Doctoral Thesis

Colocation-aware Resource Management for Distributed Parallel Applications in Consolidated Clusters

Seontae Kim

Department of Electrical and Computer Engineering
(Computer Science and Engineering)

Graduate School of UNIST

2020

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Colocation-aware Resource Management for Distributed Parallel Applications in Consolidated Clusters

Seontae Kim

Department of Electrical and Computer Engineering
(Computer Science and Engineering)

Graduate School of UNIST

Colocation-aware Resource Management for Distributed Parallel Applications in Consolidated Clusters

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy of Science

Seontae Kim

12/16/2019

Approved by

[illegible]

Advisor

Young-ri Choi

Colocation-aware Resource Management for Distributed Parallel Applications in Consolidated Clusters

Seontae Kim

This certifies that the thesis/dissertation of Seontae Kim is approved.

12/16/2019

signature

[illegible]

Advisor: Young-ri Choi

signature

[illegible]

Beomseok Nam: Thesis Committee Member #1

signature

[illegible]

Woongki Baek: Thesis Committee Member #2

signature

[illegible]

Won-ki Jeong: Thesis Committee Member #3

signature

[illegible]

Myeongjae Jeon: Thesis Committee Member #4;

Abstract

Consolidated clusters, which run various distributed parallel applications such as big data frameworks, machine learning applications, and scientific applications to solve complex problems in wide range of fields, are already used commonly. Resource providers allow various applications with different characteristics to execute together to efficiently utilize their resources.

There are some important issues about scheduling applications to resources. When applications share the same resources, interference between them affects their performance. The performance of applications can be improved or degraded depending on which resources are used to execute them based on various characteristics of applications and resources. Characteristics and resource requirements of applications can constrain their placement, and these constraints can be extended to constraints between applications. These issues should be considered to manage resource efficiently and improve the performance of applications.

In this thesis, we study how to manage resources efficiently while scheduling distributed parallel applications in consolidated clusters.

First, we present a holistic VM placement technique for distributed parallel applications in heterogeneous virtual cluster, aiming to maximize the efficiency of the cluster and consequently reduce cost for service providers and users. We analyze the effect of heterogeneity of resource, different VM configurations, and interference between VMs on the performance of distributed parallel applications and propose a placement technique that uses a machine learning algorithm to estimate the runtime of a distributed parallel application.

Second, we present a two-level scheduling algorithms, which distribute applications to platforms then map tasks to each node. we analyze the platform and co-runner affinities of loosely-coupled applications and use them for scheduling decision.

Third, we study constraint-aware VM placement in heterogeneous clusters. We present a model of VM placement constraints and constraint-aware VM placement algorithms. We analyze the effect of VM placement constraint, and evaluate the performance of algorithms over various settings with simulation and experiments in a small cluster.

Finally, we propose interference-aware resource management system for CNN models in GPU cluster. We analyze the effect of interference between CNN models. We then propose techniques to mitigate slowdown from interference for target model, and to predict performance of CNN models when they are co-located. We propose heuristic algorithm to schedule CNN models, and evaluate the techniques and algorithm from experiments in GPU cluster.

Contents

List of Figures

Chapter 1

Introduction

Consolidated clusters are commonly used to serve various application workloads from multiple users. Many users in various areas run their applications on the resource provided by the cluster. Big data frameworks such as Hadoop [1] and Spark [2] are already common for users processing and analyzing large volumes of data from various fields such as science, medicine, business, and engineering. People in various fields want to execute their scientific applications from a wide range of scientific domains (e.g., astronomy, physics, pharmaceuticals, and chemistry). In the several years, with the considerable increase in computing power, machine learning applications have attracted the spotlight for finding solutions in various research fields. Resource providers allow the simultaneously execution of various applications to utilize their resources efficiently.

Applications have different characteristics and resource requirements. Some applications require space for the huge amount of data, and the others require a large amount of computing resources. In many cases, the processes of these applications are distributed on multiple servers in parallel, in order to satisfy their requirements. By using multiple servers, they can obtain sufficient space to store the data and/or sufficient amount of computing resources.

There are several issues for consolidated clusters shared by multiple applications. Applications may share the same resources, but compete with each other to use a limited amount of these resources, so their performance will be degraded by interference. There are various factors that affect such interference, and the intensity of these effects varies depending on the characteristics and the resource requirements of the applications. Resource providers should consider interference to prevent the performance degradation of applications. Applications can cause contentions in various resources, and their consideration of interference may differ because of the different characteristics of the resources.

Resources in consolidated clusters can be heterogeneous, which implies that there are various resources with different characteristics. Because of these different characteristics of resources and applications, the mapping of applications and resources has become an important issue. The performance of applications can be better or worse depending on which resource they were executed on. Moreover, applications can have constraints such as they should be allocated on resources that satisfy their conditions for resources such as the amount of resource required an the requirement of a specific resource. Applications can have constraints for resources, and they may even have constraints between them; for instance, some applications should be allocated to the same resource, or some applications should be allocated to different resources.

In this thesis, we studied the efficient management of resources while considering the characteristics of various environments and applications, the interference between the applications, and the placement constraints of jobs. Here, we present VM placement techniques in heterogeneous clusters that improve the performance of distributed parallel applications and efficiently utilize the cluster while satisfying the placement constraints of VMs. We also present scheduling techniques for loosely-coupled applications while considering the affinity for platforms and their co-runners. We propose interference-aware resource management techniques for deep learning applications using GPU.

The main contributions of this thesis are as follows.

- We present a VM placement technique in the heterogeneous virtual cluster by placing the VMs of multiple distributed parallel applications based on a heterogeneity- and interference-aware performance model and evaluate our technique's ability to improve the performance of a heterogeneous cluster.

- We analyze the platform and co-runner affinities of many-task applications in distributed computing platforms. We then present a two-level scheduling algorithm, which distributes the resources of different platforms to applications based on the platform affinity, and map tasks of the applications to the computing nodes on the basis of the co-runner affinity. We evaluate our algorithm by using a trace-based simulator.

- We present a model of VM placement constraints between VMs, and between VMs and nodes. Then we discuss constraint-aware VM placement algorithms that optimize the performance for either energy saving or load balancing. We analyze the effect of constraints and our algorithms over various settings by using simulations and experiments in a small cluster.

- We analyze the effect of interference between CNN models in a GPU cluster. We then propose techniques to mitigate the effect of interference for the target application, and to predict the effect of interference. We also propose a heuristic algorithm to schedule CNN models while guaranteeing the QoS of the target models while maximizing the overall throughput. We evaluate our techniques and algorithms in the GPU cluster through real experiments.

Chapter 2

Holistic VM Placement for Distributed Parallel Applications in Heterogeneous Clusters

Heterogeneity of hardware configurations for physical nodes exists in a cluster, as physical machines are continuously purchased over time [3–5]. In most cases, a cluster consists of physical nodes of several different types. Each type of physical nodes is configured differently in terms of the CPU microarchitecture and clock speed, the number of cores, the amount of memory, and network and storage settings, providing different performance capabilities. In heterogeneous clusters, various applications can be deployed and executed together on the same node, due to advances in multicore and virtualization technologies.

A heterogeneous cluster is commonly used to run multiple distributed parallel applications. For a distributed parallel application, multiple virtual machines (VMs) form a *virtual cluster* (VC) to run the application in parallel and coordinate their execution by exchanging messages across the VMs. Distributed parallel applications are popularly employed to solve large scale complex scientific problems such as those in molecular dynamics [6, 7] and computational fluid dynamics [8]. They are also used to process huge amounts of data, as in Hadoop [1] and Spark [2].

In a heterogeneous cluster, for all VMs which execute a distributed parallel application together, it may not be possible to allocate homogeneous resources on which the application shows the best performance. This occurs especially with private cloud clusters on small and medium scales, and even with clusters on a public cloud, where some types of resources have been reserved for the cluster over a long term to reduce the cost significantly (as in Reserved Instances in Amazon EC2 [9]). To improve the resource utilization and performance of the cluster, we can configure the VMs of the application with heterogeneous nodes.

When different resources are used to execute a distributed parallel application, there are numerous possible ways to place the VMs of the application depending on factors such as the number of different node types used for the VMs, the number of VMs running on each type of (physical) nodes, and the number of the nodes used for each type. Moreover, even for the same VC configuration, the performance of the application widely varies depending on co-running VMs or applications, i.e., *co-runners*, which are executed on the same nodes [4, 10–12].

When running a distributed parallel application, the heterogeneous hardware configuration and/or different levels of interference on each of the nodes can slow down some of the VMs. Depending on how parallelism and synchronization are implemented for a distributed parallel application, the outcome can differ. For distributed parallel applications which are *loosely coupled* or have a load balancing feature such as big data analytics applications [1, 2] and many-task computing applications [13], a few slow VMs may not affect the performance noticeably and using favored resources in part may improve the performance of the application. However, for *tightly coupled* applications such as scientific MPI-based applications, one slow VM may cause the different execution progress rates over VMs, degrading the final performance significantly. Moreover, if the application has poor parallel efficiency, adding more VMs does not improve the performance, unlike loosely coupled applications. Thus, estimating the application performance is not trivial, as the effect of each factor such as the heterogeneity of resources, interference, VM configuration, and parallelism pattern on the performance is not clear.

Maximizing the efficiency of a cluster is crucial for service providers such as cloud providers, as doing so reduces the overall costs with an eventual price reduction for users [5]. Moreover, a strategy for efficient VM placement is necessary for providers to provide high quality services to users and to enhance user satisfaction. However, in a heterogeneous cluster, finding the best VM placement for distributed parallel applications, which maximizes the overall performance, is challenging due to the intractably large search space and complexity of estimating their performance.

Earlier studies investigate scheduling techniques which take into account the heterogeneity of hardware configurations and/or interference among applications [4, 5, 11, 12, 14, 15]. However, some prior works mainly consider single node applications [4, 12, 14], and a homogeneous cluster is assumed for an interference modeling technique for distributed parallel applications [11]. An interference and heterogeneity aware scheduling technique focuses on supporting applications popularly used in large scale clouds or datacenters such as distributed analytics frameworks, latency critical services, and web services [15]. The technique employs a greedy approach to allocate and assign the least amount of resources while still satisfying quality of service (QoS) constraints of an application, thus reducing the search space for placement.

This work investigates a holistic VM placement technique for various types of distributed parallel applications in a heterogeneous cluster, aiming to maximize the overall performance for the benefit of service providers and users. The proposed technique accommodates various factors that have an impact on performance in a combined manner, while reducing the search space and cost for the best VM placement. First, we analyze the effects of the heterogeneity of resources, different VM configurations, and interference between VMs on the performances of various distributed parallel applications. We then propose a placement technique that uses a machine learning algorithm to estimate the runtime of a distributed parallel application for various VM placement configurations.

For a heterogeneous cluster, the total search space for the VM placement of distributed parallel applications is intractable. Thus, we limit the candidate placements for the applications to a treatable subset of all possible placements in the cluster. To find the best VM placement from the huge search space, we also devise a VM placement algorithm based on simulated annealing. To generate training samples for a performance estimation model, a distributed parallel application is profiled against synthetic workloads that mostly utilize the *dominant resource* of the application, which strongly affects the application performance, reducing the profiling space dramatically.

The main contributions of this work are as follows:

- We analyze the performance of various MPI-based and big data analytics applications from SpecMPI 2007, NAS Parallel Benchmarks (NPB), Spark, Hadoop, and molecular dynamics simulators in a heterogeneous cluster.

- We explore the correlation between the dominant resource usage and performance of parallel applications ultimately to lower the profiling cost.

- To estimate the performance, we apply a machine learning algorithm in order to deal with the complex performance modeling of a parallel application, which must consider many the relevant factors discussed above in a comprehensive manner.

- We show that it is feasible to build a general model which can estimate the performance of various Hadoop and Spark applications. The model, which is essentially built based on off-line profiling runs of some number of big data analytics applications, can estimate the runtime of a target application with any size of input.

- Our extensive experiment and simulation results show that the proposed placement technique can improve the performance of a heterogeneous cluster by placing VMs of multiple applications based on a heterogeneity and interference aware performance model and the simulated annealing approach.

I Methodology

Heterogeneous Virtualized Cluster In our default experiments, we use a heterogeneous cluster which consists of 12 nodes connected via 1 GE switch. Table 2.1 shows the specifications of our heterogeneous cluster. In the cluster, there are two different types of physical nodes. For type T2, each node has 12 cores, but we use only 8 cores, 4 cores per socket, to simplify the experiments. (Note that when all 12 cores are used, there will be more possible placements, and our proposed technique has no limitation on using all of them.) Thus, 64 cores in total (i.e, 32 cores from each node type) are used to run parallel applications. For the network configuration,

Table 2.1: Specifications of our heterogeneous cluster

Type	T1	T2
CPU	Intel quad-core I7-3770 (IvyBridge) 3.40GHz	Intel hexa-core E5-2620 (SandyBridge) 2.0GHz
# sockets	1	2
L3	8MB/socket	15MB/socket
Memory	16GB	64GB (32GB/socket)
# nodes	8	4
Network (Thr.)	1GE ($\sim$ 70MB/s)	1GE ($\sim$ 110MB/s)
Storage (Thr.)	7,200 RPM HDD ($\sim$ 137MB/s)	7,200 RPM HDD ($\sim$ 109MB/s)

Table 2.2: Parallel applications used in our experiments

Type	Name	Size	Abbrev.
SpecMPI 2007	132.Zeusmp2	mtrain	Zeus
NPB	CG	Class C	CG
MPI	LAMMPS	5dhfr	LAMMPS
	NAMD	5dhfr	NAMD
Spark	GrepSpark	12.6 GB	GS
	WordCount	9.5 GB	WCS
	TeraGen	11.0 GB	TG
Hadoop	GrepHadoop	9.5 GB	GH

both T1 and T2 nodes are configured with Gigabit Ethernet, but they are configured with different networking devices, i.e., T1 nodes with a low-end device and T2 nodes with a high-end device. Therefore, T2 nodes show higher network throughput than T1 nodes. For the storage configuration, both nodes are configured with the same disk device, but due to differences in other hardware configurations, T1 nodes show higher storage throughput than T2 nodes. Note that the network and storage performances of different types of VMs on clouds vary [16].

Xen hypervisor version 4.1.4 is installed on each of the physical nodes, and for dom0, Linux kernel version 3.1.0 is used. For a VM, it is configured with two virtual CPUs and 5GB of memory. In all experiments, dom0 is pinned to all cores allocated to the active VMs, which execute the workloads, as we assume virtualized cluster systems where no dedicated cores are exclusively assigned to dom0 in order to maximize the utilization of physical CPU cores and flexibility with regard to resource use. Each parallel application is configured with 8 VMs; therefore, four applications in total can be placed on our cluster concurrently.

Note that it is possible to have a larger VM, i.e., one VM per node. However, we observed that for resource intensive applications, the performance can be improved when the VMs are spread out over multiple nodes and executed with an application which has different resource requirements. We can also use the resources more flexibly with smaller VMs.

Table 2.3: VC configurations used in our experiments

Config	# of VMs		# of nodes (# VMs per node)		List notation
	T1	T2	T1	T2	
C1	8	0	8 (1)	0	T1 $(1,1,1,1,1,1,1,1)$
C2	8	0	4 (2)	0	T1 $(2,2,2,2)$
C3	4	4	4 (1)	4 (1)	T1 $(1,1,1,1)$, T2 $(1,1,1,1)$
C4	4	4	2 (2)	2 (2)	T1 $(2,2)$, T2 $(2,2)$
C5	0	8	0	4 (2)	T2 $(2,2,2,2)$
C6	0	8	0	2 (4)	T2 $(4,4)$

Distributed Parallel Applications Table 2.2 presents parallel applications and their sizes as used in the experiments. We use different parallel workloads from SpecMPI 2007 [17], NPB [18], Spark [2], Hadoop [1], and the two molecular dynamics simulators of LAMMPS [6] and NAMD [7]. Thus, there are four MPI-based applications, which are tightly coupled, and four big data analytic applications, which are loosely coupled. In this work, we focus on scientific and big data analytics applications, as they have different characteristics on communication and synchronization patterns and can therefore show the different effects of resource heterogeneity and interference on the performance.

VC configurations In this experimental setup, the total number of different VC configurations even for a single parallel application without considering the placement of co-runners (i.e., VMs or applications running on the same node) is 80. When placing a set of four parallel applications in the default setting, the total number of possible placements exceeds one million. Therefore, searching for the optimal placement of a parallel application in a heterogeneous cluster against all possible placements is impossible. To make the search process tractable, we need to limit the candidate VC configurations of a parallel application.

In this setup, for each parallel application, we consider two types of VC configurations with six configurations in total, as shown in Table 2.3. In the table, a notation for a VC configuration, which specifies the number of VMs in a node used in the configuration for each node type, is also given. For example, T2(4,4) describes a VC configuration in which four VMs are placed in each of two T2 nodes.

We initially consider *homogeneous VC configurations* in which the application only uses one type of nodes for its VMs, with the number of VMs in each node used for the application equal. For each type of nodes, two homogeneous VC configurations, the *most scaled out* type, where the maximum number of nodes for the same type is used, and the *most consolidated* type, where the minimum number of nodes is used, are utilized. These configurations provide hardware symmetry for the application. An application which requires heavy communication among its VMs may prefer the most consolidated type, while an application that can undergo resource contention among its VMs may prefer the most scaled out type, with a chance to run with other

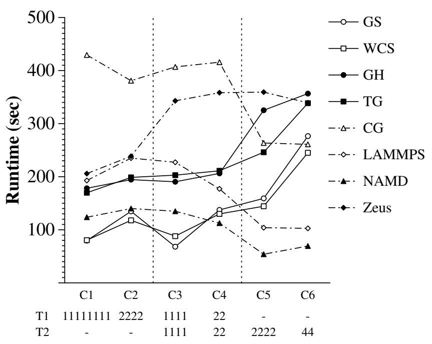


Figure 1: Runtimes over different VC configurations

applications with different resource usage characteristics.

Second, we consider *symmetric heterogeneous VC configurations*, in which an application uses two different types of nodes, but the total numbers of VMs in each node type are equal to each other. For a pair of node types, we also have two configurations, the most scaled out and the consolidated types, where the numbers of the VMs in each node are equal. By including these heterogeneous VC configurations which have hardware asymmetry as candidates, we can allow a parallel application to be assigned favored resources partially, with the VMs of the application distributed across even different types of nodes. This may improve the performance of the system.

II Performance Analysis

2.1 Effects of Different VC Configurations

Recall that each VC configuration given in Table 2.3 varies depending on the amount of resources used for each node type (i.e., the number of VMs used for each type), and the deployment of the VMs over the cluster (i.e., the number of physical nodes used for the VMs per node type). Figure 1 shows the runtimes of parallel applications without any co-runners (i.e., *solo runs*) over the six VC configurations. For the configurations, we can make three pairs, (C1, C2), (C3, C4), and (C5, C6). The two configurations in each pair are configured with the same amount of resources, i.e., the same number of VMs, for each node type. From these results, we can analyze the following.

Analysis 1: Each application has a different preferred node type. As shown in Figure 1, each application prefers to be executed in either the T1 or T2 nodes depending on its resource requirements. For the two different VC configurations of C2 and C5, which are identical except that C2 uses four T1 nodes, whereas C5 uses four T2 nodes, the resource usage patterns of each parallel application are also analyzed in Table 2.4. For the results in the table, we execute each application without any co-runners and compute the average resource usage over the VMs running the application during the execution.

In these results, storage-intensive applications have better performance in C2 compared to

Table 2.4: Runtimes and resource usages of parallel applications in C2 and C5

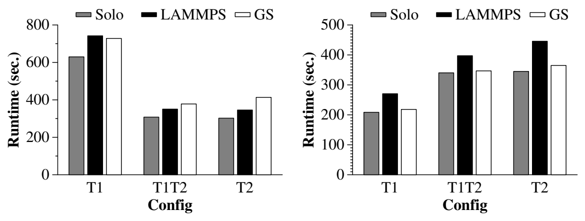
Storage App	C2					C5				
	CPU Util (%)			I/O	Time	CPU Util (%)			I/O	Time
	user	sys	wait	(MB/s)	(sec)	user	sys	wait	(MB/s)	(sec)
GS	17.20	1.06	30.99	14.60	134.82	22.59	1.81	30.37	11.49	159.14
WC	30.81	1.42	22.76	11.89	117.76	44.41	2.48	18.13	9.70	144.34
GH	37.20	2.65	7.63	6.20	194.47	41.26	3.16	6.83	3.75	325.23
TG	38.26	5.82	8.45	19.95	198.76	51.98	7.71	15.53	16.10	246.09

Network App	C2					C5				
	CPU Util (%)			I/O	Time	CPU Util (%)			I/O	Time
	user	sys	wait	(MB/s)	(sec)	user	sys	wait	(MB/s)	(sec)
CG	71.96	27.16	0.17	16.74	380.85	72.32	26.33	0.23	24.04	263.52
LAMMPS	64.41	34.32	0.38	11.05	235.08	65.90	31.20	0.91	25.71	103.97
NAMD	45.44	52.58	0.73	8.49	140.05	56.16	38.40	1.81	22.18	53.97
Zeus	39.21	23.95	1.77	9.31	238.57	41.04	22.30	0.63	6.08	359.37

C5 because their storage I/O throughputs are higher in C2 (i.e., T1 nodes). On the other hand, network-intensive applications have better performance in C5, except for 132.Zeusmp2, as they can exploit higher network I/O throughput in C5 (i.e., T2 nodes). The performance of 132.Zeusmp2 is mainly affected by the CPU performance; therefore, it has a shorter runtime in C2. Regarding CPU utilization, network-intensive applications are also compute-intensive, almost fully utilizing the CPU resources, while storage-intensive applications utilize fewer CPU resources. Thus, in our heterogeneous cluster, big data analytics applications prefer the T1 type, while MPI-based applications (except 132.Zeusmp2) prefer the T2 type.

Analysis 2: The performance of an application is not always improved proportionally to the amount of its preferred resource. We can *naively* expect that the runtime of an application is proportionally reduced as the number of VMs with the favored type increases. However, there are applications that perform quite differently from naive estimation. When we compare runtimes in the three configurations of C2, C4 and C5, which use a total of four (physical) nodes but the number of nodes per type is different, for 132.Zeusmp2 and CG, which require tight synchronization among VMs, the performance is not improved at all unless all of the VMs are executed on the preferred nodes. On the other hand, for all of the big data analytics applications, which are based on a simple communication pattern with a built-in load-balancing feature, until half of the VMs are executed on the favored T1 nodes, there is almost no performance degradation. The effect of heterogeneity on performance of parallel applications is quite dissimilar depending on the synchronization and parallelism patterns.

Analysis 3: Even with the same amount of resources for each type, the effect of how VMs are spread out or consolidated on the performance depends on the application's characteristic and used resource type(s). When the VMs of an application are spread out, they do not contend for the same storage or network resource, whereas when the VMs are consolidated, VMs running on



(a) TeraGen (b) 132.Zeusmp2

Figure 2: Runtimes of TeraGen and 132.Zeusmp2 with co-runners

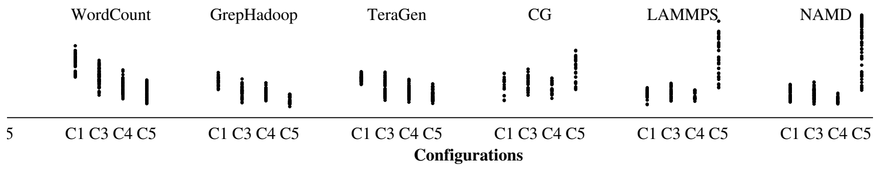


Figure 3: Performance of parallel applications with co-runners on various VC configurations

the same physical node can have fast inter-VM communication. Therefore, for storage-intensive applications, where communication among the VMs is not important, they generally prefer the scaled out configuration with less contention over storage I/O. However, for tightly coupled parallel applications, the effects of contention on the network I/O and inter-VM communication jointly affect the performance outcomes. Thus, the performance trend with a different deployment pattern is not always the same with a different resource composition. For 132.Zeusmp2, its performance on the most scaled out configuration (i.e., C1) is 15.91% higher than that on the most consolidated configuration (i.e., C2) when the T1 node type is used, while the performance on the most consolidated configuration (i.e., C6) is 5.95% higher than that on the most scaled out configuration (i.e., C5) when the T2 node type is used. This arises because it utilizes higher network bandwidth in C1 and C6 compared to C2 and C5, respectively. Similarly, LAMMPS and NAMD show different trends when the T1 node type is used (i.e., C1 and C2), and when both the T1 and T2 types are used (i.e., C3 and C4).

2.2 Effects of Interference

To investigate the effects of interference on the performance of a parallel application, we initially use a small virtual cluster composed of 4 VMs running on 4 physical nodes (i.e., in the most scaled out setting) so that we can have the same setting using each of the T1 and T2 node types in our cluster. Figure 2 shows the execution times of TeraGen and 132.Zeusmp2 over the three different VC configurations of T1(1,1,1,1), T2(1,1,1,1) and T1(1,1), T2(1,1). In the figure, LAMMPS and GrepSpark are used as co-runners.

In the results, *the interference effect caused by the same co-runner can vary from one VC configuration to another.* Unlike 132.Zeusmp2 which exhibits a similar performance trend when it co-runs with LAMMPS and GrepSpark over all three VC configurations, TeraGen's behavior with LAMMPS and GrepSpark differs. In T1(1,1,1,1), the effect of LAMMPS on the performance is similar to that of GrepSpark, but in the other configurations, the performance degradation by GrepSpark is 8.94∼22.38% higher than that by LAMMPS. Also, *the degree of the interference effect differs depending on the node type.* When TeraGen runs with GrepSpark, the performance degradation of TeraGen is 15.61% in T1(1,1,1,1), while it is 36.79% in T2(1,1,1,1) compared to solo runs on each configuration. This occurs because when TeraGen and GrepSpark, which are storage-intensive, run on T2 nodes together, the disk I/O becomes a severe performance bottleneck. Therefore, to estimate the final performance of a parallel application accurately, the resource type used and the state of the co-runners for each VM must be considered together.

We next study the effects of co-runners in our default cluster setup (as given in Table 2.1) using workloads composed of four application instances from Table 2.2. In order to understand the performance trend with co-runners, we measure the runtime of each of the parallel applications (or application instances) with various co-running applications over the four VC configurations of C1, C3, C4, and C5. (Note that for C2 and C6, each application is executed without any co-running application.) For all of the experiments running multiple applications concurrently, we repeatedly ran applications until the last one finished, as the runtime of each application is different.

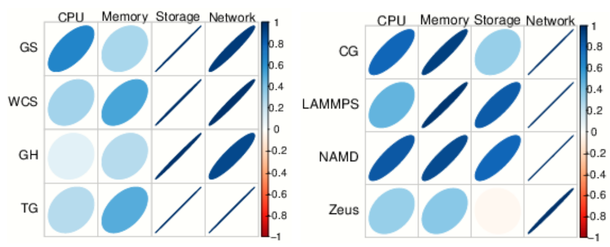
Figure 3 shows the speedup of each application over the four configurations in each experimental run. For each application, we have 148∼388 experimental runs with different co-running applications. In the figure, circles represent the runtime with co-runner(s) in each VC configuration. For the parallel applications, there is no single VC configuration in which every application has the best or worst performance. Therefore, for each of the results, the *speedup* of an application is computed as the runtime of the application in the run over the worst runtime among its solo runs in the six VC configurations.

