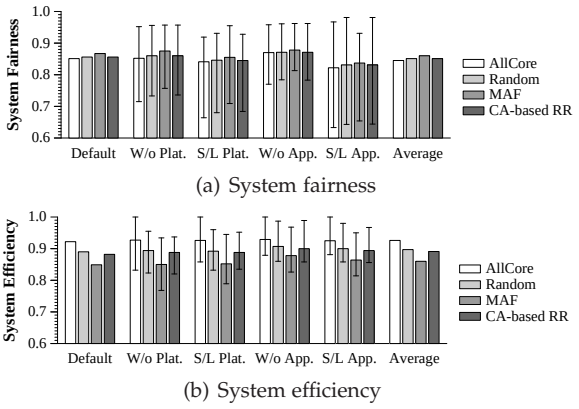
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing



(a) System fairness

(b) System efficiency

Fig. 5. Performance over various first level policies



(a) System fairness

(b) System efficiency

Fig. 6. Performance over various second level policies

combinations are selected depending on which metric is used, their performance is similar in our simulations.

However, the normalized difference can have better performance if there is a many-task application which has a short task runtime and is strongly affected by co-runners on some platform. Such an application may not be selected as the most affected one by the raw difference. Making this application select the co-runner combination with the minimum performance degradation can be helpful to improve the throughput of the platform. In each platform, we analyze all possible resource allocations for the five applications (in which 2, 3, 4, or 5 of them are assigned to the platform), and estimate the throughput of the platform based on runtime traces. In the analysis, the throughput of a platform with the normalized difference is up to 8% higher, compared with the raw difference.

6.3 Best Policy at Each Level

We investigate which policy in each level shows the best performance. For this study, we use RPA and the normalized difference as the platform affinity and co-runner affinity metrics, respectively. Figure 5 shows the system fairness and efficiency over the four first level policies with a bar that presents the minimum and maximum results in each scenario type except the default one. In the figure, for each of the first level policies, we run simulations of all the scenarios with every second level policy and average the performance over the simulations. The analysis on these simulation results are as follows.

- For the fairness policy, it distributes the equal amount of resources from each platform to applications without considering their platform affinities. Thus, it has the lowest efficiency but provides some degree of the fairness.
- For AAF, each of the users is assigned some number of cores from its favored platforms, having a shorter system makespan. However, each platform has a different number of applications that favor the resources of the platform. Thus, it is possible that some application is assigned a larger number of cores from its favored platforms, resulting in lower fairness.
- PAF only considers the perspective of the platforms. Thus, some applications are allocated a large amount of favored resources, while other applications starve for favored resources. Thus, it can provide the system efficiency similar to that with the other policies, but it achieves the low fairness.
- With PA-based RR, all users are allocated the same amount of preferred resources, unlike AAF and PAF. Thus, it achieves the highest fairness and competitively good system efficiency compared to all the other first level policies.

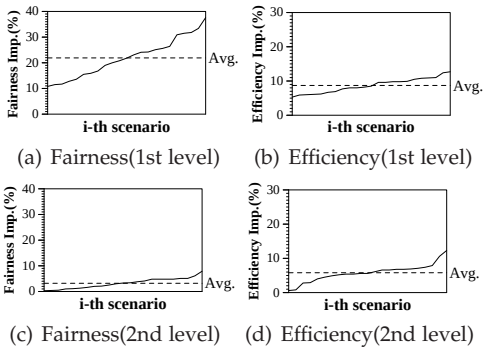
Figure 6 shows the performance over the four second level policies with a bar for the minimum and maximum results. Similar to Figure 5, each result of a second level policy is the average over all the scenarios with every first level policy. From the results, we analyze the followings.

- With `AllCore`, some applications such as CacheBench on gene show the worst performance, having the lowest fairness and the longest makespan.
- A degree of the performance effect by co-runners varies widely depending on the applications. Hence, with MAF, the most affected application selects a co-runner combination with the minimum performance degradation, improving the overall performance and consequently providing the best fairness and efficiency among the second level policies.
- With CA-based RR, each application ends up being executed with various combinations, and a combination selected by each application with a small co-runner affinity value does not help to improve the performance, even though the runtime of the application can decrease in a few cases. Thus, its performance in terms of the fairness and efficiency is similar to that of `Random`.

Note that in Figure 6, we observe that all the fairness values tend to be lower than those in Figure 5, because we execute each second level policy with all the first level policies including PAF that shows the lowest fairness.

