Journal of Network and Computer Applications 146 (2019) 102413

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Generic resource allocation metrics and methods for heterogeneous cloud infrastructures

Cem Mergenci *, Ibrahim Korpeoglu

Department of Computer Engineering, Bilkent University, Ankara, Turkey

ARTICLE INFO

Keywords:
Cloud computing
Resource allocation

ABSTRACT

With the advent of cloud computing, computation has become a commodity used by customers to access computing resources with no up-front investment, but as an on-demand and pay-as-you-go basis. Cloud providers make their infrastructure available to public so that anyone can obtain a virtual machine (VM) instance that can be remotely configured and managed. The cloud infrastructure is a large resource pool, allocated to VM instances on demand. In a multi-resource heterogeneous cloud, allocation state of the data center needs to be captured in metrics that can be used by allocation algorithms to make proper assignments of virtual machines to servers. In this paper, we propose two novel metrics reflecting the current state of VM allocation. These metrics can be used by online and offline VM placement algorithms in judging which placement would be better. We also propose multi-dimensional resource allocation heuristic algorithms showing how metrics can be used. We studied the performance of proposed methods and compared them with the methods from the literature. Results show that our metrics perform significantly better than the others and can be used to efficiently place virtual machines with high success rate.

1. Introduction

Cloud computing provides access to seemingly infinite computing resources in an on-demand and pay-as-you-go basis. Setup time of these resources are within seconds or minutes. Payment is per minutes or hours of use. Compared to traditional hosting methods, where setup time is measured by days and payment is done per month, cloud computing offers rapid and easy scaling for applications without any up front costs.

NIST defines cloud computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” (Mell and Grance, 2011). The standard also defines three basic models of service for cloud computing. These are: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

IaaS gives control to users in configuring and managing computation, networking, storage and operating systems. IaaS is much flexible than PaaS. Users can run arbitrary software on the infrastructure, but have to deal with the complexities it brings. The physical arrangement of the hardware is still hidden from users, therefore limiting architecture specific solutions. Amazon Elastic Compute Cloud (EC2) (AMAZON.com Inc., 2006) and Rackspace Cloud Servers (Rackspace US Inc., 2008) are two prominent IaaS offerings.

The challenge for IaaS provider is to allocate available resources to VM instance requests. While allocating resources, there are several factors that can be considered such as utilization, energy consumption, load balancing, task allocation, and fairness. The fact that instances are created and terminated dynamically may require the resource allocation to be dynamic as well. Another important issue is to satisfy as many VM requests as possible for a given limited physical capacity.

In this paper, we address the resource allocation problem in IaaS systems, running on heterogeneous cloud computing infrastructures, with a new approach. First, we define what constitutes a good allocation by examining the nature of multi-dimensional resource allocation where physical machine (PM) resources are of different types, like CPU, memory, disk, and network bandwidth. Based on this, we propose metrics that can reflect the allocation state and can be used by VM allocation algorithms and performance evaluations to judge how good the allocations are. These metrics can be used to compare different allocation algorithms. More importantly, they can be used as part of allocation

* Corresponding author.
E-mail addresses: mergenci@cs.bilkent.edu.tr (C. Mergenci), korpe@cs.bilkent.edu.tr (I. Korpeoglu).

https://doi.org/10.1016/j.jnca.2019.102413
Received 22 February 2019; Received in revised form 26 May 2019; Accepted 28 July 2019
Available online 31 July 2019
1084-8045/

algorithms to judge the quality of alternative allocations at a certain iteration of the algorithm or with every request arriving.

We also show how our metrics can be used in online and offline VM allocation schemes by proposing some heuristic algorithms utilizing the metrics. We consider non-migrating VMs, hence the cost of migration is not considered in our study.

We evaluated the proposed metrics and methods through extensive simulation experiments. We studied how well our metrics can solve VM placement problem. We also compared our methods with a number of existing methods. Our evaluation results show that the metrics we propose accurately capture the resource utilization state and can be used as part of VM placement algorithms with high satisfaction ratios. In majority of the cases, our methods perform much better than the other existing methods in terms of number of VM placement problem instances that can be solved successfully.

The rest of the paper is organized as follows. In Section 2, we give an overview of existing VM allocation work in the literature. In Section 3, we describe multi-dimensional resource allocation problem. In Section 4 and 5, we provide and describe our proposed metrics and methods in detail. In Section 6, we define our simulation environment, present and discuss our experiments and results. Finally, in Section 7, we present our conclusion.

2. Related work

We present related work in the literature under four categories: network-aware, energy-aware, service level agreement (SLA) based, and utilization-focused.

2.1. Network-aware

Meng et al. (2010) present a traffic-aware VM placement strategy. Authors formulate an optimization problem and prove its hardness. They provide a heuristic algorithm that solves the problem efficiently for large problem sizes. The algorithm first clusters the set of VMs and PMs. Then it matches VM clusters with PM clusters, finally assigning individual VMs to PMs. Experiments evaluate the efficiency of the allocation algorithm for various data center network architectures. A similar study is presented by Shrivastava et al. (2011), where communication dependencies and network topology is incorporated as cost metrics into migration decision.

Another network based allocation method is presented by Alicherry and Lakshman (2012). The study considers a multiple data center environment and proposes algorithms that minimizes latency between VMs of the same request hosted at different data centers, as well as algorithms that minimize inter-data center traffic and inter-rack traffic within a data center. The data center selection problem under presented model is shown to be NP-hard, and a 2-approximation algorithm is described. Datacenter network is assumed to have a tree topology. Machine selection aims to allocate VMs so that height of the communication tree is minimum.

FairCloud (Popa et al., 2012) defines three cloud network service requirements: min-guarantee, high utilization, and network proportionality. Min-guarantee states that a cloud user should be able to get a guaranteed minimum bandwidth. High utilization requires available bandwidth to be used when needed. Network proportionality means that bandwidth should be distributed among cloud users proportionally to their payments. The study defines trade-offs between these requirements and presents allocation algorithms.

To the best of our knowledge none of the allocation methods that focus on network resources consider multidimensionality of the PM resources. They assume a certain VM capacity for PMs, neglecting request and workload requirements of different VMs. In our study, we consider network resources just like others, such as CPU or memory.

2.2. Energy-aware

Mezmaz et al. (2011) propose a genetic algorithm (GA) based task scheduling optimization for both makespan and energy consumption metrics. Because there are two objectives, a solution is not unique, but is a set of Pareto points. The user is able to choose the right amount of trade-off between makespan and energy consumption among those points. The study is based on energy-conscious scheduling (ECS) heuristic proposed by Lee and Zomaya (2009), which is a greedy scheduling algorithm considering makespan and energy consumption. Proposed method uses GA to find Pareto optimal points among solutions to ECS instances. Multiple evolutionary algorithms are run in parallel. Asynchronous migrations of solutions between parallel-running instances enable exploring a larger solution space.

Beloglazov et al. (2012) present a concise survey of energy aware studies in grid and cloud computing. Authors also define general principles in energy-conscious cloud management. The study defines algorithms for initial placement and dynamic migration of VMs to decrease energy consumption due to CPU utilization. Different from Beloglazov et al. (2012), our study focuses on multidimensional resources and management of heterogeneous workloads. These aspects of energy aware resource allocation are stated as open challenges in the paper.

Whereas many energy-aware studies focus on CPU power consumption, Ye et al. (2017) also consider energy consumption by network resources while maximizing load-balancing and resource utilization. Proposed method is an extension to Knee Point-Driven Evolutionary Algorithm (KnEA) (Zhang et al., 2015), which is a many-objective optimization problem.

In our study, we do not use an energy model to reduce energy consumption. However, as a consequence of packing VMs into fewer PMs we address energy consumption concerns indirectly.

2.3. SLA-based

MorphoSys (Singh et al., 2000) describes a colocation model for SLA-based services. SLA model captures periodic resource requirements of requests. The study uses first fit and best fit heuristics for resource allocation in a homogenous environment. Two cases are considered for allocations: Workload Assignment Service allocate resources to requests, while Workload Repacking Service migrates VMs such that resources are used more efficiently. Different repacking and migration policies are also discussed. The system finds alternative allocations by transforming SLAs into equivalent forms, in case they do not fit into any of the PMs in their original forms.

Garg et al. (2014) train an artificial neural network to predict workload for two different types of applications (high performance computing and Web applications) and allocate VMs accordingly. Resource usage is monitored live so that any violations of service level agreements because of errors in prediction could be resolved by migrating VMs to a better PM. The study focuses on homogeneous cloud infrastructures and uses only one resource dimension, CPU.

2.4. Utilization-focused

Mills et al. (2011) propose a two-step resource allocation process, as inspired by Eucalyptus cloud platform (Nurmi et al., 2009). First, a cluster is selected within the cloud, then a node is selected within the cluster. Combinations of three cluster selection and six node selection heuristics are compared. Heuristics consist of the ones used in Eucalyptus, and those inspired by online bin packing literature. Experiment results show that cluster selection yields statistically significant difference in the average fraction of VMs obtained. Node selection methods, on the other hand, do not produce significant difference.

Singh et al. (2008) describe an end-to-end load balancing strategy between machine and storage virtualization for data centers. Proposed VectorDot algorithm extends Toyoda heuristic (Toyoda, 1975) for multidimensional single knapsacks to dynamic case of multiple knapsacks. VMs above a certain load threshold are migrated to PMs, so that the dot product of resource vector of the VM and those of required network resources is minimized, therefore achieving a lower cost migration.

Arzuaga and Kaeli (2010) discuss quantifying load imbalance on PMs and the overall system. Load imbalance of an individual server is defined to be a weighted sum of the imbalances for each resource type. Load imbalance of the system is defined as coefficient of variation in load distribution. The study also describes a greedy algorithm to balance load among servers. When the load of a server exceeds a threshold, VMs hosted on that server are chosen as migration candidates. The system imbalance values resulting from the migration of every candidate VM to every PM is calculated. The migration that achieves least imbalance is applied. As shown in experiment results presented by Gabay and Zaourar (2016), and reproduced in our study, a metric of weighted sum of dimensions is inferior to others.

Sandpiper (Wood et al., 2009) is a monitoring and profiling framework to detect and remove hotspots by migrating VMs. The gray-box approach cooperates with VMs in determining workloads, while the black-box method does not require an integration with VMs. The study uses the inverse of available resource volume as a metric of multi-dimensional load. However, this approach has problems as explained by Mishra and Sahoo (2011).

Mishra and Sahoo (2011) present anomalies of methods in existing VM placement literature. Based on these anomalies, they define properties of a good VM allocation and propose algorithms for static VM placement and dynamic VM placement with load balancing or server consolidation goals. Authors define properties of a good allocation as follows: it should capture the shape of the allocation, it should use total remaining capacity as well as remaining capacity of individual resources, it should consider overall utilization as well as utilization of individual dimensions. Our proposed methods obey these rules. Their proposed method uses planar resource hexagon that consists of triangles representing different resource utilization categories. VMs are allocated to PMs in complementary resource triangles (CRT), so that overall utilization is tried to be balanced.