As shown in the figure, the performances of parallel applications are significantly affected by co-running applications. For the big data analytics applications, they show similar trends over the four configurations, but their speedup values are quite different. For the MPI-based applications, they have quite different performance patterns over the configurations. For 132.Zeusmp2, the range of the speedup over various configurations is only from 0.68 to 1.65, but for NAMD, the range is from 0.67 to 5.04. Moreover, for LAMMPS and NAMD, co-runners can strongly affect their performance in C5, given the widely varying runtimes, unlike the other configurations of C1, C3, and C4 where the co-runner effects are relatively small.

To summarize the above results, in a heterogeneous cluster, the performance of a parallel application is determined by various factors, such as the heterogeneity, interference, and VM deployment across the nodes, but how each factor affects the final performance remains unclear.

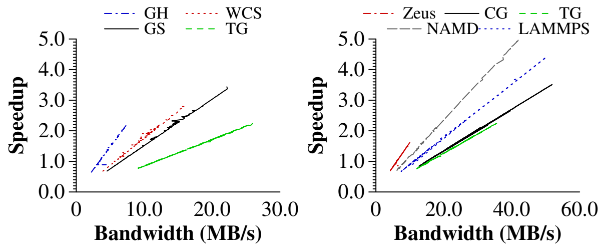
2.3 Dominant Resource

Most parallel applications demand multiple resources, such as CPU, memory, network I/O, and storage I/O, and the demand for each type of resource varies depending on the characteristics



(a) Storage I/O intensive (b) Network I/O intensive

Figure 4: Correlation between the performance and resource usage patterns



(a) Storage I/O intensive (b) Network I/O intensive

Figure 5: Dominant resource usage vs. performance

of the application. For each application, we analyze the correlation between the performance and resource usage patterns of CPU utilization, the number of LLC misses per kilo-instruction, and the network and storage bandwidths. The correlation coefficients from the analysis are presented in Figure 4. For all of the applications used in the experiments except TeraGen, we observe that the performance is closely correlated with the usage of one resource type, either the network I/O or the storage I/O. For TeraGen, there is strong correlation between its speedup and both network I/O and storage I/O. We call a resource (type) which mainly determines the performance of an application the *dominant resource* of the application.

Figure 5 shows the correlation between the dominant resource usage and the application speedup for two different types of parallel applications, storage-intensive and network-intensive applications. In the result, the speedup of a parallel application increases almost linearly as its dominant resource usage increases. Note that in our experimental setting, cases in which the speedup is bounded even when the bandwidth continues to increase because other resources become a bottleneck do not occur.

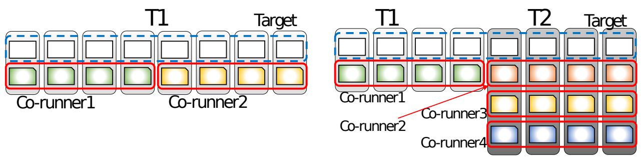
III Placement based on Machine Learning

Recall that it is not possible to search for the optimal placement of a parallel application in a heterogeneous cluster against all possible placements. Therefore, we need to limit the candidate VC configurations of a parallel application and the VC placements of multiple applications, thus reducing the complexity of performance profiling and VC placements. In this work, when placing a distributed application, we consider only homogeneous VC configurations and symmetric heterogeneous VC configurations, as discussed in Section I for a cluster with two different node types.

Consider a heterogeneous cluster composed of T types of physical nodes. To limit the number of the candidate VC configurations, for each application, we can select k types, where $k{=}2 \times \log_2 T$, out of T types. When selecting k types, we can consider the resource preferences of parallel applications but also select several types randomly. For each selected type, we have two homogeneous VC configurations (i.e., the most scaled out and consolidated ones).

For heterogeneous VC configurations, we select k types out of T types again, and we use a maximum of 2^s types out of k per VC configuration, where $s > 0$ and $2^s \leq k$. Therefore, a candidate VC configuration has 2, ..., or 2^i node types, where $0 < i \leq s$, and the numbers of VMs running on each node type are equal to each other. We assume that each VC configuration composed of R VMs, where $R \geq 2^s$ and $R \texttt{ mod } 2^s = 0$, is divided into 2^s *blocks*, and each of the VMs in the same block is executed on a (physical) node with the same type. For each i where $0 < i \leq s$, if $2^i < k$, we generate $\log_2 T$ tuples from the selected k types such that each tuple has 2^i node types, and for $\log_2 T$ tuples, each selected type appears 2^{i-1} times in total. If $2^i = k$, we generate one tuple composed of k types (as we cannot create more than one because of $\binom{k}{2^i}{=}1$). For each generated tuple, we have two heterogeneous VC configurations discussed above. In this way, the total number of the candidate VC configurations we consider is computed as $2 \times (k + \sum_{i=1}^{s} min(\binom{k}{2^i}, \log_2 T))$, which is scalable as the value of T increases. (Above, a rounded value of $\log_2 T$ is used.)

For example, when we use a maximum of two node types per VC configuration (i.e., $s{=}1$), for homogeneous VC configurations, we have k different types; thus, we have $2 \times k$ configurations. For heterogeneous VC configurations with two node types, we generate $\log_2 T$ pairs from k types, while for each pair, we have two configurations. Thus, the total number of candidate VC configurations is $6 \times \log_2 T$.



(a) C1 configuration (b) C3 configuration

Figure 6: Examples of candidate VC configurations

Moreover, we restrict the placement of VC configurations for multiple applications such that for a VC configuration composed of 2^s blocks, each of the VMs in the same block has the same set of co-runner(s). Figure 6 shows two examples of candidate VC configurations for a parallel application in our default cluster setup with $s{=}1$.

Figure 7 shows an overview of our placement technique. In a heterogeneous cluster, a set of the candidate VC configurations with homogeneous and heterogeneous resources is computed for a parallel application. A model based on a machine learning algorithm which can consider various relevant factors conjointly is built to estimate the runtime of a target application on a certain VC

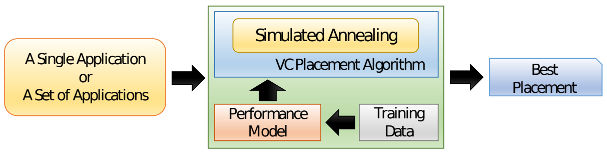
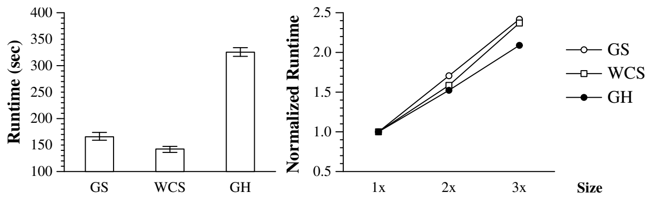


Figure 7: System overview

configuration. For MPI-based applications with a diversity of synchronization patterns, a target application that will be executed in the cluster is profiled against a synthetic workload that mainly consumes the dominant resources of the application to generate training samples. For big data analytics applications based on the same programming model, it is possible to generate training samples by exhaustively profiling a small set of big data analytics applications. The performance metrics measured during profiling are used as inputs to the model.

Our placement algorithm can place a single parallel application as the application is submitted to the cluster (i.e., in the on-line mode), and it can make placement decisions for multiple applications simultaneously (i.e., in the batch mode). Because it is still infeasible to search for all possible placements in most heterogeneous clusters, even when we restrict the candidate VC placements for parallel applications, we present a VC placement algorithm based on simulated annealing which approximates the global optimal solution [19]. The algorithm estimates the performance of a hypothetical placement of applications using the performance model, and explores the large search space for the best placement. Similar placement approaches based on simulated annealing and stochastic hill climbing have been used for parallel applications in homogeneous clusters [11] and for web-service workloads [5].

3.1 Performance Model

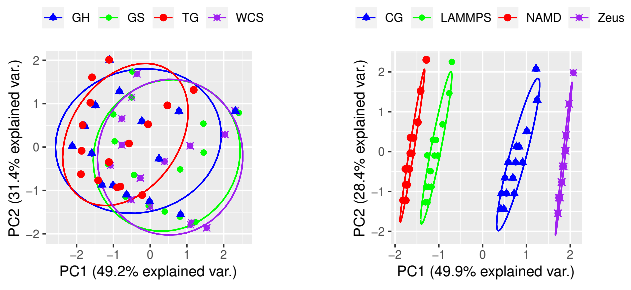


(a) Runs of the same size (b) Runs with different sizes

Figure 8: Performances of GrepSpark, WordCount, and GrepHadoop

Big data analytics applications are based on the same programming model of “map”, “shuffle”, and “reduce” to process a large amount of data [20]. In addition, the size of the input data has a significant impact on the runtimes of the applications. Figure 8(a) shows the average runtimes of GS, WCS, and GH for five different input data of the same size, on the C5 configuration. The standard deviation of the runtimes is also presented in the figure. Figure 8(b) shows their runtimes over various input sizes on the C5 configuration, where the sizes of “2x” and “3x” are two and three times the size of “1x”. In the figure, the runtimes of the applications tend to increase in proportion to the input data size. However, if the sizes of the input data are identical, the runtimes of the application are similar.

However, MPI-based applications have various communication and synchronization patterns [21,22]. Moreover, their performances are affected by various parameters which are usually specific to each application. (For example, in molecular dynamics simulators, the performance is affected by the number of atoms, the molecular topology, the cut-off distance between the atoms, etc. [10].)



(a) Big data analytics (b) MPI-based

Figure 9: Principal component analysis

Figures 9 (a) and (b) present the results of a principal component analysis (PCA) with 95% confidence ellipses, which present the regions including 95% of samples, for the big data analytics and MPI-based applications, respectively. The analysis in each case is performed on the measured runtimes of the parallel applications with various co-runner states in the C5 configuration to understand the performance trend of the applications under various interference settings. For each type of applications, an identical set of co-running synthetic workloads (which mainly utilize the dominant resource of the corresponding type as discussed in Section 3.1) is used to generate interference. In the figures, the similarity of the big data analytics applications and the dissimilarity of MPI-based applications are apparent, as the regions of big data analytics applications overlap but those of MPI-based applications are separated. Therefore, for big data analytics applications, we build one *general* performance model based on the off-line profiling of a few big data analytics applications. For a target big data analytics application that will run in the cluster, our modeling technique requires a limited number of profiling run with a certain size of input data, and the built model can estimate the runtime of the target application for different input data of any size.

For MPI-based applications, we build a model for each of the applications. In this work, we do not associate the model with different inputs (for example, different proteins and cut-off distances for molecular dynamics simulators). However, our model can be combined with other modeling techniques [10], which consider different inputs when modeling MPI-based applications such as NAMD and LAMMPS in a homogeneous cluster, in order to eliminate the need for profiling with each different input.

Generating Training Samples

We generate training samples by the off-line profiling of a parallel application over various setups. Based on the observation that the performance of a parallel application tends to be directly correlated with the usage of its dominant resource in Section 2.3, we reduce the profiling runs of the application significantly through the use of a synthetic workload that mainly consumes its dominant resource. During the profiling step, synthetic workload mixes which utilize all types of resources are not used.

We implemented two types of synthetic workload generators with different intensity levels. For a synthetic workload using the network resource, for a pair of VMs, each VM sends and receives some number of messages to/from the other VM every second. For a synthetic workload using the storage resource, each VM reads and writes some amount of data from/to its local file system every second. For each of the network and storage synthetic workloads, as the intensity of each workload increases, the network and storage bandwidths of a VM running the workload also increases. When profiling an application in each VC configuration, interference is generated in a block as a unit by running synthetic workloads with the same intensity in all of the remaining VMs on each node of the block, and each block can have a different interference level.

While profiling a parallel application, we measure various performance metrics for each VM of the application, and then compute the average or some aggregated value of each metric over the VMs. For the performance model, the metrics of CPU idle % and user %, send and receive for network I/O (in MB/s), and read and write for storage I/O (in MB/s) are selected as inputs, as they are highly correlated with the performance of the application. Note that the profiling process below needs to be done only once, unless the physical hardware configurations of the cluster are changed.

Profiling of big data analytics applications We perform exhaustive off-line profiling for a small set of big data analytics applications such that various behaviors of big data analytics applications are contained in a set of training data. For an application in the set, we run it without any co-runners and also with synthetic workloads as co-runners over all of the VC configurations. We also profile the runtimes of the application with other parallel applications in the set. To make the model estimate the runtime of an application with different input sizes, for each of the parallel applications in the set, we also profile it with several input files of different sizes. Moreover, we generate training samples that show the correlation between the runtimes and input sizes from profiled data with different input sizes.

In addition, for target Hadoop and Spark applications that will run in the cluster, we need to profile each of the applications for certain input data (whose size is reasonably small) without any co-runners and with synthetic workloads of its dominant resource in all VC configurations. Exhaustive profiling for the target applications is not necessary.

Profiling of MPI-based applications To generate training samples for each MPI-based application, we profile its runtimes without any co-runners, and only with synthetic workloads in each candidate VC configuration. To build an individual model for each application, additional profiling runs with real applications are not necessary.

Building Machine-learning based Models

In a performance model for a target application A_{target}, we basically provide the following as inputs.

- A target VC configuration VC_{target}.
- The solo run runtime of A_{target} and the average performance metrics over all of the VMs of A_{target} on VC_{target} measured during the solo run.
- In each of blocks on VC_{target}, for each co-running application C_i on the block, the average performance metrics over the VMs of C_i running on the block measured during the solo run.

For the models of MPI-based applications, their solo run runtimes (with the target input data) in all of the candidate VC configurations are profiled off-line. Therefore, the measured information is used as the input. On the other hand, for the general model of the Hadoop and Spark applications, the model uses previously profiled information pertaining to the runtime and metrics of the solo run of a target application for certain input data as inputs in order to predict the runtime with target input data whose size can differ from the profiled size. In consequence, this model additionally takes the sizes of the current target input and (estimated) output data along with those of the previously profiled run for the same application as inputs.

We attempted various machine-learning techniques using *Weka* [23], and concluded that REPTree (Reduced Error Pruning Tree) shows the best performance for our performance estimation. Machine-learning techniques such as ANN [24], SVM [25], and Gaussian Process Regression [24] do not work well if a given training data set cannot be fitted in any kernel functions. For our modeling problem, the function was not built properly from the inputs and outputs in our training set, resulting in huge error rates. Both RandomTree and REPTree are decision tree learning based algorithms [23]. In a decision tree, every non-leaf node splits the data space into subspaces based on an input attribute and a threshold, and every leaf node is assigned a target value [26]. In our decision tree, attributes which represent A_{target}, VC_{target} and applications co-running on VC_{target} discussed above are used. As the data spaces are partitioned based on important attributes that determine the performance, a decision tree is suitable for our problem. By following the root to a leaf node in the tree with the given input values, the estimated runtime of A_{target} is presented at the leaf node.

When constructing a tree, REPTree considers all of the input attributes to make a branching decision, and it prunes the tree using reduced-error pruning. On the other hand, RandomTree considers a set of K randomly selected attributes at each node to build a tree, and performs no pruning. For our runtime estimation, we selected attributes that can reflect the performance of various parallel applications carefully and provided them to the models as inputs. Hence, by considering all of the attributes, REPTree works better than RandomTree in our study.

With modeling based on REPTree, we use a regression tree mode which predicts the output in the form of a real number to estimate the runtime of A_{target} on VC_{target}. To improve the model accuracy and avoid over-fitting, we used bagging, also called bootstrap aggregating [27]. Similarly, bagging has been used to lower the error rates of performance models for HPC workloads on a multi-core system [28]. For bagging, we generate 100 training sets by sampling, from all of the training samples we have, uniformly and with replacement. We then build 100 models using the 100 training sets. Finally, we compute the average of the estimated runtimes from the models as the final estimation.

The accuracy of a model based on REPTree is affected by the quality of a training data set, as the model is built by recursively partitioning a training data set to subsets based on input attributes. To have an accurate model, training samples in the set need to have similarity to instances of real workloads running on the cluster. If the values of input attributes for an application (i.e., performance metrics) are very different from those in the training set, the model is unlikely to estimate the runtime accurately. In such a case, the model can be re-trained by adding this application's samples to the training set, as discussed in earlier work [28].

3.2 Placement based on Simulated Annealing

Our simulated annealing based placement algorithm can make placement decisions in both the batch and on-line modes. To place VMs for a given set of new applications (or a single application), we initially hypothetically distribute the VMs of the applications randomly over available nodes in a heterogeneous cluster, and estimate the performance of the applications with this random placement. To find the best placement, we change the hypothetic placement by randomly selecting one application from the new applications, placing it in a different VC configuration, and then adjusting the other new applications affected by this new placement. We then compute the performance again for a new placement state. If the estimated performance in the new state is better than that in the previous state, we can conclude that the new placement is better. In such a case, we always move to the new state. If the move results in a worse state, we probabilistically move to the new state. The above process is repeated for a predefined number of iterations. In the algorithm, we basically consider candidate VC configurations for applications and the placement of applications that satisfies our restriction on co-runners as discussed above. Note that if none of the candidate VC configurations of a parallel application is available in a cluster, then a VC configuration which is closest to one of the candidate VC configurations can be used.

In this work, when placing VMs in a heterogeneous cluster, we aim to maximize the overall speedup for parallel applications running in the cluster. To estimate the overall speedup in a certain placement, the algorithm initially estimates the runtime of each application based on the performance model, and subsequently computes the speedup over the worst runtime among its solo runs in the candidate VC configurations. It then uses the geometric mean of the speedups of all the applications for the performance, as the range of speedups under different placements differ for each application.

IV Results

4.1 Performance Estimation Accuracy

We evaluate our performance models using a heterogeneous cluster composed of 12 nodes with two node types, as described in Section I. For each Hadoop and Spark application in Table 2.2, we use two additional sizes, i.e., sizes which are two and three times larger (i.e., 2x and 3x) than the specified size (i.e. 1x) in the table to evaluate the model.

For synthetic workloads used on profiling, there are five levels of intensity for each type of synthetic workloads. For each VC configuration composed of two blocks, we run a target application with a synthetic workload on only one of the blocks and also both of the blocks (if possible). This is done for each intensity level. For example, in C3, we have five profiling runs only by running synthetic workloads on the T1 block, five runs only by running them on the T2 block, and five runs by running them on both of the blocks. In a virtualized system, dom0 is required to handle I/O requests from VMs. Thus, if a VM in a node does not use the CPU intensively, the performance of dom0, which handles I/O requests of other VMs running in the same node, can be noticeably improved. With regard to network-intensive applications used in our experiments, most of them fully utilize the CPU, unlike the storage intensive applications; thus, if co-running VMs do not fully use the CPU, their runtimes can be reduced. To model these cases, we make a network synthetic workload which can use the CPU at the two levels of ∼50% and ∼100% of a VM. For storage synthetic workloads, only one CPU level of ∼50% is used.

For each of the Hadoop and Spark applications in Table 2.2, we profile it with small size input (i.e., half the size of 1x) for six solo runs, and 40 co-runs with synthetic workloads, except for TeraGen. For a collection of applications for exhaustive profiling, we use the five additional applications of WordCountHadoop, JoinHadoop, ScanHadoop, SortHadoop and TeraSortHadoop. We also perform exhaustive profiling of the target applications. However, to demonstrate that the model can estimate the runtime of the Hadoop or Spark application $A_{BigData}$ for any input size without heavy profiling, we remove all training samples that contain any data of $A_{BigData}$ from the set of training samples obtained by exhaustive profiling. Subsequently, we build the model using a subset of training samples which does not include any samples of $A_{BigData}$ along with previously profiled samples of $A_{BigData}$ with a small input size. Finally, we predict the runtime of $A_{BigData}$ with the given target input data, similar to the evaluation method in earlier work [28]. Note that for TeraGen whose performance is correlated with both the network I/O and storage I/O, we built three models with storage synthetic workloads, network synthetic

Table 2.5: Validation results of applications

App	Error(%)	App	Error(%)
Zeus	12.71	GS	21.36
CG	21.42	WCS	16.22
LAMMPS	18.24	TG	21.72
NAMD	20.73	GH	34.27
Average	21.84	Average	23.39

workloads, and both types of synthetic workloads. There were no major differences among the error rates of the three models. We used the model with network synthetic workloads (of 80 profiling runs) in our study, as it has the lowest error rate.

To profile each MPI-based application, we have six solo runs, and 80 co-runs with network synthetic workloads. Note that many MPI-based scientific applications, including the MPI-based applications used in this experiment, iterate a given number of time steps for the execution; in many cases, the runtime of each time step (i.e. each iteration) is uniform, as demonstrated in the literature [10, 29]. Thus, for such an MPI-based application, we can have each profiling run only with a small number of time steps instead of running all of the iterations specified for the application and predict the actual runtime proportionally to the number of time steps. By exploring this iteration-based pattern, the profiling overhead can be reduced significantly.

Table 2.5 shows the average error rates of the REPTree models with bagging for the parallel applications, where 100∼180 and 166∼406 test cases were used for MPI-based and big data analytics applications, respectively. The error rate in each test case is computed as $\mid r_{est} - r_{act} \mid / r_{act} \times 100$, where r_{est} and r_{act} are the estimated and actual runtimes, respectively. Each of the built models is validated using the experimental results of the application running together with other parallel applications in our cluster. To evaluate the accuracy of the big data analytics model, which can estimate the runtime of an application with different input sizes, we used a set of testing data which includes the experimental runs using three different input sizes for each application.

In the REPTree models, for nodes in higher levels of a tree, attributes representing a target VC configuration are used the most (especially for root nodes), followed by attributes related to the dominant resource usage of a target application, as they are critical features to determine the runtime of a target application.

When ANN, SVM, and Gaussian Process Regression are used, the average error rate is very high, especially for MPI-based applications. For these algorithms, the average error rates for the MPI-based and big data analytics applications exceed 100% and 60%, respectively. These rates are not acceptable for reliable performance models. With RandomTree and REPTree (without bagging), the respective average error rates are 45.76% and 26.22% for big data analytics applications, while the corresponding error rates are 24.68% and 22.53% for MPI-based applications. REPTree with bagging provides greater accuracy than the other algorithms.

Table 2.6: Workloads used in our experiments

	App1	App2	App3	App4	S:N:B
WL1	WordCount	WordCount	TeraGen	Zeus	2:1:1
WL2	GrepSpark	WordCount	CG	NAMD	2:2:0
WL3	WordCount	TeraGen	LAMMPS	Zeus	1:2:1
WL4	GrepSpark	TeraGen	NAMD	Zeus	1:2:1
WL5	WordCount	GrepHadoop	TeraGen	CG	2:1:1
WL6	GrepSpark	GrepSpark	NAMD	LAMMPS	2:2:0
WL7	WordCount	GrepHadoop	NAMD	LAMMPS	2:2:0
WL8	GrepSpark	GrepSpark	WordCount	GrepHadoop	4:0:0
WL9	CG	LAMMPS	NAMD	Zeus	0:4:0

Note that we have attempted to have one model of the MPI-based applications based on exhaustive off-line profiling of a few MPI-based applications, similar to the general model of the big data analytics applications. For target MPI-based applications, their profiling runs only for solo runs are used in a training data set. However, the error rates are quite high as up to 45%, given that MPI-based applications show quite different characteristics, as analyzed in Section 3.1.

4.2 Experimental Results on a Cluster with Two Types

Methodology

We experimentally evaluate the performance of our ML-based placement technique using a real cluster with two node types as described in Section I. When placing parallel applications on the cluster, our simulated annealing based placement algorithm uses the performance model built in Section 4.1 to estimate the runtimes of applications for a hypothetical placement. We compare our technique (`ML-G`) with the following five heuristics:

- Random placement (`Random`): This heuristic randomly places parallel applications among the candidate VC configurations.

- Greedy placement (`Greedy`): This algorithm places a parallel application in one of the most consolidated homogeneous VC configurations, where the application has the smallest solo run runtime, if it is available. This is inspired by the greedy algorithm in Quasar [15], where for an application, the scheduler initially sorts server types according to their performance and then selects servers which is capable of higher performance in the sorted order while attempting to pack the application within a few servers.

- Heterogeneity-aware interference-ignorant placement (`HA`): This algorithm considers the effect of different VC configurations with heterogeneous resources on the performance of a parallel application. It places a parallel application in one of its candidate VC configurations based on their solo run runtimes in a greedy manner. It assigns the application an available VC configuration with the smallest runtime, without reflecting possible interference effects caused by co-runners.

- Interference-aware heterogeneity-ignorant placement (`IA`): If two parallel applications which have the same type of dominant resource are placed together on a set of the same nodes, their performance will be degraded, as they compete for the same type of resources in the nodes. To reduce the interference effect, this algorithm creates groups of two for given applications such that two applications with different dominant resource types are paired, if possible, and then places each pair randomly (on C1, C3 or C5 in our setup), being oblivious to the heterogeneity of resources. Note that this approach can be used only in the batch mode.

- ML-based placement with individual models (`ML-I`): This algorithm places a parallel application in the same manner as `ML-G` except that the individual modeling approach is also used for Hadoop and Spark applications. For target Hadoop and Spark applications, the profiling overhead for `ML-I` can exceed that for `ML-G`, which can use small input size for profiling.

For the experiments in this subsection, we provide a set of four applications as input to the placement algorithms (i.e., batch mode). In `Greedy` and `HA`, we need to order multiple applications in the set. For each application, we compute the maximum speedup as r_{worst}/r_{best}, where r_{worst} and r_{best} are the runtimes of its worst (i.e., longest) and best (i.e., shortest) solo runs between the C2 and C6 configurations for `Greedy` and among the six candidate VC configurations for `HA`. Then, we sort parallel applications in a workload in a decreasing order of their computed maximum speedups. For each application in the sorted order, `Greedy` and `HA` make a placement decision. For `Random` and `IA`, the average performance of five random placements is shown in the results.

Table 2.6 presents the nine workloads used in our experiments. The ratio of the number of storage-intensive applications (denoted as "S"), that of network-intensive applications (denoted as "N"), and that of applications utilizing both network and storage resources (denoted as "B") in each workload is also given in the table. For the big data analytics applications, the input sizes (i.e., 1x) described in Table 2.2 are used in these workloads. The workloads are selected such that there are various ratios of the numbers of storage-intensive, network-intensive, and storage-network-intensive applications, including two homogeneous workloads that are composed of only one type of applications. In our experimental setup, the total number of candidate placements for four applications is 74. For each workload, we explore all of the candidate placements to find the best placement of the four applications (`Best`), which has the maximum geometric mean of the speedups.