With the best first level policy (i.e. PA-based RR), the improvement of MAF on the fairness is 2.0% on average and 5% at maximum, and that on the efficiency is 3.7% on average and 9% at maximum, compared to `Random`. As we can see from the research on real Grid workload, users can be categorized into "normal users" who frequently submit relatively small numbers of jobs and "data challenge users" who occasionally submit much larger numbers of jobs [30]. This type of data challenge pattern can be also demonstrated by some national labs in U.S. where only a few

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2836444, IEEE Transactions on Services Computing



(a) Fairness(1st level) (b) Efficiency(1st level)

(c) Fairness(2nd level) (d) Efficiency(2nd level)

Fig. 7. Impacts of first and second level policies

number of mission critical workloads are supported on their supercomputing resources. Therefore, the service providers can appropriately adopt our proposed two-level scheduling mechanisms based on their resource usage patterns and associated workloads. When supporting a small number of data challenge users in a heterogeneous computing system, the performance improvement by MAF can make a significant difference on the users and system even with its higher time complexity. However, if the system needs to support a large number of normal users, `Random` might be a better choice.

6.4 First Level Effect vs. Second Level Effect

We investigate how the decision at each level can affect the performance of the system. Figure 7(a) and (b) show the performance improvement caused by a different choice for a first level policy, where all the four first level policies are used with RPA metric and MAF in the second level for total 23 scenarios in Table 8. Figure 7(c) and (d) show the performance improvement caused by a different choice for a second level policy, where all the four second level policies are used with PA-based RR and RPA metric in the first level for the scenarios. For each scenario, we compute how the performance with the best (first or second level) policy is better than that with the worst policy on that scenario.

As shown in the figure, the fairness is strongly affected by which policy is used in the first level, and a used policy in the second level has a small effect on the fairness, because possible co-runner combinations depend on which applications are assigned to each platform by a first level policy. For the efficiency, the effect by a first level policy is higher than that by a second level policy on average. However, at the worst case, the effect of the second level decision on efficiency is similar to that of the first level decision.

6.5 Discussion

Varying D_{unit} and P_{unit} In PA-based RR, when D_{unit} increases to 200 cores (i.e. 8% of the total cores in the system) for the default setting, the fairness is degraded by 3% while the efficiency remains the same, compared to the performance when $D_{unit} = 1$. Similarly, in CA-based RR, when P_{unit} increases to 20 nodes (i.e. 4% of the total nodes), there is almost no performance difference, compared to the performance when $P_{unit} = 1$.

Reducing the complexity of MAF In the above results, all the combinations of co-runners analyzed in Section 2.3 were considered and the co-runner affinity value of a user was kept recomputed before selecting a new combination. We observe that the effect of considering the combinations with up to two users on the performance is negligible. Updating the co-runner affinity value may be critical when several applications with high co-runner affinities are allocated relatively a small number of cores for a platform. However, in our simulations, there is almost no performance difference even when its value is computed only once. These results show that the complexity of MAF can be potentially further reduced without significant performance degradation.

Dynamic submission We also studied the performance trend when applications in a workload are submitted dynamically based on a Poisson distribution with four different mean rates. For the experiments, we used the default and *W/o App.* workloads in the default resource configuration. In general, the first level and second level policies show a similar performance trend as all the applications are submitted at the same time. However, the fairness of PAF improves. This is because in PAF, an early submitted application can continuously execute its tasks on previously assigned platforms, and consequently, more applications can be assigned to a platform.

7 Related Work

Defining the fairness In earlier studies, the fairness of users was defined based on their performance slowdowns similar to our work, such as the maximum slowdown [31], [32] and the ratio of the maximum slowdown to the minimum slowdown [33]. With the fairness index [34], the best fairness is achieved when all users have the same degree of the performance slowdown [35]. However, the ideal performance of each application is computed as that when all the available resources are used without considering interference effects among its own tasks [32], or when an even share of each type of heterogeneous resources is used without reflecting different behaviors of application performance on platforms [20], [31].

Scheduling many-task applications In a heterogeneous computing system, several scheduling algorithms for many-task applications have been investigated to improve the efficiency [6], [20], [22], [36]. A scheduling algorithm that considers the suitability (i.e. importance) of a platform to an application as well as that of an application to a platform was proposed [36]. Also, preference rankings of applications for different platforms provided by users are used on scheduling many-task applications [22]. In these studies, the effect of co-runners is not taken into account. A two-level scheduling algorithm is proposed to allocate the cores of each platform to users whose EPA values are higher than or equal to the average EPA value of the platform in the first level, similar to PAF, and use MAF in the second level [6]. However, in the first level, the number of users who are assigned to a platform can be quite different from each other, generally having lower fairness compared to PAF. In the second level, the co-runner affinity values of the users for each platform are computed once by considering all the users in the system and are used in all cases without reflecting the current assignment of the users to the platform. Resource allocation policies were proposed to

achieve efficiency, fairness, and/or user satisfaction without considering the co-runner effects [20]. A fairness-aware scheduling policy for bag-of-task applications was studied for a large-scale platform in which the number of tasks for each application is much less than the number of nodes [32].