Chen and Shen (2014) use a similar idea of placing complementary VMs in the same PM. Complementary VMs are chosen by profiling the performance requirements in terms of resources and time. VMs that use the same resource at different times, or those that use different resources at the same time are considered complementary. VMs are allocated to PMs for which the remaining capacity weighted by weights assigned to resource dimensions is minimum.

Rather than explicitly finding VMs with complementary resource requirements, we focus on designing a good fitness function, minimization or maximization of which will have the same effect.

Other methods model VM placement as a bin packing problem. For single dimensional bin packing there are efficient heuristics to approximate the optimal solution. *First-fit decreasing* (Johnson, 1974) is a popular and very simple such heuristic. It sorts the items to be placed in decreasing order. Beginning from the largest item, it places them in the first bin they fit, opening a new bin if none of the existing ones are suitable. This heuristic does not use more than 11/9 OPT +1 bins, where OPT defines the value of an optimal solution.

Generalizing the first-fit decreasing heuristic to vector packing case is not trivial. Different methods are presented by Panigrahy et al. (2011b). Panigrahy et al. (2011a) propose the dot product metric that we used in our experiments.

Stillwell et al. (2010) compare the performance of greedy, LP-based, genetic, and vector packing algorithms. Even though they only consider clouds with homogeneous resources, they conclude that vector packing approaches are superior to others.

Gabay and Zaourar (2016) formally define vector bin packing with heterogeneous bins (VBPHB) problem. A weighted sum of dimensions of vectors are proposed as a method of ordering multi-dimensional items and bins. By using different weights and calculating the weighted sum statically or dynamically, various measures are defined. These measures are used in combination with item- and bin-centric allocation heuristics as well as balancing-focused ones. Authors present a benchmark that consists of five classes of problem instances with different item and bin properties. Combinations of proposed measures and heuristics are evaluated on this benchmark. Authors also apply their theoretical work to a real-world machine reassignment problem. We compare our novel methods with the ones presented by Gabay and Zaourar (2016). Results show that our methods perform much better in the majority of the cases.

A survey of VM placement schemes are presented by Masdari et al. (2016). According to their taxonomy, our method can be considered as a resource-aware bin packing-based method for heterogeneous environments. According to another survey of resource provision algorithms in cloud infrastructures (Zhang et al., 2016), our study classifies as a bin packing method for server selection with objectives of node cost minimization, energy efficiency, and utility maximization.

There is a need for a VM allocation method for heterogeneous cloud infrastructures, because clouds rarely consist of homogeneous resources. As cloud computing develops, ever more types of computing resources are offered to customers, such as SSD storage, GPUs, application specific integrated circuits (ASICs); therefore, the allocation method should support an arbitrary number of dimensions. We know that simple weighted sum of resource dimensions is not good enough. We improve upon these points. Our main contributions are as follows:

- We define two design principles to quantify the quality of an allocation so that allocation alternatives could be compared in a consistent manner.
- Using these design principles, we propose two novel measures.
- The measures we propose are parametric so that they could be adapted to the specific environment in which they will be used.
- Our measures are suitable to be used with different allocation algorithms, as we demonstrate in this study.
- We evaluate the performance of our proposed methods using a benchmark. Results show that our metrics perform better than the ones in the literature.

3. Multi-dimensional VM allocation problem

Resource allocation problem for virtual machines in cloud computing infrastructures can be considered from different perspectives and at different levels of complexities.

According to the way requests are processed, resource allocation can be done in an online (incremental) or offline (batch) manner. Online algorithms process every VM request individually as they arrive. Considering the current state of the system resources, the request is allocated to the best PM. The algorithm continues allocation with the next request. Offline algorithms process a set of requests at the same time, and resources are allocated accordingly. Ideally, an offline method knows all requests before starting allocation.

In practice, VM requests arrive as the system operates and the request information is not available to the system beforehand. Therefore, we consider offline allocation as batch allocation where resource requests arrive in sets. These sets may consist of dependent or independent requests. Online and offline algorithms are reducible to each other. Individual requests could be buffered to be processed in batches, or batch requests could be allocated individually. Online algorithms are simple in expression and should be fast but may not achieve the same efficiency as offline algorithms. Offline algorithms can be more sophisticated and efficient but it may not be possible to delay requests to form a batch.

Current virtualization technologies allow VMs to migrate from one PM to another. Although migration does not interrupt VM operation, it causes delay and overhead. Migration is also costly to cloud operator. Depending on the size of the VM, network resources of the infrastructure, processing and IO resources of PMs will be used. Therefore some systems may not apply migration, and in those systems VMs will be *static*, running in the same PM until termination. Additionally, depending on the class of applications running on VMs, the lifetime of VMs may be short or very long. In the latter case, the resource allocation algorithm can ignore VM termination in a round of execution. As VMs come and go, the resources can be fragmented since different VMs may have very different resource requirements. In the case of *dynamic* VMs, where VMs can be migrated, efficiency may be increased by migrating VMs to more suitable PMs. However, as mentioned above, migration costs are an important concern. More migrations than necessary would decrease overall performance and consume network resources.

Under the resource model we define, resource allocation problem is a multi-objective optimization problem. Resource allocation aims to improve energy efficiency, resource utilization, load balancing, and server consolidation. Because of the NP-hard nature of the problem, it is not feasible to search the problem space efficiently and reach an optimal solution. Additionally, a utility function combining different objectives into a single value is usually necessary to decide among alternative solutions and make search space smaller. Heuristic or metaheuristic methods can be used to find practically good enough results. Heuristic algorithms run faster and therefore could be preferable for real time tasks.

In this paper we focus on request oriented and static VM allocation. This is very similar to well-known *bin-packing* problem, which is an NP-hard combinatorial optimization problem. There are *n items*, i, of different sizes, w_i. These items are placed into at most *m bins*, j, with a certain *capacity*, C. The objective is to minimize the number of bins used while placing the items. Bin packing can be described with the following integer linear program:

$$\text{minimize} \sum_{j=1}^{m} y_j \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_{ij} \leq C y_j, \qquad 1 \leq j \leq m \tag{2}$$

$$\sum_{j=1}^{m} x_{ij} = 1, \qquad 1 \leq i \leq n \tag{3}$$

where binary decision variable x_{ij} defines whether item i is placed in bin j, and y_j defines whether bin j is used.

VM allocation in cloud infrastructures can be modeled as a bin packing problem, where VMs correspond to items and PMs correspond to bins. It would be oversimplification to use the basic model without any modifications. First of all, properties of VM instances cannot be captured with a single weight value. VMs are multi-dimensional resource packs, best defined as a vector of values. Secondly, cloud infrastructures are heterogeneous environments, where there is no single capacity that applies to all hardware resources. We can modify Equation (2) to satisfy these requirements:

$$\sum_{i=1}^{n} w_{ik} x_{ij} \leq C_{jk} y_j, \qquad 1 \leq j \leq m, \qquad 1 \leq k \leq d \tag{4}$$

where d is the number of resource types (dimensions). This version of the problem is called *vector bin packing*, or simply *vector packing* problem.

In this paper, we present our own vector packing approach based on *fitness* functions, i.e., *utilization metrics*. A fitness function maps each point of a multi-dimensional resource utilization space to a single numeric value so that different points in space can be compared easily. At the end, our approach aims to increase the utilization of physical resources that are powered up in cloud infrastructures so that a minimum number of physical machines are used.

4. Proposed utilization metrics

In this paper, we propose utilization metrics, i.e., fitness functions, and related methods as a solution to VM placement problem. We will first introduce our metrics that can reflect how good a possible allocation is. Then we will provide sample algorithms to show how our metrics can be used in efficient resource allocation. Before describing our proposed metrics, however, we start with some preliminaries.

We consider a vector-based resource allocation model based on the following definitions from Mishra and Sahoo (2011):

- *Normalized Resource Cube* (NRC): A unit cube representing total available resources of a PM. Each dimension of the cube corresponds to a resource type (such as CPU, memory, disk space, I/O bandwidth, etc.). All other normalized vectors are located inside this cube. If there are d different resource types in PM or VM, we consider the problem as d dimensional resource allocation problem.
- *Total Capacity Vector* (*TCV*): The vector along principal diagonal of NRC. It is equal to the vector $\mathbf{1^d}$.
- *Resource Requirement Vector* (RRV): The amount of resources that are needed by a VM. RRVs are not normalized, they have absolute values. However, they are assumed to be normalized by the scaling factor of the NRC in the context of NRCs.
- *Resource Utilization Vector* (RUV): The amount of utilized resources inside an NRC. RUV is the sum of RRVs of VMs that are hosted in a PM.
- *Remaining Capacity Vector* (RCV): The amount of unutilized resources inside an NRC. ($\mathbf{RCV} = \mathbf{TCV} - \mathbf{RUV}$)
- *Resource Imbalance Vector* (RIV): The difference between RUV and its projection onto TCV.

Fig. 1 pictures an NRC with its vectors. The number of dimensions of the vector space depends on the number of resource types considered. Here, a three dimensional space is depicted with x, y, and z resource dimensions. Notice that addition of an RRV cannot decrease the utilization, therefore RUV always increases towards full utilization planes $x = 1, y = 1, z = 1$. The best utilization happens when RUV = TCV; sum of allocated RRVs is $(1, 1, 1)$.

Although the best allocation is obvious, it is not trivial to compare arbitrary allocations. Fig. 2 illustrates a two dimensional vector space with two utilization vectors. Note that an NRC is a square in two dimensions. Two RUVs may have resulted from two sets of RRVs, which may share identical subsets. We need to decide whether RUV_1 or RUV_2 is a better allocation. Comparing RUVs is equivalent to comparing corresponding RCVs. We use an RCV-based comparison, because a comparison that is based on the availability of resources is more meaningful than a comparison based on the used amount.

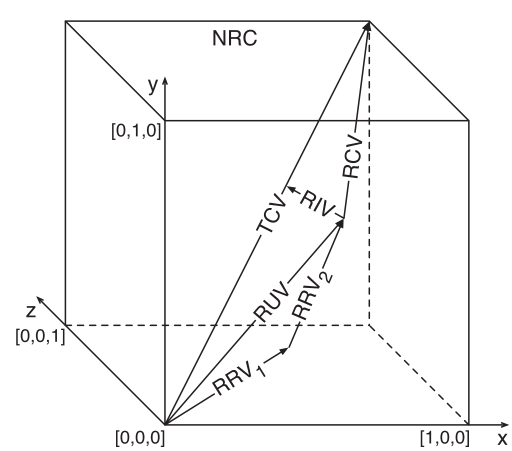


Fig. 1. Normalized Resource Cube (NRC) in vector model for resource request and allocation representation.