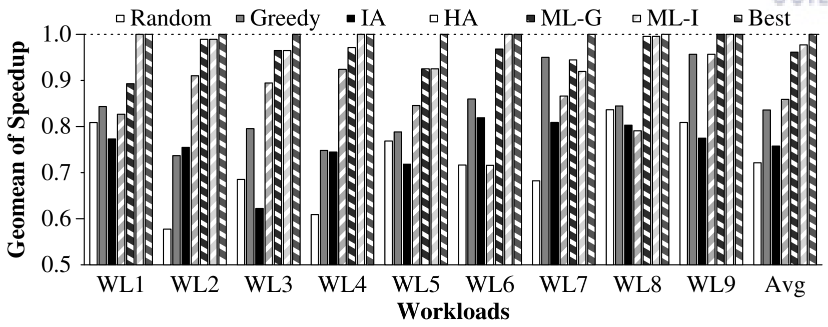


Figure 10: Speedups of workloads, normalized to the best placement

Table 2.7: Best configuration for each workload

	App1	App2	App3	App4
WL1	WCS: C1	WCS: C3	TG: C3	Zeus: C5
WL2	GS: C3	WCS: C2	CG: C3	NAMD: C5
WL3	WCS: C3	TG: C3	LAMMPS: C5	Zeus: C2
WL4	GS: C3	TG: C3	NAMD: C5	Zeus: C1
WL5	WCS: C1	GS: C5	TG: C1	CG: C5
WL6	GS: C3	GS: C3	NAMD: C5	LAMMPS: C1
WL7	WCS: C1	GH: C1	NAMD: C6	LAMMPS: C6
WL8	GS: C1	GS: C3	WCS: C3	GH: C5
WL9	CG: C5	LAMMPS: C5	NAMD: C2	Zeus: C2

Experimental Results

Figure 10 shows the speedup of each of the nine workloads (on the geometric mean), normalized to that of the best placement. Our ML-based placement technique `ML-G` achieves 96.13% of the performance of `Best` on average. The performance improvements of `ML-G` compared to `Random`, `Greedy`, `HA`, and `IA` are 24.80%, 12.95%, 10.61%, and 21.11% on average, respectively. The performance of `ML-G` is fairly comparable to that of `ML-I`, which achieves 97.72% of `Best` on average. With regard to homogeneous workloads, they are less sensitive to the placement configuration, as they consist of similar applications in terms of preferred resources and interference effects. Thus, the performance difference between good and poor configurations becomes smaller. Our performance models have modest error rates, but the models can compute the relative performance gap between different placements and distinguish between good and poor placements reasonably well.

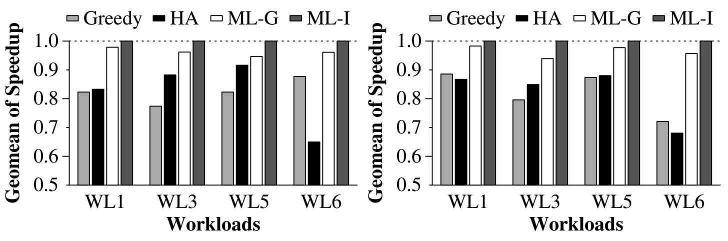
For various algorithms, we analyze the complexity of profiling overhead for a target application with the corresponding target input in a cluster which has T node types. With `Greedy`, `HA`, `IA`, and `ML-I`, the profiling complexity is $O(\log T)$, while with `ML-G`, no additional profiling is not required, because we use the previously profiled information for the application (possibly with a different input size). In our setup of a cluster with two types, for `Greedy`, `HA`, and `IA`, two, six, and three solo runs are profiled, respectively. For `ML-I`, 46 and 86 runs are profiled for storage and network intensive applications.

We subsequently analyze the best placements of the workloads in our heterogeneous cluster.

Table 2.7 indicates that there is no single best placement that works for all workloads, despite the fact that there is a tendency of network-intensive applications to favor T2 nodes that provide a higher network I/O bandwidth, while storage-intensive applications prefer to be placed on T1 nodes where they have only one co-runner VM and may use more of the storage I/O resource. Five different sets of VC configurations are used for the best placements. Even when the same set of VC configurations is used to place four applications, depending on the combination of applications, the same application can be assigned a different VC configuration.

Recall that we include two heterogeneous VC configurations as the candidate VC configurations for a parallel application. All of the best placements except for WL5, WL7 and WL9 use the heterogeneous VC configuration of C3, and all of the applications placed in C3, except for CG in WL2, are storage-intensive big data applications. For big data analytics applications, the performance can be improved by allocating the favored nodes partially, and the performance can be enhanced if their VMs are executed with other applications over different nodes, with less contention over the same types of resources. This result shows that it is beneficial to exploit heterogeneous VC configurations with hardware asymmetry.

4.3 Large Scale Simulations



(a) Batch mode (b) On-line mode

Figure 11: Speedups in a large scale cluster

To evaluate our placement technique in a large-scale heterogeneous cluster, we simulated the placement algorithms using runtime traces collected from real experiments in our 12 node clusters. In our simulations, a heterogeneous cluster consists of 80 T1 nodes and 40 T2 nodes, 120 nodes in total, with the same configuration of VMs and candidate VC configurations used as in Sections 4.1 and 4.2.

The simulator, which was implemented in C++, computes the placement of 40 parallel applications. In our ML-based algorithm, for a hypothetical placement, it estimates the speedup for a parallel application based on the performance model built in Section 4.1. For a final placement determined by the algorithm, the simulator computes the geometric mean of the speedups of 40 applications using their actual runtimes as measured in our 12 node cluster with the same VC and co-runner configurations. Note that due to the limited cluster setup used here, in some cases we were not able to generate precisely the same placement of a parallel application with co-runners computed by the algorithm. In such cases, we used the runtime of an application measured in the configuration closest to the computed case. We also implemented the simulator of HA in a similar manner, while for Greedy, we were able to evaluate the performance based on measured solo run runtimes.

Figure 11 shows the speedups of the four workloads of WL1, WL3, WL5 and WL6 (in Table 2.6) on a large cluster in the batch and on-line scheduling modes. In a workload, ten instances of each application are submitted. In the figures, the speedups are normalized to those of ML-I, as searching for the best placement is infeasible with a large cluster. In the on-line mode, we assume that 40 applications are submitted at the same time, but there is an order among them. Each application is placed in sequence in the submission order. For a workload, five submission orders are randomly selected. The average value of the five runs is given in Figure 11 (b).

In the batch mode, compared to Greedy and HA, our ML-G improves the performance by 14.32% and 14.70% on average, respectively. In the on-line mode, the performance improvements with ML-G are 15.09% and 15.05% on average, compared to Greedy and HA, respectively. With individual models in ML-I, the performances improve correspondingly by 3.79% and 3.60% in the batch and on-line modes, compared to ML-G.

Because the on-line algorithm only considers the placement of one application at a time, and it cannot change the placements of existing applications in the cluster, the numbers of possible VC configurations and placements for the application are reduced during on-line scheduling. A submission order of applications in a workload affects the performance of each placement technique. However, in our simulation runs, in no case do Greedy and HA show better performance than ML-G. The minimum performance improvements of ML-G, compared to Greedy and HA, over all of the runs of the four workloads, are 4.54% and 1.34%, respectively.

4.4 Experimental Results on a Cluster with Four Types

We investigate the performance of our placement technique in a heterogeneous cluster composed of 20 nodes with four different node types. In addition to T1 and T2, each node in T3 is configured with two Intel octa-core E5-2640 v3 (Haswell) 2.60GHz processors, 20MB L3 per socket 32 GB memory, and Gigabit Ethernet, and there are four T3 nodes in the cluster. Each of T3 nodes has two sockets, but we only use one socket with 8 cores. To have another node type T4, we change the CPU frequency of four nodes that have the same specification as T3 nodes to 1.60GHz. The network throughput of T3 and T4 is similar to that of T2.

In the experiment, we use WL1 shown in Table 2.6. We have two instances of each application in WL1 and the workload is composed of a total of eight application instances. For the on-line mode experiments, five submission orders of eight applications are randomly selected, and the average value of the five runs is presented below. Recall that in a cluster with T node types, we select $k = 2 \times \log_2 T$ types out of T, and we use up to 2^s types per heterogeneous VC configuration, where $s > 0$ and $2^s \leq k$. In the cluster with four node types (i.e., T=4), we evaluate the performance of our placement technique for two cases where a candidate VC configuration uses a maximum of two node types (i.e., s=1) and four node types (i.e., s=2).

There is a trade-off between the potential performance enhancement using more node types per VC configuration and the overhead to profile applications and search for the best placement. As T and s increase, the numbers of candidate VC configurations for an application and profiling runs of the application increase. For each parallel application, the total numbers of the candidate VC configurations with s=1 and s=2 are 12 ($=6\times \log_2 T$) and 14 ($=6\times \log_2 T+2$), respectively, as discussed in Section III. (Note that due to the limited cluster setup used in the experiments, we have 13 candidate VC configuration with s=2.) When building models for the applications in WL1, the number of profiled runs of each application against synthetic workloads with s=2 is 2.5 times larger than that with s=1.

When up to two types are used (i.e., s=1) for a VC configuration, for the batch mode, the performance of `ML-I` is 11.66%, and 13.29% better than those of `Greedy` and `HA`, respectively. For the on-line mode, the performance of `ML-I` is 14.45%, and 13.93% better than those of `Greedy` and `HA`, respectively. For the batch and on-line modes, the respective performance improvements with using a maximum of four types (i.e., s=2) are 6.38% and 3.64%, compared to using a maximum of two types.

In our experimental results, when using more node types per VC configuration, much higher profiling overhead is required, but the performance gain tends to be relatively small with considering the overhead. For 132.Zeusmp2, it tends to have no performance gain by utilizing heterogeneous resources and it is less affected by interference as shown in Section II. Thus, 132.Zeusmp2 still prefers to be placed on nodes with the favored type, i.e., T1 nodes, regardless of which application co-runs on T1 nodes, and consequently, a heterogeneous VC configuration with four node types is not selected for 132.Zeusmp2. With regard to WordCount, the performance can be better on a VC configuration with all of T1, T2, T3 and T4 node types, compared to a VC configuration with T3 and T4 node types. Therefore, WordCount is placed on a VC configuration with all of the four node types to improve the performance. However, a VC configuration with T3 and T4 types is also used for WordCount, when one of the four node types is exhausted by other applications in the workload. Depending on the combination and submission order of applications in a workload, it is possible that heterogeneous VC configurations is not well utilized, having small performance improvement. Even when a cluster has a large number of heterogeneous node types, it may be effective enough to use a small number of node types to run a parallel application (instead of using all the node types per VC configuration) while keeping the profiling overhead of the application within reasonable bounds.

4.5 Discussion

Effects of profiling against mixed synthetic workloads In our 12 node cluster described in Section I, the storage and network intensive applications have 40 and 80 profiling runs, respectively, with synthetic workloads with five intensity levels, which mainly use the dominant resource. However, when a synthetic workload which utilizes the CPU, network and storage resources in a mixed manner is used, both types of the applications end up having 400 profiling runs. For WordCount and 132.Zeusmp2, we build the performance model based on mixed synthetic workloads. The accuracy of these models is decreased by less than 1.1%, compared to that of the models using only the dominant resource, demonstrating the effectiveness of using the dominant resource on the profiling process.

A different number of VMs for applications In the above results, all of the applications are configured with 8 VMs. However, applications have different resource requirements; thus, they use different numbers of VMs for their virtual clusters (i.e., different VC sizes). We evaluate our placement method for a workload composed of applications with 4, 8, 16 VMs using the 12-node cluster with two node types. With a given VC size, each VC configuration has four homogeneous and two heterogeneous candidate VC configurations as discussed in Section I. (Note that due to the limited size of the cluster used in our experiments, there are only three candidate VC configurations for 16 VMs.) To use 4 or 16 VMs for an application, additional profiling runs of the application are done using synthetic workloads, and these runs are added to a training data set to build a model.

Each application has two blocks regardless of a used VC size, but the assumption that each of the VMs in the same block has the same set of co-runner(s) is relaxed. This assumption is to restrict possible VC configurations, reducing the overhead of profiling and placement, but it needs to be relaxed if it restricts too many VC configurations, preventing a resource provider from utilizing the cluster resource effectively. In general, heterogeneous co-runners can exist for a block of a VC configuration, if there are two candidate VC configurations that use the same type of physical nodes, but the numbers of used nodes differ. For example, in the configuration of T1(1,1,1,1),T2(1,1,1,1) for 8 VMs, a block in a T1 node or a T2 node has heterogeneous co-runners if there are two co-runners using 4 VMs each with the T1(1,1),T2(1,1) configuration. When a block has heterogeneous co-runners, we provide the average performance metrics of the co-runners to the performance model as inputs.

In our experiments, we use the two workloads of WL3 and WL5 in Table 2.6, which consist of two applications with 4 VMs each, one application with 8 VMs and one application with 16 VMs, and use individual models for all applications in WL3 and WL5. For WL3 and WL5, our technique shows 91.48% and 100% of the performance with `Best`, respectively. Adding a different VC size to the model requires more profiling of an application, but when we use 50% of profiling runs which include ones with intensities of three and five, the accuracy of the model is degraded only by around 1% on average. With the model of 50% samples, the performance of WL5 becomes 96.19% of `Best`, while that of WL3 remains the same.

VM placement with VM live migration For the on-line scheduling mode in Section 4.3, we do not change the placement of existing applications to place a new application. It is possible to migrate VMs of an existing application to ensure better placement of applications. However, if the VM migration overhead is too high, the efficiency of a cluster will be decreased. In the on-line scheduling for the four workloads in Section 4.3, if we allow existing VMs to be migrated whenever a placement with VM migration increases the efficiency, the performance will be similar to that with the batch model, showing around 7% higher performance than the on-line mode of `ML-G` and `ML-I` on average, under the assumption that the VM migration overhead is none. To study the performance of `ML-G` over various VM migration overheads, we run simulations in the on-line mode with VM migration for WL3 (similar to Section 4.3). We modify our simulated annealing algorithm to consider migrating already placed VMs to improve the efficiency. Note that for a parallel application, VMs in the same block must be migrated together as a unit. For WL3, the possible maximum improvement by employing VM migrations is 10.15%. As the overhead to migrate an application to a new placement increases to 2% of the average runtime of all the applications used in Table 2.2, the performance of WL3 cannot match that in the on-line mode without VM migration. The VM migration overhead must be considered when re-arranging existing VMs in an effort to improve the performance.

V Concluding Remarks

In this work, we investigated a placement technique for distributed parallel applications in a heterogeneous cluster, aiming to maximize the overall performance for the benefits of service providers and users. Using the experiments on heterogeneous clusters and large scale simulations, we demonstrated the feasibility of a heterogeneity and interference aware placement approach for distributed parallel applications, which considers various factors to influence the performance of a distributed parallel application in a combined manner for maximizing the efficiency.

Chapter 3

Platform and Co-runner Affinities for Many-Task Applications in Distributed Computing Platforms

Recent emerging applications from a wide range of scientific domains (e.g., astronomy, physics, pharmaceuticals, chemistry, etc.) often require a very large number of loosely coupled tasks (from tens of thousands to billions of tasks) to be efficiently processed with potentially large variances of task execution times and resource usage patterns. This makes existing computing paradigms such as High-Throughput Computing (HTC) [30] or Volunteer Computing [31] expand into Many-Task Computing (MTC) [13], and brings many research issues in the design and implementation of middle-ware systems that can effectively support these challenging scientific applications.

To support loosely coupled applications composed of many number of tasks effectively, all the available resources from different types of computing platforms such as supercomputers, grids, and clouds need to be utilized. However, exploiting heterogeneous resources from distributed computing platforms for multiple loosely coupled many-task applications with various resource usage patterns raises a challenging issue to effectively map the tasks of the applications to computing nodes in the different platforms.

The heterogeneous platforms have different characteristics, since different hardware, system software stack, middle-ware, network and storage configurations are used for each of the platforms. Thus, the runtime of a task for an application can vary dramatically depending on which platform is used to run it. Moreover, the runtime of the task can be significantly affected by co-running tasks, from the same application or different applications, in the same computing node. As a result, in order to minimize the total makespan of multiple (loosely coupled) many-task applications and consequently maximize the overall throughput, the platform affinity as well as the co-runner affinity of the applications must be fully analyzed, and considered for scheduling.

In this work, we analyze the platform and co-runner affinities of many-task applications in

Table 3.1: Heterogeneous computing platforms used in our experiments

Type	Computing Platform	CPU	Memory Size	# Nodes	Network
PLSI	kias.gene (gene)	AMD Opteron 246 2.0GHz (2 Core)	8 GB	64	1Gbps
	unist.cheetah (cheetah)	Intel Xeon 2.53GHz (8 Core)	12 GB	61	1Gbps
Grid	darthvader.kisti.re.kr (darth)	Intel Xeon 2.0GHz (8 Core)	16 GB	8	1Gbps
Cloud	Local cloud (lcloud)	Intel Xeon 2.0GHz (12 Core)	32 GB (2.4GB per VM)	6	1Gbps

distributed computing platforms with different types of supercomputers, grids, and clouds. We perform a comprehensive experimental study using four computing platforms, and five many-task applications from real scientific domains (except one). We then define two metrics, one for the platform affinity, and the other for the co-runner affinity, and present a two-level scheduling algorithm, aiming to minimize the system makespan. It distributes the resources of different platforms to each of applications based on the platform affinity in the first level, and for each platform, it maps tasks of the applications, which are allocated some resources of the platform in the first level, to computing nodes based on the co-runner affinity in the second level. Finally, we evaluate the performance of our scheduling algorithm, using a trace-based simulator.

In our simulations, we compare our scheduling algorithm with two other algorithms, fair-AllCore and worst-AllCore, which are ignorant of and/or contrary to the platform and corunner affinities of applications, trying to show the maximum possible performance gain by being aware of the affinities. In the first level, the fair-AllCore algorithm allocates the resources of each platform equally to every application, while the worst-AllCore algorithm allocates the resources to an application with the worst platform affinity. In the second level, both of the algorithms basically map tasks from the same application to the same node for each platform. Our simulation results show that our algorithm can improve the performance up to 30.0% and 18.8%, compared to the worst-AllCore and fair-AllCore algorithms, respectively.

I Summary

This work has extended version with other authors, so we focus on part for resource management. In this section, we summarize base knowledge about our environment and experimental analysis.

1.1 Target platform and applications

In this section, we describe distributed computing platforms with three different types of supercomputers, grids and clouds, and five different many-task applications.

Heterogeneous Computing Platform

Table 3.1 shows hardware specifications of our four different computing platforms. PLSI stands for "Partnership and Leadership for the nationwide Supercomputing Infrastructure" and consists

Table 3.2: Size of input and output data and memory usage of a task for each application

Application	Input data	Intermediate data	Output data	Memory Usage
AutoDock	8.0MB	-	3.1KB	157.5MB
Blast	1.5GB	-	1.9MB	410.1MB
CacheBench	-	-	1.4KB	1 4GB
Montage	74.7MB	970.3MB	2.8MB	63.4MB
ThreeKaonOmega	-	-	6.3KB	0.6MB

various supercomputers. PLSI provides a global shared storage system based on GPFS [32], but it can be a performance bottleneck when many I/O operations are performed. The grid environment that has been used in our experiments consists of a single computing platform, operated by KISTI. Our grid platform is composed of a computing element (CE) and a storage element (SE), so that input/output data and executables are stored at the SE and worker nodes in the CE process tasks by utilizing the SE. It makes additional file transfer overhead. Our private cloud uses virtual machine (VM), and their image files are stored in the local disk of each host machine running the VMs, and all the worker VMs are created and started to run, before submitting tasks of applications to the platform.

Many-Task Applications

Table 3.2 shows the sizes of input/output data, and the average memory usage (over all the platforms) of a task for each of our five different many-task applications, AutoDock [33], Blast [34], CacheBench [35], Montage [36], and ThreeKaonOmega [37]. AutoDock is a suite of automated docking tools to predict how small molecules (such as substrates or drug candidates) bind to a receptor of known 3D structure (docking). In our case, we use this AutoDock to perform the docking of ligands to a set of target proteins to discover potential new drugs for several serious diseases such as SARS or Malaria. AutoDock is a mainly CPU-intensive application with small sizes of memory usage and input/output data files.

BLAST(Basic Local Alignment Search Tool) is a tool to find regions of local similarity between genome sequences by comparing nucleotide or protein sequences to a database of sequences and calculating the statistical significance of matches. BLAST requires a relatively larger input file to be used for sequence matching process so that the data staging cost from the storage to each compute node can be substantial compared to other applications.

Montage, an Astronomical Image Mosaic Engine, is a toolkit for assembling Flexible Image Transport System (FITS) images into composite images called mosaics. Unlike the other applications, each task of Montage generates a substantial amount of intermediate files which can result in total hundreds of MBs. Therefore, Montage is a very I/O-intensive application which requires efficient support of simultaneous I/O operations.

ThreeKaonOmega is a simulation study of the multi-particle production scattering problem in the nuclear physics area. The overall process of ThreeKaonOmega consists of high dimensional matrix computations, so it is pure CPU-intensive application with almost no memory usage and disk I/O operations.

We have used CacheBench to study behaviors of our platforms under heavily memory-intensive operations. For CacheBench, a different amount of memory is used for each platform, based on the amount of memory and number of cores per node.

1.2 Experimental Analysis

We investigated the performance of the five many-task applications, AutoDock(A), Blast(B), CacheBench(C), Montage(M), and ThreeKaonOmega(T) in the four different computing platforms. To understand the effect of performance interference among tasks running on the same computing node, i.e. co-runners, from either the same application or different applications, we measured the performance over various combinations of the applications in a node with n cores as follows:

- OneCore: only one task of an application runs in the node, regardless of available cores of the node.
- AllCore: n tasks from the same application runs in the node concurrently.
- TwoApps: $n/2$ tasks from each of two different applications runs in the node concurrently.
- FourApps: $n/4$ tasks from each of four different applications runs in the node concurrently.

Effects of Co-runners

We investigated the affinities of the five many-task applications to various combinations of co-runners. We execute applications with various combinations as mentioned above in four different computing platforms, but we can't execute FourApps combinations in gene because each node in gene has only two cores.

For AutoDock, the maximum performance degradation compared to OneCore is 15.6%. For different combinations, the effect of co-runners on the slowdown of AutoDock is similar to each other. For CacheBench, co-running tasks have no effect on its performance except Allcore in gene and darth. In these two cases, the size of memory of a computing node is not sufficient slightly to run 2 and 8 tasks, respectively, ending up using the swap disk of the node. For ThreeKaonOmega, the effect of co-runners tends to be modest, except TwoApps with Montage in cheetah, in which its runtime increases by 25.6%, due to the effects of co-running Montage tasks.

For Blast, it reads a large amount of input data during its execution, so its performance is significantly degraded in cheetah, when it is executed together with Montage. When a large amount of data access to GPFS occurs in chetah, then GPFS becomes a performance bottleneck,

Table 3.3: Runtime without co-runners over different platforms (seconds)

App	gene		cheetah		darth		lcloud	
	Runtime	SD	Runtime	SD	Runtime	SD	Runtime	SD
AutoDock	480.42	108.4	244.05	48.6	347.11	69.3	294.16	56.3
Blast	63.97	42.6	37.9	27.1	46.35	32.6	38.47	29.0
CacheBench	376.35	6.5	336.11	0.1	357.93	0.6	354.83	0.4
Montage	309.21	16.0	150.77	9.7	142.41	12.5	106.63	1.8
ThreeKaonOmega	200.18	8.9	71.02	3.0	101.08	10.3	86.88	0.6

an the CPU utilization drops significantly, degrading the performance of Blast. For Montage, it requires to write a large amount of data, so its runtime increases dramatically when multiple tasks of it run together due to contention in GPFS.

As co-runners, the effect of CacheBench is stronger than that of Montage in darth. When the length becomes larger than the size of the last-level cache, the performance degradation in darth is larger than that in lcloud and cheetah.

Effects of Platform

We investigated the affinities of the five applications to the different platforms. In case of darth, the characteristics of the grid should be reflected. We measure the file transfer overhead between CE and SE for five applications, and blast has significant overhead due to transferring it's large input file. To reduce this overhead, we assumed that 20 tasks are packaged as one bundle (as similar to [38]).

In Table 3.3, the average runtimes and standard deviations of tasks for each of the applications with OneCore over the different platforms are presented. For AutoDock, Blast and ThreeKaonOmega, the order of platforms with better performance, i.e. a higher affinity, is cheetah (the best), lcloud, darth, and gene. For Montage, its performance is significantly degraded in the PLSI platform due to GPFS overhead, so the best platform for it is lcloud. Cachebench shows almost same performance in the different platforms.

II Two-level scheduling mechanism

In this section, we first discuss two metrics, representing the platform and co-runner affinities of the applications. We then present a two-level scheduling algorithm based on the affinities for a system composed of multiple computing platforms.

2.1 Platform and co-runner affinity metrics

To measure the effects of platforms and co-runners on the performance of many-task applications, we define two metrics, platform affinity, and co-runner affinity. The platform affinity of an

Table 3.4: Platform affinity metric

	gene	cheetah	darth	lcloud
AutoDock	0.614	1.532	0.978	1.214
Blast	0.640	1.308	1.009	1.284
CacheBench	0.929	1.080	0.994	1.006
Montage	0.431	1.234	1.326	1.883
ThreeKaonOmega	0.431	1.822	1.181	1.428

Table 3.5: Co-runner affinity metric

	gene	cheetah	darth	lcloud
AutoDock	0.062	0.104	0.090	0.092
Blast	0.027	0.273	0.024	0.050
CacheBench	0.101	0.003	0.027	0.004
Montage	0.080	0.706	0.098	0.119
ThreeKaonOmega	0.042	0.058	0.161	0.043

application K to a platform P indicates how suitable platform P is to run application K. The platform affinity metric of K to P can be computed as follows. First, the average runtimes of tasks for K with OneCore in various platforms are normalized to the runtime in P. Then, the platform affinity of K to P is computed as the average of the normalized runtimes of the other platforms (not including P). For example, the average task runtimes of Montage on gene, cheetah, and darth, normalized to that of lcloud, are 2.90, 1.41, and 1.34, respectively, and so the platform affinity metric of Montage to lcloud is computed as $(2.90 + 1.41 + 1.34)/3 = 1.883$.

This metric of the platform affinity represents the relative runtime of application K, if only the other platforms are equally used to run all the tasks of K. Thus, if the computed value is larger than 1, we can conclude that it is beneficial to run K on P, since if not, the runtime of K will increase. The higher a value of the platform affinity is, the more suitable P is to run K. If the computed value is less than 1, then it means that the runtime of K decreases if P is not used to execute it, and so P is not suitable to run K.

The co-runner affinity of an application K to a platform P indicates how strongly the performance of K is affected by any co-runners in platform P. To compute the co-runner affinity of K to P, we first compute the co-runner affinity of K to each combination C, which is composed of some number of applications including K, in P. It indicates how suitable the tasks of the applications in C are to run together with a task of K in the same node in P.