Scheduling independent tasks Techniques to map and schedule independent tasks on heterogeneous computing platforms have been investigated [37]–[39], in which the effects of different machine types on the performance are taken into account on scheduling. Moreover, the scheduling heuristics for independent tasks such as Min-min and Sufferage [37], [38] map a single task to a machine at a time. Thus, when they are used without modification in our setting, basically only one application is assigned to a platform, which usually shows poor performance.

Heterogeneity and interference aware scheduling The impact of heterogeneity of platforms and interference caused by co-runners has been considered to support web-service applications [40], single-thread and multi-threaded applications [41], distributed analytics frameworks and latency critical services [42] in large-scale datacenters. Nathuji et al. proposed a framework to capture the effects of interference, and provision additional resource to guarantee the QoS for CPU bound applications [43]. The application and platform aware resource allocator was proposed, but a single multi-core server was considered [44].

Fairness on clouds and big data analytics platforms The fairness among multiple users has been considered in big data analytics frameworks [45]–[47]. The fairness regarding the locality of input blocks is considered on scheduling [45], [46]. Xu et al. presented a proportional share I/O scheduler to achieve the fairness for competing data-intensive applications [47]. For allocating multiple types of resources to users with diverse resource demands, the dominant resource fairness (DRF) is proposed to allocate the equal amount of the dominant resource to each user [48] without considering the interference effect. Tang et al. presented the resource-as-you-contributed fairness, where each user can use the resource proportional to the resource contribution of the user to other users [49]. For privately shared datacenters, a task co-location technique was presented, where the co-location of two tasks is fair if their performance is degraded in a similar level [27].

8 Concluding Remarks

In this paper, we studied the fairness in two-level scheduling for heterogeneous distributed computing systems to support multiple many-task applications with various resource requirements. We discussed three resource allocation policies used in the first-level and two task mapping policies in the second level. We showed that the fairness of the system is mostly affected by which resource allocation policy is used in the first level, because possible co-runner combinations in task mapping are limited by the decision in the first level. In our evaluation results, the performance difference on the fairness is up to 38% depending on a first level policy, while it is up to 8% depending on a second level policy.

Our study can also be useful for resource providers (such as KISTI [13]) to build and maintain a heterogeneous computing system that can effectively support different types of challenging applications. If a resource provider needs to support various many-task applications whose performance is strongly affected by which platforms are used, the fairness can be improved by building the system to have a similar amount of resources for each type of platforms respectively in order to maintain the best performance of each application type. Otherwise, the provider can simply build the system to secure a large amount of resources as much as possible to increase the overall system throughput. As a future work, we are planning to build a pilot service for large-scale many-task applications in a few selected domains with our collaborating resource providers.

Acknowledgment

This work was partly supported by Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2015-0-00590, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and National Research Foundation of Korea (NRF) funded by the Korea government(MSIT) (NRF-2015R1C1A1A02037400 and NRF-2016M3C4A7952604).

References

[1] M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP journal*, vol. 11, no. 1, 1997.
[2] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *1st Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.
[3] "Tivoli workload scheduler loadleveler." https://www-03.ibm.com/systems/power/software/loadleveler/.
[4] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters.," in *Annual Linux Showcase & Conference*, 2000.
[5] "Torque resource manager." http://www.adaptivecomputing.com/products/open-source/torque/.
[6] S. Kim, E. Hwang, T. kyung Yoo, J.-S. Kim, S. Hwang, and Y. ri Choi, "Platform and co-runner affinities for many-task applications in distributed computing platforms," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.
[7] "AutoDock." http://autodock.scripps.edu/.
[8] "Blast." http://blast.ncbi.nlm.nih.gov/Blast.cgi.
[9] "CacheBench." http://icl.cs.utk.edu/llcbench/.
[10] "Montage." http://montage.ipac.caltech.edu/.
[11] H.-Y. Ryu, A. I. Titov, A. Hosaka, and H.-C. Kim, "ϕ photoproduction with coupled-channel effects," *Progress of Theoretical and Experimental Physics*, vol. 2014, no. 2, 2014.
[12] "PLSI." http://www.plsi.or.kr/.
[13] "Korea Institute of Science and Technology Information." http://en.kisti.re.kr/.
[14] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.
[15] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. Foster, "MTC Envelope: Defining the capability of large scale computers in the context of parallel scripting applications," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, 2013.
[16] J.-S. Kim, S. Rho, S. Kim, S. Kim, and S. Hwang, "HTCaaS: Leveraging distributed supercomputing infrastructures for large-scale scientific computing," in *6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, 2013.
[17] B. Zhang, Y. Ruan, T. L. Wu, J. Qiu, A. Hughes, and G. Fox, "Applying twister to scientific applications," in *IEEE 2nd International Conference on Cloud Computing Technology and Science*, 2010.
[18] M. Rynge, G. Juve, J. Kinney, J. Good, G. Berriman, A. Merrihew, and E. Deelman, "Producing an infrared multiwavelength galactic plane atlas using Montage, Pegasus and Amazon web services," in *23rd Annual Conference for Astronomical Data Analysis Software and Systems*, 2013.