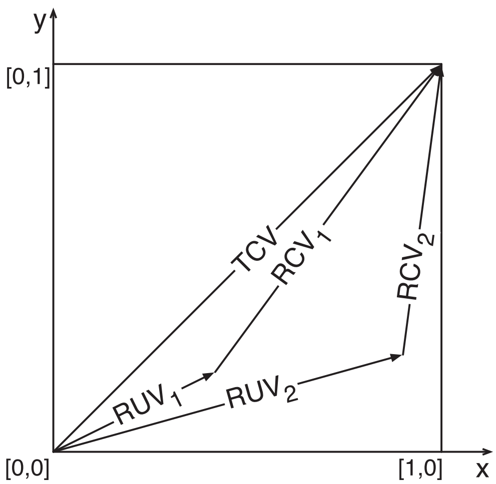


Fig. 2. Vector comparison.

We would like to achieve the best utilization. Therefore a naive approach is to use Euclidean distance to *TCV*, which would correspond to the magnitude of *RCV*. Further inspection reveals problems with this approach. In Fig. 2, $\|\mathrm{RCV}_2\|$ is smaller than $\|\mathrm{RCV}_1\|$. However, RUV_2 is very close to full utilization of resource *x*, while leaving resource *y* relatively under-utilized. On the other hand RUV_1 utilizes both resource types in a more balanced way. Therefore, we conclude that the angle between *TCV* and *RCV* is an important factor in comparison.

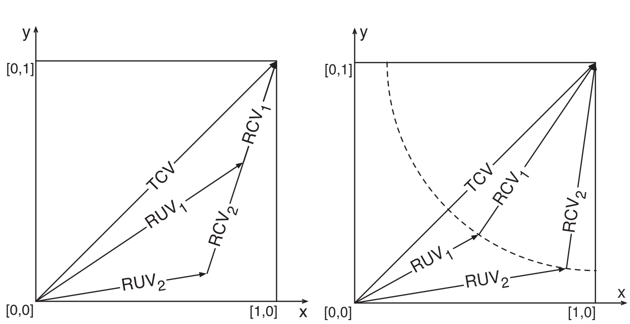
Simply prioritizing either the angle or the magnitude would not work. For two *RCV*s with the same angle, the one with smaller magnitude is closer to best utilization, therefore is better (Fig. 3a). For two *RCV*s with the same magnitude, the one with the smaller angle yields better balance of resource types, therefore is better (Fig. 3b). We conclude that a good fitness function should distinguish allocations based on both angle and the magnitude of the RCV.

4.1. Our metrics

After these preliminaries, we now present our utilization metrics that can be used in judging goodness of alternative allocations. We propose two parametrized metrics: TRfit and UCfit.

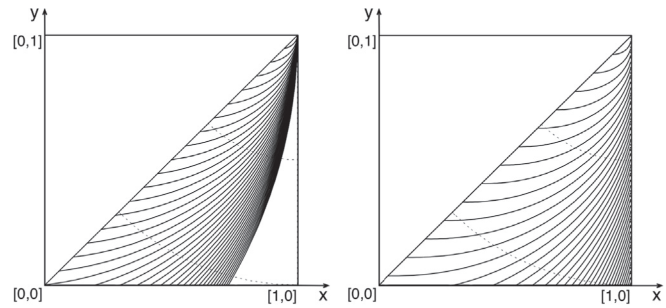
4.1.1. Our first metric (TRfit)

We think that the suitability of allocation can be determined by a fitness function *f* of *RCV*. Fitness function defines a total order in the allocation vector space, therefore providing a way of comparing allocation alternatives. Although the allocation scheme would work for an arbitrary definition of *f*, it is best to define one that achieves an efficient allocation. We came up with the following fitness function,



(a) RCV_1 is better. (b) RCV_2 is better.

Fig. 3. Comparison of two RCVs with the same angle or magnitude.



(a) $\alpha = 0$ (b) $\alpha = \pi/12$

Fig. 4. TRfit fitness function with different parameters.

which we call *TRfit* (TCV-RCV fitness). It considers total capacity vector and remaining capacity vector. It has one parameter, which is α.

$$f_{TRfit}(\mathbf{RCV}) = \frac{\|\mathbf{RCV}\|}{\cos^{-1}\left(\frac{1}{\sqrt{d}}\right) - \theta_{\mathbf{TCV}}(\mathbf{RCV}) + \alpha} \tag{5}$$

where $\theta_{\mathbf{TCV}}(\mathbf{RCV})$ is the angle between *TCV* and *RCV*, given by the formula:

$$\theta_{\mathbf{TCV}}(\mathbf{RCV}) = \cos^{-1}\left(\frac{\mathbf{TCV}\cdot\mathbf{RCV}}{\|\mathbf{TCV}\|\|\mathbf{RCV}\|}\right) = \cos^{-1}\left(\frac{1}{\sqrt{d}}\cdot\frac{\sum_{i=1}^{d}\mathbf{RCV}[i]}{\sum_{i=1}^{d}(\mathbf{RCV}[i])^2}\right) \tag{6}$$

and α is a parameter that results in fitness functions with different properties. *d* is the number of resource types (number of dimensions). Fig. 4a and b depict isometric curves of fitness functions with different α values, $\alpha = 0$ and $\alpha = \pi/12$ respectively. *RCV*s on the same curve have the same fitness value. As shown by the figures, our fitness function is in accordance with the comparison guide described above. Among two *RCV*s with the same angle to *TCV*, the one with less magnitude has better fitness value. Among two *RCV*s with the same magnitude, the one with smaller angle to *TCV* has better fitness value. Note that in our definition of fitness function, *lower* values denote *better* fitness.

An important property of this function is the fact that the same fitness value achieved by allocating a new RRV leads to increasingly higher utilization. In this sense, α is the parameter that defines the amount of increase. When $\alpha = 0$, further allocations that yield the same fitness value lead to full utilization point, where as for $\alpha = \pi/12$ the highest achievable utilization for a given fitness value is suboptimal. Fig. 5 gives TRfit heatmaps for various values of α parameter. A heatmap renders the multi-dimensional resource space based on the fitness function values. The smaller the value, the lighter is the color in the heatmap, and vice versa: the larger the value, the darker is the color. The points that have the same color have the same fitness value. As mentioned earlier, smaller values (lighter colors) indicate better fitness.

4.1.2. Our second metric (UCfit)

There are many fitness functions that conform to our proposed properties. TRfit function presented in previous section is one of them. We present an alternative function in Equation (7), which we call BasicUCfit (utilization-capacity fitness). It considers both resource utilization and remaining capacity. Again, it is a function of *RCV*. BasicUCfit is a simple fitness function that combines the Euclidean distance to full utilization point ($\|\mathbf{RCV}\|$) and the angle between RUV and RCV (θ).

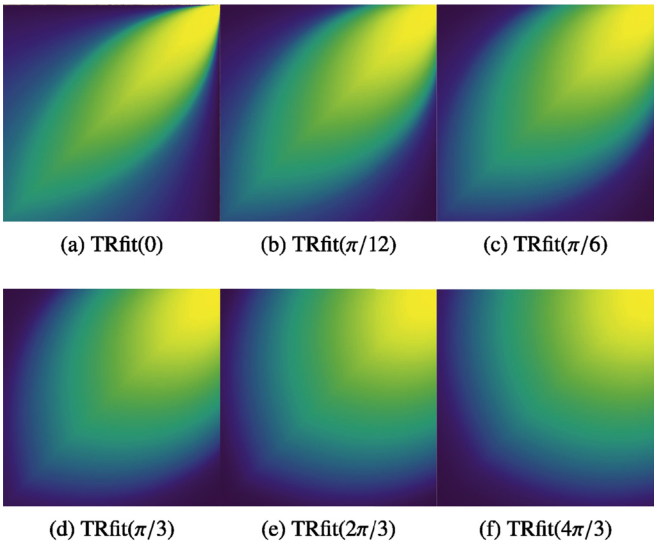


Fig. 5. Visualization of TRfit for different parameter values.

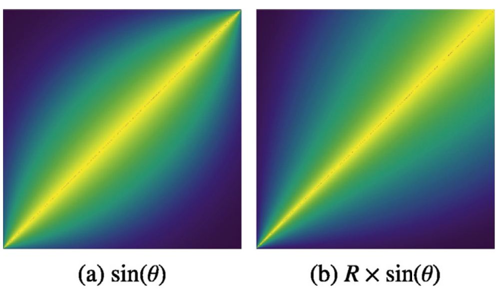


Fig. 6. BasicUCfit heatmap.

Euclidean distance to full utilization point is visualized in Fig. 8c. Sin of the angle between RUV and RCV is visualized in Fig. 6a. Multiplication of both, results in visualization in Fig. 6b. The effect of the Euclidean distance component to the fitness value is more pronounced towards full utilization point. In further points, the effect of the angle is higher.

$$f_{BasicUCfit}(\mathbf{RCV}) = \|\mathbf{RCV}\| \cdot \sin\left(\theta_{\mathbf{RUV}}(\mathbf{RCV})\right) \tag{7}$$

BasicUCfit function is not flexible in its current form. It would be much better if contributions of each component in the formula could be adjusted. Equation (8) introduces its parametrized version, which we call *UCfit*. It has three parameters a, b, c. The parameters a and b adjust the contributions of Euclidean distance and sin terms respectively. One important difference from the original version is the normalization of the Euclidean distance with the magnitude of TCV. Normalization brings the range of Euclidean distance component to $[0, 1]$, same as the range of sin component. This makes values of parameters meaningful relative to each other. The parameter c is used to make sin component not equal to zero, so that the metric can distinguish values along TCV, for which sin term is 0.

Fig. 7 gives heatmaps for various a and b values of UCfit. Parameter c is zero in those figures.

$$f_{UCfit}(\mathbf{RCV}) = \left(\frac{\|\mathbf{RCV}\|}{\|\mathbf{TCV}\|}\right)^a \cdot \left(\sin\left(\theta_{\mathbf{RUV}}(\mathbf{RCV})\right) + c\right)^b \tag{8}$$

The angle θ is always between 90 and 180°, regardless of the dimension of the vector space. This property makes applying UCfit fitness trivial to any number of dimensions.

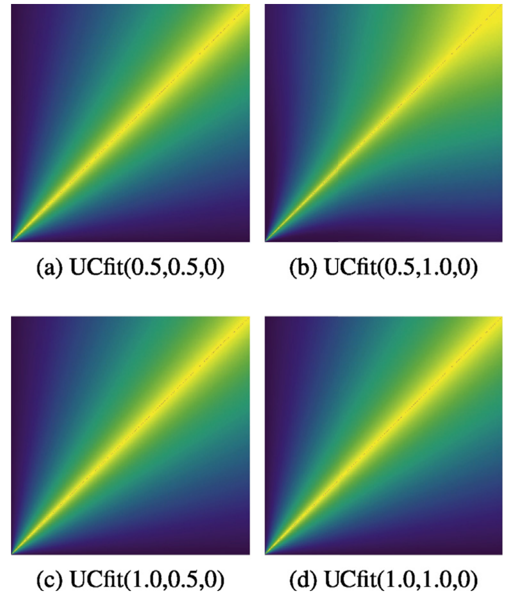


Fig. 7. Visualization of UCfit for different parameter values.