For each combination C (of AllCore, and two/four applications), the difference between the runtime of K with C and that with OneCore (i.e. runtime with C - runtime with OneCore) in platform P is computed. Then, the computed difference is normalized by the runtime with OneCore. Note that it is unlikely, but it is possible, that the difference is negative, and in this case, we replace the negative value to 0. The lower a value of the co-runner affinity of K to C is, the more suitable combination C is to run K in P. Finally, the co-runner affinity of K to P is computed as the average of the co-runner affinity values of K to all the combinations.

This metric of the co-runner affinity of K to P represents the performance degradation of K when a task of K runs together with other tasks, from the same application or other applications, in the same node in P. Therefore, if the value of this metric is small, then we can conclude that the performance of application K is not strongly affected by any co-running tasks in P.

Table 3.4 and Table 3.5 show the platform and co-runner affinity metrics of the applications computed as the above for each of the platforms.

2.2 Two-level scheduling algorithm

To demonstrate the importance of being aware of the platform and co-runner affinities, we present a two-level scheduling algorithm that optimizes the system performance in a basic way based on the affinities. Our goal is to minimize the system makespan by allocating the computing cores of a platform equally to some number of applications that have relatively high platform affinity values for the platform, and then finding combinations, using those applications, which have a small effect of co-runners in the platform. Therefore, our algorithm decides the number of cores to be allocated to an application for each of the platforms based on the platform affinity in the first level, and for each platform, it maps tasks of the applications to nodes in the platform based on the co-runner affinity in the second level. Note that it can be advantageous to allocate the resources of a platform to several applications with diverse resource usage patterns, rather than one application with the highest platform affinity, for finding combinations that result in less performance degradation due to co-runners.

In this work, pilot jobs are used to reduce the overhead to schedule a large number of tasks for many-task applications. For a pilot job based system, there is a separate queue for each of submitted applications, and once a core of a platform is allocated to an application, a pilot job starts to execute on that core. It continuously fetches a task from the queue of the application, and executes until there are no tasks left in the queue. This mechanism is similarly implemented in [38–40].

For this affinity-aware scheduling algorithm, the following inputs are initially given.

- The number of nodes, and the number of cores per node for each of the platforms
- The number of (remaining) tasks, N_tasks, for each of the applications
- The platform and co-runner affinity metrics of each of the applications over all the platforms

In the first level scheduling, for each of the platforms, we first compute the average of the platform affinity values of all the applications, and the number of applications with a platform affinity value higher than the average. Basically, the available cores of a platform are distributed and allocated equally to the applications with a platform affinity value higher than the average. However, the number of cores allocated to each application should be limited by N_task. Thus, we allocate the cores of a platform to each of the applications in increasing order sorted by N_task of the applications, so that if an application needs less than the equal share of cores for the platform, the remaining cores will be allocated to the other applications.

Also, for an application K, we decide the number of cores to be allocated to K for each of the platforms in decreasing order sorted by its platform affinity. In this way, if N_task for K is less than the total number of cores which can be allocated to K from the different platforms, we can allocate cores from a platform with a higher value of the platform affinity first to application K.

After this process, it is possible that the available cores of some platforms have not been distributed yet, and some applications still need to get more cores. In this case, we repeat the above process. When we recompute the average value of the platform affinity for a platform with some available cores, we only consider the applications that still need to get more cores. Thus, the newly computed average can become larger than the old one. For example, initially the average of the platform affinity for gene is 0.609, but if ThreeKaonOmega is allocated all the needed cores and so it is not included to compute the average, the newly computed average becomes 0.653. Using the newly computed average, some application like Blast that has been allocated the resources from gene in the previous round cannot get the resources from gene any more. To avoid such cases, we remain to use the old average value.

Once the first level scheduling is done, for each platform, a set of the applications assigned to the platform and the number of cores allocated to each of the applications (i.e. the number of pilot jobs to be executed for the application) from the platform are computed. In the second level scheduling, for each of the platforms, we first search an application with the maximum co-runner affinity (whose performance can be most affected negatively by co-runner tasks) to the platform. For the selected application, we find a combination C with the smallest co-runner affinity. We then attempt to map this combination to a node as much as possible in the platform. If the remaining number of pilot jobs for some of the applications in C is not sufficient to create a whole combination any more, we repeat the above process to find a new combination from the remaining applications. If only a few number of pilot jobs, which are not enough to make a combination, from some of the applications remain, we make a random combination and assign it to a node.

In this scheduling algorithm, an application with low values of the platform affinity over all the platforms may suffer from starvation if applications with high values of the platform affinity are submitted continuously. In this case, an aging technique can be used to increase the values of the platform affinity periodically for such starving applications. Note that the scheduler monitors

Table 3.6: Default resource configuration

	gene	cheetah	darth	lcloud
# nodes	60	15	15	10
# cores per node	2	8	8	12
# total cores	120	120	120	120

Table 3.7: The number of tasks in default workload

	AutoDock	Blast	CacheBench	Montage	ThreeKaonOmega
# Tasks	9,360	68,400	9,000	18,000	27,648

and adapts to changes in the system, such as the submission and termination of applications, and the addition and removal of platforms. For a change, the scheduler redistributes the resources to the applications and re-map pilot jobs to nodes, but our system allows running tasks to complete without preemption as in [39].

III Simulation Results

To evaluate the performance of our affinity-aware scheduling algorithm, we have implemented a trace-based simulator in C++. The trace of task runtimes obtained from the experiments discussed in Section I was used in the simulations. For `AllCore`, we have measured runtimes for all the tasks, but for the other combinations, we have measured runtimes for a subset of the tasks. For the simulations, to estimate the runtime of a task in these combinations, we used regression analysis based on the measured runtimes for a subset of the tasks. For an application K and a combination C in a platform, it estimates the relationship between the runtime of a task for K with `OneCore` and the runtime of the task with C.

In the simulations, we assume that all applications are submitted at the same time. Table 3.6 shows the default resource configuration used in our simulations. The total number of cores in the system was 480. Table 3.7 shows the default workload. For this workload, we replicated tasks of the applications used in Section I. It was generated such that the five applications finish their execution around the same time, if the resources of every platform are allocated equally to each of the applications, and each task is executed with `OneCore`.

For each simulation run, a particular task of an application can be executed on different platforms with a different combination. This is because when two or more pilot jobs of the application from multiple platforms try to fetch a new task in the queue of the application at the same time, the task at the head of the queue is assigned to one of the pilot jobs randomly.

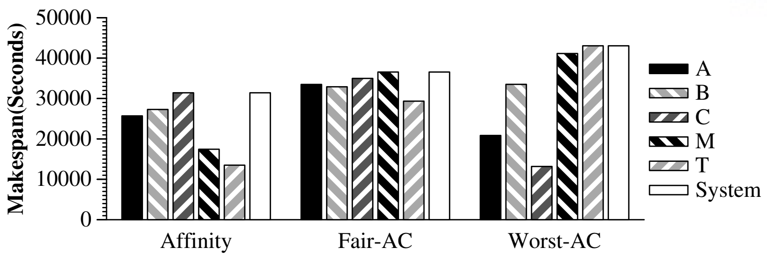


Figure 1: Makespans with the three algorithms

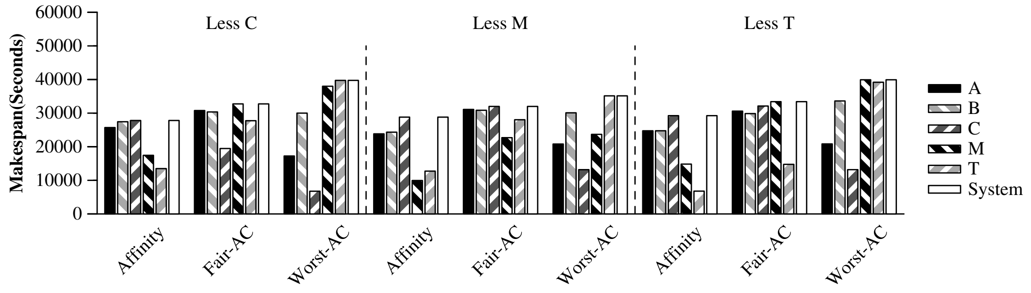


Figure 2: Makespans over various workloads

For the result of each simulation, the average of 100 simulation runs was computed.

To analyze the maximum possible performance improvement by being aware of the platform and co-runner affinities, we implement two other algorithms, *fair-AllCore* and *worst-AllCore*, and compare our algorithm with them. In the fair-AllCore algorithm, the scheduler allocates the resources of each platform *fairly* (i.e. equally) to every application in the first level, and for each platform, it basically maps tasks of the applications with `AllCore` in the second level. For a system that is unaware of the platform and co-runner affinities, this algorithm may be commonly used. In the worst-AllCore algorithm, the scheduler allocates the resources of a platform to an application with the worst platform affinity in the first level, and then maps tasks to nodes with `AllCore` in the second level. The performance of this algorithm may be almost close to that of the worst case, and so we use its performance as a baseline.

Figure 1 shows the makespans of the applications and the system when the default setting is used for our affinity-aware algorithm (Affinity), the fair-AllCore algorithm (Fair-AC), and the worst-AllCore algorithm (Worst-AC). The performance improvements of Affinity and Fair-AC algorithms against Worst-AC algorithm are 27.0% and 14.1%, respectively. In Affinity algorithm, ThreeKaonOmega and Montage complete their execution earlier than the other applications, since they have relatively higher values for the platform affinity. Thus, ThreeKaonOmega and Montage are allocated a larger number of cores than the other three applications. Also the performance of Montage can improve by using a combination other than `AllCore`. However, CacheBench is allocated 40 cores of gene, and continues to use only the cores until AutoDock is done. Thus, it finishes the execution at last. In Fair-AC algorithm, Montage has the longest makespan, because its performance is significantly degraded with `AllCore`.

Figure 2 shows the makespans over various workloads with the default resource configuration.

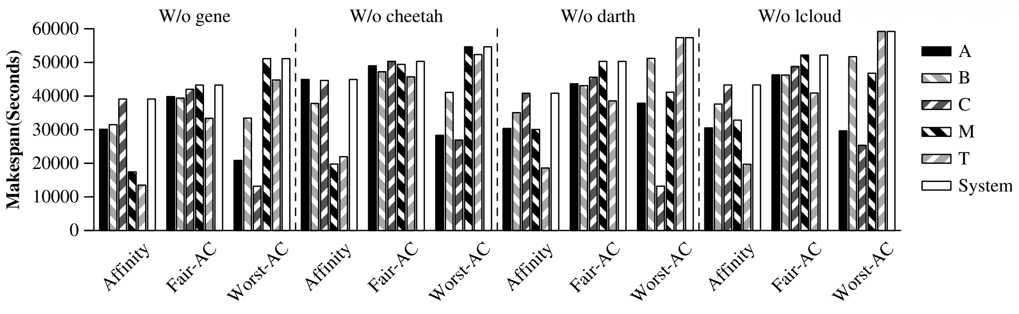


Figure 3: Makespans over various resource configurations

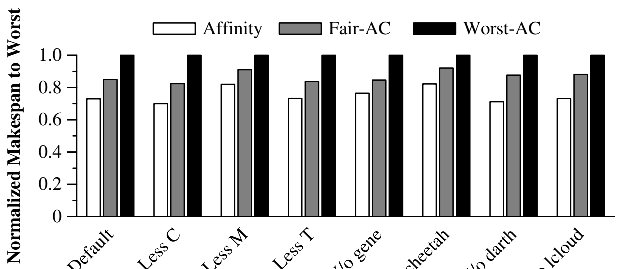


Figure 4: Makespans normalized by the worst-AllCore algorithm

In each *"Less K"* workload, the workload of application K is reduced to a half of the default workload. Figure 3 shows the makespans over various resource configurations of the system for the default workload. In each *"W/o P"* configuration, the resources from platform P are not used, resulting that the total number of cores in the system is 360. In these results, we can observe the similar trends for the three algorithms as in Figure 1.

In Less M and W/o cheetah, the improvement of Affinity algorithm against Worst-AC algorithm is reduced. Montage is affected considerably by the platform and co-runner affinities, and so in Less M, the workload that can be executed more efficiently by Affinity algorithm is reduced. When the resources of cheetah, which is a platform with a higher value of the platform affinity for all the applications except Montage, are not available, Affinity algorithm cannot use these resources for improving the performance. In W/o darth and W/o lcloud, the performance gap between Affinity and Fair-AC algorithms is a bit increased, compared to the default setting in Figure 1. Because Montage has the high platform affinity to lcloud and darth, the performance of Montage degrades in W/o darth and W/o lcloud. However, with Affinity algorithm, Montage is assigned the resources of darth in W/o lcloud, and those of lcloud in W/o darth, and so the performance degradation of Montage is less severe, compared to Fair-AC algorithm.

Figure 4 presents the system makespans of the three algorithms normalized to those of Worst-AC algorithm for the simulations discussed above. The performance improvement of our algorithm against Worst-AC algorithm is 30.0% at maximum and 24.8% on average, and that against Fair-AC algorithm is 18.8% at maximum and 13.5% on average.

IV Concluding Remarks

In this work, we analyzed the platform and co-runner affinities of many-task applications in distributed computing platforms with different types of supercomputers, grids, and clouds. Our comprehensive experimental analysis showed that both platforms and co-runners can affect the performance of applications dramatically. We also demonstrated that it is critical to consider the platform and co-runner affinities on scheduling. The affinity-aware algorithm discussed in this work is based on basic optimization. A sophisticated affinity-aware scheduling algorithm that can optimize the performance effectively over various scenarios is considered as our future work.

Chapter 4

Constraint-aware VM Placement in Heterogeneous Computing Cluters

Virtualized systems such as virtualized data centers and cloud computing systems consist of a large number of machines that are configured with different hardware and software [41, 42]. This heterogeneity occurs because it is impossible to configure a large-scale system only with one generation of machines that have the same hardware configurations. In such virtualized systems, a large number of applications with various resource requirements can be executed on virtual machines (VMs) by sharing the resources of the systems. To initially or dynamically place a VM on one of nodes in a system, the resource requirement of the VM basically needs to be considered. In addition, there can be other placement conditions to be satisfied for the placement of the VM. Simple placement constraints of the VM for CPU compatibility [43], special hardware requirement, and license requirement [44] can be imposed based on machine attributes. Also, more complex constraint conditions between VMs can be requested such that two VMs must not be co-located on the same node due to the possibilities of security leaks [45], or a set of certain VMs must be separated in different nodes for increasing the availability of a service running on them [46]. Furthermore, to improve performance, the co-location of VMs, which potentially share many identical memory pages or exchange a large number of messages, on the same node can be requested [47, 48].

It has been reported that in Google clusters, approximately 50% of jobs have constraints on machine attributes, while 11% of the jobs have complex constraints [41]. In [42], Google trace analysis shows that around 6% of jobs are submitted with some constraints, and most of them are simply based on machine attributes. However, we expect that more complex constraints will be requested by users or administrators of the system, as the heterogeneity of resources and applications increases. Moreover, in Cloudera and Yahoo cluster traces, the scheduling delay of tasks is increased more than twice [49], and in Google traces, the delay is increased up to six times [41], because of the constraints of the tasks. As a task needs to wait to be scheduled on a particular node that satisfies the constraints of the task, the utilization of some number of nodes in a cluster may be decreased.

Existing VM management systems such as VMware vSphere [50], Microsoft SCVMM [51], and XenServer [52] allow users to specify VM placement constraints. However, by using the features provided in the systems, the users may not easily specify complex constraints of VM placement based on attributes of VMs and nodes. Also, these constraints can be specified in different paths by multiple users, administrators, and even system developers (to provide some functionality like fault tolerance [46]). Therefore, the constraint conditions can *conflict* with each other, but currently there is no mechanism to easily figure out these conflict conditions. In the existing systems, the older constraints take precedence or the newer constraints overwrite, which may result in ignoring critical constraints.

A VM placement algorithm selects one of nodes in a system for a VM, aiming to improve the performance of the system toward the optimization objective such as balancing the load of nodes, or reducing energy consumption. VM placement constraints can increase the complexity of VM placement, limit the choice of nodes for VMs, and affect the performance of the system negatively. Also, we may not find any node satisfying the constraints of a VM, failing to create the VM in the system. Various combinations of VM placement constraints can affect the performance of the system differently. Thus, it is important to understand the net effects of having the constraints on VM placement.

In this work, we study constraint-aware VM placement in heterogeneous computing clusters. We first present a model of VM placement constraints that supports all types of constraints between VMs, and between VMs and nodes. In a cluster, a constraint that two or more VMs must be placed together on the same node can be specified due to various reasons. For example, if a user has two VMs that exchange critical information, then the user can request that the two VMs be co-located on the same node. In this way, the user can ensure that messages exchanged between them are not sent outside of the node. In our model, we partition VMs in VM units, such that VMs in the same unit must be co-located on the same node and thus, the VMs are equivalent with respect to VM placement constraints. We discuss how to compute a set of candidate nodes for a VM unit, based on the constraints of each VM in the unit. Second, we discuss six constraint-aware VM placement algorithms that place a new VM on one of nodes in a cluster. For the VM placement algorithms, we consider two different optimization objectives of energy saving and load balancing, which are commonly used in virtualized systems. Third, using simulation, we study the effects of VM placement constraints on VM placement, and evaluate the performance of the algorithms over various settings. We also run experiments of the algorithms in a small cluster.

Our extensive simulation results demonstrate that the effects of VM placement constraints vary, depending on the optimization goal, the types of the constraints, and the system configurations. From the results, we also conclude that an *activeness-aware algorithm* which attempts to place a new VM on one of active nodes by utilizing VM migrations, and a *spot migration algorithm* which attempts to migrate some VMs from a selected node for a new VM, i.e. a potential hotspot, to another node provide good performance for energy saving and load balancing, respectively.

The main contributions of this work are as follows.

- We present a model of VM placement constraints which can simplify and compact complex constraint conditions of VMs, and reduce overhead to check the conditions on placing VMs on nodes in a heterogeneous cluster.
- We discuss total six constraint-aware VM placement heuristics, optimizing toward two common goals of energy saving and load balancing.
- We perform extensive simulation study and experiment to provide comprehensive analysis on the performance of the six constraint-aware VM placement algorithms as well as the effects of VM placement constraints over widely diverse cluster and constraint configurations.
- We compare the constraint-aware VM placement algorithms that have the same goal with each other, and analyze the effect of different configurations on the algorithms. Our work can be useful for administrators to build and manage their systems properly based on common constraints requested for the systems.

I Virtualized Systems and Optimization Goals

We consider a virtualized system as follows. The system consists of some number of virtualized nodes, which are configured with heterogeneous hardware and software, and these nodes form a cluster. For each VM, VM placement constraints and requirements for resources such as CPU, memory, and I/O can be specified. In this system, a VM can be placed on any node in the cluster, if the node satisfies the placement constraints and resource requirements of the VM. Moreover, we assume that the cluster is a live migration domain of running VMs. In other words, a VM can be dynamically migrated to any other node, which satisfies its placement constraint, in the cluster. Note that if the scale of a system becomes too large, then we can divide the system into multiple clusters, and have a management policy to select which cluster to be used for a new VM.

For placing VMs in the system, there are two popularly used optimization goals, *energy saving*, and *load balancing* as in [53–55].

1. Energy saving: A VM placement algorithm needs to minimize the number of *active* nodes, on which at least one VM is placed. The system can save energy by turning off unnecessary (i.e. inactive) nodes, or putting those nodes in a lower power state.
2. Load balancing: A VM placement algorithm aims to balance load over all the nodes, preventing the performance degradation of VMs running on overloaded nodes in the system, and utilizing the resources of the system efficiently.

II A model of VM placement constraints

2.1 Types of VM placement constraints

Each constraint for a VM can be expressed as a predicate on attributes of either nodes or VMs in a cluster (similar to [41, 56, 57]) along with a constraint type which will be discussed below. An example of predicates for node attributes is that the CPU architecture of a node is Intel, its Ethernet speed is faster than x, and it is configured with GPGPU. A predicate example for VM attributes is that the owner of a VM is company Y. In this work, we consider that all specified VM placement constraints for VMs are *mandatory*, which must be satisfied.

Note that for a VM, constraints regarding resource usage patterns of other VMs on the same node can be provided as predicates, along with the requirement of the VM for various resources such as CPU, memory and I/O. For example, it can be requested that a CPU-intensive VM be co-located with I/O-intensive VMs to avoid contention for the same resource type [58]. Also, it can be requested that VMs having many identical memory pages (because they execute the same operation system/applications or process the same data) be placed together. In this way, identical memory pages can be consolidated in a single shared page reducing memory usage [47, 59].

Each VM placement constraint can be specified as one of the following types:

- *VMtoNode-Must*: This type of a constraint describes a node on which a VM must be placed. For this type, a predicate on node attributes is given, and the VM must be placed on one of the nodes satisfying the predicate. Thus, a VMtoNode-Must constraint can be computed as a set of all the nodes for which the given predicate is true. (Note that a “VMtoNode-MustNot” constraint can be easily converted to a VMtoNode-Must constraint.)

- *VMtoVM-Must*: This type of a constraint describes a VM with which a VM must be placed on the same node. For this type, a predicate on VM attributes is given, and the VM must be placed with all the VMs satisfying the predicate. Thus, a VMtoVM-Must constraint can be computed as a set of all the VMs for which the given predicate is true, and VMs in the same set must be placed on the same node.

- *VMtoVM-MustNot*: This type of a constraint describes a VM with which a VM must not be placed on the same node. For this type, a predicate on VM attributes is given, and the VM must not be placed with any of the VMs satisfying the predicate. Thus, a VMtoVM-MustNot constraint can be computed as a set of all the VMs for which the given predicate is true, and VMs in the same set must be placed on different nodes.

Examples of the above constraint types are as follows. For the VMtoNode-Must constraint, a VM must be placed on a node with SSDs and 10 GE network to utilize high storage and network bandwidth. For the VMtoVM-Must constraint, two VMs must be placed on the same node as they exchange a large amount of messages with each other [48]. Finally, for the VMtoVM-MustNot constraint, a primary VM and its backup VM must not be placed on the same node to provide the fault tolerance [60].

If multiple constraints of the same type are given for a VM, we can combine them as a single set of either nodes or VMs as follows. For multiple VMtoNode-Must constraints, we compute the intersection of the constraints, $S_{VtoHMust}$. For multiple VMtoVM-Must constraints, we compute the union of the constraints, $S_{VtoVMust}$. For multiple VMtoVM-MustNot constraints, we also compute the union of them, $S_{VtoVMustNot}$.

2.2 VM equivalence with respect to VM placement constraints

In our model, a list of constraints is given for each VM. Thus, VMtoVM-Must and VMtoVM-MustNot constraints can be ambiguous if a VMtoVM-Must or VMtoVM-MustNot constraint between two VMs, i and j, is specified only for one of the VMs, say i. In this work, we simply remove such ambiguity by applying a symmetric relation for VMtoVM-Must and VMtoVM-MustNot constraints. Thus, the constraint is added to the constraint list of VM j.

The VMtoVM-Must constraint is also a transitive relation. If $i \sim j$ (i.e. a VM i must be co-located with a VM j), and $j \sim k$, then $i \sim k$. Moreover, the VMtoVM-Must constraint is reflexive (i.e. $i \sim i$). Thus, the VMtoVM-Must constraint between VMs is an equivalence relation. By the relation of the VMtoVM-Must constraint of VMs, we can partition all VMs into *VM units*, such that a VM is a member of one and only one VM unit, and the union of all the VM units is equal to a set of all the VMs. VMs in the same VM unit are equivalent with respect to VM placement constraints, since they must be placed together on the same node.

Next, we discuss how to compute the VMtoNode-Must and VMtoVM-MustNot constraints for a VM unit u. The VMtoNode-Must constraint for u, i.e. a set of nodes called $N_{VtoHMust}$, is computed as the intersection of the VMtoNode-Must constraint sets of all the VMs in u. The VMtoVM-MustNot constraint for u, i.e. a set of VMs called $VM_{VtoVMustNot}$, is computed as the union of the VMtoVM-MustNot constraint sets of all the VMs in u.

For a VM unit u, if $N_{VtoHMust}$ is empty, or the intersection between u and $VM_{VtoVMustNot}$ is not empty, then we consider that the constraints of the VMs in u *conflict* with each other. It means that there is no possible placement of u. Conflicting constraints should be informed to users or administrators, and they should resolve conflicts before requesting to create the VMs again. Also, if the total resource requirement of u is larger than the capability of any node in the cluster, the placement of u will fail. Similar to conflicting constraints, such a large resource requirement should be informed to users or administrators so that they can adjust the constraints and/or resource requirements of the VMs in u.

To compute a set of candidate nodes that satisfy all the placement constraints and resource requirements of a VM unit u, we compute a set of nodes that satisfy the VMtoVM-MustNot constraints for u, $N_{VtoVMustNot}$, as

$$N_{VtoVMustNot} := N_{all} - N_{exclude}$$

where H_{all} is a set of all nodes in the cluster, and $H_{exclude}$ is a set of nodes on which running VMs in $VM_{VtoVMustNot}$ are currently placed. We also need to compute a set of nodes that satisfy the total resource requirements for u, N_{res}. To compute N_{res}, we first compute the total resource requirements for all the VMs in u. We then filter out nodes that currently do not have enough resources to satisfy its resource requirements from a set of all nodes in the cluster. Finally, we can compute a set of candidate nodes for u, N_{cand}, as

$$N_{cand} := N_{VtoHMust} \cap N_{VtoVMustNot} \cap N_{res}$$

In this model, for each VM unit, the constraints of the VMs in the unit are *merged*, and their redundant placement constraints, if any, are *compacted*, reducing the overhead to check the same conditions. Moreover, conflicts among constraints of the VMs can be simply checked.

III VM Placement Algorithms

3.1 Overview

A VM placement problem that maps each VM to one of the nodes in a cluster is considered as a bin packing problem, which is known to be NP-hard. In this work, we view a VM placement problem as placing a VM unit, instead of an individual VM, on a node in the cluster. For our algorithms, we assume that VM placement constraints given for VMs in the same VM unit do not conflict with each other.

Without a notion of a VM unit, every time we (re)place an individual VM i, we need to compute a set of VMs with which i must be co-located, and consider the current placements and constraints of the VMs in the computed set. Also, we cannot migrate i alone to another node, if i must be placed together with some other VMs. Thus, we better maintain some information about VM units. If not, the VM placement algorithm becomes extremely complicated. Therefore, at runtime, we assume that the system keeps track of a list of VM units and their constraint information.