[19] A. P. Norgan, P. K. Coffman, J.-P. A. Kocher, D. J. Katzmann, and C. P. Sosa, "Multilevel parallelization of AutoDock 4.2," *Journal of cheminformatics*, vol. 3, no. 1, 2011.

[20] E. Hwang, S. Kim, T.-k. Yoo, J.-S. Kim, S. Hwang, and Y.-r. Choi, "Resource allocation policies for loosely coupled applications in heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, 2016.

[21] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[22] E. Hwang, S. Kim, J.-S. Kim, S. Hwang, and Y.-R. Choi, "On the role of application and resource characterizations in heterogeneous distributed computing systems," *Cluster Computing*, vol. 19, no. 4, 2016.

[23] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A fast and light-weight task execution framework," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[24] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: Uniform resource provisioning and access for clouds and grids," in *Fourth IEEE International Conference on Utility and Cloud Computing*, 2011.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[26] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[27] Q. Llull, S. Fan, S. M. Zahedi, and B. C. Lee, "Cooper: Task colocation with cooperative games," in *IEEE International Symposium on High Performance Computer Architecture*, 2017.

[28] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in a shared internet hosting platform," *ACM Transactions on Internet Technology*, vol. 9, no. 1, 2009.

[29] S. Kim, J.-S. Kim, S. Hwang, and Y. Kim, "Towards effective science cloud provisioning for a large-scale high-throughput computing," *Cluster Computing*, vol. 17, no. 4, 2014.

[30] B. T. Quang, J.-S. Kim, S. Rho, S. Kim, S. Kim, S. Hwang, E. Mednernach, and V. Breton, "A comparative analysis of scheduling mechanisms for virtual screening workflow in a shared resource environment," in *Proceedings of CCGrid-Life Workshop*, 2015.

[31] Y. Kwon, C. Kim, S. Maeng, and J. Huh, "Virtualizing performance asymmetric multi-core systems," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[32] J. Celaya and U. Arronategui, "Fair scheduling of bag-of-tasks applications on large-scale platforms," *Future Generation Computer Systems*, vol. 49, 2015.

[33] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[34] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*, vol. 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, 1984.

[35] J. Wang, N. Abu-Ghazaleh, and D. Ponomarev, "Controlled contention: Balancing contention and reservation in multicore application scheduling," in *IEEE International Parallel and Distributed Processing Symposium*, 2015.

[36] J. Xiao, Y. Zhang, S. Chen, and H. Yu, "An application-level scheduling with task bundling approach for many-task computing in heterogeneous environments," in *Network and Parallel Computing*, 2012.

[37] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[38] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings Eighth Heterogeneous Computing Workshop*, 1999.

[39] E. U. Munir, S. Mohsin, A. Hussain, M. W. Nisar, and S. Ali, "SDBATS: A novel algorithm for task scheduling in heterogeneous computing systems," in *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2013.

[40] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[41] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[42] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[43] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[44] P. Tembey, A. Gavrilovska, and K. Schwan, "Merlin: Application- and platform-aware resource allocation in consolidated server systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[45] 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*, 2010.

[46] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[47] Y. Xu and M. Zhao, "IBIS: Interposed big-data I/O scheduler," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016.

[48] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[49] S. Tang, B.-S. Lee, and B. He, "Fair resource allocation for data-intensive computing in the cloud," *IEEE Transactions on Services Computing*, 2016.

Eunji Hwang received the BS and MS degree in informatics from Gyeongsang National University, Jinju, Gyeongsangnam-do, Republic of Korea, in 2011 and 2013, respectively. She is currently working toward the PhD degree under the supervision of Professor Young-ri Choi at Ulsan National Institute of Science and Technology (UNIST). Her research interests include cloud computing, big-data processing and resource scheduling.

Jik-Soo Kim received his B.S. and M.S. in Computer Science and Statistics from Seoul National University in Korea, Ph.D. in Computer Science from University of Maryland at College Park, USA. He is currently an Assistant Professor at the Department of Computer Engineering of Myongji University. His primary research interests are in the design and analysis of distributed computing infrastructures to support Many-Task Computing, Cloud Computing and Data-intensive Computing.

Young-ri Choi received the BS degree in computer science from Yonsei University, Seoul, Korea, in 1998. She received the MS and PhD degrees in computer science from the University of Texas at Austin in 2002 and 2007, respectively. Her research interests include cloud computing, scientific computing, big data analytics platforms, and network protocols. She is currently an associate professor in the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology (UNIST).