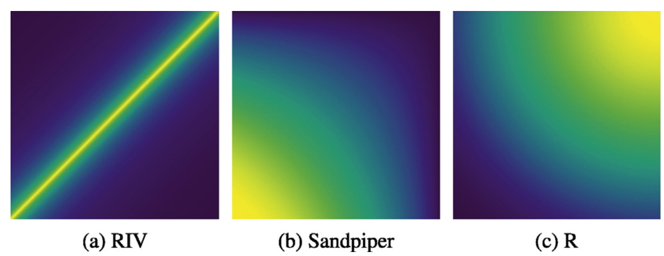


Fig. 8. Heatmaps for other metrics.

4.2. Other metrics

Below we also present some existing metrics that we use in our performance evaluation.

4.2.1. RIV metric

A metric can be based on the RIV vector as well (Mishra and Sahoo, 2011). RIV vector is resource imbalance vector and is a function of RCV. We will simply call this metric as RIV metric. This metric is not our contribution. We use it for comparison purposes. Fig. 8a gives the heatmap for RIV.

4.2.2. SandFit metric

The Sandpiper (Wood et al., 2009) uses a metric based on inverse of the available volume of a resource. Again it is a function of RCV, but it is inversely proportional. We call such a metric simply as Sandpiper metric in this paper. This metric is not our contribution, but we present and use it here for comparison. Fig. 8b gives the heatmap for Sandpiper.

4.2.3. R metric

R is a very simple metric that directly depends on the Euclidian distance to full utilization point. It is a naive metric that could be used in a utilization maximization scenario. Formally, it is equal to magnitude of *RCV*. Fig. 8c gives its heatmap.

5. Allocation heuristics based on our metrics

In this section we provide some efficient VM placement heuristic methods that also show how our metrics can be used.

As the first method, Algorithm 1 presents an *online* algorithm that allocates a given VM instance request to a suitable PM among infrastructure resources. Main strategy is to allocate the VM to a PM that is switched on. If the VM does not fit into any of the switched-on PMs, a sleeping PM is woken up to allocate the VM. It may be the case that the request exceeds capacity of resources available on the infrastructure (including switched-off physical machines), in which case the request is denied.

Algorithm 1 Online VM allocation.
ONLINE-ALLOCATE(Resources, VM)
1: $bestOn \leftarrow nil$
2: $bestOff \leftarrow nil$
3: **for all** $PM \in Resources$ **do**
4: **if** $RRV_{PM}(VM) \preceq RCV[PM]$ **then**
5: $rcv \leftarrow RCV(RUV[PM] + RRV_{PM}(VM))$
6: **if** $On[PM]$ and $f(rcv) < f(bestOnRCV)$ **then**
7: $bestOnRCV \leftarrow rcv$
8: $bestOn \leftarrow PM$
9: **else if** $Off[PM]$ and $f(rcv) < f(bestOffRCV)$ **then**
10: $bestOffRCV \leftarrow rcv$
11: $bestOff \leftarrow PM$
12: **end if**
13: **end if**
14: **end for**
15: **if** $bestOn \neq nil$ **then**
16: $RUV[bestOn] \leftarrow RUV[bestOn] + RRV_{bestOn}(VM)$
17: **else if** $bestOff \neq nil$ **then**
18: $RUV[bestOff] \leftarrow RRV_{bestOff}(VM)$
19: SWITCH-ON(Resources, bestOff)
20: **end if**
21: return *nil*

Capacity constraints are ensured in Line 4. A PM could host a VM if and only if RRV of VM is less than or equal to RCV of PM (i.e., demand is less than remaining capacity for all dimensions). This is denoted by $\preceq$ symbol and formally defined as:

$$\mathbf{u} \preceq \mathbf{v} \triangleq \mathbf{u_i} \leq \mathbf{v_i}, \forall i = 1 \ldots d \quad (9)$$

Similarly, $\prec$ symbol is defined as:

$$\mathbf{u} \prec \mathbf{v} \triangleq \mathbf{u_i} \leq \mathbf{v_i}, \forall i = 1 \ldots d \text{ and } \mathbf{u_j} < \mathbf{v_j}, \exists j = 1 \ldots d \quad (10)$$

Allocation vectors of PMs (RUV, RIV, RCV, and TCV) are normalized, they range from $\mathbf{0^d}$ to $\mathbf{1^d}$, whereas RRVs of VMs are not. In order to be able to compare RCV and RRV, we scale RRV of VM by the normalization factor of PM. This operation is denoted with a subscript of the normalization factor on scaled vector, such as $RRV_{PM}(VM)$.

When a sleeping PM is determined to host the request, $\text{Switch} - \text{On}()$ method is called to bring machine into operation. Basically, it removes the given PM from the set of sleeping machines, Off[Resources], and adds it to the set of machines in operation, On[Resources].

Online allocation runs in $O(m)$ time, where m is the number of resources.

We can also use our metrics in *offline* allocation methods that have a batch of VMs to place into a set of PMs. Next we describe an offline method using our metrics. But before describing it in detail, we introduce an auxiliary procedure, Algorithm 2, which allocates as much VMs as possible from a given set of requests onto a given PM. At each iteration, R contains requests that could fit into remaining space of the PM. This invariant is satisfied by the Lines 2 and 8. Among the requests in R the best fitting VM is allocated, until either all of the requests are allocated or there are no requests left that could fit into remaining space.

Algorithm 2 VM allocation to a given PM.
FILLPM(PM, Requests)
1: $Allocated \leftarrow \emptyset$
2: $R \leftarrow \{VM \in Requests \mid RRV_{PM}(VM) \preceq RCV[PM]\}$
3: **while** $R \neq \emptyset$ **do**
4: $BestVM \leftarrow \arg\min_{VM \in R} f(RCV(RUV[PM] + RRV_{PM}(VM)))$
5: $RUV[PM] \leftarrow RUV[PM] + RRV_{PM}(BestVM)$
6: $Allocated \leftarrow Allocated \cup \{BestVM\}$
7: $R \leftarrow R - \{BestVM\}$
8: $R \leftarrow R - \{VM \in R \mid RCV[PM] \prec RRV_{PM}(VM)\}$
▷*Remove VMs requesting larger than remaining capacity*
9: **end while**
10: $Requests \leftarrow Requests - Allocated$
11: **return** *Allocated*

Arg min operation in Line 4 can be implemented by scanning the set of requests and keeping the running minimum of calculated fitness function and the VM for which it is minimum. This takes $O(n)$ time, where n is the number of requests. Similarly, eliminating VMs in Lines 2, 7, and 8 could be achieved with a single pass over R, therefore taking $O(n)$ time. Line 10 could be performed by iterating over allocated items and removing them from requests in $O(kn)$ time, where k is the number of requests that could be allocated to the PM. The while loop iterates k times allocating a single VM in each iteration; therefore, total time complexity of the algorithm is $O(n + kn)$. In case of an allocation, kn term dominates. In the worst case, all of the requests could be allocated, which takes $O(n^2)$ time.

Algorithm 3 is our offline method that allocates a set of VM requests to a set of physical machines. It chooses a suitable PM for a given VM. Our offline algorithm uses complementary strategy, choosing suitable VMs for each PM. The offline algorithm begins by allocating requests to PMs that are currently switched on. Switched-on machines are sorted in descending order of their fitness value, so that imbalanced PMs are given priority to choose the best fitting VM for them. Remaining requests are allocated to sleeping/switched-off machines.

Algorithm 3 Offline VM allocation.
OFFLINE-ALLOCATE(Resources, Requests)
1: **for all** $PM \in \text{Sort}_{>_{f(RCV)}} On[Resources]$ **do**
2: FILLPM(PM, Requests)
3: **end for**
4: $Total \leftarrow \sum_{VM \in Requests} RRV[VM]$
5: $R \leftarrow \text{SORT}_{<_{f(RCV_{Total})}} Off[Resources]$
6: **for all** $PM \in R$
7: **if** FILLPM(PM, Requests) $\neq \emptyset$ **then**
8: $R \leftarrow R - \{PM\}_n$
9: SWITCH-ON(Resources, PM)
10: **else** ▷*PM cannot host any of the VMs*
11: $R \leftarrow R - \{r \in R \mid r \equiv PM\}_n$
12: **end if**
13: **end for**

Sum of RRVs of remaining requests are calculated so that PMs with resource ratios similar to that of overall requests get allocated first. Resource ratio similarity is calculated by the fitness function. Normally, RCVs of different PMs are not comparable, because they are normalized by different factors. Therefore, we renormalize them with the sum of RRVs to be able to compare. Beginning from the best fitting PM to worst fitting one, offline algorithm uses Algorithm 2 to allocate VMs. Note that for practical reasons, it would be wise to return from the procedure immediately when all of the requests are satisfied. The algorithm terminates when there are no requests left to allocate, or when remaining requests would not fit into any of the PMs because of capacity constraints.

It is important to define the data structures of resources, On[Resources] and Off[Resources], properly. On[Resources] is the set of PMs that are switched on. We use a simple list to implement this set. We could use a simple list to also keep the set of sleeping PMs, Off[Resources], but that would cause redundancy; instead, we use a counted set. Switched off PMs of the same type can be represented by a single element, since all of them have the same amount of each resource. There are many fewer distinct types of PMs than there are individual PMs, therefore it would be better to keep the list of PM types along with the number of individual machines of that type. This approach would not be suitable for On[Resources], because individual PMs would have different allocations resulting in different residual capacities even though they belong to the same PM type. Therefore, they cannot be practically aggregated as a single element.

Algorithm 3 uses counted set operations while processing Off[Resources]. Sorting in Line 5 is done with respect to elements (PM types) regardless of associated counts (individual machines). For loop in Line 6 iterates over individual machines. Loop variable *PM* is set to PMs corresponding to each count of each element in R. Note that R is modified inside the loop, therefore the loop should be considered to get the first element at each iteration rather than passing over a static list.

When a PM gets allocated some requests, count of its type is decreased by one. The iteration continues with next instance of the same type, assuming the count has not reached zero. When count of an element reaches zero, it is removed from the set. If a PM could not be allocated any requests, none of the instances of the same type could be. Therefore, corresponding PM type element is removed from the counted set, and iteration continues with an instance of next PM type.

Modifications on R is formally defined as a set difference operation. For counted sets, set difference operator returns a set in which the associated counts of elements in the first set is decreased by the associated counts of corresponding elements in the second set. Elements that have a count of zero are assumed to be removed from the set. In Line 8 only a single instance of a PM type is differentiated from R, therefore causing the count of the PM type decrease by one. In Line 11 the count of the PM type is set to zero by differentiating the set of all instances of a certain type. Therefore, the type of the PM is removed from R. Counted set is denoted by a subscripted n on closing set brace.

Allocating requests to On[Resources] takes $O(t \log t + tn + k_{on} n)$ time, where $t = |\text{On[Resources]}|$ and k_{on} is the number of requests that could be allocated to On[Resources]. $t \log t$ term is due to sorting. tn term comes from iterating all requests for each PM. Time complexity of allocating k_{on} requests is independent of the number of PMs to which they are allocated, therefore it takes $O(k_{on} n)$ time.