A constraint-aware VM placement algorithm is invoked when a new VM is created. It tries to place the new VM on one of nodes satisfying its constraints, while optimizing the system for a target goal. The overview of a constraint-aware VM placement algorithm is given in Algorithm 1. To place a new VM v, we first resolve the constraint ambiguity for a new VM v. On doing so, we need to check not only the constraints of v, but also the constraints of all running VM units to see if there are any constraints regarding v. We then generate a VM unit u for v based on the VMtoVM-Must constraints of v and all running VMs in the system ($v \in u$). VM v can be added to an existing VM unit, or it can become a new VM unit whose member is only itself. In some cases, two or more existing VM units may need to merge to one unit due to the VMtoVM-Must constraint of v. When v is added to an existing unit, its VMtoVM-MustNot and VMtoNode-Must constraints need to be updated to reflect the constraints of v. On generating a VM unit

Algorithm 1 VM Placement Algorithm

```
 1: input: a new VM $v$ with constraints $S_{VtoNMust}$, $S_{VtoVMust}$ and $S_{VtoVMustNot}$, and snapshot of a
    cluster
 2: var $u$: set of VMs;
 3:      $n_{tar}$: a node;
 4: resolveAmbiguity($v$, $S_{VtoVMust}$, $S_{VtoVMustNot}$);
 5: $u$:=generateUnit($v$, $S_{VtoNMust}$,$S_{VtoVMust}$,$S_{VtoVMustNot}$);
 6: $N_{cand}$ := compH_cand($u$, $N_{all}$);
 7: if $H_{cand} \neq null$ then
 8:    $n_{tar}$ := selectNode($u, N_{cand}$);
 9:    for each $v_i$ in $u$ where $v_i \neq v$ do
10:       if getCurrentNode($v_i$) $\neq n_{tar}$ then
11:          migrateVM($v_i, n_{tar}$);
12:       end if
13:    end for
14:    placeVM($v, n_{tar}$);
15: else if rearrangeVMsInSequence($v$, $u$, $N_{all}$) $= FAIL$ then
16:    return $FAIL$;
17: end if
18: return $SUCCESS$;
```

and computing constraints for the unit, we only need to consider the constraints of currently running VMs. Once we have a unit u for v, we compute a set of candidate nodes N_{cand} for u as discussed in the previous section. The algorithm selects one of the nodes in N_{cand}, which maximizes the performance toward the goal.

If N_{cand} is empty, then we can conclude that there is no node satisfying the VM placement constraints and resource requirements for the new VM v in the current state, and the creation of v is failed. However, besides optimizing the goal, VM creation failure is another very important aspect to consider for placing VMs. To reduce VM creation failure, we can attempt to rearrange existing VMs in some sequence to create v, using VM live migration (similar to [61]). In this *VM rearrangement* feature, basically some VM unit on a node n is migrated to another node, so that node n now satisfies all the constraints and resource requirements for u. Then the existing VMs in u are migrated to n in sequence, and v is finally placed on n.

In this case, we count the *length of the sequence* as 2. (VM live migration is similarly used to solve the violation of constraints in [61].) Note that it is possible to attempt the rearrangement of VMs more exhaustively, if the above attempt fails. For example, we can try to migrate a VM unit u_1 from a node n_2 to a node n_1, and then migrate another VM unit u_2 from a node n_3 to n_2, so that u and v can be placed on n_3. In this case, the length of the sequence is 3.

```
 1: FUNCTION rearrangeVMsInSequence($v, u, N_{cand}$)
 2: $N_{VtoHMust}$ := compN_VtoNMust($u \cup v, N_{cand}$)
 3: $N_{VtoVMustNot}$ := compN_VtoVMustNot($u \cup v, N_{cand}$)
 4: $N_{res}$ := compN_res($u \cup v, N_{cand}$)
 5: for each node $n$ in $N_{VtoNMust} \cap N_{VtoVMustNot}$ in increasing order of index do
 6:    $U_{res}$ := findUnitsLargeRes($n, u \cup v$)
 7:    for each unit $u'$ in $U_{res}$ do
 8:       if migrateVMsInSequence($v, u, u', n, N_{cand}$) then
 9:          return $SUCCESS$;
10:       end if
11:    end for
12: end for
13: for each node $n$ in $N_{VtoHMust} \cap N_{res}$ do
14:    $U_{VtoVMustNot}$ := findUnitsVtoVMustNot($n, u \cup v$)
15:    for each unit $u'$ in $U_{VtoVMustNot}$ do
16:       if migrateVMsInSequence($v, u, u', n, N_{cand}$) then
17:          return $SUCCESS$;
18:       end if
19:    end for
20: end for
21: return $FAIL$;
```

3.2 VM Placement for Energy Saving

For VM placement algorithms to maximize energy saving of a cluster, we basically use the *first fit* heuristic to place a VM on one of active nodes, if possible, aiming to reduce the total number of active nodes used for currently running VMs.

ES-Basic Algorithm. This algorithm simply uses the first fit heuristic for selecting a node among candidate nodes (i.e. in line 8 for Algorithm 1). This algorithm is used as a baseline for comparison. On selecting a node based on the first fit heuristic, we assume that nodes are basically sorted by their index in increasing order.

Attribute-Aware Algorithm. This algorithm works in the exact same way as the ES-basic algorithm, except that it sorts computed candidate nodes by the number of attributes that a node possesses and satisfies in decreasing order. (It uses the index to break a tie.) Therefore, VM v can be placed on one of nodes having many satisfied attributes, and consequently, this selected node can be used for other VMs that have been already running or will be created later.

Activeness-Aware Algorithm. This algorithm aims to place a new VM v on one of currently active nodes, if possible. It first computes candidate nodes for v only among currently active nodes unlike the other algorithms. If one or more active candidate nodes exist, it places v on one of them based on the first fit heuristic. However, if there is no active candidate node, it tries to rearrange existing VMs within the currently active nodes (as discussed below), so that the VM unit of the new VM unit can be placed on one of the active nodes. If rearranging VMs fails for creating v, it concludes that it is impossible to place v on one of the active nodes. The algorithm then computes candidate nodes again for v considering all the nodes in the system, and selects a node as it does in the ES-basic algorithm.

The VM rearrangement feature discussed above can be implemented in several ways. In this work, for energy saving, rearranging VMs within a given set of nodes, G, for a new VM v and its VM unit u is performed as follows. It first searches a set of nodes in G, which satisfy the VMtoNode-Must and VMtoVM-MustNot constraints of u (i.e. $N_{VtoHMust} \cap N_{VtoVMustNot}$). Then for each node h in the computed set (in increasing order of index), it searches VM units whose resource requirements are equal to or larger than those of u. If there is such a unit u', it attempts to migrate u' from n to another node which satisfies the constraints of u' in G, so that it can migrate each of existing VMs in u to n, and finally place v on h. If the above attempt fails, it searches a set of nodes in G which satisfy the VMtoNode-Must constraints and resource requirements of u (i.e. $N_{VtoHMust} \cap N_{res}$). Then for each node n in the computed set, it searches VM units that cannot be placed with u due to the VMtoVM-MustNot constraint. If there is such a unit u', the rearrangement of VMs in u' and u, followed by the placement of v, is attempted as described above.

3.3 VM Placement for Load Balancing

For VM placement algorithms to achieve load balancing (LB), we basically use the *worst fit* heuristic to place a VM on a node which has the largest amount of free resources. In this work, we focus on CPU resources of nodes in a cluster for load balancing. Thus, the algorithms aim to minimize the standard deviation of CPU utilization for all the nodes in the cluster.

LB-Basic Algorithm. This algorithm simply uses the worst fit heuristic for selecting a node among candidate nodes, and it is used as a baseline.

Periodic Migration Algorithm This algorithm selects a node for the new VM exactly in the same way as the LB-basic algorithm. However, it periodically checks the overall VM placement state, and dynamically migrates some number of running VMs to minimize the load imbalance metric. For dynamic VM migration, this algorithm first finds a node with the maximum CPU utilization. Among VM units in that node, it then finds a unit u and another node n such that the imbalance metric is reduced if u is migrated to n. This process is repeated until the imbalance metric cannot be reduced any more. (Similar dynamic placement features have been studied [55, 62].)

Spot Migration Algorithm. This algorithm first selects a node for the new VM in the exact same way as the LB-basic algorithm. However, after then, the algorithm sees if it is possible to decrease the load imbalance metric by migrating some VM unit u' running on the selected node, which becomes a potential *spot* for high resource utilization by placing the VM unit of the new VM, to another node n' (that satisfies the constraints of u'). If there is such a VM unit on the selected node, and the improvement is larger than a predefined threshold, it migrates the unit to node n'. For each running VM unit u' in the selected node in decreasing order of the VM unit size, we attempt to migrate u' based on the worst fit heuristic. This *spot migration* can be done for only one VM unit at the creation of each VM, and only VM units running on the selected node are considered for dynamic migration.

In the VM rearrangement feature for load balancing, we similarly rearrange VMs to place the new VM and its VM unit u as discussed above for energy saving. However, we need to check all possible ways to rearrange VMs in sequences to find the one that can maximize the load balance of the cluster. Thus, we hypothetically try to migrate every unit whose resource requirement is equal to or larger than that of u on each of nodes in $N_{VtoNMust} \cap N_{VtoVMustNot}$, and also try to migrate every unit that has the VMtoVM-MustNot constraint with u on each of nodes in $N_{VtoNMust} \cap N_{res}$. Then for all the possible sequences to place the new VM, we select one with the minimum value of the load imbalance metric.

Note that without considering VM rearrangement, the complexity of the above ES and LB algorithms except the spot migration and periodic migration algorithms is $O(N_{node})$, and the complexity of VM rearrangement is $O(VN_{node})$, where N_{node} is the number of nodes and V is the number of currently running VMs. The complexity of the spot migration algorithm is $O(VN_{node})$, and that of the periodic migration algorithm is $O(V^2N_{node})$ when it periodically migrate VMs (under the assumption that each VM is migrated to another node at most once).

IV Simulation Results

4.1 Methodology and Metrics

For our simulation, we modified CloudSim [63] such that VMs and nodes can have some attributes, and VMs can have placement constraints based on the attributes of VMs and nodes. We also added features to create VMs dynamically and deal with VM units, and implemented the six constraint-aware VM placement algorithms discussed in Section III. To provide comprehensive analysis on the effects of VM placement constraints, we identify various parameters for the VM placement constraints and cluster configurations, which may affect the performance of the VM placement algorithms. We then synthetically generate a diversity of constraint scenarios, and run simulations of the algorithms over those scenarios.

Cluster Setting. Table 4.1 presents the configurations of nodes and VMs used for our simulations. For the runtime and migration overhead of a VM, we use the values measured from one of SPEC CPU2006 applications given in Table 4.2.

We assign attributes to nodes in a cluster as follows. The nodes have N_{att} attributes, att_0, att_1, ..., $att_{N_{att}-1}$, where $N_{att} \geq 2$, and each attribute has two possible values. For att_0, the possible values are 1 and 2, while for the other attributes, those are 0 and 1. Value 0 indicates that a node does not possess the specified attribute, while different positive values indicate the different types of the attribute. Note that att_0 represents one kind of attributes, for which each node must possess the attribute, such as the CPU architecture with Intel and AMD types,

Table 4.1: Cluster configurations

Total num. of nodes (N_{node})	128, 256, 512	
Num. of cores per node (N_{core})	4, 8, 16	
Num. of nodes with same attribute values	8, 16, 32	
VM types	Require 100% of one core	40%
	Require 75% of one core	40%
	Require 50% of one core	20%
Total CPU requirement of VMs	75% of total CPU resources	

and the other attributes represent the other kind, for which some of nodes in the cluster can be selectively configured to have the attributes such as SSD and GPU. For each attribute, we assign possible values to the nodes, such that the cluster is configured to have an equal number of nodes with the same attribute value. After assigning values for all the attributes, we sort the nodes in a random order. For the default cluster of 256 nodes, N_{att} is three, and the number of nodes with the exactly same node attributes is 32. After assigning values for all the attributes, we sort the nodes in a random order, and assign them index. We set the max CPU utilization threshold of each node to 85% to have some amount of headroom (as discussed in [64]).

Constraint Generation. Each user or user group can own multiple VMs. We call a set of VMs owned by the same user (or user group) a *VM group*. A user can have multiple VM groups based on his or her need. For VMs in the same VM group, a subset of the VMs may need to place on the same node, becoming a VM unit. On generating simulation scenarios, we consider the following four relationships among VM units in the same group.

1. All the VM units have the VMtoVM-MustNot constraint with respect to each other, and have the same VMtoNode-Must constraint condition.

2. All the VM units have the VMtoVM-MustNot constraint with respect to each other, but have no VMtoNode-Must constraint.

3. All the VM units have the same VMtoNode-Must constraint condition, but have no VMtoVM-MustNot constraint.

4. All the VM units have neither VMtoVM-MustNot constraint nor VMtoNode-Must constraint.

We define the *VM unit size* as the total CPU resource requirement of a VM unit considering that the value of the CPU resource for one CPU core is 100% since each VM can have a different requirement for the CPU resource. For example, for a VM unit u, if the size of u is 500%, then the total CPU resource requirement of all the VMs in u is 5 cores. In addition, we define the

VM group size as the total number of VM units in a VM group. We assign VM placement constraints of VMs as follows.

1. For all the VMs to be created, we first randomly create multiple VM units and form multiple VM groups from the generated VM units such that VM unit sizes and VM group sizes are randomly selected within ranges limited by the maximum VM unit size M_u and the maximum VM group size M_g, respectively.

2. Each group may have the VMtoNode-Must and/or VMtoVM-MustNot constraints. If the VMtoNode-Must constraint rate, $CR_{VtoNMust}$, is $x\%$, then $x\%$ of the groups have the VMtoNode-Must constraint. Similarly, if the VMtoVM-MustNot constraint rate, $CR_{VtoVMustNot}$, is $y\%$, then $y\%$ of the groups have the VMtoVM-MustNot constraint.

3. For each VM group with the VMtoNode-Must constraint, it has an equal chance of selecting one of the values for each node attribute. For an attribute, when value 0 is assigned to a group, all the VM unit in the group can be placed in any nodes regarding the attribute.

It is important to note that on generating constraints for VMs, we tried to avoid a constraint setting in which the placement of all VMs are not feasible at all. For example, we ensure that the total CPU requirement of VMs that must be placed on nodes with a certain attribute is at most the total amount of CPU resources of the nodes that possess the attribute.
Metrics. For each simulation run, VM creation requests are arrived dynamically to the system based on the normal distribution, and the last VM creation is requested at 3,600 seconds, regardless of the total number of VMs. A created VM continues to execute its application iteratively, but once the last VM is created, it finishes the remaining execution and does not restart its execution again. The performance of each algorithm is evaluated over the whole period. Two different performance metrics are used depending on the optimization goal.

- *The number of active nodes* (N_{active}): The time-weighted average of the number of active nodes is measured for the ES algorithms.

- *Load imbalance* (σ_{load}): The time-weighted average of the standard deviation of CPU utilization is computed during the period from a time when 20% of the VM creations are requested to a time when 80% of them are requested for the LB algorithms. (In the excluded time periods, the load balancing of a cluster is not affected noticeably by different algorithms.)

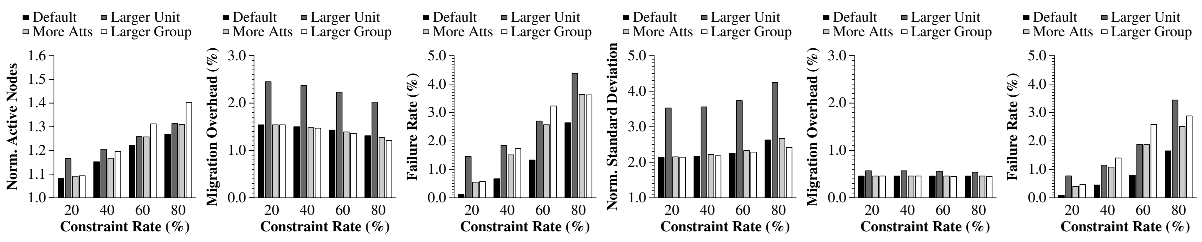
In addition, we measured the following two metrics for all the algorithms:

- *VM creation failure rate* (R_{fail}): The percentage of VMs that fail to be created, because there are no nodes satisfying their VM placement constraints and resource requirements.

- *Migration overhead* (T_{mig}): The percentage of the average of additional VM runtimes caused by VM migrations to the average runtime of all VMs in no constraint case.

The result of each simulation in this section is the average value over 200 simulation runs. For the spot migration algorithm, the threshold was experimentally selected, and for the periodic migration algorithm, the periodic dynamic migration of VMs were performed every 100 VM creations.

4.2 Effects of VM placement constraints



(a) N_{active}(ES) (b) T_{mig}(ES) (c) R_{fail}(ES) (d) σ_{load}(LB) (e) T_{mig}(LB) (f) R_{fail}(LB)

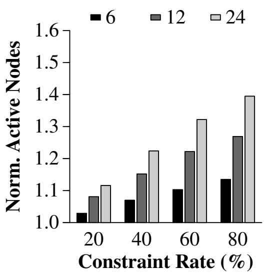
Figure 1: Effects of VM placement constraints on ES-basic and LB-basic, compared to no constraint case

We first investigate the effects of VM placement constraints using ES-basic and LB-basic algorithms. We compare their performance over various settings with that of no VM placement constraint case in which only resource requirements are considered. The default setting (*Default*) for the simulations is configured as N_{node}=256, N_{core}=12, M_u=500% (i.e., the total CPU resource requirement of a VM unit is at most 500% considering that one CPU core is 100%), M_g=64, and N_{att}=3 (thus, the number of nodes with the same attribute values is 32). For the VMtoVM-Must and VMtoVM-MustNot constraints, their effects become strong if there are many VM units with large size, and many VM groups with large size, respectively. Furthermore, the effects of the VMtoNode-Must constraint increase if there are many different node attributes, and so the number of nodes with the exactly same attribute values becomes small. To study how each type of the constraints affects the performance of VM placement, we also run ES and LB basic algorithms over the following three additional settings, *Larger Unit*, *Larger Group*, and *More Attributes*, which use the same setting as *Default* except M_u=1000%, M_g=128, and N_{att}=4, respectively.

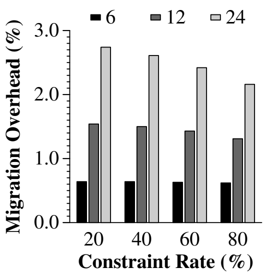
Figure 1 shows the performance of ES and LB basic algorithms, in which the x-axis is the rate of the VMtoVM-MustNot and VMtoNode-Must constraints i.e. $CR_{VtoVMustNot}$ and $CR_{VtoNMust}$ In Figure 1(a) and (d), the number of active nodes (N_{active}) of ES-basic and load imbalance (σ_{load}) of LB-basic are normalized to those with no constraints in the same setting of the VMs and nodes, respectively. Note that in all of the settings, without any constraints, the VM migrations and VM creation failures never occurred.

For ES-basic algorithm, as $CR_{VtoVMustNot}$ and $CR_{VtoNMust}$ increase, the normalized N_{active} tends to increase, while T_{mig} tends to decrease, since more nodes become active and so for a new

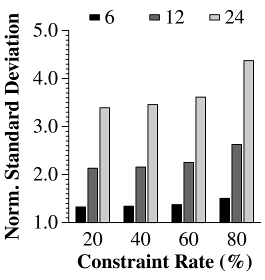
VM, the chance for its VM unit to migrate to some other node for satisfying the resource requirement reduces. For *Larger Unit*, the average VM unit size is larger than the other settings, having higher VM migration overhead. For LB-basic algorithm, as $CR_{VtoVMustNot}$ and $CR_{VtoNMust}$ increase, the normalized σ_{load} increases slightly, and there is no effect on the migration overhead, since VMs are basically distributed over nodes for load balancing. The VMtoVM-Must constraint (i.e. a large VM unit size) affects σ_{load} badly.



(a) N_{active}(ES)



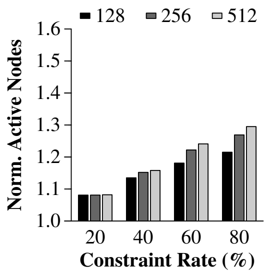
(b) T_{mig}(ES)



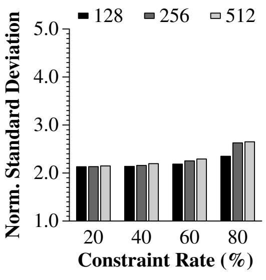
(c) σ_{load}(LB)

Figure 2: Effects of N_{core} on ES/LB-basic, compared to no constraint

The VMtoVM-Must constraint is bounded by the number of cores per node, N_{core}, since a VM unit size must be less than or equal to the total resources offered by a node. Figure 2 shows the performance on *Default* setting except varying N_{core}. Note that due to the space limitation, we only represent some of the results in the rest of the section. As N_{core} increases, we can have a VM unit which consists of a larger number of VMs with a larger resource requirement, and thus, the maximum VM unit size (M_u) increases such that the values of M_u for 6, 12 and 24 cores are 250%, 500% and 1000%, respectively. For ES-basic, with larger N_{core}, the normalized N_{active} increases, because the size of a VM unit becomes larger and also the effects of the VMtoVM-MustNot constraint become strong, and T_{mig} also increases. For LB-basic, a large VM unit size affects σ_{load} significantly, but T_{mig} slightly increases as N_{core} increases.



(a) N_{active}(ES)

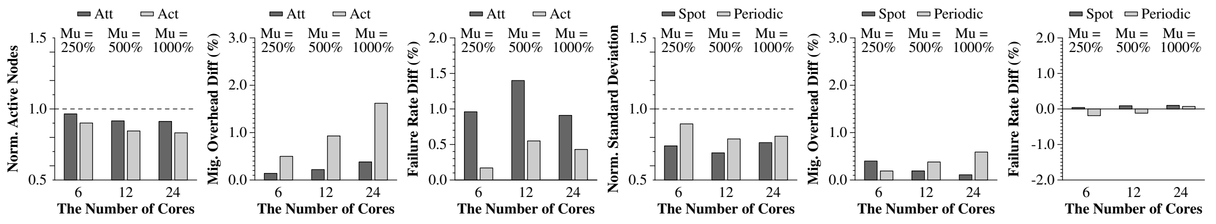


(b) σ_{load}(LB)

Figure 3: Effects of N_{node} on ES/LB-basic, compared to no constraint

The VMtoVM-MustNot constraint is bounded by the number of nodes, N_{node}, in a cluster, since the size of a VM group cannot be larger than N_{node}. For a cluster with a large number of nodes, the size of a VM group can be large. Figure 3 shows the performance on *Default* setting except varying N_{node}. In this figure, as N_{node} increases, the maximum VM group size (M_g) also increases as 32, 64 and 128, and the number of node attributes (N_{att}) also increases as 2, 3 and 4, for 128, 256 and 512 nodes, respectively. Thus, the number of nodes with the same attribute values is 32 in all cases. For ES-basic, as N_{node} increases, the normalized N_{active} increases. However, for LB-basic, the effect of the VMtoVM-MustNot constraint is not as strong as that of the VMtoVM-Must constraint (in Figure 2(c)) on σ_{load}. In both of ES/LB-basic, with a larger N_{node}, T_{mig} remains similarly, but R_{fail} increases, since N_{att} increases.

4.3 Performance comparison of the algorithms



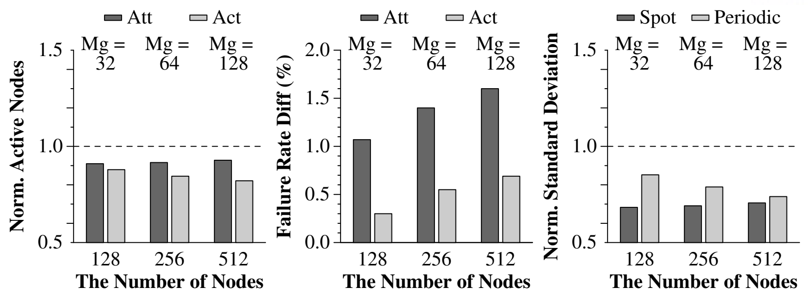
(a) N_{active}(ES) (b) T_{mig}(ES) (c) R_{fail}(ES) (d) σ_{load}(LB) (e) T_{mig}(LB) (f) R_{fail}(LB)

Figure 4: Effects of N_{core} on ES and LB, compared to ES/LB-basic

In this section, we compare the performance of the constraint-aware placement algorithms for the same goal with each other over various settings. Compared to ES-basic or LB-basic, the other ES or LB algorithms may have better performance on N_{active} or σ_{load}, but T_{mig} and R_{fail} may increase. Figure 4 shows the performance of Attribute-aware (Att-aware or Att), Activeness-aware (Act-aware or Act), Spot-migration (Spot-mig or Spot) and Periodic-migration (Periodic-mig or Periodic) algorithms on *Default* setting except varying N_{core}. In Figure 4 (a) and (d), N_{active} and σ_{load} are normalized to that with ES/LB-basic, respectively. For T_{mig} and R_{fail}, the difference from each algorithm to its basic algorithm is shown. Due to the space limitation, we only represent the results when both of $CR_{VtoVMustNot}$ and $CR_{VtoHMust}$ are 60%. The simulation results for the other rates show the similar trend as in the previous results.

In this figure, as N_{core} increases, M_u also increases as in Figure 2. For energy saving, Act-aware algorithm has the best performance regarding N_{active}, but it has higher migration overhead than Att-aware algorithm. Both of the algorithms have a higher VM creation failure rate than that of ES-basic, since they strategically attempt to use some subset of nodes to reduce the number of active nodes. Thus, the placement of VMs is possibly concentrated on those nodes, ending up the creation failure of some VMs that must be placed on them. For load balancing, Spot-mig algorithm generally shows the good performance for σ_{load} with reasonable migration overhead in all settings. For Spot-mig algorithm, with a larger VM unit, it becomes harder to perform spot migrations, decreasing T_{mig}. For Periodic-mig algorithm, if the periodic migration is done more frequently, imbalance can be lower with higher overhead.

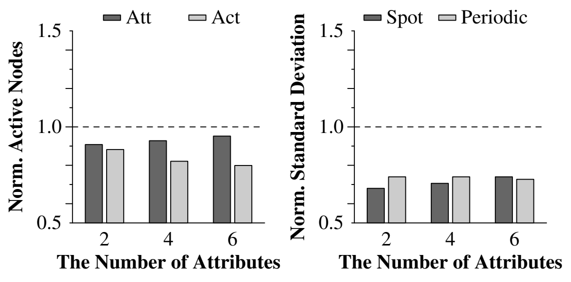
Figure 5 shows the performance of the ES and LB algorithms on *Default* setting except



(a) N_{active}(ES) (b) R_{fail}(ES) (c) σ_{load}(LB)

Figure 5: Effects of N_{node} on ES and LB, compared to ES/LB-basic

varying N_{node}. In this figure, as N_{node} increases, M_g also increases as in Figure 3. As N_{node} increases, N_{active} of Act-aware(ES) and σ_{load} of Periodic-mig(LB) decrease unlike the other algorithms. For both of the ES and LB algorithms, T_{mig} difference is not strongly affected by N_{node}. R_{fail} of the LB algorithms is very small as in Figure 4, and for Periodic-mig, it becomes lower than that of LB-basic when N_{node}=512.