Line 4 takes linear time in n. Line 5 takes $O(s \log s)$ time, where s is the number of PM types.

Allocating requests to Off[Resources] takes $O(sn + k_{off}(n + s))$ time, where k_{off} is the number of requests that could be allocated to Off[Resources]. Allocating k_{off} resources takes $O(k_{off} n)$ time. For each allocation, PM type is removed from R and Off[Resources] in $O(s)$ time. When no resources could be allocated to a PM type, the for loop iterates s times scanning the list of requests for a suitable VM; hence the term sn.

Overall time complexity of Algorithm 3 is $O(t \log t + tn + k_{on} n + s \log s + sn + k_{off}(n + s))$. Ordering the terms semantically, we obtain $O(t \log t + s \log s + n(t + s) + n(k_{on} + k_{off}) + sk_{off})$. To simplify the terms, we can consider a common configuration of the cloud where $s \ll t \leq m$, in which case the time complexity becomes $O(m \log m + n(m + k))$, where $k = k_{on} + k_{off}$.

An alternative offline allocation algorithm is presented in Algorithm 4. It shares the outline of Algorithm 3, except it calculates the sum of remaining requests at each iteration and chooses the best fitting PM accordingly. Although costlier than the previous version, it handles high variance request sets better, since remaining requests would result in a different overall resource balance after each allocation to a PM. In terms of time complexity, $s \log s$ term is replaced by s^2 term because of the arg min calculation in each iteration of the while loop. The other difference, Line 13, takes $O(k_{off} n)$ time in the aggregate, therefore doesn't affect the time complexity because $k_{off} n$ term is already accounted for. Time complexity of the whole algorithm is, therefore, $O(t \log t + s^2 + n(t + s) + n(k_{on} + k_{off}) + sk_{off})$. Common case still runs in $O(m \log m + n(m + k))$, assuming that $s^2 < m \log m$, which is a safe assumption for large-scale cloud infrastructures.

Algorithm 4 Offline VM allocation alternative.
OFFLINE-ALLOCATE(Resources, Requests)
1: **for all** $PM \in \text{Sort}_{>f(RCV)} On[Resources]$ **do**
2: FILLPM(PM, Requests)
3: **end for**
4: $Allocated \leftarrow \emptyset$
5: $R \leftarrow$ Off[Resources]
6: $Total \leftarrow \sum_{VM \in Requests} RRV[VM]$
7: **While** $R \neq \emptyset$ or $Total \neq 0$ **do**
8: $BestPM \leftarrow \arg\min_{PM \in R} f(RCV_{Total}[PM])$
9: $Allocated \leftarrow$ FILLPM(BestPM, Requests)
10: **if** $Allocated \neq \emptyset$
11: $R \leftarrow R - \{BestPM\}_n$
12: SWITCH-ON(Resources, BestPM)
13: $Total \leftarrow Total - \sum_{VM \in Allocated} RRV[VM]$
14: **else** ▷*BestPM cannot host any of the VMs*
15: $R \leftarrow R - \{PM \in R \mid PM \equiv BestPM\}_n$
16: **end if**
17: **end While**

If all PMs are turned on and we have all VM requests in hand to place, we can look to each possible VM-PM pair while placing VMs into PMs to find out the best metric value. More specifically, we can consider VMs one by one and for each VM we can calculate the fitness value of placing that VM to each of the PMs and select the best fitting PM, and place the VM there. Then we can continue with the next VM in the list. The pseudocode of the method is shown in Algorithm 5. This is the algorithm that we use to compare various metrics in the evaluation section of the paper. We use this algorithm in our evaluations to do a fair comparison with other methods in literature (Gabay and Zaourar, 2016).

Algorithm 5 Offline VM allocation considering all pairs.
ALLOCATE(Resources, Requests)
for all $PM \in Resources$ **do**
for all $VM \in Requests$ **do**
if $RRV_{PM}(VM) \preccurlyeq RCV[PM]$ **then**
$rcv \leftarrow RCV(RUV[PM] + RRV_{PM}(VM))$
$fitness[PM, VM] \leftarrow f(rcv)$
else
$fitness[PM, VM] \leftarrow \infty$
end if
end for
end for
While $\min\{fitness[PM, VM] \mid \forall PM \in Res, \forall VM \in Req\} \neq \infty$ **do**
$BestPM, BestVM \leftarrow \arg\min_{PM \in Res, VM \in Req} fitness[PM, VM]$
$RUV[BestPM] \leftarrow RUV[BestPM] + RRV_{BestPM}(BestVM)$
$Requests \leftarrow Requests - \{BestVM\}$
for all $VM \in Requests$ **do**
if $RRV_{BestPM}(VM) \preccurlyeq RCV[BestPM]$ **then**
$rcv \leftarrow RCV(RUV[BestPM] + RRV_{BestPM}(VM))$
$fitness[BestPM, VM] \leftarrow f(rcv)$
else
$fitness[BestPM, VM] \leftarrow \infty$
end if
end for
end While

Fitness values are initialized in $O(mn)$ time. The while loop iterates k times by definition, because minimum of fitness values is ∞ when there are no more requests that could be allocated to any PM. Each iteration takes $O(mn)$ time due to arg min operation in Line 12 which considers all VM-PM pairs. Therefore, the whole algorithm runs in $O(kmn)$ time.

6. Simulation experiments and evaluation

To measure the performance of the proposed resource allocation metrics and methods, we developed two custom simulation environments.

The first one is implemented in Java and measures the effect of heterogeneity in cloud resources to allocation performance. The design is simple with few components. *PhysicalMachine* and *VirtualMachine* classes are subclasses of *AllocationResource* which is a model of vector operations. *Cloud* class consists of allocation algorithms and random generation of PM resources and VM requests. Many instances of *Cloud* are generated and allocation results are gathered by *Simulation* class. The code runs on any system with Java 5 or above.

The second simulation environment we developed aims to measure the performance of allocation methods we proposed and compare them with the ones from literature. It is based on open-source Python code provided by TeamJ19ROADEF2012 (2016). The codebase consists of four main files. *container.py* file models a problem *Instance* that consists of *Items* and *Bins*. *generator.py* file generates problem *Instances* according to different set of specifications and parameters. *measures.py* file contains allocation metrics which are used by the allocation algorithms defined in *heuristics.py* file. Finally, *benchmark.py* file runs a simulation consisting of many different problems.

We extended the existing code by implementing our fitness functions and integration code with the rest of the simulation environment in *measures.py* file. We also implemented *trace.py* file that keeps a trace of the simulation so that results are persisted and could be analyzed by importing them into *pandas* (NumFOCUS Foundation, 2008). We modularized *benchmark.py* file by introducing a *config.py* file that keeps a set of parameters and measures for which a simulation was run. Using these configuration files, we were able to implement *parallel.py* file that runs many simulations with different parameters in parallel, taking advantage of many-core architectures. The code runs on a Linux system with python 2.7 and *numpy* (NumFOCUS Foundation, 2006).

6.1. Effect of resource heterogeneity to allocation performance

Before evaluating our metrics and comparing them with others, we wanted to see the effect of heterogeneity. For this experiment we considered number of physical machine types as the metric of heterogeneity. When the cloud infrastructure consists of a single type of physical machine, it has a homogenous architecture. The more types of different physical machines make it more heterogeneous. Physical machine types are specified by the number of resource dimensions and minimum and maximum capacities for each dimension. The capacity of a resource in a dimension is drawn uniform randomly between the minimum and maximum capacity for that dimension. The number of physical machines available in the cloud for each type is also drawn uniform randomly from a specified interval. We examined the effect of number of machine types to request satisfaction ratio, i.e., the ratio of VMs that are requested and placed, to all VMs that are requested. In order to minimize the effect of randomness to simulation results, an average of 200 runs is presented.

Fig. 9 shows that request satisfaction increases as heterogeneity increases. Having a more heterogeneous infrastructure enables virtual machines of different resource demand compositions to find suitable resources, therefore increasing the request satisfaction ratio. As can be seen, request satisfaction ratio does not increase monotonically as the number of types of PMs increases. Peaks and lows of request satisfaction ratio correlate with those of cloud capacity in dimension 1, which is the resource constrained capacity that dominates the satisfaction ratio for this scenario.

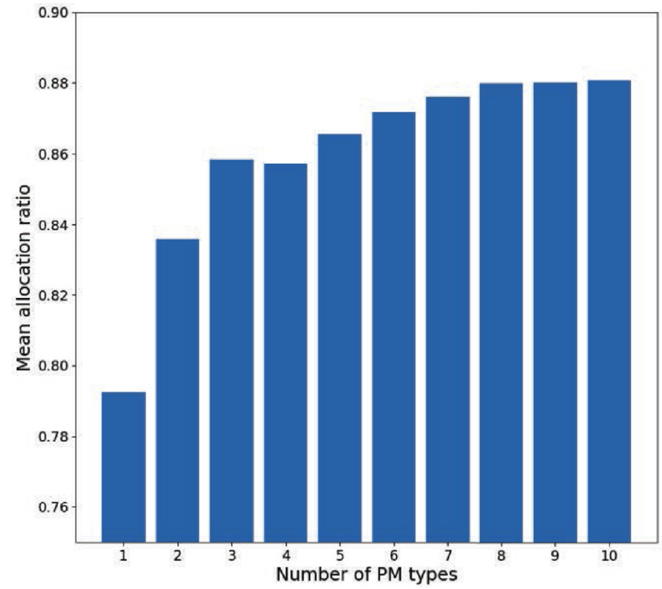


Fig. 9. Effect of heterogeneity.

6.2. Performance of proposed methods

Next we present our main results: the results of performance and comparison experiments. In these experiments, we want to compare various metrics and algorithms in terms of how well VMs are placed. For this we consider the following evaluation strategy. We generate feasible problem instances. A problem instance has a set of VMs (items) with d dimensional resource requirements to be placed into a set of PMs (bins) with d dimensional resource capacities. In a feasible problem instance all VMs can be placed into PMs. Various resource dimensions (d) and number of bins (PMs) are considered. A problem instance can be for example: place 300 items with some random request requirements into 50 bins with some random capacities. We generate the items to make an instance feasible. If placed in the generation order, all items will be placed. But while placing the items, the algorithm does not know the feasible order (which is very hard to enumerate).

Consider a toy-example cloud of 2 PMs with 2 resource dimensions. PMs have capacities (RCVs) of $\begin{pmatrix} 7 \\ 7 \end{pmatrix}$ and $\begin{pmatrix} 5 \\ 6 \end{pmatrix}$. 3 VM requests are to be allocated with resource requirements (RRVs) of $\begin{pmatrix} 4 \\ 3 \end{pmatrix}$, $\begin{pmatrix} 2 \\ 4 \end{pmatrix}$, and $\begin{pmatrix} 5 \\ 5 \end{pmatrix}$. VMs 1 and 2 could be allocated to PM 1, and VM 3 could be allocated to PM 2. Therefore, the problem is feasible.

We consider 5 different problem instance classes, as defined by Gabay and Zaourar (2016). These are: 1) random uniform, where bin capacities are chosen independently and in a uniform random manner (we call it as *uniform* in our result tables); 2) random uniform with rare resources, where bin capacities are again chosen randomly but one resource dimension is scarce (we call it as *uniform-rare*); 3) correlated capacities, where dimension capacities of a bin are correlated but resource requests are not (we call it as *correlated-false*); 4) correlated capacities and requirements, where both bin dimension capacities and resource dimension requirements are correlated (we call it as *correlated-true*); 5) similar items and bins, where resource requests are similar to bin capacities (we call it as *similar*).

Depending on the metric, not all VMs can be placed. If at least one VM in a problem instance cannot be placed, we consider that instance as unsolved (unsuccessful). If all VMs in a problem instance are placed,

Table 1
Number of problem instances solved (i.e., success count) by various methods for various number of resource dimensions (d). Each column gives the results for a different d value. There are 30 physical machines. Success count can be at most 500.

# bins # resources	30 2	30 3	30 4	30 5	30 6	30 7	30 8	Total
UCfit-2-1-02	480	384	289	193	123	121	121	1711
UCfit-2-1-025	475	392	287	191	129	111	117	1702
UCfit-2-15-02	465	368	289	191	130	119	123	1685
UCfit-3-2-01	469	378	282	197	125	119	115	1685
UCfit-2-1-03	477	386	289	194	118	110	105	1679
UCfit-3-1-01	476	386	293	183	119	111	111	1679
UCfit-2-1-01	463	374	279	189	124	120	128	1677
TRfit-pi-3	453	414	231	181	122	103	100	1604
TRfit-pi-2	470	399	228	177	119	100	100	1593
TRfit-pi-6	436	385	252	179	125	105	100	1582
TRfit-pi-12	410	348	256	160	109	102	100	1485
dp	450	366	186	129	103	99	100	1433
dp_normC	425	312	196	116	100	100	100	1349
bc_dyn_1/C	417	281	142	102	100	100	100	1242
bc_dyn_R/C	414	272	150	103	100	100	100	1239
R/C	410	270	145	104	100	100	100	1229
bc_dyn_1/R	403	269	132	100	100	100	100	1204
1/C	385	273	142	103	100	100	100	1203
1/R	378	263	141	103	100	100	100	1185
ic_dyn_R/C	350	254	152	104	100	100	100	1160
ic_dyn_1/C	344	245	144	103	100	100	100	1136
ic_dyn_1/R	323	232	141	105	100	100	100	1101
sbb_st_R/C	368	190	108	100	100	100	100	1066
sbb_st_1/R	363	193	108	100	100	100	100	1064
sbb_dyn_R/C	370	181	107	100	100	100	100	1058
sbb_st_1/C	366	184	107	100	100	100	100	1057
sbb_dyn_1/C	331	186	104	100	100	100	100	1021
sbb_dyn_1/R	318	186	108	100	100	100	100	1012
bb_dyn_1/R	221	128	101	100	100	100	100	850
bb_dyn_1/C	215	130	102	100	100	100	100	847
bb_st_1/R	214	125	100	100	100	100	100	839
bb_st_R/C	212	122	100	100	100	100	100	834
bb_st_1/C	207	125	101	100	100	100	100	833
bb_dyn_R/C	197	128	101	100	100	100	100	826
sandpiper	101	100	100	100	100	100	100	701
shuff1	141	81	85	71	65	68	68	579
nothing	152	78	73	75	64	67	66	575
dp_normR	173	100	36	15	30	46	56	456
sbb_nothing	63	40	33	31	16	20	17	220
sbb_shuff1	58	35	28	29	24	13	22	209
riv	92	34	6	1	0	1	6	140
bc_shuff	26	14	8	5	3	3	4	63
sbb_shuff	12	2	6	4	2	1	1	28
ic_shuff	8	1	0	0	0	0	0	9
bb_shuff1	4	1	0	1	0	0	1	7
bb_nothing	1	0	0	0	1	0	0	2
bb_shuff	0	0	0	0	0	0	2	2

we consider that as solved (success). We count, for each algorithm, the number of instances solved. The more, the better. For example, if use of a metric A enables 4200 of 5000 instances to be solved, we consider it better than a metric B that enables only 3700 of the same instances to be solved.

Consider the toy-example cloud defined above. Using UCfit$(2, 1, 0.2)$ we first calculate fitness values for all VM-PM pairs. Among 6 pairs VM 3 $\begin{pmatrix} 5 \\ 5 \end{pmatrix}$ - PM 2 $\begin{pmatrix} 5 \\ 6 \end{pmatrix}$ pair has the best fitness value of 0.013, therefore VM 3 is allocated to PM 2. PM 2 now has residual capacity (RCV) $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, which is less than RRVs of VM 1 or 2. Next, VM 1 $\begin{pmatrix} 4 \\ 3 \end{pmatrix}$ is placed into PM 1 $\begin{pmatrix} 7 \\ 7 \end{pmatrix}$ because its fitness value, 0.12, is better than that of VM 2 $\begin{pmatrix} 2 \\ 4 \end{pmatrix}$, 0.25. Finally, VM 2 is placed into remaining capacity of PM 1 $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$. As a result, UCfit$(2, 1, 0.2)$ is able to find a feasible solution to this problem instance.

Using dot product (dp) metric, for example, results in a different allocation. dp metric calculates dot products of RCV and RRV, and considers higher values to fit better. Among all VM-PM pairs, VM 3 $\begin{pmatrix} 5 \\ 5 \end{pmatrix}$

Table 2
Success count of various methods for various instance classes. Each column is for a different instance class. Success count of a method for a class can be at most 5400: 9 different d values and 6 different bin count values are used. That means a total of 54 different configurations considered. For each configuration, there are 100 instances.

Class ==>	cor-False	cor-True	similar	unif	unif-rare	Total
UCfit-2-1-02	2175	5395	3030	1240	700	12,540
UCfit-2-1-03	2198	5394	2790	1307	718	12,407
UCfit-2-1-01	2115	5394	3095	1158	629	12,391
UCfit-3-1	2088	5387	3021	1190	609	12,295
UCfit-2-2-02	2142	5394	3009	1091	610	12,246
UCfit-3-1-02	2192	5395	2219	1329	758	11,893
TRfit-pi-4	2595	5396	1443	1426	603	11,463
UCfit-2-3-02	1846	5395	2444	1081	621	11,387
UCfit-2-1	1802	5292	2885	884	443	11,306
TRfit-pi-2	2480	5393	1072	1477	803	11,225
TRfit-3pi-4	2365	5391	869	1394	842	10,861
UCfit-1-1-02	1330	5370	2691	768	460	10,619
dp	1695	5372	1142	1441	595	10,245
dp_normC	1919	5388	817	1325	319	9768
r	2003	5387	405	1035	667	9497
TRfit-0	593	5390	2341	647	111	9082
sandpiper	47	5369	5	3	7	5431
dp_normR	402	219	3027	122	177	3947
UCfit-1-1	100	387	2161	119	175	2942
riv	0	1	1573	0	38	1612

and PM 1 $\begin{pmatrix}7\\7\end{pmatrix}$ has the best fitness value of 70. Once VM 3 is placed into PM 1, RCV of PM 1 becomes $\begin{pmatrix}2\\2\end{pmatrix}$ which is not large enough to hold any of the remaining VMs. Next, VM 1 $\begin{pmatrix}4\\3\end{pmatrix}$ is allocated to PM 2 $\begin{pmatrix}5\\5\end{pmatrix}$, because its dot product, 35, is better than that of VM 2 $\begin{pmatrix}2\\4\end{pmatrix}$, 30. Residual capacity (RCV) of PM 2 is now $\begin{pmatrix}1\\3\end{pmatrix}$, therefore VM 2 is left unallocated. dp metric is considered to be unsuccessful because it fails to find a feasible allocation.

We compare our metrics, TRfit and UCfit, with some other metrics that are defined in other studies and by Gabay and Zaourar (2016). Table 1 shows the first set of results. The number of resource dimensions d is varied between 2 and 8. There are 30 physical machines (bin count is 30). There are 5 problem classes. For each class 100 instances are generated. Hence, for each resource dimension count d there are a total of 500 instances. Therefore the maximum success count can be 500 for a given resource dimension count, meaning that all 500 problem instances are solved. Total number of problems is 3500.

As can be seen in Table 1, our methods TRfit and UCfit perform much better than various other methods proposed in literature and by Gabay and Zaourar (2016). While our two methods with various parameters achieve a total success count of at least 1485, 42.4% of the problems for all resource dimensions, a lot of other methods are providing a total success count below 1000 (28.5%). Our best result is obtained by UCfit with 1711 problems (48.8%) solved successfully, whereas the

Table 3
Success counts of methods for various number of resource types (d) for correlated capacities (correlated-false) instance class. d is varied between 2 and 10. Each column shows the results for a different d value. There are 6 different bin counts considered (10, 20, 30, 40, 50, 100). The success count can be at most 600.

d ==>	2	3	4	5	6	7	8	9	10
TRfit-pi-4	599	576	524	427	279	134	55	1	0
TRfit-pi-2	599	589	536	415	232	102	6	1	0
TRfit-3pi-4	600	590	530	392	184	68	1	0	0
UCfit-2-1-03	594	567	502	335	151	48	0	0	1
UCfit-3-1-02	595	581	504	329	145	38	0	0	0
UCfit-2-1-02	593	559	490	335	152	46	0	0	0
UCfit-2-2-02	587	545	474	322	151	61	2	0	0
UCfit-2-1-01	585	549	477	320	136	48	0	0	0
UCfit-3-1	580	553	481	306	138	28	1	0	1
r	599	572	476	266	85	5	0	0	0
dp_normC	595	562	469	217	75	1	0	0	0
UCfit-2-3-02	590	521	394	231	91	18	1	0	0
UCfit-2-1	554	486	398	244	102	18	0	0	0
dp	590	517	367	169	48	3	1	0	0
UCfit-1-1-02	556	403	244	101	23	3	0	0	0
TRfit-0	315	176	67	30	5	0	0	0	0
dp_normR	271	113	17	1	0	0	0	0	0
UCfit-1-1	94	5	1	0	0	0	0	0	0
sandpiper	39	7	1	0	0	0	0	0	0
riv	0	0	0	0	0	0	0	0	0

Table 4
Success counts of methods for different *d* values and for uniform instance class. Success count can be at most 600.

d ==>	2	3	4	5	6	7	8	9	10
TRfit-pi-2	598	519	255	52	1	2	12	13	25
dp	590	508	231	42	1	7	11	20	31
TRfit-pi-4	592	491	237	52	3	5	10	11	25
TRfit-3pi-4	597	508	210	22	1	4	13	13	26
UCfit-3-1-02	589	482	197	6	1	2	12	14	26
dp_normC	586	468	201	16	2	1	5	15	31
UCfit-2-1-03	582	456	197	20	1	2	11	12	26
UCfit-2-1-02	575	423	174	18	1	1	11	12	25
UCfit-3-1	553	423	152	9	1	2	11	13	26
UCfit-2-1-01	564	397	142	4	1	0	10	14	26
UCfit-2-2-02	570	372	101	2	1	1	8	13	23
UCfit-2-3-02	575	386	86	1	2	2	5	9	15
r	580	375	39	1	0	2	10	10	18
UCfit-2-1	488	289	57	0	1	1	9	13	26
UCfit-1-1-02	521	200	13	0	1	2	6	6	19
TRfit-0	478	138	8	0	1	0	4	3	15
dp_normR	103	7	0	0	0	0	1	0	11
UCfit-1-1	84	18	0	0	1	0	4	3	9
sandpiper	3	0	0	0	0	0	0	0	0
riv	0	0	0	0	0	0	0	0	0

closest performing other method, dp, provides 1433 feasible solutions (40.9%). Our metric improves over performance of dp by 19.3%. There are metrics with very poor performance (less than 2.0%), which proves that choosing a good heuristic is important.