(a) N_{active}(ES) (b) σ_{load}(LB)

Figure 6: Effects of N_{att} on ES and LB, compared to ES/LB-basic

Figure 6 shows the performance of the ES and LB algorithms over various numbers of node attributes with N_{node}=512 and N_{core}=8. In the simulations, only one attribute has two possible values of 1 and 2, while the other attributes have two possible values of 0 and 1. Act-aware(ES) algorithm provides good energy saving performance even with strong VMtoNode-Must constraints. For the ES algorithms, R_{fail} tends to increase as N_{att} increases. On the other hand, for load balancing, N_{att} has almost no effect on the performance of both of Spot-mig and Periodic-mig except that R_{fail} difference for Periodic-mig decreases a bit with a larger N_{att}. For both of the ES and LB algorithms, T_{mig} are similar over varying N_{att}.

4.4 Discussion

VM Rearrangement

We studied the effects of the VM rearrangement feature on the performance of all the VM placement algorithms using the default cluster of 256 nodes with 8 cores each. From the simulation results, we observe that without this feature, generally for all the algorithms, the failure rate increases, and the migration overhead decreases. For the ES algorithms, the effect on the normalized number of active nodes is very small. However, this feature is critical for Att-aware and Act-aware algorithms on reducing VM creation failure. Att-aware algorithm first places VMs on nodes which possess many node attributes, and then rearrange existing VMs to place a new one, if needed. Act-aware algorithm employs this feature in two places, one for rearranging VMs only within active nodes, and the other for rearranging VMs within all nodes in the cluster. Thus, without this feature, VM failure rates of Att-aware and Act-aware algorithms increase to 5.32% and 3.00% on average, respectively, from 1.46% and 0.99% with the feature.

For the LB algorithms, we observe that the effects of the VM rearrangement feature is very small on their performance. However, avoiding the VM creation failure may be very important and critical, even for a small number of VMs, and thus, this feature may be certainly worth for most of cases.

VM Group Placement

We investigate VM placement scenarios where VMs that belong to the same VM group are created at the same time. Such VM group placement is requested commonly for multi-tier or parallel applications. For the simulations, a cluster composed of 512 nodes with 8 cores per node is used, and four different node attributes are used. For a VM group, we compute the placement for every VM unit, and then place all the VMs in the group accordingly on the selected nodes at the same time.

With VM group placement, in general, the performance regarding energy saving or load balancing gets worse, compared to the individual VM placement, especially for the LB algorithms. This is because a VM unit whose size is larger than that of an individual VM needs to be placed at once. For the LB algorithms, the values of σ_{load} increase more than two times, compared to those with the individual VM placement. For all the algorithms, the migration overheads and VM creation failure rates decrease, because by computing a node for a VM unit at once, there is no need to migrate a part of already running VMs in the unit to another node for the creation of a new VM which belongs to the same unit.

Dynamic VM Migration

In this work, although we focus on the initial placement of VMs, our algorithms can be extended to dynamically migrate running VMs to improve the performance of a cluster. In fact, the periodic migration algorithm can be used to dynamically migrate running VMs for load balancing.

In the case of energy saving, the activeness-aware algorithm can be modified and invoked periodically. Instead of finding a node for a new VM, we can attempt to empty VMs out of a node. To do so, we first compute a set of nodes with low resource utilization based on some threshold and then sort the nodes in increasing order of the utilization. For a node n in that order, we check if we can migrate each VM unit u in node n to one of the nodes having the utilization higher than the given threshold. Similar to the activeness-aware algorithm for the initial VM placement, we may attempt to rearrange VMs running on the nodes with high utilization so that u can be migrated to one of them. If it is possible to vacate all the VM units in h, we migrate them to the newly selected nodes and then finally power off n. This process is repeated until the number of active nodes is reduced enough.

V Experiment Results on a Small Cluster

Table 4.2: Runtimes and overheads of SPEC CPU2006 applications

CPU-intensive			Memory-intensive		
Name	Runtime	Overhead	Name	Runtime	Overhead
gobmk	741.15s	9.45s	libquantum	599.08s	65.43s
hmmer	741.17s	2.20s	soplex	380.87s	42.13s
tonto	767.39s	1.80s	gcc	481.04s	27.93s
namd	653.86s	0.92s	lbm	581.88s	73.82s

We perform experiments using a small cluster which consists of four physical machines connected via 1GE switch. Each physical machine has two hexa-core Intel Xeon E5-2620 2.0GHz processors and 64GB memory. In a physical node, Xen hypervisor version 4.1.4 is installed. Each virtual machine is configured with one virtual CPU and 3GB memory. In an experiment scenario, we generate total 36 VMs, and we assume that the CPU requirement of a VM is 100% of one core. The last VM creation is requested at 6,300 seconds.

Table 4.2 shows the SPEC CPU2006 benchmark applications used in our experiments. The runtime and VM live migration overhead for each application are also given in the table. The VM live migration overhead is measured as the additional time to complete an application, when a VM running the application is migrated to another node during the execution. Note that the default VM live migration scheme of Xen which is based on the pre-copy migration [65] was used in our experiments. The performance of live migration varies depending on a different migration algorithm [66], but the migration overhead (i.e. T_{mig}) is proportional to the number of the migrations performed during the experimental run. Therefore, the performance trend will remain similar to that with different migration algorithms.

We run experiments for three different scenarios for each of the ES and LB algorithms. For each scenario, M_u is 500% (i.e., the number of VMs in each VM unit is randomly selected in the range from one to five), and M_g is 3 (i.e., the number of VM units in each group is randomly selected in the range from one to three). Also, the number of node attributes is 2 such that node

1 satisfies both attributes of $att1$ and $att2$, node 2 satisfies $att1$ only, node 3 satisfies $att2$ only and finally node 4 satisfies none. Thus, each VM can have zero, one or two VMtoNode-Must constraints. The VMtoNode-Must and VMtoVM-MustNot constraint rates are 60% (i.e., the numbers of VMs with the VMtoNode-Must and VMtoVM-MustNot constraints are around 21). For the scenarios used in our experiments, we have 12 VM units and 5.7 VM groups on average. Similar to our simulation, VM creation requests are arrived to the cluster based on the normal distribution.

We basically generate a scenario by randomly selecting one of the applications in Table 4.2 for each VM, and also randomly assigning its VM placement constraints and creation order. For scenarios 1 and 2, the same random scenarios are used for both of the ES and LB algorithms. For scenario 3, we modify the creation order of some VMs for a randomly created scenario, such that the effects of VM placement constraints become significant, and a different scenario is used for the ES and LB algorithms.

Table 4.3: Effects of VM placement constraints, compared to no constraint case

		Scenario 1	Scenario 2	Scenario 3
ES	N_{active}	1.613	1.615	1.642
	T_{mig}	0.68% (26.58s)	0.59% (21.75s)	0.83% (30.54s)
	R_{fail}	0.00%	0.00%	0.00%
LB	σ_{load}	2.136	2.396	15.701
	T_{mig}	0.11% (4.43s)	0.33% (12.36s)	0.49% (18.12s)
	R_{fail}	0.00%	0.00%	0.00%

Table 4.3 shows the performance of ES/LB-basic algorithms, compared to no constraint case. In general, the experiment results show similar trend as the simulation results. For ES-basic, N_{active} in the experiments is little larger than that in the simulations, since N_{node} in the cluster is small, and the used VM group size is larger than *larger Group* in Figure 1. For LB-basic, σ_{load} is similar to that in the simulations except scenario 3 that demonstrates the dramatic impact of VM placement constraints on the performance. Note that for energy saving, we cannot generate a scenario where the effect becomes exceptionally strong, because N_{node} is too small.

Table 4.4 shows the performance of the ES and LB algorithms on the real cluster. In the table, N_{active} and σ_{load} are normalized to those with ES/LB-basic, respectively, and "T_{mig} Diff" and "R_{fail} Diff" are the differences of T_{mig} and R_{fail} from each algorithm to its basic algorithm, respectively. Generally, the ES and LB algorithms show the similar performance to that in the simulations. In the results, the migration overhead of Act-aware and Att-aware algorithms becomes similar in the cluster composed of a small number of nodes. There is one VM creation failure in scenarios 2 and 3 for the ES algorithms. The migration overhead of Spot-mig algorithm is larger than that of Periodic-mig algorithm in scenarios 1 and 3. In the simulations, we observe

Table 4.4: Performance of ES and LB algorithms, compared to ES/LB-basic

Scenario	Scenario 1		Scenario 2		Scenario 3	
ES Alg.	Att	Act	Att	Act	Att	Act
N_{active}	0.759	0.683	0.762	0.665	0.749	0.651
T_{mig} Diff	0.55% (21.59s)	0.44% (17.18s)	0.55% (20.52s)	0.92% (34.24s)	1.81% (66.37s)	1.59% (58.40s)
R_{fail} Diff	0.00%	0.00%	2.78%	2.78%	2.78%	2.78%
Scenario	Scenario 1		Scenario 2		Scenario 3	
LB Alg.	Spot	Periodic	Spot	Periodic	Spot	Periodic
σ_{load}	0.790	0.975	0.533	0.862	0.375	0.547
T_{mig} Diff	0.59% (23.32s)	0.13% (5.14s)	0.30% (10.98s)	0.50% (18.44s)	0.57% (20.81s)	0.16% (5.98s)
R_{fail} Diff	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

that usually T_{mig} of Periodic-mig is higher than that of Spot-mig, but as N_{node} decreases, the difference of T_{mig} between them becomes smaller.

VI Concluding Remarks

The proposed model of VM placement constraints can provide a clear view of complex constraints of VMs, which are imposed in different paths by several parties such users, administrators, and developers. From our results, we observed that the VMtoVM-Must constraint strongly affects the performance of ES/LB-basic algorithms (in terms of N_{active} and σ_{load}, respectively), while the VMtoVM-MustNot constraint affects the performance of ES-basic algorithm, but has almost no effects on that of LB-basic algorithm. We also concluded that Act-aware and Spot-mig algorithms have good performance for energy saving and load balancing, respectively, in most of settings.

Our work can be helpful for administrators of a virtualized system to understand the effects of various VM placement constraints and employ proper constraint-aware VM placement techniques that satisfy the performance requirement of a target system in terms of the optimization goal, VM creation failure rate, and VM migration overhead. The VM creation failure rate can be reduced, especially for the ES algorithms, if we enable more exhaustive search of possible VM rearrangement by increasing the length of a rearrangement sequence. However, the exhaustive search will increase the complexity of finding a sequence of VM rearrangement dramatically and also will cause higher VM migration overhead. Thus, for normal VMs, administrators can use a default length, which is small, but for VMs with high importance, they can increase the sequence length to place them successfully in the cluster.

In addition, for the effective cluster management for VMs with placement constraints, constraint analysis tools to predict the possible performance effect of having additional constraints and to rank the existing constraints in order of the negative performance effects can be investigated. Using those tools, administrators will be able to check the effect of placing a new VM with a certain set of constraints beforehand, and relax some constraints with strong effects if possible. In this way, administrators will be able to filter unnecessarily strong constraints of the VMs in the cluster.

Chapter 5

Interference-aware Resource Management Techniques for Deep Learning Applications

Graphics processing units (GPUs) have become an attractive resource for accelerating applications in various computation-intensive areas over the past few years. General-purpose computing on GPUs (GPGPU) is a widely used technique to accelerate applications by utilizing the massive parallel computing power of GPUs. GPGPU is applicable to various applications such as physical simulations, solving various types of equations, and machine learning. As GPGPU has become a trend, resource providers have also focused on it. A number of large-scale cloud providers and datacenter operators such as Amazon EC2 [67] offer GPU services. Three of the top five supercomputers in the Top500 List use GPUs [68], and the percentage of GPU-equipped systems is increasing.

One of the most important reasons for using GPUs for applications is that they have a large amount of computations that takes too long time to process using a CPU. Accelerating that computation using GPU can achieve much better performance, but it still takes too long for some applications. One of the best solutions to this problem is to use multiple GPUs similar to the existing parallel applications that use multiple CPUs across distributed servers. In recent years, some GPU applications have become parallel to utilize multiple GPUs in multiple servers, such as machine learning frameworks like TensorFlow [69].

Deep learning techniques were proposed a long time ago, but they have become emerging techniques in various domains because of the development of hardware and the related researches. Convolutional neural network (CNN) is one of the famous classes of deep neural networks that has been spotlighted in various fields such as image classification. There are many popular CNN models such as AlexNet [70], VGG [71], and ResNet [72]. We have to use multiple GPUs to train these CNN models in parallel because of their huge computational cost.

Solving the under-utilization problem of the resource is a well-known challenge, and GPUs have a similar issue. One of the characteristics of GPU applications is that they cannot be executed only using GPUs. GPU applications also have to use the CPU in many cases such as accessing data and communicating with other processes. This means that GPU applications have to use the CPU in some portion of their runtime and the GPU will remain idle in this portion. Therefore, GPU is usually under-utilized when executing GPU applications.

Co-locating multiple CNN models can improve the utilization of GPU. However, when the multiple CNN models are executed in the same GPU concurrently, their performance is degraded by the interference with each other. There are various factors that cause the interference between parallel GPU applications, because they have to use resources from both the device- and the host-side. The effect of interference can vary depending on the characteristics and the resource requirement of the applications. When co-locating parallel GPU applications, the scheduler has to consider the effect of interference from the applications.

There are some prior works to schedule GPU applications while considering the interference [73–76]. However, they mainly consider single-node applications [73, 74, 76]. A colocation-aware job scheduler considering the host-side interference is proposed, but it focuss on the performance of CPU applications and the performance of GPU applications was not considered [75]. An integer programming-based scheduler for the SLURM resource manager to improve the utilization of the GPU cluster is proposed [77].

In this work, we propose a performance prediction model considering the interference between the distributed parallel CNN models in TensorFlow. We first analyze the effect of interference to show why this interference should be considered. We then propose our kernel-split technique for mitigating the slowdown of CNN models. We also propose a performance prediction model for the interference that uses a GPU pattern simulator to estimate the slowdown of the distributed parallel CNN models because of the interference with low cost. We also evaluate our heuristic algorithm by using this model to show that we can schedule the distributed parallel CNN models efficiently with two goals - resource harvesting and maximizing throughput.

I Effect of interference

In this section, we investigate the factors that cause interference on both the device- and the host-side, and how their effect on the performance of parallel GPU applications.

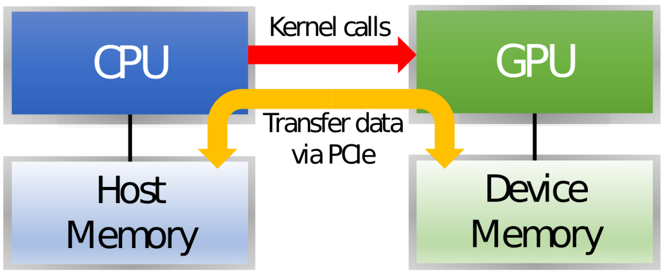


Figure 1: Simplified architecture of GPU server

Table 5.1: List of distributed parallel CNN models and their characteristics

Dataset	Name	# Parameters	Batch size	GPU util (%)	Network BW (MB/s)
Cifar10	Alexnet	1.9M	128	26.46	152.12
	Resnet32	0.5M	128	45.74	40.84
	Trivial	4.1M	32	8.04	232.06
Imagenet	Googlenet	7.0M	32	71.39	170.75
	Nasnet	5.3M	32	72.18	69.48
	Resnet50	25.6M	64	85.31	233.47
	Resnet152	60.2M	32	81.62	360.54
	VGG16	138.4M	32	73.47	385.65

Figure 1 shows the architecture of the GPU server briefly. GPU has many processing elements (PE) like the cores in the CPU, and it has a memory called the device memory. When a GPU application is executed, the data for computation should be copied from the host memory to the device memory. The application submits functions called kernel for computation to the GPU. After the computation is completed, the result is stored in the device memory, but it should to be copied to the host memory. CPU and GPU have a PCIe connection; therefore, the data are transferred via the PCIe bus.

On the device-side, interference may occurr from the processing elements and the memory bandwidth. When the multiple GPU applications are executed on the same GPU, kernels are submitted in the GPU. The processing elements can be used for only one kernel at once; therefore, the other kernels need to wait for the completion of the currently running kernel, and their waiting time becomes slowdown of the application. To run the kernel, the applications have to transfer data from the host memory to the device memory through PCIe. The result data of the kernel is stored in device memory, so the data can be transferred from the device memory to the host memory. However, if multiple applications transfer data in the same direction, they have to share te limited bandwidth, thereby degrading the performance of data transfer.

On the host-side, interference can occur from the host memory and I/O. To send data to the device memory, the applications have to store data to the host memory; therefore, they have to share memory. For the parallel GPU applications, they have to communicate periodically; therefore, they have to use the network devices. Moreover, it is not possible that transfer data in the device memory to other machines without the special device; therefore, data should be copied from the device memory to the host memory for network communication. Therefore, if the multiple parallel GPU applications are co-located, they have to share the memory and network devices.

To investigate the effect of interference on both the device- and the host-side, we perform experiments with some distributed parallel CNN models. Table 5.1 presents the distributed parallel CNN models that we used and their characteristics when they were executed in a 4-

node GPU cluster. We use three CNN models with the Cifar10 dataset [78], and five CNN models with the Imagenet dataset [79] from the TensorFlow benchmark [80]. We used the default batch size in the benchmark, and computed the number of parameters from the code. Network BW means the average bandwidth of the network I/O, and it is a sum of send and receive. Each CNN model uses four nodes, and they have four parameter servers and workers with one for each node.

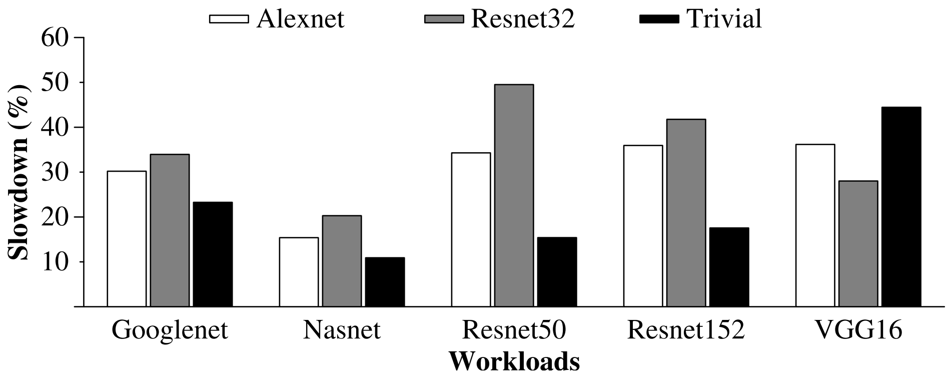


Figure 2: Slowdown of Cifar10 models when co-located with sharing GPU

Figure 2 shows the performance effect of the Cifar10 models when they were co-located with the Imagenet models in the same nodes. In these experiments, the CNN models shared the same GPU and the same CPU socket in each node. We computed the slowdown as the additional execution time of the target model because of the co-location with the other models divided by the original execution time. The performance of the Resnet32 model was more considerably affected than that of the other models when co-located with most of the Imagenet models, and the performance of the Trivial model was less affected than that of the others. However, when the Cifar10 models were co-located with the VGG16 model, they showed a different pattern with the others. Because the Imagenet models except VGG16 showed high GPU utilization, the performance of the Resnet32 model that had higher GPU utilization than the others was more affected than that of the others when co-located with them. In contrast, VGG16 model shows larger parameter size than the other Imagenet models, so the main reason of interference was the network I/O contention, and the performance of Trivial model was more considerably affected than that of the other Cifar10 models.

From this result, we observed that sharing A GPU can affect the performance of GPU applications badly. Moreover, this result shows that the effect of interference cannot be predicted easily. In some co-location cases, the performance of THE model is degraded even the sum of GPU utilization of two models are below 100%. Interference could occur on both the device- and the host-side, and the interference from the host-side might have a more considerable effect than that from the device-side in some cases such as those in which the Cifar10 models are co-located with the VGG16 model.

Figure 3 shows the performance effect of the Cifar10 models when they were co-located

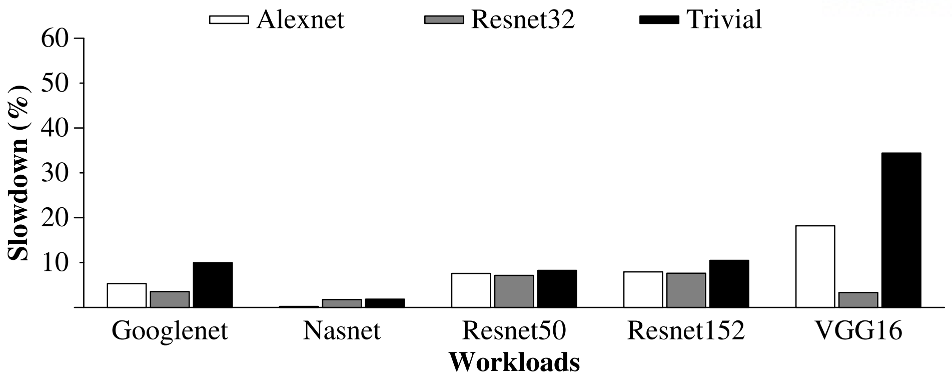


Figure 3: Slowdown of applications when co-located with sharing CPU

with the Imagenet models in the same nodes but used different GPUs. The models shared the same CPU socket in each node, but they did not share the GPU. Therefore, we can observe the effect of interference from only the host-side resources such as the host memory and the network I/O. The Alexnet and Resnet32 models showed a low slowdown when co-located with the Imagenet models except VGG16, because in these co-location pairs, the interference from the host-side resources was not critical. However, the Trivial model showed higher slowdown than other Cifar10 models, and the performance of this model was significantly affected when it was co-located with VGG16. The performance of the other Cifar10 models was also highly affected when they were co-located with VGG16. The CNN model with a large parameter size such as VGG16 required considerable large network communication between the nodes because of the parameter update; therefore, the co-location with VGG16 might have caused interference from the network I/O even when the models did not share the GPU. The Trivial model has highest parameter size among the Cifar10 models, so it is more considerably affected from co-location with the Imagenet models than other Cifar10 models, but the performance of the Resnet32 model, which has a small parameter size was almost the same as the default even VGG16 model shared the same CPU socket and network device in the same nodes.

II Techniques for managing interference

In this section, we introduce our techniques for managing the interference from the co-location of the CNN models. The first technique, called kernel-split, could mitigate the performance degradation of one CNN model by splitting kernels of other CNN models sharing the same GPUs. The second technique, called the GPU usage pattern simulator, could estimate the performance of the CNN models when they are co-located in the same nodes and sharing the same GPUs by simulating their GPU usage pattern. Our techniques focus on the device-side interference because the host-side interference showed low effect except a few cases.

2.1 GPU usage pattern

Our techniques are based on the characteristics of the GPU usage pattern, so we will explain this concept first. As we mentioned in Section I, the GPU usage pattern of GPU applications is of two types. GPU applications use two types of functions, namely kernel and memcpy. A kernel function deals with computations that have to processed on the GPU processing elements, and a memcpy function handles the data transfer between the host memory and the device memory.

In general, the GPU can only process one kernel at once. There are some techniques for using two or more kernels together, such as MPS [81], but they have certain limitations such as the difficulty of managing the sharing of processes with scheduling decisions. When two or more GPU applications share the same GPU, the kernel of an application has to wait until the kernel of another application is completed.

Unlike kernel functions, multiple memcpy functions can be simultaneously processed by a GPU. However, the data transfer consumes the memory bandwidth and the PCIe bandwidth. Therefore, processing multiple memcpy functions in the same direction can create a contention for the memory and the PCIe bandwidth, and their performance can be degraded because of the sharing of the limited bandwidth.

2.2 Kernel Split

We propose kernel-split technique to mitigate the performance degradation of a GPU application sharing the GPU with other applications. This technique focuses on kernel functions. As we mentioned in Section 2.1, the kernel of an application has to wait until the kernel of another application is completed when two or more GPU applications share the same GPU. If the execution time of a kernel is long, the other kernel has to wait for a long period of time. Our kernel-split technique slices the long kernel into multiple sub-kernels by executing the kernels repeatedly with each part of the divided data. The CNN models use one batch for a single iteration. A batch contains images as numbers as the batch size. We split this batch to sub-batch and execute multiple kernels with each sub-batch. We define the value "split factor" as the number of sub-kernels to split each kernel for the GPU applications. If the split factor of the GPU application is 1, kernel-split is not applied to it.

Figure 4 shows the example of the kernel split. There are two GPU applications, A and B, sharing the same GPU. The graph shows when their kernels were started for the GPU processing, and finished. In the original case, the execution time of the first kernel for B was 12s, so the second kernel for A had to wait 12s until the first kernel is completed. When the kernel-split technique (with spilt factor of 6) was applied on GPU application B, its kernel was split into six sub-kernels. In this case, the second kernel of A could be executed only after the first sub-kernel of B was completed, and its waiting time was reduced to only 2s.

Unfortunately, splitting the kernel resulted in an overhead, and the performance of the GPU application degraded when this technique is applied. However, this technique could mitigate the

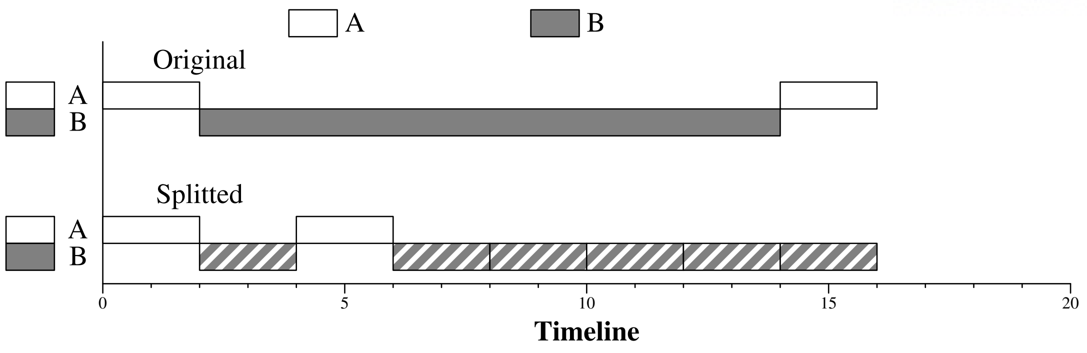


Figure 4: Kernel Split

slowdown of other CNN models that are co-located with it. Therefore, our kernel-split technique could be used to guarantee the performance of a high-priority job or to meet the QoS limit while co-locating other jobs to increase the GPU utilization.

2.3 GPU usage pattern simulator

We propose a GPU usage pattern simulator [82] to predict the performance of parallel GPU applications when they are co-located in the same nodes and share the same GPU. Our simulator needs the information of the GPU usage pattern of the applications as the input, and simulates the behavior of their kernel and memcpy functions on the basis of their characteristics to compute the execution times and the slowdown of the applications. The GPU usage pattern of the GPU applications could be profiled by using the nvprof [83] tool that records information about every kernel and memcpy function call during the execution of the GPU applications. From the profiled data, we used the start time and the duration for each kernel and memcpy call, and the PCIe bandwidth for only the memcpy calls.