In the subsequent experiments we compared our parametric methods TRfit and UCfit with the best performing three methods from Gabay and Zaourar (2016), dp, dp_normC, and dp_normR, and three other methods from literature, r, riv, and sandpiper. 6 different bin counts are considered, 10, 20, 30, 40, 50, and 100, each having 9 different resource dimensions: 2–10. For each bin count and dimension count pair, we generate 100 problem instances for each instance class. Therefore, a total of 5400 instances are generated for each class, and the total number of instances is 27,000 across all classes.

Table 2 shows the performance of the selected best performing methods and our methods with different parameters for various problem classes. When the bin capacities at various dimensions are correlated, heuristics perform the best compared to other problem classes. When overall success count is considered, our UCfit metric performs the best. In total, it finds a feasible solution to 12,540 problem instances (46.4% of all problems). Our TRfit metric performs close to UCfit, where the best TRfit value is 11,463 (42.4%). Performance of other metrics from the literature is worse. The closest one, dp, solves 10,240 instances (37.9%), Compared to dp, UCfit performs 22.4% better. Riv performs the worst among the compared algorithms, satisfying just 1612 instances (5.9%), which is only 12.8% of our best result.

Tables 3 and 4 compare various methods for different number of resource dimensions (d). In Table 3 we see results for correlated capacities only (correlated-False class). The first thing that we notice is that when d increases, the placement becomes more difficult and the number of satisfied problem instances decreases dramatically for all metrics. For $d > 8$, problems become practically unsolvable with the proposed heuristics. TRfit achieves highest scores compared to other metrics, solving 2595 of 6000 problems (43.2%). Closest performing method from literature, r, is able to solve 2003 problems (33.3%). TRfit achieves 29.5% better performance than r. The difference between two methods are lower for easier problems. Both methods achieve above

Table 5
Success counts of methods for different *bin count* values and for *correlated-false* instance class (that means only dimension capacities are correlated not the dimension requirements). Success count can be at most 900.

bin count ==>	10	20	30	40	50	100	Total
TRfit-pi-4	236	333	410	463	504	649	2595
TRfit-pi-2	254	328	394	428	481	595	2480
TRfit-3pi-4	254	319	372	401	452	567	2365
UCfit-2-1-03	224	285	352	388	417	532	2198
UCfit-3-1-02	230	294	344	386	412	526	2192
UCfit-2-1-02	223	286	340	382	416	528	2175
UCfit-2-2-02	191	269	341	385	417	539	2142
UCfit-2-1-01	212	275	332	369	405	522	2115
UCfit-3-1	206	267	331	368	406	510	2088
r	222	271	321	349	371	469	2003
dp_normC	206	257	297	336	360	463	1919
UCfit-2-3-02	195	239	295	321	362	434	1846
UCfit-2-1	141	212	284	323	368	474	1802
dp	219	255	275	302	310	334	1695
UCfit-1-1-02	110	157	197	237	257	372	1330
TRfit-0	101	108	107	102	99	76	593
dp_normR	44	38	46	57	71	146	402
UCfit-1-1	25	15	10	17	13	20	100
sandpiper	38	8	1	0	0	0	47
riv	0	0	0	0	0	0	0

Table 6
Success counts of methods for different *bin count* values and for *correlated-true* instance class (both dimension capacities and requirements are correlated). Success count can be at most 900.

bin count ==>	10	20	30	40	50	100	Total
TRfit-pi-4	896	900	900	900	900	900	5396
UCfit-2-1-02	895	900	900	900	900	900	5395
UCfit-2-3-02	895	900	900	900	900	900	5395
UCfit-3-1-02	895	900	900	900	900	900	5395
UCfit-2-1-01	894	900	900	900	900	900	5394
UCfit-2-1-03	894	900	900	900	900	900	5394
UCfit-2-2-02	894	900	900	900	900	900	5394
TRfit-pi-2	893	900	900	900	900	900	5393
TRfit-3pi-4	892	899	900	900	900	900	5391
TRfit-0	890	900	900	900	900	900	5390
dp_normC	889	899	900	900	900	900	5388
r	888	899	900	900	900	900	5387
UCfit-3-1	889	898	900	900	900	900	5387
dp	873	900	899	900	900	900	5372
UCfit-1-1-02	885	894	899	896	897	899	5370
sandpiper	875	895	899	900	900	900	5369
UCfit-2-1	842	876	887	894	894	899	5292
UCfit-1-1	125	72	59	53	48	30	387
dp_normR	97	54	31	23	10	4	219
riv	1	0	0	0	0	0	1

95% performance for $d < 4$. The difference becomes larger for harder problems with $d > 4$, where TRfit solves 24.8% of the problems while r solves just 9.8%.

In Table 4 we see results for instances with uniform random capacities (uniform class). First of all, compared to correlated capacities, success rate of the methods are in general lower for uniform instance class. It is easier to place VMs into instances with correlated capacities, than with uniform random capacities. It is also interesting to note that when d gets larger, sometimes the success count can increase as well. For example, for various versions of UCfit heuristic, success count is more when d is 10 compared to when d is 5. This shows that depending on the capacities, having more dimensional resources does not always mean less success rate. The performance difference between TRfit and closest performing method from literature, dp, is significantly lower compared to correlated capacities class. TRfit solves 1477 of 6000 problems (24.6%) and dp solves 1441 (24.0%).

Tables 5–9 evaluate the performance of various methods for different classes and bin counts. Instances with correlated capacities and requirements (correlated-True class) enable the best success rate for the methods. Because both capacities and requirements are correlated, it is easier to find good placements for VMs compared to other problem classes where it is harder to distinguish between good and bad placements. Uniform random capacities with rare resources (uniform-rare class), on the other hand, is the most difficult to solve instance class.

Table 5 gives the success rate of the methods for various bin counts for the class of instances where only resource capacities but not requirements are correlated (correlated-False class). Note that the problem gets easier as the number of bins increases. The best performing method is TRfit with $\alpha = \pi/4$. It solves 2595 of 5400 problems (48.0%). Other TRfit metrics with different parameters are at the top as well, except TRfit(0). Then comes UCfit. The metrics r and dp rank around the middle, by solving 37.0% and 31.3% of the problems respectively. TRfit-pi-4 improves over the performance of r by 29.5%. The metrics sandpiper and riv are at the bottom. sandpiper is able to solve less than 1% of the problems, where riv fails to solve any.

Table 7
Success counts of methods for different *bin count* values and for *similar* instance class. Success count can be at most 900.

bin count ==>	10	20	30	40	50	100	Total
UCfit-2-1-01	733	571	466	425	431	469	3095
UCfit-2-1-02	708	553	463	414	414	478	3030
dp_normR	779	614	518	437	378	301	3027
UCfit-3-1	714	553	459	403	413	479	3021
UCfit-2-2-02	729	548	462	425	401	444	3009
UCfit-2-1	703	539	438	398	377	430	2885
UCfit-2-1-03	671	496	399	386	391	447	2790
UCfit-1-1-02	681	502	397	365	353	393	2691
UCfit-2-3-02	604	417	350	338	339	396	2444
TRfit-0	571	385	332	333	334	386	2341
UCfit-3-1-02	553	353	307	301	321	384	2219
UCfit-1-1	608	383	297	281	272	320	2161
riv	548	291	192	179	170	193	1573
TRfit-pi-4	239	202	218	236	250	298	1443
dp	160	157	186	200	202	237	1142
TRfit-pi-2	129	149	182	190	195	227	1072
TRfit-3pi-4	93	116	140	150	169	201	869
dp_normC	105	113	123	135	146	195	817
r	35	50	65	74	82	99	405
sandpiper	5	0	0	0	0	0	5

Table 8
Success counts of methods for different *bin count* values and for *uniform* instance class. Success count can be at most 900.

bin count ==>	10	20	30	40	50	100	Total
TRfit-pi-2	201	188	213	247	280	348	1477
dp	210	184	210	236	263	338	1441
TRfit-pi-4	194	176	208	241	263	344	1426
TRfit-3pi-4	205	176	197	235	261	320	1394
UCfit-3-1-02	182	172	198	229	243	305	1329
dp_normC	176	161	199	227	248	314	1325
UCfit-2-1-03	172	156	192	217	250	320	1307
UCfit-2-1-02	154	140	181	215	233	317	1240
UCfit-3-1	146	138	171	206	225	304	1190
UCfit-2-1-01	147	132	169	198	216	296	1158
UCfit-2-2-02	148	123	159	181	207	273	1091
UCfit-2-3-02	142	132	156	184	201	266	1081
r	145	128	158	174	200	230	1035
UCfit-2-1	110	85	117	156	172	244	884
UCfit-1-1-02	93	80	107	132	153	203	768
TRfit-0	81	80	94	111	124	157	647
dp_normR	32	14	11	11	7	47	122
UCfit-1-1	35	14	10	11	15	34	119
sandpiper	3	0	0	0	0	0	3
riv	0	0	0	0	0	0	0

Table 6 gives the success rate of the methods for various bin counts for class of instances with correlated capacities and requirements (correlated-True). This is the easiest class of all instance classes. Almost all methods perform above 98.0%.

Table 7 gives the success count of the methods for various bin counts for the class of instances that have similar capacities (similar). The problems become harder to solve as the number of bins increases. This time UCfit method performs the best, solving 3095 of 5400 problems (57.3%). The metric dp_normR is very close with 56.0% performance. This is the only class of problems for which dp_normR shows reasonable performance. It performs poorly in all other cases. riv has the same performance characteristic with dp_normR, though it has worse performance, 29.1%. The metric r performs poorly, 7.5%, and sandpiper is the worst.

Table 8 gives the success count of the methods for various bin counts when bin capacities are drawn randomly from a uniform distribution (uniform). The problems become easier to solve as the number of bins increases. TRfit-pi-2 and dp are performing the best, with 27.3% and 26.6% performance respectively. In general, TRfit is ahead of UCfit for this class of instances. The best performing TRfit improves over the performance of the best performing UCfit by 11.1%. sandpiper and riv have the worst performance.