One of the important characteristics of the CNN models is that their training process is iterative. Therefore, we do not have to profile the entire execution of the model, and we can profile only some of the iterations to simulate their performance accurately. Because the GPU usage patterns of the applications are different, the number of kernel and memcpy calls of applications may differ; the iteration time of applications may also differ. Therefore, we make the simulator to repeat the behavior of applications as much as we want to simulate their behavior when they are co-located more accurately.

We use a different method to simulate the kernel calls and the memcpy calls. In case of the kernel calls, they cannot be executed concurrently. Therefore, if there are multiple kernels submitted to the GPU, many of them have to wait for the others. The simulator applies this behavior; therefore, the kernels have to wait for the completion of the other kernels. In fact, there is no information on how kernels of multiple applications are scheduled in a real GPU, so we used the FCFS policy for our simulation. Even when the start time of the kernel is delayed, the interval between the calls does not change. Therefore, if the kernel waits, the entire runtime is increased.

In the case of the memcpy calls, they can be executed concurrently but have to share the limited PCIe bandwidth. If a memcpy is submitted and there is no running memcpy, it is just scheduled. However, if a memcpy is submitted when there is one or more memcpy functions currently running, their duration is changed after scheduling the new memcpy. New durations for the running memcpy functions are calculated on the basis of their PCIe bandwidth and maximum bandwidth. We fixed the maximum bandwidth as the theoretical maximum bandwidth (15.75GB/s). The durations of the running memcpy functions were also changed when one of them was completed.

III Interference-aware resource management system

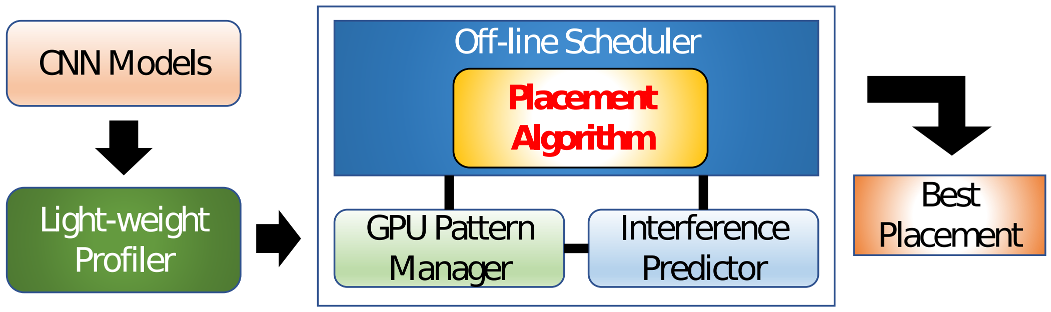


Figure 5: Interference-aware resource management system for GPU cluster

In this section, we present the details of our interference-aware resource management system. Figure 5 shows an overview of our system.

3.1 Off-line profiling

When we have CNN models to schedule, we profile them first to get their GPU pattern data. Our GPU pattern simulator can estimate the performance of the models with the data obtained by profiling only a small number of iterations. We profiled the models with 50 iterations. We used the information about the kernel and memcpy function calls executed in the GPU. We can get the detailed information about every kernel and memcpy call of the CNN models from the start to the finish by using nvprof [83].

Some CNN models can be the target of the kernel-split technique, and their GPU usage pattern will change. We do not have to profile the models for each possible split factor, because there are rules for changing split factor. To estimate the performance of the models when the kernel-split technique is applied to them, we can profile only two runs to cover with any split factor. We profile once without applying the kernel-split, and once with the application of the kernel split. When we apply the kernel-split for profiling, We split the kernels of the models into two sub-kernels.

3.2 GPU pattern manager

The GPU pattern manager generates the input for the interference predictor (GPU usage pattern simulator). First, it parses the profile data to get important metrics such as start time, duration, bandwidth, and data size. Then, it analyzes the data to determine the changes in the GPU usage pattern from the kernel-split. Finally it generates the input data with the target split factor (including 1) given by the off-line scheduler.

Change in GPU usage pattern from kernel-split

In this section, we explain the prediction of the GPU usage pattern with any split factor. There are some rules about the change in the GPU usage pattern. From off-line profiling, we can get two cases of the GPU usage pattern with the kernel-split and without the kernel-split. Comparing two cases using the following rules, we can check which part of the GPU usage pattern is changed and how it changed because of the kernel split.

For the kernel functions, there are two types of changes. Most of the kernels are split as many sub-kernels, so it will be called as many as split factor, and execution time of each call will shorten. In this case, we can use simple regression to predict the execution time of kernels when they are split. The manager marks these kernels to define them as the splitting kernels. Some kernels may not split even when the kernel-split technique is applied to the model. In this case, they are not changed, so we can just use their information. We can check the names of the kernels to compare them between the two data. However, there are some cases in which the names of kernels can be changed, because the library may use a different kernel when the size or shape of the data is changed. In this case, we use the pairwise kernels from each data with similar names, and treat them as the same kernel.

Even though our technique deals with splitting kernels, memcpy functions also can be split because they are closely related to the kernel functions. The memcpy functions have the same name, so we check the size of the data to determine which functions to compare. After this, we can deal with memcpy calls in a manner similar to that used for dealing with the kernel calls. If the size of the data of the memcpy functions is divided by the split factor, we use simple regression to predict the execution time, and they are marked as split. If the size of the data is the same as that in the case of memcpy functions, we can just use this information.

Generating input data for simulator

In this section, we explain how to generate the input data for the GPU usage pattern simulator. When the off-line scheduler needs to check the slowdown of the CNN models when they are co-located, it requests the information from the interference predictor. At this time, the scheduler also requests the input data for the simulator from the GPU pattern manager. When it sends a request, it also gives the split factor for each model. Therefore, the GPU pattern manager has to generate the input data for each CNN model with the given split factor.

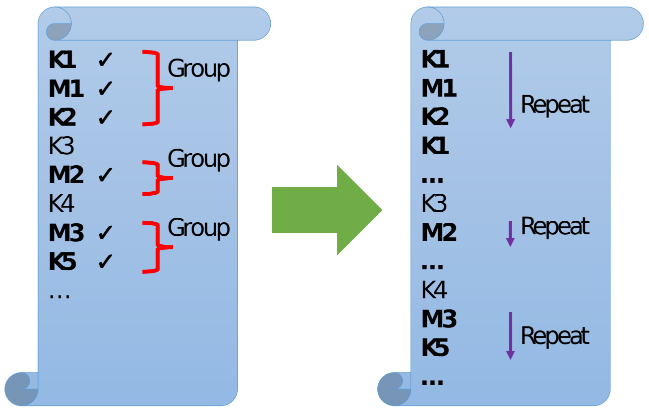


Figure 6: Generating GPU usage pattern

As explained in section 3.2, we can predict the GPU usage pattern with any split factor. From comparing two profiled GPU usage patterns, we can determine which kernel or memcpy call is split and which is not, and how the execution time of the kernel or memcpy calls changes when they are split. Figure 6 shows how to generate the GPU usage pattern with the given split factor. On the basis of the GPU usage pattern without the kernel split, the manager follows the function calls from the start to the end, and groups the function calls that are marked. Thereafter, the manager computes the execution time of the kernel and memcpy calls that are in any group on the basis of the given split factor and regression result of each call. Finally, the manager repeats the call of the groups according to the given split factor.

3.3 Interference predictor

To predict the interference between the CNN models, the interference predictor uses the GPU usage pattern simulator. When the interference predictor gets a request to predict the interference between the CNN models from the off-line scheduler, it waits for the GPU pattern manager to get the input data of the models. When the GPU pattern manager generates the input data for the CNN models, the interference predictor gives them to the GPU usage pattern simulator to estimate the slowdown of each CNN model. Then, the interference predictor sends the result of the simulator to the scheduler.

3.4 Off-line scheduler

The off-line scheduler can try to find the best placement for the CNN models with the given goal. From other modules, we can predict how the performance of CNN models changes when they are co-located. The placement algorithm of our scheduler can use this information to find which CNN models have to be co-located for efficiency.

Performance goals and metrics

We focused on two of the popularly used goals for resource management, namely guaranteeing Qthe uality of Service, and maximizing the throughput.

Guaranteeing Quality of Service The placement scheduler needs to guarantee the performance of latency-sensitive applications. As the QoS threshold, we used the slowdown of the CNN model. For example, if the QoS threshold is 10%, the slowdown of the latency-sensitive CNN models should be less than 10%.

Maximizing throughput The placement scheduler needs to maximize the throughput of the applications, which implies using the resource efficiently. In general, the throughput of the application can be computed as the reciprocal of the execution time. However, CNN models may have different execution time, so we defined *normalized throughput* (Original execution time / actual execution time). For example, if the slowdown of a model is 100%, the normalized throughput is 0.5.

Min-max pairing (MMP) algorithm

We propose a heuristic placement algorithm called the Min-Max Pairing(MMP). Our algorithm pairs the CNN models, and co-locates the CNN models in the same pair to the same resources. To minimize the slowdown of the CNN models, we have to pair CNN models with a low slowdown when they are co-located. From the interference predictor, we can get the performance effect of every possible pair of CNN models. Therefore, we can pair the models on the basis of this information. Our MMP algorithm has two parts, namely generating a matrix and pairing the models.

Generating Matrix Before pairing the CNN models, MMP generates a matrix that has the information of every possible pair of models. Let the number of latency-sensitive CNN models be N, and the number of batch CNN models be M. We assume that M is equal to or greater than N. Then, the size of the matrix will be $N * M$. For each pair, the GPU pattern manager generates one input for the latency-sensitive CNN models to which the kernel-split is not applied, and many inputs for the batch CNN models with various split factors. Then, the interference predictor predicts the slowdown of the CNN models with various split factors for the batch CNN model. From the result, the algorithm selects the case in which the slowdown of latency-sensitive CNN model is less than the QoS threshold, and places the slowdown of the batch CNN model in this case into the matrix. If there is no case in which the slowdown of the latency-sensitive CNN model is less than the QoS threshold, the algorithm leaves the element of the matrix for this pair empty. If there are two or more cases in which the QoS threshold can be satisfied, algorithm selects the minimum slowdown of the batch CNN model.

Min-Max Pairing After the matrix generation, our MMP algorithm chooses appropriate pairs to co-locate. First, it removes the completely empty rows or columns. If a row of the matrix is empty, there is no way to guarantee the QoS of the latency-sensitive CNN model while

Algorithm 2 Min-Max Pairing Algorithm

```
1: input: $M = m_1 1, m_1 2, ...m_2 1, ..., m_N M$: slowdown matrix
2:      $A = a_1, ...a_N$: latency-sensitive CNN models
3:      $B = b_1, ...b_M$: number of batch CNN models
4: $x, y = N, M$
5: Remove empty columns of $M$ and decrease $y$ by the number of empty columns
6: for each row $R_i$ in $M$ do
7:    if $R_i$ is empty then
8:       Schedule $a_i$ alone
9:       Remove $R_i$ and decrease $x$ by 1
10:   end if
11: end for
12: while $x > 0$ do
13:    for each row $R_i$ in $M$ do
14:       if $R_i$ has only one element then
15:          $j$ = the index of element in $R_i$
16:          Schedule $a_i$ with $b_j$, decrease $x$ and $y$ by 1
17:          Remove all elements in row $i$ and column $j$ in $M$
18:          break
19:       end if
20:    end for
21:    for each column $C_j$ in $M$ do
22:       if $C_j$ has only one element and $x == y$ then
23:          $i$ = the index of element in $C_j$
24:          Schedule $a_i$ with $b_j$, decrease $x$ and $y$ by 1
25:          Remove all elements in row $i$ and column $j$ in $M$
26:          break
27:       end if
28:    end for
29:    Remove the maximum element $m_i j$ in $M$
30:    if Column $R_i$ is empty then
31:       Decrease $y$ by 1
32:    end if
33: end while
```

co-locating any batch CNN models; therefore, this model will be scheduled alone. If a column of the matrix is empty, co-locating the batch CNN model makes it impossible to guarantee the QoS of any latency-sensitive CNN model; therefore, this model will not be used. Then, it finds the row that has only one element left. If it exists, there is only way left to schedule the latency-

Table 5.2: The configurations of GPU cluster

Resource	Specification
CPU	Intel(R) Xeon(R) octa-core CPU E5-2640 v4 @ 2.60GHz * 2
GPU	Tesla K80 (Kepler) * 2
Memory	32GB for CPU, 12GB for each GPU
# nodes	4
Network	Infiniband

sensitive CNN model. Therefore, the algorithm selects the pair and removes all the elements in the same row and the same column from the matrix. Thereafter, the algorithm finds the column that has only one element left. If it exists, there is only way left to schedule the batch CNN model. However, if there is a sufficient number of remaining batch CNN models, we do not have to use this model. Therefore, it checks the number of remaining batch and latency-sensitive CNN models. If the number of remaining models is the same, the algorithm selects the pair, and removes all the elements in the same row and the same column from the matrix. Next, the algorithm finds the maximum slowdown in the matrix and removes that element, and repeats the procedure from finding a row. Algorithm 3.4 shows the second part of this algorithm.

3.5 Implementation

We modified TensorFlow [69] to implement our kernel-split technique. Our original plan for implementing our technique was to modify the code right before calling the kernel functions, but it was difficult to find the point to modify because many of the GPU kernel functions used in the TensorFlow model execution were in libraries. Therefore, we modified the code for generating the executors from the run function of the TensorFlow session (i.e. session.run()). We divided the tensor that contained the input data by the split factor and called an executor for each divided input tensor. We did not have to split a short kernel, so we checked whether the input tensor existed and the size of the input if it existed, and we divided only the input tensors with larger than 1M. After the execution of the executors was completed, we added the code for gathering results.

We implemented the components of our system, such as the GPU usage pattern simulator and the GPU pattern manager in C++.

IV Evaluation

4.1 Experimental Setup

Table 5.2 shows the configurations of our GPU cluster. In the cluster, each node has two sockets with eight cores each. Every node in the cluster is connected via Infiniband. We installed Ubuntu 18.04 with Linux kernel version 4.15.0 for each node, and CUDA 10.0 is installed for GPU. We modified TensorFlow 1.13 to implement the kernel-split technique. We implemented the GPU pattern usage simulator using C++. Every experiment in this work was carried out using this cluster.

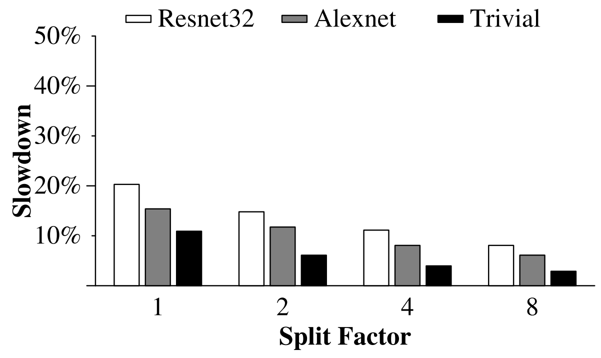
For each node, two GPUs are connected with the first CPU socket via PCIe. If the GPU application is pinned to the second socket, it has to communicate with the GPUs via not only the PCIe but also the QPI between sockets. We tested the performance effect of this architecture, but there was no significant difference was observed from that observed using QPI. We always pin CNN models to the first CPU socket in each node. When the CNN models are co-located, they use same nodes, and they share the same socket and the same GPU in each node.

For the applications, we use eight CNN models as described in Table 5.1. We used the code in the TensorFlow benchmark [80] for all of the CNN models. We used the Cifar10 models as the latency-sensitive CNN models, and the Imagenet models as batch CNN models. We paired only one Cifar10 model with one Imagenet model. The Imagenet models consumed a considerably large amount of device memory, so it is impossible to co-locate any Imagenet model pair in the same GPU. We also didn't co-locate two Cifar10 models.

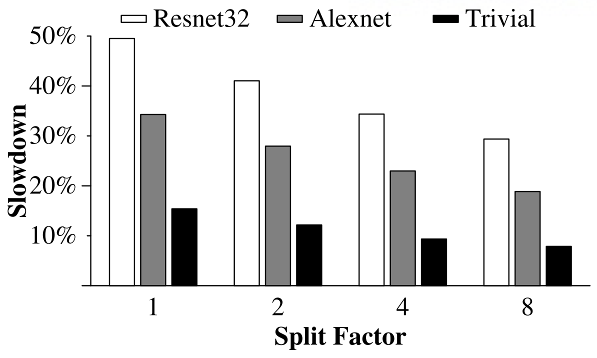
4.2 Effect of Kernel Split

We run experiments of co-locating CNN models without using the kernel-split techniques (split factor = 1). Then, we applied the kernel-split technique to the batch CNN models while increasing the split factor to 2, 4, and 8 to observe the effect of our technique. The split factor could be any integer up to the batch size of the CNN model, but increasing the split factor resulted in an additional overhead for the batch CNN models.

Figure 7 shows the slowdown of the Cifar10 models co-located with (a) Nasnet and (b) Resnet50 when the split factor of the Imagenet models was increased from 1 to 8. We observed that increasing the split factor of the Imagenet models could mitigate the performance degradation of the co-located Cifar10 models. The use of the other Imagenet models also showed similar results: the slowdown of the Cifar10 models decreased when the split factor was increased. Therefore, we just present the results for two models that have the highest average slowdown (Resnet50) and the lowest average slowdown (Nasnet). From all the possible pairs without the kernel-split, the Resnet32 model showed 36.38% slowdown on average, but that with kernel-split with a split factor of 8 exhibited a slowdown decreased to 22.07%. The average slowdown of Alexnet was 28.95% when the kernel-split technique is not applied, and it decreased to 12.56%



(a) With Nasnet



(b) With Resnet50

Figure 7: Effects of Kernel Split

Table 5.3: Validation result of performance prediction model

# Split	Alexnet	Resnet32	Trivial	Average
1	6.81%	8.21%	6.15%	7.06%
2	5.64%	7.31%	5.49%	6.15%
4	4.68%	7.06%	4.93%	5.56%
8	4.27%	6.58%	4.44%	5.09%
Average	5.35%	7.29%	5.25%	5.96%

with a split factor of 8. The average slowdown of Trivial was decreased from 16.78% to 7.09%.

Splitting the kernel to half showed highest effect at the first time, but the effect of splitting the kernel further worsened. Because doubling the split factor implied decreasing the kernel length to half, the kernel length decreased exponentially. In contrast, the performance of the Imagenet models was degraded by splitting their kernels. Their slowdown was almost directly proportional to the number of additional kernels (i.e. split factor - 1), and the slowdown from the kernel-split was 13 20% per kernel. The slowdown of the Imagenet model was not bad when the split factor was low, but it became significant in the case of a high split factor. To co-locate CNN models efficiently, we need to find the appropriate split factor to reduce the overhead of the batch models while guaranteeing the QoS of the latency-sensitive models.

4.3 Accuracy of prediction model

In this section, we evaluate the accuracy of the GPU usage pattern simulator. We compare the result in section 4.2 and the result of the simulator to observe how well our simulator predicts the performance of the CNN models. As mentioned in section 3.1, we use the off-line profiled data to predict.

Table 5.3 shows the error rates of our GPU usage pattern simulator. The error rate in each case is computed as the difference between the real slowdown and the predicted slowdown, and each value is the average of the five error rates from the slowdown of the Cifar10 model co-located with each Imagenet model. We evaluated the accuracy of our simulator from the total 60 cases (15 pairs * 4 split factors), and the average error rate was 5.73%. Moreover, the maximum error rate of our simulator was 9.73%, which implied that our simulator also showed high accuracy for every single case. As mentioned in section 3.1, we used two input data items and a split factor of 2 for input with the kernel split. We also use other values for the split factor of 4 and 8 and evaluate the accuracy. When we used the input with split factor of 4, the average error rate was 5.96%. Furthermore, it became 6.32% when we use the split factor of 8. Our GPU usage pattern simulator also showed high accuracy when we use a different split factor for the input.

As mentioned in section II, we considered only the device-side interference, because the host-side interference showed a low effect except in a few cases in our setup. In our environment, the host-side interference was not the main reason for the performance degradation except some pairs, including VGG16, and it had only a small effect for many cases. However, we already observed that the Cifar10 models had different patterns for the slowdown when they were co-located in the VGG16 model in section I. Further, our GPU usage pattern simulator showed a higher overhead when it predicted the performance of the Cifar10 models when they were co-located with the VGG16 model. Therefore, we predicted the performance of the CNN models when they were co-located while considering the effect of both the device- and the host-side interference.

Our GPU usage pattern simulator could not simulate the host-side interference, but it could reflect the slowdown from the host-side interference while simulating the device-side interference. In the GPU usage pattern, there are idle times when the CNN models use the CPU-side resources. Our simulator get the slowdown of the CNN models from the host-side interference as the input, and increase the idle times to represent the lengthened time for host-side processing. We measure the slowdown of the CNN models from only the host-side interference by using different GPUs while the sharing the same nodes and the same sockets. Then, we provide the measured slowdown to the simulator as the input, and predicted the slowdown of the CNN models. The average error rate of the simulator was decreased from 5.96 to 5.11%.

4.4 Off-line Scheduling

Methodology

We experimentally evaluated the performance of our scheduling algorithm. Our min-max pairing (MMP) algorithm use the GPU pattern manager and the interference predictor that contained the GPU pattern simulator as explained in section II to find the best placement for the CNN models and schedule them. The scheduler focused on guaranteeing the QoS threshold for the latency-sensitive CNN models, and the batch CNN models could be scheduled according to the scheduling decision. We compared our algorithm with the following heuristics:

- Random pairing (Random): This heuristic randomly pairs up each latency-sensitive CNN model with the random batch CNN model. It can apply the kernel-split to the batch CNN model to reduce the slowdown of the latency-sensitive CNN model. If the slowdown of the CNN model is lower than QoS threshold, the CNN models in the pair will be co-located. If not, the latency-sensitive CNN model will be scheduled alone.

- MMP without split (W/o Split): This algorithm is the same as our min-max pairing except that it cannot use the kernel-split.

- Greedy placement (Greedy): This heuristic also uses the matrix and generates the matrix in the same manner as MMP. Then, it chooses the pair with the lowest slowdown.

We used three QoS thresholds, namely 10%, 20%, and 30%, to evaluate the scheduling algorithms. The split factor for the kernel split is limited to 8 for our algorithm and the other heuristics because using a higher split factor makes the batch CNN models very slow. Our GPU cluster is too small to execute all the CNN models simultaneously, so we execute the models in a pair (or only one model) simultaneously, and executed the models in the next pair after the execution of the earlier models was completed.

Experimental Results

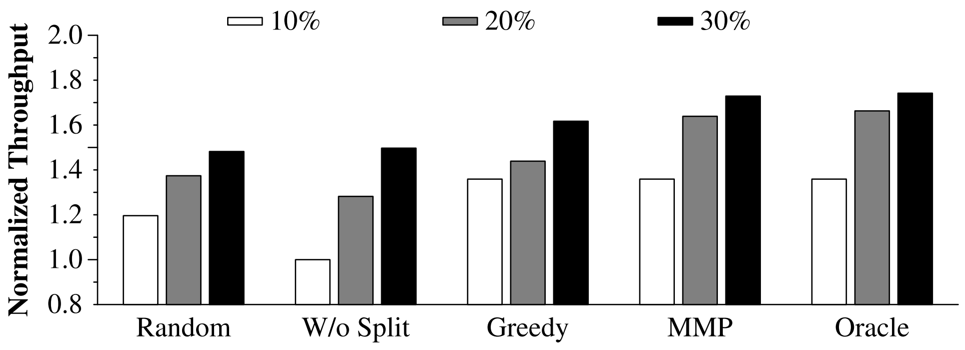


Figure 8: Normalized Throughput of Algorithms

Figure 8 shows the normalized throughput with the QoS threshold varying from 10% to 30%. We compared our MMP algorithm with three heuristics and oracle (the best placement). We calculated the sum of the normalized throughput for each pair and computed the average of these values. When a latency-sensitive CNN model was scheduled alone without a pair, its normalized throughput was 1. Our MMP algorithm achieves a higher normalized throughput than the other heuristics. Moreover, our algorithm achieves 99.26% of the oracle on average. The greedy heuristic shows the same normalized throughput when the QoS threshold was 10%, because there were only a small number of possible pairs that they could choose because of the QoS violation. When the QoS threshold was relaxed, the greedy heuristic exhibited lower performance than our MMP algorithm. The normalized throughput of the W/o split was only

1 when the QoS threshold was 10%, because any latency-sensitive CNN model cannot be co-located with lower than 10% of slowdown without using kernel split. When the QoS threshold was relaxed, we obtained a higher normalized throughput, because the batch CNN models could be co-located with the latency-sensitive CNN models without a kernel split or a lower split factor.

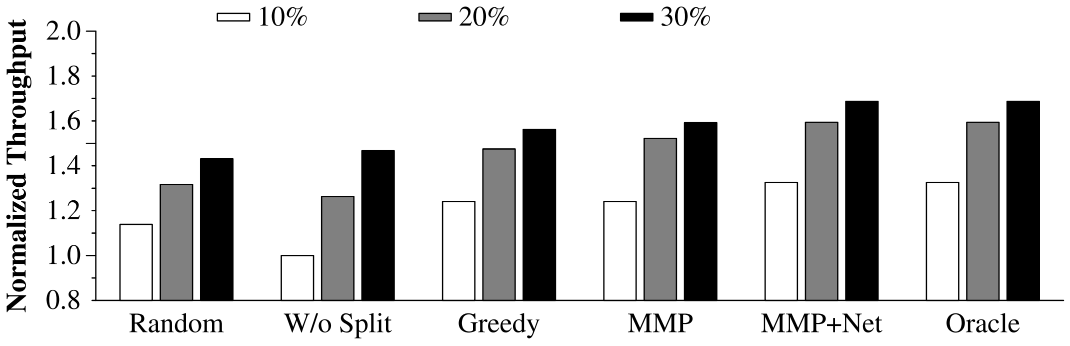


Figure 9: Normalized Throughput of Algorithms in Restricted Scenario

In the first result, the latency-sensitive applications do not pair up with VGG16 model except random heuristic. The VGG16 model has a large size of parameters, and it shows high network bandwidth. So, there is no performance degradation from the network interference in the first result. Unfortunately, we cannot co-locate two CNN models with large parameter size, because they consume a large amount of the device memory, and our GPUs does not have sufficient amount of the device memory to support them simultaneously. However, we can use the Trivial model that consumes high network bandwidth without large parameter size. To show the effect of network communication with distributed parallel models, we do experiment with restricted scenario that only three batch models are exist. In this scenario, the number of latency-sensitive models is same with the number of batch models, and that means there is no extra batch models. We exclude GoogleNet, Resnet50 batch models in this scenario.

Figure 9 shows the normalized throughput with varying QoS threshold in the restricted scenario. In this scenario, VGG16 model is paired up with latency-sensitive models in some cases. Our MMP algorithm (including W/o split) and greedy heuristic uses the prediction result of simulator, and it shows higher error rate when the latency-sensitive model is colocated with VGG16 models than other cases, because there is no consideration of network interference in our simulator. In this scenario, our MMP algorithm achieves 94.48% of oracle in average. As we mentioned as section 4.3, we also have prediction model that considers the effect of network. We use this model for our MMP algorithm to schedule CNN models, and we call this as MMP+Net. Our MMP+Net shows exactly same scheduling decision with oracle with all QoS thresholds.