Table 9 gives the success count of the methods for various bin counts when bins are generated in the same manner as in uniform class, but the capacity of the last dimension is set to zero with certain probability, indicating a rare resource in the cloud (uniform-rare). Note that this is the most difficult of all problem classes. The problems become slightly harder to solve for intermediate number of bins. They are easier for lower and higher number of bins. The best performing TRfit solves 842 of 5400 problems (15.5%). The best performing UCfit is slightly behind with 758 problems (14.0%). Performance of metric r is relatively good (12.3%), but TRfit is able to achieve 26.2% better performance. Metrics riv and sandpiper again perform the worst with less than 1% performance.

Table 9
Success counts of methods for different *bin count* values and for *uniform-rare* instance class. Success count can be at most 900.

bin count ==>	10	20	30	40	50	100	Total
TRfit-3pi-4	228	115	114	109	116	160	842
TRfit-pi-2	229	114	104	95	101	160	803
UCfit-3-1-02	220	103	98	98	100	139	758
UCfit-2-1-03	205	94	86	93	103	137	718
UCfit-2-1-02	205	90	93	91	93	128	700
r	183	89	92	91	94	118	667
UCfit-2-1-01	187	82	82	81	87	110	629
UCfit-2-3-02	159	80	83	86	93	120	621
UCfit-2-2-02	169	82	78	81	84	116	610
UCfit-3-1	182	75	72	76	82	122	609
TRfit-pi-4	219	91	73	58	65	97	603
dp	196	79	63	65	75	117	595
UCfit-1-1-02	136	65	51	58	60	90	460
UCfit-2-1	145	53	49	51	60	85	443
dp_normC	116	31	30	31	37	74	319
dp_normR	93	31	21	9	13	10	177
UCfit-1-1	79	31	18	18	11	18	175
TRfit-0	82	15	10	2	1	1	111
riv	31	6	0	0	1	0	38
sandpiper	7	0	0	0	0	0	7

7. Conclusion

As a cloud computing service, Infrastructure as a Service (IaaS) operators provide public access to their infrastructures. Cloud customers use cloud resources via virtual machines, on which they can run arbitrary software. One of the challenges faced by providers in this type of cloud computing is efficient allocation of physical resources to VM requests so that the number of customers whose requests are satisfied can be increased.

In this paper we provide metrics and methods that enable efficient packing of VMs into physical machines so that consumer requests are totally satisfied in as many cases as possible. For this, we first propose two novel and generic resource utilization metrics, called TRfit and UCfit, that measure the goodness of a current allocation. A measure is always monitored and used while virtual machines are being placed into physical machines to have a utilization state that is as suitable as possible. Hence the metrics are used in selecting VMs and PMs to make a good assignment. We also show how these metrics can be used as part of some sample VM placement algorithms. In this way we provide complete and generic VM placement methods that can be applied for various resource types and counts. Our measures are parametric. Each different selection of a parameter enables a new sub-metric that can be more suited for particular cases. The methods are heuristic-based, therefore they run very fast and in polynomial time with respect to request count and machine count.

We compared our methods both internally, for different parameter settings, and also externally with some other methods from literature in terms of number of VM placement problems that can be solved completely—without having any unplaced VMs. We investigated the effect of physical machine count, resource type count, and also problem instance classes. For all cases, our methods perform better than existing methods, by 22.4% overall, and by up to 29.1% when individual problem classes are considered. Relative ranks of our methods against each other varies depending on the case.

As a future work, the methods we proposed could be extended for allocating requests with physical proximity constraints, so that VMs that would serve for the same application could communicate faster among each other. Such a method could consider cloud as a tree of resources, and allocate subsets of a set of VMs in neighboring sub-trees according to fitness values.

Declaration of interests

None.

Acknowledgements

This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) projects 113E274 and 116E048.

References

Alicherry, M., Lakshman, T.V., 2012. Network aware resource allocation in distributed clouds. In: IEEE INFOCOM, pp. 963–971, https://doi.org/10.1109/INFCOM.2012.6195847.

AMAZON.com Inc, 2006. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. (Accessed 26 June 2012).

Arzuaga, E., Kaeli, D.R., 2010. Quantifying load imbalance on virtualized enterprise servers. In: Workshop on Software and Performance, pp. 235–242, https://doi.org/10.1145/1712605.1712641.

Beloglazov, A., Abawajy, J., Buyya, R., 2012. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. Future Gener. Comput. Syst., https://doi.org/10.1016/j.future.2011.04.017.

Chen, L., Shen, H., 2014. Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters. In: IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, pp. 1033–1041, https://doi.org/10.1109/INFOCOM.2014.6848033.

Gabay, M., Zaourar, S., 2016. Vector bin packing with heterogeneous bins: application to the machine reassignment problem. Ann. Oper. Res. 242, 161–194, https://doi.org/10.1007/s10479-015-1973-7.

Garg, S.K., Toosi, A.N., Gopalaiyengar, S.K., Buyya, R., 2014. Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter. J. Network Comput. Appl. 45, 108–120, https://doi.org/10.1016/j.jnca.2014.07.030.

Johnson, D.S., 1974. Approximation algorithms for combinatorial problems. J. Comput. Syst. Sci. 9, 256–278, https://doi.org/10.1016/S0022-0000(74)80044-9.

Lee, Y.C., Zomaya, A.Y., 2009. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. In: Cluster Computing and the Grid, pp. 92–99, https://doi.org/10.1109/CCGRID.2009.16.

Masdari, M., Nabavi, S.S., Ahmadi, V., 2016. An overview of virtual machine placement schemes in cloud computing. J. Netw.Comput. Appl. 66, 106–127, https://doi.org/10.1016/j.jnca.2016.01.011.

Mell, P., Grance, T., 2011. The Nist Definition of Cloud Computing. NIST Special Publication 800-145. .

Meng, X., Pappas, V., Zhang, L., 2010. Improving the scalability of data center networks with traffic-aware virtual machine placement. In: IEEE INFOCOM, pp. 1154–1162, https://doi.org/10.1109/INFCOM.2010.5461930.

Mezmaz, M., Melab, N., Kessaci, Y., Lee, Y.C., Talbi, E.G., Zomaya, A.Y., Tuyttens, D., 2011. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. J. Parallel Distrib. Comput. 71, 1497–1508, https://doi.org/10.1016/j.jpdc.2011.04.007.

Mills, K., Filliben, J., Dabrowski, C., 2011. Comparing VM-placement algorithms for on-demand clouds. In: IEEE International Conference on Cloud Computing Technology and Science, https://doi.org/10.1109/CloudCom.2011.22.

Mishra, M., Sahoo, A., 2011. On theory of VM placement: anomalies in existing methodologies and their mitigation using a novel vector based approach. In: IEEE International Conference on Cloud Computing, pp. 275–282, https://doi.org/10.1109/CLOUD.2011.38.

NumFOCUS Foundation, 2006. NumPy. https://www.numpy.org. (Accessed 15 May 2019).

NumFOCUS Foundation, 2008. Pandas: Python Data Analysis Library. https://pandas.pydata.org. (Accessed 15 May 2019).

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D., 2009. The eucalyptus open-source cloud-computing system. In: 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 124–131, https://doi.org/10.1109/CCGRID.2009.93.

Panigrahy, R., Prabhakaran, V., Talwar, K., Wieder, U., Ramasubramanian, R., 2011a. Validating Heuristics for Virtual Machines Consolidation. Technical Report https://www.microsoft.com/en-us/research/publication/validating-heuristics-for-virtual-machines-consolidation/.

Panigrahy, Rina, Talwar, Kunal, Uyeda, Lincoln, Wieder, Udi, 2011b. Heuristics for Vector Bin Packing. Technical Report. Microsoft Research.

Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., Stoica, I., 2012. FairCloud: Sharing the Network in Cloud Computing, pp. 187–198, https://doi.org/10.1145/2342356.2342396.

Rackspace US Inc, 2008. Cloud Server and Virtual Server Hosting by Rackspace. http://www.rackspace.com/cloud/servers. (Accessed 26 June 2012).

Shrivastava, V., Zerfos, P., won Lee, K., Jamjoom, H., Liu, Y.H., Banerjee, S., 2011. Application-aware virtual machine migration in data centers. In: IEEE INFOCOM, pp. 66–70, https://doi.org/10.1109/INFCOM.2011.5935247.

Singh, H., hau Lee, M., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Filho, E.M.C., 2000. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. Comput. 49, 465–481, https://doi.org/10.1109/12.859540.

Singh, A., Korupolu, M., Mohapatra, D., 2008. Server-storage virtualization: integration and load balancing in data centers. In: Supercomputing Conference, https://doi.org/10.1109/SC.2008.5222625.

Stillwell, M., Schanzenbach, D., Vivien, F., Casanova, H., 2010. Resource allocation algorithms for virtualized service hosting platforms. J. Parallel Distrib. Comput. 70, 962–974, https://doi.org/10.1016/j.jpdc.2010.05.006.

TeamJ19ROADEF2012, 2016. Variable-Size-Vector-Bin-Packing. https://github.com/TeamJ19ROADEF2012/Variable-Size-Vector-Bin-Packing. (Accessed 15 May 2019).

Toyoda, Y., 1975. A Simplified Algorithm for Obtaining Approximate Solutions to Zero-One Programming Problems.

Wood, T., Shenoy, P.J., Venkataramani, A., Yousif, M.S., 2009. Sandpiper: blackbox and gray-box resource management for virtual machines. Comput. Netw. 53, 2923–2938, https://doi.org/10.1016/j.comnet.2009.04.014.

Ye, X., Yin, Y., Lan, L., 2017. Energy-efficient many-objective virtual machine placement optimization in a cloud computing environment. IEEE Access 5, 16006–16020, https://doi.org/10.1109/ACCESS.2017.2733723.

Zhang, X., Tian, Y., Jin, Y., 2015. A knee point-driven evolutionary algorithm for many-objective optimization. IEEE Trans. Evolut. Comput. 19, 761–776, https://doi.org/10.1109/TEVC.2014.2378512.

Zhang, J., Huang, H., Wang, X., 2016. Resource provision algorithms in cloud computing: a survey. J. Netw. Comput. Appl. 64, 23–42, https://doi.org/10.1016/j.jnca.2015.12.018.

Cem Mergenci is currently a PhD candidate in the Computer Engineering Department at Bilkent University, Turkey. He received his MS and BS degrees from Bilkent University, both in Computer Engineering. His research interests are cloud computing, resource allocation algorithms, wireless networks and distributed systems.

Ibrahim Korpeoglu received his Ph.D. and M.S. in Computer Science from University of Maryland at College Park in 2000 and 1996, respectively, under the supervision of Prof. Satish Tripathi. He received his B.S. degree (summa cum laude) in Computer Engineering, from Bilkent University in 1994. He is a full professor in Department of Computer Engineering at Bilkent University. Before joining Bilkent, he worked in Ericsson, IBM T.J. Watson Research Center, Bell Laboratories, and Bellcore, in USA. He received Bilkent University Distinguished Teaching Award in 2006 and IBM Faculty Award in 2009. He speaks Turkish, English and German. His research interests include computer networks, wireless networks, cloud computing, and distributed systems. He is a member of ACM and a senior member of IEEE.