V Discussion

5.1 Architectures for Communication

The performance modeling for three different system architectures for the communication of deep learning models, namely parameter servers (PS), peer-to-peer (P2P), and ring allreduce (RA) is proposed. Research to model the behavior of systems to predict performance of the computation and the communication, and to analyze the performance of the systems is exist [84]. In our work, we used parameter servers for the CNN models, but had the same number of parameter servers and workers as the number of nodes. The experiment in that work showed the use of RA resulted in a higher throughput than the use of PS or P2P, because of the reduced network overhead. Even the host-side interference showed a low effect except in a few cases in our experiments, it will show lower effect when we use the RA system.

5.2 Overheads

As mentioned in section 2.3, our GPU usage pattern simulator repeats the behavior of applications to improve accuracy. Our MMP algorithm requires the result of the simulation for all possible pairs, so we checked the overhead of the simulation. The running time of the simulator depends on the length of the simulation and the number of kernel and memcpy calls of applications. The default configuration of our simulator is 10000s simulation, and we used this configuration for all of the experiments. Our simulator took less than 1s with 10000s simulation for each pair. When we increased the length of simulation to 100000s, it took 5s, but the accuracy of the simulation was almost the same as that before.

As mentioned in section 3.1, we profiled twice for each CNN model with 50 iterations. Each profiling run was completed in less than 1 min.

The time complexity of our MMP algorithm is $O(M^2N)$.

VI Concluding Remarks

In this work, we proposed the interference-aware resource management system for the distributed parallel CNN models in a GPU cluster, aiming to predict the effect of the interference from the co-location and find a scheduling method to minimize the slowdown of the CNN models. We analyzed the effect of interference on both the device- and the host-side of the parallel GPU applications, and showed that the interference on the host-side could be the main reason for the performance degradation for some CNN models with large parameter size or high network bandwidth. We present a kernel-split technique that could mitigate the slowdown from the co-location of the CNN models. We built an interference-aware performance prediction model that could predict the effect of the device-side interference accurately at a low cost. Furthermore, we showed the possibility for considering the host-side interference. Using the experiments performed on the GPU cluster, we also showed that the proposed interference-aware resource management system could achieve a near-optimal performance.

Chapter 6

Related Works

Several techniques to schedule applications in heterogeneous and consolidated clusters have been studied. Paragon is a QoS-aware scheduler that considers heterogeneous resources and interference in a large scale datacenter for single node applications [4]. A fair placement technique based on game theory is proposed to perform pairwise task collocations [85]. A QoS-aware management system called Quasar is proposed to handle resource allocation and assignment of incoming workloads, including distributed applications, in a heterogeneous cluster [15]. The above three techniques use collaborative filtering to estimate the performance of applications.

In Quasar [15], for an application, it performs four classification techniques for estimating the effects of scale-up, scale-out, heterogeneity and interference separately. It also employs a greedy scheduler that exams and selects nodes, one by one, from the highest performance platform for the application to find a placement that satisfies a QoS constraint. This greedy scheduler is well suited for applications commonly used in large scale datacenters such as web server, latency critical and stateful services, and distributed analytics frameworks. Quasar mostly considers non-virtualized systems. Mage considers the heterogeneity across and within servers when scheduling latency-critical and batch applications, improving the performance while satisfying QoS requirements [86].

The effects of different platforms and co-runners on the performance of applications have been investigated in the domain of large-scale datacenters and web-service workloads [5]. Also, several mapping heuristics for a set of independent tasks on heterogeneous computing systems have been studied [87, 88]. In the above work, the heterogeneity is mainly caused by different machine types like different microarchitectures. In [5], usually two long running web-service applications such as websearch are co-located in a node, due to core and memory requirements. Thus, the effects of microarchitectural heterogeneity are much stronger than those of co-runner heterogeneity. In their results, the performance degradation due to co-runners is usually less than 30%, but that due to platform affinity is as high as 3.5 times. However, in our work, we analyze the affinities of diverse many-task applications to heterogeneous platforms, which are different not only in microarchitectures of nodes, but also in software and middleware stack, and network and storage configurations. Furthermore, for many-task applications, a higher degree of co-runners need to be considered, since a task is usually assigned to each core in a node. In our results, the degradation due to co-runners can be higher than that due to platforms.

In [89], for a heterogeneous computing system with supercomputers, grids, and clouds, a proposed scheduling algorithm matches applications and resources, by considering the importance of a platform to an application (which is similar to the platform affinity), and the importance of an application to a platform. However, the effect of co-runners was not considered, and synthetic workloads were used for analysis. In [90], a scheduling algorithm was proposed to mitigate I/O contention for file transfer in grids for MTC.

Various metascheduling approaches to deal with multiple distributed computing environments such as HPC, grids, and clouds were discussed in [91, 92], including a hierarchical scheduler. In our two-level scheduler, the first level metascheduler decides the number of cores to be allocated to an application for each platform, and the second level scheduler in each platform maps tasks of applications (i.e. pilot jobs) to nodes.

Interference-aware management techniques that normalize a different level of interference to a score have been studied [11,12,14]. An off-line profiling based model [12] and a runtime profiling model [14] are presented for single node applications. An interference modeling technique for distributed parallel applications [11] is proposed for a homogeneous cluster, not considering hardware heterogeneity.

Several techniques to schedule GPU applications with sharing the same GPU have been studied. Prophet [73] is an approach to predict the interference of GPU applications from the sharing GPU; it guarantees the QoS while maximizing the GPU utilization. Our work is similar to this in certain respects, but we considered distributed parallel CNN models and developed a technique for mitigating the performance degradation of the QoS target. The technique for managing the interference on GPU sharing by the performance prediction on the basis of the GPU usage pattern has been studied [74]. It uses the concept of a timed petri-net to predict the performance of GPU applications when they share a GPU, but it focuses only the kernel functions. Mystic [76] uses a collaborative filtering technique to identify the similarity between GPU applications, and guide the scheduler to minimize the interference by co-locating different types of GPU applications.

There have been several efforts to consider job or task placement constraints for performance modeling and scheduling of clusters. The issue of increased scheduling delays due to task placement constraints was discussed and methodologies to incorporate the effects of the constraints on the performance benchmark were proposed [41]. Thinakaran et al. proposed a scheduler that considers various task placement constraints from Google datacenter traces [93], while ensuring low tail latency in heterogeneous datacenters [49]. Tumanov et al. proposed a scheduler that considers soft constraints, which are not mandatory, to maximize the benefit in heterogeneous clouds [56]. Ghodsi et al. proposed an allocation mechanism that extends max-min fairness to consider job constraints only based on machine attributes [57].

An algorithm was proposed to achieve the high availability property called *k-resilient*, based on VM placement that considers the VMtoVM-MustNot and VMtoNode-Must constraints [94]. A hierarchical placement algorithm for an application composed of a set of VMs on a datacenter (of regions, zones, and racks) was studied, supporting three types of constraints, resource demand, communication and availability for the VMs of the application [60]. VM placement techniques for multi-VM applications (such as a 3-tier web application) were proposed where an application owner can specify constraints of single-rack, affinity, and anti-affinity for VMs of the application [95]. Similar to our work, Jhawar et al. investigated resource management that considers constraints between VMs, and between VMs and nodes. However, they have three types for global constraints of resource capacity, infrastructure constraints of forbid and count, and application constraints of restrict, distribute, and latency, and no performance evaluation was provided for the algorithm [96]. For maximizing the revenue of a provider, Shi et al. proposed VM provisioning approaches in which for a set of VMs, one of some defined constraints, regarding full deployment of all the VMs in the set, anti-colocation, and security, can be specified, and the proposed algorithms compute VM placement for given sets of VMs [97].

Constraint-aware placement techniques on clouds were studied, but they consider constraints regarding resource requirements without focusing on constraints based on node and VM attributes [98, 99]. Similar to these techniques, constraint solvers may be used, but the cost and complexity of using the solvers may become high to satisfy not only resource requirements but also VMtoNode-Must, VMtoVM-Must, and VMtoVM-MustNot constraints of VMs. In our work, we provide a general model of constraints that supports all types of constraints, not only for service availability, fault tolerance, or multi-VM applications, and present six constraint-aware VM placement heuristics. We also analyze the effects of different types of constraints on the performance over various system and constraint configurations using the heuristics.

For grid systems, constraint-aware scheduling techniques have been studied [100, 101]. These techniques consider constraints like hardware constraints and co-locality of jobs similar to our work. However, they do not focus on presenting a model of constraints. Note that in grid systems, it is harder to utilize job migration in order to satisfy given constraints unlike virtualized systems.

Techniques to find the best virtual cluster configuration for distributed parallel applications have been investigated [10, 102]. For MapReduce applications, a system to automatically find a good cluster configuration regarding its size and resource types is presented for clouds [102]. A configuration guidance framework for scientific MPI-based applications with various inputs is investigated to find the optimal VM configuration that satisfies the cost and runtime requirements on clouds [10]. The optimal or near-optimal cloud configuration for big data analytics applications can be found efficiently based on Bayesian Optimization [103]. A technique to estimate the performance of advanced analytics such as machine learning by using a small size input is proposed [104]. Machine learning algorithms have been used to predict the performance of virtualized applications [105] and HPC workloads on a single multicore machine [28], and also predict the effects of GPGPU hardware configurations [106].

The performance of many-task applications has been analyzed in a certain platform such as supercomputers [107, 108] and clouds [109, 110]. However, the performance difference caused by unique characteristics of platforms as well as diverse combinations of co-runners has not been investigated. For virtualized environments, the performance interference has been analyzed, but the main focus was to understand the effects of virtualization [111, 112].

Resource harvesting techniques have been investigated, which co-locate latency-critical services and batch workloads to increase the resource utilization of a cluster while minimizing the performance degradation of the latency-critical services [113]. In our work, the ES algorithms attempt to place VMs on the minimum number of nodes by increasing the resource utilization of each node, while satisfying the placement constraints of the VMs.

Various quality of service (QoS) metrics in cloud computing systems have been discussed [114]. In our work, we support two popular optimization goals, energy saving, and load balancing, of cloud computing systems. Our work can be extended to support more complex optimization goals (such as minimizing the number of active nodes while balancing the load among them). Also, QoS metrics such as performance (i.e., the execution time, throughput, or response time) and availability of a VM can be supported by specifying constraints properly. For example, to satisfy QoS of a VM which executes a web server, the VMtoNode-Must constraint, which is to place the VM on a node with high CPU performance, and the VMtoVM-MustNot constraint, which is not to co-locate it with resource-heavy VMs, can be specified for the VM.

The technique for QoS support with fine-grained sharing on GPU has been investigated [115]; it converts the QoS goal into IPC goals to provide epoches for the kernels of the QoS job. Gandiva [116] is a scheduling framework that utilizes domain-specific knowledge to improve the latency and efficiency of training DL models in a GPU cluster. It focuses on the time-slicing of DL models on the basis of a mini-batch to improve the cluster efficiency. Salus [117] also focused on time-slicing based on a mini-batch to enable two GPU sharing primitives: fast job switching and memory sharing.

Existing VM management systems allow users to specify VM placement constraints. VMware vSphere [61] has features of VM-Node anti-affinity/affinity rules in which constraints between a group of VMs and a group of nodes can be specified, and VM-VM anti-affinity/affinity rules in which constraints between individual VMs are specified. In SCVMM [51], placement conditions between VMs and nodes can be specified based on their custom properties, and also possible owners (i.e. nodes) of an individual VM can be provided by users. For a VM, an availability set of VMs which cannot be placed on the same node with the VM can be defined. In XenServer [52], users can specify that a VM must be placed on a specific node. Using the above features, users may not easily describe complex conditions of all different types of constraints based on VM and node attributes. Moreover, the users may not easily figure out conflict conditions between constraints. In VMware's Distributed Resource Scheduler (DRS), the load balancing algorithm can be automatically or manually called to improve the load balance of a cluster [62, 118]. The DRS algorithm based on a greedy hill-climbing technique basically selects the best VM migration case among all possible ones to minimize the imbalance of the cluster, and continues this process until the imbalance becomes less than a given threshold. Therefore, the DRS algorithm will show the performance trend similar to our periodic migration algorithm.

Some techniques similar to our kernel-split techniques have been studied. GPipe [119] is a scalable pipeline parallelism library for giant DNNs with model parallelism, and it slices the forward and backend processes to reduce the waiting time for devices. Mesh-TensorFlow [120] allows users to split tensors across any dimensions for supporting model parallelism. GPU-SAM [121] is a real-time multi-GPU scheduling framework that splits the GPU application into smaller computation units and executes them in parallel to reduce the execution time. A locality-aware matrix operation technique that divides a kernel to fit the size of cache for the cache efficiency has also been studied [122]. Split-CNN [123] splits the input image into small patches and operates on these patches independently to deal with the memory space problem.

ROADEF/EURO challenge 2012 [124] dedicated to a machine reassignment problem which reassigns processes to machines with some constraints (such as conflict, spread and dependency) in collaboration with Google. OptaPlanner [125] provides various algorithms to solve scheduling problems, and it considers the machine reassignment problem as one of examples.

Chapter 7

Conclusion

In this thesis, we study how to manage resources efficiently while considering the characteristics of various resources and applications, interference between applications, and placement constraints of jobs. We present VM placement techniques in heterogeneous clusters that considering heterogeneity and interference to improve the overall performance of distributed parallel applications, and considering placement constraints to analyze the effect of constraints and improve the performance of the cluster. We also present scheduling techniques for many-task applications in distributed computing systems while considering the affinity of applications for platform and corunners. We propose interference-aware GPU sharing techniques for deep learning applications that maximize the throughput of all clusters by using batch models while guaranteeing QoS for latency-sensitive models.

To conclude our work, we will discuss how to consider all of the issues concurrently by extending the concept of constraints in the third work to first work. Our first work considers distributed parallel applications, and VMs are used to execute them. Placement constraints came from the characteristics of applications and resources, so these constraints should be extended to the relationship between applications and resources, and between applications in this work. Our constraint model can be applied as follows:

- **VMtoNode-Must** Applications use a set of VMs (virtual cluster) to execute the application. When the application has this constraint, VMs for this application should have the same constraint.

- **VMtoVM-Must** This constraint becomes the relationship between applications. If this constraint is applied without any modification, VMs of applications should be placed in the same node. That means, a single node have to execute all VMs for two applications. It is not possible for many systems doesn't have huge nodes that can allocate every VMs, and even if it is possible to place all VMs in the same node, it restricts the VC configuration of applications too much. This model should be extended as two sets of nodes to place VMs for each application should be same.

- **VMtoVM-MustNot** This constraint becomes the relationship between applications. VMs for one application should be placed in different nodes with VMs of another application.

When we choose k types of resources out of T types in the heterogeneous cluster with many types, we have to check VMtoNode-Must constraints to pick resources that satisfy these constraints. When our simulated annealing choose VC configuration for application hypothetically, it has to check all constraints to use configurations that satisfy constraints.

In this thesis, we present resource allocation techniques for a variety of distributed parallel applications and consolidated clusters. From our works, we show that our techniques can effectively support various applications and resources by considering their characteristics, interference and constraint.

References

[1] "Apache Hadoop," http://hadoop.apache.org/.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[3] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool Publishers, 2009.

[4] 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 (ASPLOS)*, 2013.

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

[6] "LAMMPS," http://lammps.sandia.gov/.

[7] "NAMD," http://www.ks.uiuc.edu/Research/namd/.

[8] "OpenFOAM," http://http://www.openfoam.com/.

[9] "Amazon EC2 Reserved Instances Pricing," https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/.

[10] J. Han, C. Kim, J. Huh, G. J. Jang, and Y. r. Choi, "Configuration guidance framework for molecular dynamics simulations in virtualized clusters," *IEEE Transactions on Services Computing*, vol. 10, no. 3, pp. 366–380, 2017.

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

[12] 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 (MICRO)*, 2011.

[13] 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.

[14] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[15] 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 (ASPLOS)*, 2014.

[16] "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types/.

[17] "SPEC MPI 2007," https://www.spec.org/mpi2007/.

[18] "NPB," https://www.nas.nasa.gov/publications/npb.html.

[19] R. Eglese, "Simulated annealing: a tool for operational research," *European journal of operational research*, vol. 46, no. 3, 1990.

[20] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004, pp. 137–149.

[21] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder, "Spec mpi2007-an application benchmark suite for parallel systems using mpi," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 2, pp. 191–205, Feb. 2010.

[22] R. Riesen, "Communication patterns," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06, 2006, p. 8.

[23] E. Frank, M. A. Hall, and I. H. Witten, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann, 2016.

[24] C. M.Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[25] S. R. Gun, "Support vector machines for classification and regression," *ISIS technical report*, vol. 14, pp. 85–86, 1998.

[26] L. Rokach and O. Maimon, "Top-down induction of decision trees classifiers - a survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 35, no. 4, pp. 476–487, Nov 2005.

[27] C. McCue, *Introduction Data Mining and Predictive Analysis (Second Edition).* Butterworth-Heinemann, 2015.

[28] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012.

[29] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005.

[30] M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP journal*, vol. 11, no. 1, 1997.

[31] D. P. Anderson, C. Christensen, and B. Allen, "Designing a runtime system for volunteer computing," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov 2006, pp. 33–33.

[32] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, no. 19, 2002.

[33] "Autodock," http://autodock.scripps.edu/.

[34] "Blast," http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[35] "Cachebench," http://icl.cs.utk.edu/projects/llcbench/cachebench.html.

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

[37] 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, p. 020003, Feb. 2014.

[38] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a fast and light-weight task execution framework," in *Proceedings of the ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 43.

[39] J.-S. Kim, S. Rho, S. Kim, 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*. ACM, 2013.

[40] S. Rho, S. Kim, S. Kim, S. Kim, J.-S. Kim, and S. Hwang, "HTCaaS: A Large-Scale High-Throughput Computing by Leveraging Grids, Supercomputers and Cloud," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1341–1342.

[41] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[42] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[43] "VMware VMotion and CPU Compatibility," http://www.vmware.com/files/vmotion-info-guide.pdf.

[44] "ISV Licensing in Virtualized Environments," https://www.vmware.com/08Q1_wp_vmw _ISV_Licensing.pdf.

[45] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[46] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 44, 2010.

[47] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," in *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.

[48] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration," in *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, 2010.

[49] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: A constraint-aware scheduler for heterogeneous datacenters," in *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 977–987.

[50] "VMware vSphere," http://www.vmware.com.

[51] "Microsoft SCVMM," http://www.microsoft.com.

[52] "XenServer," http://www.xenserver.org.

[53] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, May 2007, pp. 119–128.

[54] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng, “Energy-saving virtual machine placement in cloud data centers,” in *Proceedings of Cluster, Cloud and Grid Computing (CCGrid), the 13th IEEE/ACM International Symposium on*, 2013.

[55] E. Arzuaga and D. R. Kaeli, “Quantifying load imbalance on virtualized enterprise servers,” in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, 2010.

[56] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, “alsched: algebraic scheduling of mixed workloads in heterogeneous clouds,” in *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[57] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Choosy: max-min fair sharing for datacenter jobs with constraints,” in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[58] J. Sonnek and A. Chandra, “Virtual putty: Reshaping the physical footprint of virtual machines,” in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*, ser. HotCloud ’09, 2009.

[59] C. A. Waldspurger, “Memory resource management in VMware ESX server,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[60] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible, “Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement,” in *Proceedings of Services Computing (SCC), IEEE International Conference on*, 2011.

[61] “VMware Distributed Resource Management,” https://labs.vmware.com/academic/publications/gulati-vmtj-spring2012.

[62] A. Gulati, G. Shanmuganathan, and A. Holler, “Cloud-scale resource management: Challenges and techniques,” in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[63] 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,” *Softw. Pract. Exper.*, vol. 41, 2011.

[64] “BMC documentation,” https://docs.bmc.com/docs/display/bcmco90/Managing+Cloud+Capacity+Visibility+view+threshold+settings.

[65] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proceedings of the 2nd Conference on Symposium*

on Networked Systems Design & Implementation - Volume 2, ser. NSDI'05, 2005, pp. 273–286.

[66] C. Jo, Y. Cho, and B. Egger, "A machine learning approach to live migration modeling," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, 2017, pp. 351–364.

[67] "Amazon EC2," https://aws.amazon.com/ec2.

[68] "Top 500 List," https://www.top500.org/.

[69] "TensorFlow," http://www.tensorflow.org/.

[70] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[71] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[72] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[73] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 17–32, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093315.3037700

[74] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar, "Interference-driven resource management for gpu-based heterogeneous clusters," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287091

[75] J. Wu and B. Hong, "Collocating cpu-only jobs with gpu-assisted jobs on gpu-assisted hpc," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 418–425.

[76] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 353–362.

[77] S. Soner and C. Özturan, "Integer programming based heterogeneous cpu-gpu cluster scheduler for slurm resource manager," in *2012 IEEE 14th International Conference on*

High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems, June 2012, pp. 418–424.

[78] "cifar10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.

[79] "Imagenet dataset," http://www.image-net.org/.

[80] "TensorFlow benchmark," https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks.

[81] "Multi-Process Service (MPS)," https://docs.nvidia.com/deploy/mps/index.html.

[82] Y. ri Choi, D. Kim, S. Kim *et al.*, "C-2018-039480 gpu application performance prediction program," https://scholarworks.unist.ac.kr/handle/201301/29513, 2018.

[83] "NVIDIA visual profiler," https://developer.nvidia.com/nvidia-visual-profiler.

[84] S. Alqahtani and M. Demirbas, "Performance analysis and comparison of distributed machine learning systems," 2019.

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

[86] F. Romero and C. Delimitrou, "Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT18)*, November 2018.

[87] 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.

[88] M. Maheswaran, S. Ali, H. J. Siegal, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Heterogeneous Computing Workshop, Proceedings. Eighth.* IEEE, 1999, pp. 30–44.

[89] 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*, ser. LNCS, 2012, vol. 7513, pp. 1–13.

[90] Y. Zhang, S. Chen, and Z. Hu, "A scheduling algorithm for many-task computing optimized for io contention in heterogeneous grid environment," in *Fifth International Conference on Computational and Information Sciences.* IEEE, 2013, pp. 1541–1544.

[91] S. Sotiriadis, N. Bessis, F. Xhafa, and N. Antonopoulos, "From meta-computing to interoperable infrastructures: A review of meta-schedulers for hpc, grid and cloud," in *IEEE 26th International Conference on Advanced Information Networking and Applications (AINA)*, March 2012, pp. 874–883.

[92] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "Distributed job scheduling on computational grids using multiple simultaneous requests," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 359–366.

[93] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[94] E. Bin, O. Biran, O. Boni, E. Hadad, E. Kolodner, Y. Moatti, and D. Lorenz, "Guaranteeing high availability goals for virtual machine placement," in *Proceedings of Distributed Computing Systems (ICDCS), the 31st International Conference on*, 2011.

[95] G. Keller and H. Lutfiyya, "Dynamic management of applications with constraints in virtualized data centres," in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015.

[96] R. Jhawar, V. Piuri, and P. Samarati, "Supporting security requirements for resource management in cloud computing," in *Proceedings of Computational Science and Engineering (CSE), IEEE 15th International Conference on*, 2012.

[97] L. Shi, B. Butler, D. Botvich, and B. Jennings, "Provisioning of requests for virtual machine sets with placement constraints in iaas clouds," in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2013.

[98] H. N. Van, F. D. Tran, and J. Menaud, "Performance and power management for cloud infrastructures," in *Proceedings of the IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 329–336.

[99] L. Zhang, Y. Zhuang, and W. Zhu, "Constraint programming based virtual cloud resources allocation model," *International Journal of Hybrid Information Technology*, vol. 6, pp. 333–344, 11 2013.

[100] R. Raman, M. Livny, and M. Solomon, "Matchmaking: distributed resource management for high throughput computing," in *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, July 1998, pp. 140–146.

[101] T. Majumder, M. E. Borgens, P. P. Pande, and A. Kalyanaraman, "On-chip network-enabled multicore platforms targeting maximum likelihood phylogeny reconstruction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 1061–1073, July 2012.

[102] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[103] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 469–482.

[104] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 363–378.

[105] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[106] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[107] K. Wang, Z. Ma, and I. Raicu, "Modeling many-task computing workloads on a petaflop ibm blue gene/p supercomputer," *IEEE CloudFlow*, 2013.

[108] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2010, pp. 1–10.

[109] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, 2011.

[110] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Multicloud deployment of computing clusters for loosely coupled mtc applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 924–930, 2011.

[111] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 51–58.

[112] Y. Koh, R. C. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments." in *ISPASS*, 2007, pp. 200–209.

[113] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, "History-based harvesting of spare cycles and storage in large-scale datacenters," in *Proceedings of*

the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 755–770.

[114] T. Guérout, S. Medjiah, G. D. Costa, and T. Monteil, "Quality of service modeling for green scheduling in clouds," *Sustainable Computing: Informatics and Systems*, vol. 4, pp. 225–240, 2014.

[115] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 269–281.

[116] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[117] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," *CoRR*, vol. abs/1902.04610, 2019. [Online]. Available: http://arxiv.org/abs/1902.04610

[118] M. I. Sal, M. Vikas, and J. Adarsh, "DRS Performance - VMware vSphere 6.5," *VMware Inc., White Paper*, pp. 1–27, 2016.

[119] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, 2018. [Online]. Available: http://arxiv.org/abs/1811.06965

[120] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," 2018.

[121] W. Han, H. S. Chwa, H. Bae, H. Kim, and I. Shin, "Gpu-sam: Leveraging multi-gpu split-and-merge execution for system-wide real-time support," *Journal of Systems and Software*, vol. 117, pp. 1–14, 2016.

[122] Y. Chen, A. B. Hayes, C. Zhang, T. Salmon, and E. Z. Zhang, "Locality-aware software throttling for sparse matrix operation on gpus," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 413–426. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/chen-yanhao

[123] T. Jin and S. Hong, "Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization," in *Proceedings of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 835–847. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304038

[124] "Google ROADEF'12," http://challenge.roadef.org/2012/en/.

[125] "OptaPlanner," https://www.optaplanner.org.

Acknowledgements

First, I would like to express gratitude to my advisor - Prof. Young-ri Choi for the continuous of my Ph.D study. She guided and encouraged me in all time of research and writing this thesis. I couldn't imagine that I can be granted a degree without her support.

I also would like thank my thesis committee: Prof. Beomseok Nam, Prof. Woongki Baek, Prof. Won-ki Jeong, Prof. Myeongjae Jeon, for their valuable comments and encouragement.

I thank my fellow labmates, especially Eunji Hwang. She set an example as a senior of our lab, and helped me for many times. I also thank other labmates and friends for encouraging me and giving me a pleasure for the last seven years.

Last but not the least, I would like to thank my parents, for supporting me throughout my life.